2010-09-01

Fexprs as the basis of Lisp function application; or, $vau: the ultimate abstraction

John N. Shutt
Worcester Polytechnic Institute

Fexprs as the basis of Lisp function application or `$vau`: the ultimate abstraction

by

John N. Shutt

A Dissertation
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy
in
Computer Science

August 23, 2010

Approved:

Prof. Michael A. Gennert, Advisor
Head of Department

Prof. Daniel J. Dougherty

Prof. Carolina Ruiz

Prof. Shriram Krishnamurthi
Computer Science Department, Brown University

Abstract

Abstraction creates custom programming languages that facilitate programming for specific problem domains. It is traditionally partitioned according to a two-phase model of program evaluation, into syntactic abstraction enacted at translation time, and semantic abstraction enacted at run time. Abstractions pigeon-holed into one phase cannot interact freely with those in the other, since they are required to occur at logically distinct times.

Fexprs are a Lisp device that subsumes the capabilities of syntactic abstraction, but is enacted at run-time, thus eliminating the phase barrier between abstractions. Lisps of recent decades have avoided fexprs because of semantic ill-behavedness that accompanied fexprs in the dynamically scoped Lisps of the 1960s and 70s.

This dissertation contends that the severe difficulties attendant on fexprs in the past are not essential, and can be overcome by judicious coordination with other elements of language design. In particular, fexprs can form the basis for a simple, well-behaved Scheme-like language, subsuming traditional abstractions without a multi-phase model of evaluation.

The thesis is supported by a new Scheme-like language called Kernel, created for this work, in which each Scheme-style procedure consists of a wrapper that induces evaluation of operands, around a fexpr that acts on the resulting arguments. This arrangement enables Kernel to use a simple direct style of selectively evaluating subexpressions, in place of most Lisps' indirect quasiquotation style of selectively suppressing subexpression evaluation. The semantics of Kernel are treated through a new family of formal calculi, introduced here, called *vau calculi*. Vau calculi use direct subexpression-evaluation style to extend lambda calculus, eliminating a long-standing incompatibility between lambda calculus and fexprs that would otherwise trivialize their equational theories.

The impure vau calculi introduce non-functional binding constructs and unconventional forms of substitution. This strategy avoids a difficulty of Felleisen's lambda-v-CS calculus, which modeled impure control and state using a partially non-compatible reduction relation, and therefore only approximated the Church–Rosser and Plotkin's Correspondence Theorems. The strategy here is supported by an abstract class of Regular Substitutive Reduction Systems, generalizing Klop's Regular Combinatory Reduction Systems.

Preface

The concept of *first-class object* is due to Christopher Strachey (1916–1975).

A first-class object in any given programming language is an object that can be employed freely in all the ways that one would ordinarily expect of a general value in that language. The "rights and privileges" of first-class objects —the ways one expects they may be freely employed— depend on the language. Although one might draw up a partial list of first-class rights and privileges for a given language, e.g. [AbSu96, §1.3.4], a complete list is never quite possible, because (as in human societies) the rights and privileges of first-class-ness are difficult to recognize until they are missed.

Strachey's prime example, then, of *second*-class objects was procedures in Algol. Procedures in Algol can be called, or passed as arguments to procedure calls, but —unlike numbers— cannot be stored as the values of variables, nor returned as the results of procedure calls. Thus Algol procedures cannot be denoted by compound expressions, nor by aliases; as [Stra67, §3.5] put it, they "need to appear in person under their own names."

The Scheme programming language pioneered first-class treatment of several types of objects, notably procedures (whose promotion to first-class status will be discussed in this dissertation in §3.3.2). However, for those cases in which the operands in an expression are not to be evaluated automatically, special reserved operator symbols are used; and the evaluation rules denoted by these reserved symbols cannot be evoked in any other way — they always have to appear in person under their own names.[1]

The Kernel programming language is a redesign of Scheme for the current work[2] that grants first-class status to the objects named by Scheme's special-form operators (both built-in and user-defined). Why, how, and with what consequences it does so are the concerns of this dissertation.

In place of Scheme's single type of first-class procedures, Kernel has two types of first-class combiners: applicatives, which are analogous to Scheme procedures; and operatives (known historically in the Lisp community as *fexprs*), which take their operands unevaluated and correspond to Scheme's special-form combiners. To avoid gratuitous confusion between the two, Kernel conventionally prefixes the names of its

[1] A more detailed discussion of first-class-ness, both in general and in the particular case of Scheme, appears in [Shu09, App. B (First-class objects)].

[2] That is, Kernel and this dissertation are both part of the same work, by the same author. See §3.5.

operatives with "$" (a convention that enhances the lucidity of even mundane Kernel code, independent of any exotic use of first-class operatives).

The elegance of Kernel's support for first-class operatives arises from the synergism of two distinct innovations.

The flashier innovation is the *$vau* operative, which behaves nearly identically to *$lambda* except that its result is operative rather than applicative. *$vau* also differs from *$lambda* by having an extra parameter, appearing after the usual parameter tree, which locally binds the dynamic environment from which the constructed operative is called. One could, for example, write

```
($define! $if
   ($vau (x y z) env
      ($cond ((eval x env) (eval y env))
             (#t           (eval z env)))))
```
(0.1)

deriving a compound operative *$if* from pre-existing operative *$cond*. As an alternative to hygienic macro declarations, *$vau* gains clarity by using ordinary Scheme/Kernel tools rather than a separate macro sublanguage, and by specifying its evaluations —and its uses of the dynamic environment— explicitly; but the more elegant, and subtler, innovation of Kernel's operative support lies in its treatment of first-class *applicatives*.

A Kernel applicative is simply a shell, or *wrapper*, to induce argument evaluation, around some other underlying combiner (which may even be another applicative). The constructor of applicatives is an applicative *wrap*. The introduction of *wrap* as a primitive, evidently orthogonal to *$vau*, obviates the need for a primitive *$lambda*, since *$lambda* can then be constructed as a compound operative:

```
($define! $lambda
   ($vau (ptree . body) static-env
      (wrap (eval (list* $vau ptree #ignore body)
                  static-env))))
```
(0.2)

The introduction of an inverse operation *unwrap* completes the orthogonalization of Kernel combiner semantics, by allowing *apply* to be constructed as well:

```
($define! apply
   ($lambda (applicative argument-tree)
      (eval (cons (unwrap applicative) argument-tree)
            (make-environment)))).
```
(0.3)

Content of the dissertation

The main content of the dissertation is divided into two parts: Part I addresses Kernel (the practical instrument of the thesis, designed by the author for the extended work), while Part II addresses vau calculi (its theoretical instrument, designed as part of the

dissertation). Each part begins with a chapter of background materials preliminary to that part. Chapter 1 contains background materials that motivate and clarify the thesis, and therefore logically precede both parts. Chapter 2 contains background materials on the use of language in the dissertation that cross-cut the division between theory and practice.

Much of the background material is historical. Historical background serves three functions: it clarifies where we are, by explaining how we got here; it explains *why* we got here, laying motivational foundation for where to go next; and it reviews related work.

The background chapters also concern themselves not only with how programming languages specify computations (semantics), but with how programming languages influence the way programmers think (psychological bias). This introduces a subjective element, occasionally passing within hailing distance of philosophy; and the thesis, since it is meant to be defended, mostly avoids this element by concerning itself with semantics. However, the background chapters are also meant to discuss language design motivations, and psychology is integral to that discussion because it often dominates programmer productivity. Semantics may make some intended computations difficult to specify, but rarely makes them impossible — especially in Lisp, which doesn't practice strong typing; whereas psychological bias may prevent entire classes of computations outright, just by preventing programmers from intending them.

Chapter 16 contains concluding remarks on the dissertation as a whole.

Several appendices contain material either tediously uninsightful (and therefore deferred from the dissertation proper) but technically necessary to the thesis defense; or not directly addressing the thesis, but of immediate interest in relation to it. Appendix A gives complete Kernel code for the meta-circular evaluators of Chapter 6. Appendix B discusses efficient compilation of Kernel programs. Appendix C discourses briefly on the history of the letter vau, and how its typeset form for this document was chosen.

Acknowledgements

My work on this dissertation owes a subtle intellectual debt to Prof. Lee Becker (1946–2004), who served on my dissertation committee from 2001 until his passing in July 2004; and who introduced me to many of the programming-language concepts that shaped my thinking over the years, starting at the outset of my college career (in the mid-1980s) with my first exposure to Lisp.

I owe a more overt debt to the members of my dissertation committee, past and present —Lee Becker, Dan Dougherty, Mike Gennert, Shriram Krishnamurthi, and Carolina Ruiz— for their support and guidance throughout the formal dissertation process.

Between these extremes (subtle and overt), thanks go to my current graduate ad-

visor, Mike Gennert, and to my past graduate advisor, Roy Rubinstein, for nurturing my graduate career so that it could reach its dissertation phase.

Particular thanks to Mike Gennert, also, for drawing me back into the Scheme fold, after several years of apostasy in the non-Lisp wilderness where I would never have met *$vau*.

For discussions, feedback, and suggestions, thanks to the members of NEPLS (**N**ew **E**ngland **P**rogramming **L**anguages and **S**ystems Symposium Series).

Thanks also to my family, especially my parents, for putting up with me throughout the dissertation ordeal; and to my family, friends, and colleagues, for letting me try out my ideas on them (repeatedly).

l

Contents

List of Tables and Figures

List of Definitions

List of Theorems

Chapter 1

The thesis

1.1 Abstraction

> The acts of the mind, wherein it exerts its power over its simple ideas, are chiefly these three: (1) Combining several simple ideas into one compound one; and thus all complex ideas are made. (2) The second is bringing two ideas, whether simple or complex, together, so as to take a view of them at once, without uniting them into one; by which way it gets all its ideas of relations. (3) The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction: and thus all its general ideas are made.
>
> — John Locke, *An Essay Concerning Human Understanding* ([Lo1690]), Bk. II Ch. xii §1.[1]

Ideally, one would like to express each program in terms of an abstract view of computation —i.e., in a programming language— that is natural to its intended problem domain. The ideal is pursued by a process of *abstraction*, in which new abstract views of computation are derived from pre-existing views.

In other words, abstraction uses the facilities of an existing programming language to define a new programming language (presumably, a new language closer to what we want for some intended problem domain).[2]

[1] Traditionally, epigraph attributions gave only the name of the author and perhaps the title of the work; but that approach is only manageable if the body of literature is small and tightly controlled. Epigraphs tend to be worth repeating, being often selected for just that property; and in the modern electronic culture, any underattributed quotation that is worth repeating will tend over time to become a misattributed misquotation: if not enough information is provided to readily double-check the source, replication errors are unlikely to be corrected when they occur.

[2] For the philosophically inclined reader:

What we first called an *abstract view of computation*, and then (and hereafter) a *programming*

This section traces the history of abstraction in programming, and explains why the development of well-behaved fexprs is central to advancing the state of the abstractive art. Fexprs are contrasted with macros; and most of the major elements of the thesis are derived.

1.1.1 Some history

In the early years of computing, people called *programmers* designed programs and wrote them down in mnemonic shorthand, which was then given to people called *coders* who translated the shorthand into binary machine code. Later, programmers wrote input for assembler programs, and later still, programmers in most problem domains wrote input for compiler programs (over some objections that this reduces programmer control of the machine[3]).

Throughout this historical progression, the *source*, or *programming*, language (which the programmer writes) exists ultimately to be translated into a *target* language of a different order (which the machine reads). Translation of assembly source to machine target is nearly one-to-one (except for expansion of calls to macros, which were in use by the late 1950s and will be discussed momentarily).

The idea of an abstract target language for a programming language was promoted by the UNCOL movement, which peaked around 1960 and faded in the following few years.[4] An UNCOL (*UNiversal Computer Oriented Language*) was supposed to be an intermediate-level language into which all programming languages could be translated, and from which all machine languages could be generated ([Sa69, §x.2]).

In the UNCOL model, programming languages are treated as monolithic entities, each specified by its translator into UNCOL. The idea that each program could define its *own* specialized programming language was advanced by the *extensible-languages*

language, corresponds to what John Locke in the above epigraph called an *idea* — though Locke notably didn't delineate just what an "idea" is. Our requirement that the new programming language be defined using facilities of the pre-existing language corresponds to Locke's constraint that abstraction only 'separates' a general idea from a complex of ideas in which it was already latently present.

When relating the principle of abstraction in programming to the same-named principle in metaphysics, we prefer Locke's account of abstraction over that of other classical philosophers because he casts abstraction in the role of a constructive process, not because we have any essential commitment to Locke's conceptualism. As computer scientists, we don't care whether our programming languages exist in reality (Plato), in concept (Locke), or only in name (Ockham), as long as we can design and use them.

[3]The objectors were quite right, of course.

[4]We're simplifying our selective tour of programming-language design history by identifying major ideas with major trends in which they occurred. Each idea has earlier roots — as does each trend. In the case of UNCOL, [Share58, p. 14] remarks,

> It might not be difficult to prove that "this was well-known to Babbage," so no effort has been made to give credit to the originator, if indeed there was a unique originator.

movement of the 1960s.[5] Classically, an extensible language consists of a source-level *base language* together with meta-level *definitional facilities* to extend the base language ([Chr69]). A program then consists of two parts, a meta-level part that specifies an extension of the base language followed by a source-level part written in the extended language.

Traditional macros use simple polynomial substitution to perform localized source-code transformations from the extended language to the base language, which can then correspond substantially one-to-one with a simple abstract target language. Because polynomial substitution is entirely independent of the semantics of the base (and of the target) language, it was well suited for language extension in an era when unstructured programming languages were commonplace; and, because its simplicity incurred no start-up development time, it was available for immediate deployment. Accordingly, traditional macros were the principal means of extension used in the extensible-languages movement, to the point of near-synonymity with the movement as a whole. (A second technology of *adaptive grammars* was also begun by the extensible-languages movement, but required significant development time, and consequently would not mature until long after the extensible-languages movement had died.[6])

The extensible-languages movement peaked around 1970, and faded rapidly over the next few years. Its fatal flaw was complexity: one layer of extension to the base language might be effected by an ordinary user, but additional layers were difficult to manage, as each additional layer would have to attend to the details of all the layers that preceded it ([Sta75]). In retrospect, this was a limitation especially of the extension technology used, traditional macros. Because a traditional macro call expands source-to-source, the call is only permissible if the code it expands to would also be permitted, so that every facility used by the macro must be visible where the macro is called. Hence, if multiple layers of macros are built up, the macro programmer has to contend simultaneously with the cumulative detail of all the layers.

The idea of selectively hiding information, thus mitigating the information overload from successive language extensions, was advanced by a new *abstraction* movement in language design, which emerged as the extensible-languages movement ebbed (and which would later come to be dominated by the object-oriented paradigm, much as extensibility had been by macros). The new movement did view abstraction as a process of language construction (e.g., [Dij72]); but as just noted, traditional macros

[5] M.D. McIlroy's 1960 paper on macros, [McI60], is usually cited in association with the beginning of the extensible-languages movement.

[6] The idea of adding new grammar rules to a base language occurs in one of the earliest articulations of the extensibility principle, [BrMo62] (note that the paper was received in 1960). The idea of an internally adaptive grammar came somewhat later ([Car63]), and adaptive grammar formalisms only began to emerge in the 1970s (e.g., [Wegb80]), as the extensible-languages movement was winding down. Adaptive grammars then developed largely as a fringe technology until around 1990; see [Shu03b, Ch. 3], and for some more recent developments, `http://www.pcs.usp.br/~lta/` (the Laboratório de Linguagens e Técnicas Adaptivas at the Universidade de São Paulo).

can't work properly in the presence of information-hiding, so language-construction by traditional macros wasn't viewed as abstraction. For roughly the next two decades, macros had no part in the abstraction movement. Macros did continue in two mainstream non-assembly-language settings: the C language (where macros provide basic support for several key features[7]); and the Lisp language family (where the base-language syntax is extremely sparse, and there is a proportionately vigorous tradition of language extension).

Around 1990, there was a climate shift in the programming-language design community. Existing abstraction techniques were showing signs of reaching their limits (see for example [GaSt90]); also, efforts by the Lisp community during the 1980s had produced hygienic variants of macros that were less aggressively anti-encapsulatory than the traditional form. In this changed atmosphere, macros were tentatively admitted under the abstraction umbrella, by granting them the pseudonym *syntactic abstractions* ([Ga89]).

The term *syntactic abstraction* ties in neatly with the two-phase model of language extension. Most modern forms of abstraction (notably, OOP class hierarchies) are integrated into the semantics of the base language, so that they appear to the programmer as run-time phenomena, i.e., as part of computation by the target abstract machine; while traditional macros, being by definition source-to-source translations, are eliminated before the target program is generated. Following this distinction, we will call abstractions *semantic* when they are integrated into the programmer's abstract model of run-time behavior, *syntactic* when they are, observably to the programmer, eliminated before run-time.

Syntactic abstractions do not have to be macros. A class of non-macro syntactic abstractions is proposed, under the name *micros*, in [Kr01]; a micro specifies a transformation directly source-to-target (rather than source-to-source, as macros), which is clearly syntactic since its processing must be completed before the target abstract machine begins computation.[8] Micros are a universal syntactic abstraction, in that they can be used to assign any possible (computable) semantics to source programs. (However, they won't play a significant role in this dissertation, because they aren't a traditional abstraction, so aren't within the scope of the thesis.[9])

[7] In traditional C ([KeRi78]), the macro preprocessor supports symbolic constants, `for` loops, and simple inlining (for efficiency by avoiding function calls), simplifying the implementation of a minimal C compiler. On the wider significance of simple implementation, see [Ga91].

[8] [Kr01] reserves the term *syntactic abstraction* for source-to-source transformations, i.e., macros, and uses *linguistic abstraction* for the class including both macros and micros.

[9] One might ask whether the claims of the thesis could be extended to include micros — that is, whether the fexpr-based approach described in the dissertation can subsume abstractions achieved via micros. The present work provides no basis to conjecture either yes or no on this question. Micro-based abstraction connects closely with issues of separately compiled modules; and since fexprs intrinsically defy the compilation/execution phase distinction, how to handle separately compiled modules in the presence of fexprs is itself a major question beyond the scope of the dissertation.

1.1.2 Abstractive power

> Each extensible language is surrounded by an envelope of possible extensions reachable by modest amounts of labor by unsophisticated users.
>
> — [Sta75, p. 20].

We now turn to the question of how to design a programming language so as to maximize its abstractive power.

First of all, we need to be clear on what we mean by *abstractive power*. The extensible-languages movement had always viewed the process of language extension (mostly, syntactic abstraction[10]) as literally specifying a programming language; and the abstraction movement has traditionally viewed the process of abstraction the same way, though usually not in quite such a technically literal sense.[11] We therefore take the view here (similarly to the above epigraph) that the measure of the abstractive power of a programming language is the class of programming languages that it can readily specify — or, more to the point (but also more subjectively), the class of *problem domains* that the language, including its readily achieved abstractive extensions, can readily address.

Our goal, then, is to construct arbitrarily long sequences of languages, hopscotching by abstraction from a starting language to ever more conceptually distant languages. As a convenient metaphor, we refer to the maximum feasible distance thus reachable from a given starting language as its *radius of abstraction*.

So, what would a language with a very high radius of abstraction look like?

We immediately dismiss from consideration the issue of call syntax, i.e., the mechanics of how the parts of an expression (generically, operator and operands) are put together — as writing `(a+b)` versus `(+ a b)`, `add(a,b)`, etc. While call syntax is a highly *visible* feature of programming languages (and, accordingly, received a good deal of attention from the extensible-languages movement; see [Wegb80]), it is also largely superficial, in that it should affect convenience of expression more-or-less uniformly across the board, with no impact on what can be expressed — especially, no

[10] Despite the dominance of macros in the movement, there were always some extensible languages that admitted semantic abstraction as extension (e.g., Proteus, [Be69]). However, by 1975 the extensible-languages and abstraction movements were carefully segregating themselves from each other; so that [Sta75], despite a very broad-minded definition of extension that *technically* seems to admit semantic abstraction, was in practice a critique only of languages that still self-identified as extensible. The schism was recent. Only a few years earlier, Simula 67 had presented itself as an extensible language ([Ic71]); and even its defining document was titled *Common Base Language* ([DaMyNy70]). A few years later, [Gua78] would recount the history of programming abstraction from the invention of assembly language up to 1978 without any hint that the extensibility tradition had ever existed.

[11] Usually, abstraction research that takes the language-specification concept literally has direct technological ties back to the extensible-languages movement — e.g., [Chr88] building on adaptive grammars, or [Kr01] building on macros.

impact on the kinds of abstraction possible (beyond its uniform effect on convenience of expression in the abstracting and abstracted languages). Intuitively, call syntax should account for a constant factor rather than an asymptotic class distinction in abstractive radius. (However, a constant factor can have great practical importance within an asymptotic class, so it should be well worthwhile —though outside the scope of the current work— to revisit the issue of call syntax once the underlying semantics of abstraction have been settled.)

A language property of great interest here is that the abstractive facilities apply to the base language in a free and uniform way. This is roughly what M.D. McIlroy, in the context of syntactic abstraction, called *smoothness* ([McI69]); and we will adopt his term. In semantics —where there are no crisp bounds to what could be considered abstraction, since semantic abstractions are, by definition, integrated into the semantics— similar properties have long been advocated under the names *orthogonality* (due to Adriann van Wijngaarden, [vW65]), and *first-class objects* (due to Christopher Strachey, [Stra67]).

Our interest in smoothness is based on the conjecture that

(Smoothness Conjecture) Every roughness (violation of smoothness) in a language design ultimately bounds its radius of abstraction.

The intuition here is that a butterfly effect occurs, in which the consequences of roughness are compounded by successive abstractions until, sooner or later depending on the degree of roughness, cumulative resistance drags down further abstraction through the fuzzy threshold from feasible to infeasible.

The intuited butterfly effect —thus, chaos— underlying the conjecture makes it an unlikely subject for formal defense: we actually *expect* to be unable to anticipate, at design time, specific advantages of a smooth facility that will later emerge from its actual use. (See, for example, [Li93, §3.4.4].) However, the conjecture brings us a step closer to a thesis —i.e., to what we do propose to formally defend— by allowing us to select a thesis based on immediate, though still subjective, properties (smoothness) rather than long-term effects (abstractive power).[12]

Syntactic abstractions are, by definition, restricted in their interaction with semantic abstractions (since syntactic abstractions are observably processed before runtime). Therefore, by reasoning from the conjecture, syntactic abstractions should bound abstractive power. This bound is inherent in the syntactic approach to abstraction, so the only way to eliminate the bound would be to eliminate support for syntactic abstraction; but if the semantic facilities of the language can't achieve

[12]As an alternative to steering clear of the problem of abstractive power, one might choose to confront it directly, by developing a rigorous mathematical definition of abstractive power, and subsequent theory. The definition would seem to be justifiable only by accumulated weight of evidence, in the subsequent theory, that the mathematical phenomena of the theory really correspond to the subjective phenomena of abstractive power. A mathematical definition of abstractive power is proposed in [Shu08].

all the abstractions of the eliminated syntactic facilities, then the elimination, while removing the fuzzy bound caused by roughness, would also introduce a hard bound caused by loss of capabilities. So we want to eliminate syntactic abstraction, but in doing so we need a semantic facility that can duplicate the capabilities of syntactic abstraction.

A semantic facility of just this kind, traditionally called *fexprs*, was used by Lisp languages of the 1960s and 70s. (The term *fexpr* identifies the strategy, but in a modern context is purely ad hoc; so when we don't need to identify the strategy and aren't discussing history, we prefer the more systematic terminology developed hereafter in §2.2.2.) Each fexpr specifies a computation directly from the (target-language) operands of the fexpr call to a final result, bypassing automatic operand evaluation and thus giving the programmer complete semantic control over computation from the target language.[13] Unfortunately, as traditionally realized, fexprs are even more badly behaved than traditional macros: by making it impossible to determine the meanings of subexpressions at translation time, they destroy locality of information in the source-code — thus undermining not only encapsulation (as do traditional macros), but most everything else in the language semantics as well. So around 1980, the Lisp community abandoned fexprs, turning its collective energies instead to mitigating the problems of macros.

Hygienic macros, which partially solve the anti-encapsulation problem of traditional macros, were developed by around 1990. To address the problem, some assumptions had to be made about the nature of encapsulation in the base-language (otherwise one wouldn't know what kind of encapsulation to make one's facility compatible with), and therefore the solution is only valid for the class of languages on which the assumptions hold. However, hygienic macros are still syntactic abstractions — which is to say that, despite the interaction with encapsulation required for hygiene, they don't interact at all with most of the language semantics, nor interact closely even with encapsulation. This limited interaction is both an advantage and a disadvantage: on one hand, the solution assumes only one facet of the language semantics, and should therefore be valid on a broad class of languages satisfying this weak assumption (roughly, the class of all languages with static scope); while on the other hand, the solution can't exploit the language semantics to do any of its work

[13] A remarkably parallel statement can be made about micros ([Kr01]): each micro specifies a translation directly from the (source-language) operands of the micro call to a target expression, bypassing automatic operand translation and thus giving the programmer complete syntactic control over translation from the source language. Thus, micros are to translation what fexprs are to computation. The parameters of the analogy are that micros bypass processing that would happen after a macro call, and are inherently syntactic; while fexprs bypass processing that would happen before a procedure call, and are inherently semantic.

The analogy also extends to internal mechanics of the devices: micros, as treated in [Kr01], rely heavily on a function `dispatch` that explicitly performs source translations, compensating for the loss of automatic operand translations — just as fexprs (treated here) rely heavily on a function `eval` that explicitly performs evaluations, compensating for the loss of automatic operand evaluations.

for it, and so is technically complicated to implement.

Because fexprs are semantic abstractions, we expect any well-behaved solution for fexprs —if such exists— to contrast with hygienic macros on all the above points. Fexprs must interact closely with the entire language semantics, so well-behavedness should require much stronger assumptions (though smoothness, which is the point of the exercise, should alleviate this as it entails *simple* interactions), and any solution should be valid only on a proportionately narrower class of languages; while, with the entire language semantics available to lean on, it should be possible to achieve a technically simple solution (which is also an aspect, or at least a symptom, of smoothness).

The essence of our thesis —to be defended by demonstration— is that such a solution does in fact exist: a combination of semantic assumptions that supports fexprs in a well-behaved and simple way.

1.1.3 Scheme

The smoothness criterion is suited to improving an existing language design, but doesn't provide enough detail to design a language from scratch; so in staging a demonstration of the thesis, we need to choose an existing design from which to start. The language we've chosen is Scheme (primarily *R5R* Scheme, [KeClRe98]).

The choice of a Lisp language has two main technical advantages:

- Lisp uses the same trivial syntax for all compound expressions, thus dismissing concrete syntax from consideration, as recommended above (early in §1.1.2).[14]
- Lisp treats programs as data. This is a signature feature of Lisp languages, and fexprs can't be supported smoothly without it: the technical utility of fexprs is in their ability to explicitly manipulate their target-language operands, and the only way to achieve that without introducing a phase distinction is to treat target-language expressions as data.[15]

Among Lisps, Scheme is particularly suited to the work, in several ways:

- Scheme is a particularly smooth Lisp, with stated design goals to that effect and various pioneering smooth features (though not under the name *smoothness*; see

[14]Following the recommendation, we disregard various attempts to add more features to Lisp syntax, such as keyword parameters in Common Lisp ([Ste90, §5.2.2]).

[15]In contemplating the impact of the programs-as-data feature on abstractive power, note the following remark of M.C. Harrison at the 1969 extensible languages symposium ([Wegn69, p. 53]):

> Any programming language in which programs and data are essentially interchangeable can be regarded as an extendible language. I think this can be seen very easily from the fact that Lisp has been used as an extendible language for years.

(Harrison preferred *extendible* rather than *extensible* because, he said, *extensible* sounds too much like *extensive*.)

[KeClRe98, p. 2]). So, starting from Scheme gives us a head start in achieving a smooth demonstration language. Smoothness of the demonstration language is not explicitly claimed by the thesis, but should facilitate well-behavedness and simplicity, which *are* claimed. (We also wouldn't be upset if our demonstration language had an unprecedentedly high radius of abstraction, both for the value of the achievement itself, and because it would lend additional post-facto justification to the pursuit of the thesis.)

- Scheme is a particularly simple language. Besides being specifically claimed in the thesis, simplicity is at least a strong correlate with smoothness; and, plainly, a simple system is easier to work with.

- Scheme is representative of the class of languages on which the classical hygienic-macro solution is valid (which is to say, mostly, that it has static scope). So, placing well-behaved fexprs in a similar setting should facilitate comparison and contrast between the two technologies.

1.2 Semantics

This section considers a second thread in the history of programming languages: the use of mathematical calculi to describe programming-language semantics. Whereas the previous section explained *why* well-behaved fexprs are of interest, this section explains broadly *how* well-behavedness of fexprs can be addressed by calculi. The principal programming activity of interest here will be meta-programming. There will be no role in the discussion for macros, which address abstraction but not meta-programming; and fexprs, which address both, will be contrasted with quotation (which addresses meta-programming but not abstraction).

Of interest here are, on the programming side, Lisp, and on the mathematics side, λ-calculus.

Alonzo Church created the λ-calculus in the early 1930s ([Bare84, §1.1]; see also §8.1). The original theory treated logic as well as functions; but inconsistencies (antinomies) were quickly discovered in the logic, and in [Chu41] he recommended a far less ambitious subset that omitted the logical connectives, leaving only an intensional theory of functions.[16] Pure λ-calculus as usually formulated today is a slight generalization of Church's 1941 calculus.[17]

[16] That's *intensional*, with an *s*, an adjective contrasted in logical discourse to *extensional*. Roughly, a function definition by extension is a graph, i.e., a set of input-output pairs, while a function definition by intension is a rule for deriving output from input; see [Chu41, §2]. The idea of viewing functions extensionally is commonly attributed to Dirichlet, who more-or-less described it in a paper in 1837 (a huge step in a progressive generalization of the notion of function that had been going on since the later 1700s; see [Kli72, §40.2]).

[17] The usual modern λ-calculus is sometimes called the λK-calculus to distinguish it from Church's

In the late 1950s, John McCarthy set out to design an algebraic list processing language for artificial intelligence work ([McC78]; see also §3.3.2). Seeking a suitably 'algebraic' notation for specifying functions, and being aware of Church's intensional theory of functions, he adapted λ-notation for his language. He named the language Lisp (acronym for *LISt Processing*).

1.2.1 Well-behavedness

Programmers spend much of their time reasoning informally about programs. Formal reasoning about programs may be used, on occasion, to prove that particular programs are correct; but more powerfully, to prove that general classes of program transformations preserve the meanings of programs. The general transformations are then safe for use in meta-programs —such as optimizing compilers— and also for use in subsequent program correctness proofs. The general classes of transformations manifest as equation schemata in an *equational theory*, i.e., a formal theory whose provable formulae equate expressions, $M = N$. (When M and N must be closed terms, $M = N$ is an equation; when they may contain free variables, it's a schema.[18] Re formal theories, see §8.2, §8.3.1.)

The stronger the equational theory —i.e., the more expressions it equates— the more opportunities it will afford for program optimization. The more general the schemata —i.e., the more equations are expressed by each schema, hence the more strength the theory derives per schema— and the fewer separate schemata must be considered, the more feasible it will be for a compiler to automatically discover suitable optimizations. Ideal for optimization would be a strong theory based on a few schemata. On the other hand, from the programmer's perspective, informal reasoning will be facilitated by language simplicity; and by separability of interpretation concerns, a.k.a. *good hygiene* (Chapter 5). But language simplicity manifests formally as a small number of schemata; and, likewise, separability of interpretation concerns manifests formally as the existence of very general schemata (whereas interference between concerns would tend to fragment the theory into many special cases). So a simple, clear language fosters a simple, strong theory, and vice versa.

Lisp began as a simple, clear (if exotic) programming language; and these properties have formed a definite theme in the subsequent evolution of the Lisp family

1941 λI-calculus, which differs by the constraint that a term $\lambda x.M$ is syntactically permissible only if x occurs free in M. [Chu41] does mention λK-calculus, but strongly recommends λI-calculus (which he calls simply λ-calculus) for the theorem —true for λI-calculus but not for λK-calculus— that, for all terms M, if M has a normal form then every subterm of M has a normal form. (We will have more to say on this point in Footnote 25 of §8.3; for the undiluted technical arguments, see [Chu41, §17], [Bare84, §2.2].)

[18] Here we mean *semantically* closed terms, versus free *semantic* variables, *semantic* meaning interpreted by the human audience, rather than interpreted by the formal system itself. Semantic versus syntactic variables will be discussed in §2.1; by the notation introduced there, "x = y" is an equation (likely unprovable), "$x = y$" a schema (enumerating an equation for each choice of x, y).

— and, in particular, of the Scheme branch of the family. Therefore, Lisp/Scheme ought to support a simple, strong formal theory. On the other hand, λ-calculus *is* a simple, strong theory (or to be technically precise, its theory, called $\boldsymbol{\lambda}$, is a simple, strong theory); and with the `lambda` constructor playing such a prominent role in Lisp, naturally there has been much interest over the decades in using λ-calculus as a semantic basis for Lisp/Scheme.

Unfortunately, despite their superficial similarities, at the outset Lisp and λ-calculus were profoundly mismatched. λ-calculus had been designed to describe functions as perceived by traditional (actually, eighteenth-century) mathematics; while Lisp was first and foremost a programming language, borrowing from λ-calculus only its notation for function definitions.

1.2.2 Order of operations

In traditional mathematics, the calculation of a function is specified by a single expression with *applicative* structure. That is, each compound expression is an operator applied to subexpressions; and, in computing the value of the expression, the only constraint on the order of operations is that each subexpression must be evaluated by the time its resultant value must be used by a primitive function (as opposed to a *compound* function constructed from primitives).[19] λ-calculus, in particular, is strictly applicative; and the theory $\boldsymbol{\lambda}$ derives its considerable strength from this thoroughgoing disregard for detailed order of operations (see §2.3.3, §8.3).

In the earliest programming languages —assembly languages— the order of operations was completely explicit. Later languages tended toward specifying functions algebraically; which, besides aiding program clarity, also eliminated accidental syntactic constraints on the order of operations; and Lisp, with its thoroughly applicative syntax and its `lambda` constructor, represented a major advance in this trend. As more abstract programming language syntax has provided opportunities for flexibility in actual order of operations, (some) programming language designers have used that flexibility to eliminate more and more accidental semantic constraints on order of operations — and, in doing so, have gradually revealed how the essential order-constraints of programming deviate from the applicative model.

Scheme takes the moderately conservative step of allowing the arguments to a function to be evaluated in any order.[20] This could be viewed as a separation of

[19]This is the general sense of *applicative* throughout the dissertation. It is borrowed directly from [Bac78, §2.2.2], which refined it from the sense disseminated by Peter Landin (as a result of his theoretical study of programming languages encouraged by his mentor, Christopher Strachey; [La64], [Cam85, §4]). There is an unrelated and incompatible term *applicative-order* sometimes used to describe eager argument evaluation, and opposed to *normal-order* for lazy argument evaluation ([AbSu96, §1.1.5]); for this distinction, we will prefer the more direct adjectives *lazy* and *eager*.

[20]Demonstrating the lack of consensus on eliminating order constraints, PLT Scheme restricts standard Scheme by specifying the exact order of argument evaluation ([Fl10, §2.7]).

interpretation concerns (good hygiene), since it means that the syntactic device for function application is separate from the syntactic device for explicit command sequencing. Scheme does require that all arguments must be evaluated before the function is applied — so-called *eager argument evaluation*, which isn't purely applicative because the applicative model only requires arguments to be evaluated when the function to be applied is primitive; but, against this possible drawback, eager argument evaluation separates the concern of forced argument evaluation from that of compound-versus-primitive functions (good hygiene again), thereby enhancing the treatment of compound functions as first-class objects by rendering them indistinguishable from primitives.

Some languages, such as Haskell, allow argument evaluation to be postponed past the application of a compound function — lazy argument evaluation; but in general this takes out too much of the order of operations, so that the language interpreter must either do additional work to put back in some of the lost order, or else lose efficiency.[21] A few languages, such as Haskell, have taken the applicative trend to its logical extreme, by eliminating non-applicative semantics from the language entirely — rendering these languages inapplicable to general-purpose programming, for which time-dependent interaction with the external world is often part of the purpose of the program. Haskell has addressed this shortcoming, in turn, by allowing functions to be parameterized by a mathematical model of the time-dependent external world (originally, a monad) — thus allowing purely applicative programs to describe non-applicative computations, but dumping the original problem of the programming/mathematics mismatch back onto the mathematical model. It is not yet clear whether the mathematical models used in 'monadic' programming will eventually find a useful balance between expressiveness and simplicity; see [Shu03a], [Hu00].

As an alternative to radically altering the structure of programming languages, one could approach the problem entirely from the mathematical side by *modifying the λ-calculus* to incorporate various non-applicative features. A watershed effort in this direction was Plotkin's λ_v-calculus, introduced in [Plo75], which uses eager argument evaluation. Plotkin recommended a paradigmatic set of theorems to relate language to calculus and establish the well-behavedness of both. In the late 1980s, Felleisen ([Fe87]) refined the paradigm and developed variant λ-calculi incorporating imperative control structures (first-class continuations) and mutable data structures (variables).

[21] The problem is with the use of tail recursion for iteration. A properly tail recursive implementation prevents tail calls from needlessly filling up memory with trivial stack-frames; but if lazy argument evaluation is practiced naively, then memory may fill up instead with lazily unevaluated expressions. This was the original reason why Scheme opted for eager argument evaluation; [SuSt75, p. 40] notes, "we ... experimentally discovered how call-by-name screws iteration, and rewrote it to use call-by-value." That paper discusses the issue in more detail on [SuSt75, pp. 24–26].

1.2.3 Meta-programming

In meta-programming, a representation of an *object-program* is manipulated, and in particular may be executed, by a *meta-program*. How this impacts the applicativity of the meta-language —and the simplicity and clarity of the meta-language—and the simplicity and strength of its theory— depends on a somewhat muddled convergence of several factors. (If it were straightforward, the outstanding issues would have been resolved years ago.)

1.2.3.1 Trivialization of theory

First of all, we may divide the meta-language capabilities that support meta-programming into two levels: construction and evaluation of object-programs (nominal support); and examination and manipulation of object-programs (full support). For short, we'll call these respectively *object-evaluation* and *object-examination*.

Object-evaluation permits a meta-program p to evaluate an object-expression e and use the result; since e is evaluated before use, applicativity may be preserved. Object-examination, however, is inherently non-applicative: in order for p to examine the internal structure of e, p must use e without evaluating it first — directly violating the operation-order constraint imposed by the applicative model. Object-examination thus differs qualitatively from the non-applicative features addressed by the aforementioned variant λ-calculi (of Plotkin and Felleisen), all of which, broadly speaking, added further order-constraints to the model without disturbing the one native order-constraint central to λ-calculus.

More technically, expressions in ordinary λ-calculus are considered equal —thus, interchangeable in any context— if they behave the same way under evaluation. Using object-examination, however, a meta-program might distinguish between object-expressions e_1, e_2 even if they have identical behavior, so that e_1, e_2 are no longer interchangeable in meta-programs, and cannot be equated in the theory of the meta-language. Thus the meta-language theory will have little or nothing useful to say about object-expressions. Moreover, it is a common practice among Lisps to use the meta-language as the object-language[22] — so that a meta-language theory with nothing useful to say about object-expressions has nothing useful to say at all. (See §8.4.)

The trivialization of equational theory is actually not inherent in meta-programming. It emerges from an interaction between the native structure of λ-calculus,

[22]McCarthy wanted to show the existence of a universal Lisp function, which by definition requires that the object-language be isomorphic to all of Lisp. McCarthy's early papers used an explicit isomorphism between partially distinct sublanguages; but most implemented Lisps use a single syntax for object- and meta-languages. We assume the single-syntax approach in our discussion here, as the additional complexity of the dual-syntax approach would tend to obscure the issues we are interested in. The dual-syntax approach is useful in addressing certain other issues, relating to object-languages whose meta-programming capability may be strictly less than that of the meta-language; see [Mu92].

and the means used to specify which subexpressions are and are not to be evaluated before use. We consider here two common devices for this specification: *quotation*, and *fexprs*.

1.2.3.2 Quotation

The quotation device, in its simplest form, is an explicit operator designating that its operand is to be used without evaluation. An explicit operator usually *does* something (i.e., designates an action), and so one may naturally suppose that the quotation operator *suppresses* evaluation of its operand. One is thus guided subtly into a conceptual framework in which all subexpressions are evaluated before use unless explicitly otherwise designated by context. We will refer to this conceptual framework as the *implicit-evaluation paradigm*. Just as quotation suggests implicit evaluation, implicit evaluation suggests quotation (the most straightforward way for a context to designate non-evaluation), and so the device and paradigm usually occur together in a language — even though they are not inseparable, as will emerge below.

Quotation and implicit evaluation are the norm in natural-language discussions, where most text is discussion but occasionally some text is the subject of discussion; and, given this familiar precedent, it's unsurprising that quotation was used to support meta-programming in the original description of Lisp.

In principle, simple quotation is sufficient to support the paradigm; but in practice, the demands of the paradigm naturally favor the development of increasingly sophisticated forms of *quasiquotation*, a family of devices in which object-expressions are constructed by admixtures of quotation with subexpression evaluation.[23] Support for quasiquotation generally entails a distinct specialized quasiquotation sublanguage of the meta-language; and, accordingly, *use* of quasiquotation generally entails specialized programmer expertise orthogonal to programmer expertise in the rest (i.e., the applicative portion) of the meta-language. On the evolution of quasiquotation in Lisp, see [Baw99]; for an example of full quasiquotation support outside the Lisp family of languages, see [Ta99].

Mathematically, the addition of quotation to λ-calculus is the immediate technical cause of the trivialization of equational theory mentioned earlier. Any potentially evaluable expression becomes unevaluable in context when it occurs as an operand to quotation; thus, the equational theory can never distinguish the evaluation behavior of any expression from its syntax, and only syntactically identical expressions can be equated. The trivialization could be prevented by introducing a notation into the calculus that *guarantees* certain designated subexpressions will be evaluated regardless of context. An equation relating the evaluation behaviors of two expressions would then be distinct from an equation relating the expressions themselves. This solution,

[23] This pattern of development occurred not only in the computational setting of Lisp, but also (to a lesser degree) in the natural-language context of the mathematical logic of W.V. Quine, who coined the term *quasi-quotation* circa 1940 ([Baw99, §4]).

however, requires the researcher to step outside the implicit-evaluation paradigm — which is not an obvious step, because the paradigm commands considerable credibility by providing, through quotation and quasiquotation, both full support for the functionality of meta-programming (though the support may not always be entirely convenient), and a conventionally well-behaved reduction system (though without an associated useful equational theory).

The alternative paradigm implicit in this technical fix, in which subexpressions are *not* evaluated unless their evaluation is explicitly designated by context, we will refer to as *explicit evaluation*. Practical use of explicit-evaluation techniques as an alternative to quotation will be explored in §7.3.

1.2.3.3 Fexprs

The fexpr device, in its simplest form, allows arbitrary, otherwise ordinary functions to be tagged as *fexprs*, and provides that whenever a fexpr is called, its operands are passed to it unevaluated. Which particular subexpressions are and are not to be evaluated before use must then be determined, in general, at run-time, when each operator is evaluated and determined to be or not to be a fexpr. Thus, static program analysis can't always determine which subexpressions will and will not be evaluated before use, retarding optimization. Worse, in a statically scoped language (where the body of a compound function is evaluated in the static environment where the function was constructed), it is necessary to give the fexpr access to its dynamic environment (the environment where the function is called), compromising language hygiene and causing difficulties for the programmer, the optimizer, *and* the equational theory. Mitigation of these difficulties is discussed in Chapter 5.

The fexpr device is crudely compatible with both the implicit- and explicit-evaluation paradigms — the distinction being simply whether fexprs suppress evaluation of their operands (implicit evaluation), or non-fexpr functions *induce* evaluation of their operands (explicit evaluation).[24] From the latter view, it is a small step to view each non-fexpr function as the literal composition of a fexpr with an operand-evaluation wrapper. This approach is the central subject of the current dissertation.

Traditionally, however, fexprs have been treated as a separate type of function, coexisting independently with ordinary functions — and coexisting also, in Lisp, with a quasiquotation sublanguage. When fexprs are juxtaposed with quasiquotation, the quasiquotation operators themselves appear as fexprs, promoting association of fexprs with the implicit-evaluation paradigm.

Around 1980, the Lisp community responded to the various difficulties of fexpr technology by abandoning it in favor of macros (§1.1, §3.2). Even as fexprs disappeared from mainstream Lisps, though, they were taken up by a new, largely experimental family of *reflective* Lisps.

[24] Fexprs do have a bias toward explicit evaluation, which will be brought out in §3.1.3.

1.2.4 Reflection

Reflection, in programming, means giving a program the capacity to view and modify the state of its own computation. Reflection as a programming activity does not bear directly on the current work. However, owing to the fact that reflective Lisps have been the primary consumers of fexpr technology in the past two decades,[25] the common perception of fexprs has been colored by their use in reflection. We therefore overview the interplay of fexprs with reflection, and ultimately consider why it has tended to reinforce the association of fexprs with the implicit-evaluation paradigm.

In order to achieve reflection, reflective Lisps allow a running program to capture concrete representations of various aspects of computational state that, in a traditional non-reflective Lisp, would not be made available to the program. The three aspects usually captured are operands, environments, and continuations. Unhygienic properties of fexprs that would otherwise be frowned upon —that they capture operands and environments— thus become virtues in a reflective setting; and some reflective Lisps use a variant form of fexprs that capture continuations as well (subsuming the functionality of Scheme's `call-with-current-continuation`). Fexprs used in a reflective setting are generally called *reifying procedures*, and the act of capturing these various partial representations of state is called *reification*.

The verb *reify* is borrowed from philosophy. Derived from the Latin *res*, meaning thing or object, it means to "thingify": to treat a noun as if it refers to an actual thing when it does not. Philosophers *accuse* one another of reification. As a few examples among many, abstractions, concepts, or sensations might be reified. To understand each such accusation, one must fine-tune one's understanding of *thing* accordingly (e.g., one might be accused of reification for treating an infinity as a number, or for treating a number as a material object); so that, rather than belabor the definitions of words, one might more usefully understand the mistake of reasoning as a *type error*, in which a thing of one type is treated as if it belonged to a different type [Bl94, "reification"].

Because Lisp has a fairly well-defined notion of *object* as —mostly— something that can be the value of a variable,[26] it seems that reification in reflective programming would be the act of giving object status to something that otherwise wouldn't have it. Environments are the most clearcut example: they are not objects in (modern)

[25] Reflective Lisp research began with Brian Smith's 3-LISP, on which see [Sm84]. Subsequent work used additional reflective Lisps to explore the roots of the reflective power of 3-LISP; see [FrWa84, WaFr86, Baw88].

[26] We're really referring to the notion of *first-class* object, which was discussed in the Preface, and for which nameability by variables is just one particular requirement; but in Lisps, most objects nameable by variables have all the rights and privileges of first-class-ness.

Christopher Strachey, by the way, originally evolved the notion of first-class value from W.V. Quine's principle *To be is to be the value of a variable* ([La00]); and Quine, after deriving the principle in his essay "On What There Is" [Qu61, pp. 1–19], had applied it to reification in another essay, "Logic and the Reification of Universals" [Qu61, pp. 102–129].

non-reflective Lisps, but they become objects when captured —reified— by fexprs. This definition of Lisp reification is not without difficulty, as we shall see; and, as in philosophy, the choice of terminology will be superficial to the underlying issue. But for the choice of terminology to be credible, any definition of *reification* ("making into an object") in programming must include the granting of object status — that is, there is no reification if the thing already was an object, or if it doesn't become one.

Based on this criterion alone, we observe that whether or not operand capturing, the hallmark behavior of fexprs, can possibly qualify as reification depends on whether one considers it in terms of the implicit-evaluation paradigm, or the explicit-evaluation paradigm. Under the implicit-evaluation paradigm, the operands in a combination are initially expected to be evaluated, thus expected to be unavailable to the program, and so, arguably, they do not qualify as objects. Hence, their subsequent capture assigns (or perhaps restores) object status to things that had been without it, and the capture meets our necessary criterion for reification. Under the explicit-evaluation paradigm, however, since there is no prior expectation that the operands would be evaluated, the operands are objects to begin with; so their capture isn't reification.

Our suspicion that this criterion is not *sufficient* stems from the observation that merely providing an object to the program is an empty gesture if one does not additionally provide suitable operations on the object (*methods*, in OOP parlance). Recalling the case of environment capturing, it is of no use for a fexpr to capture its dynamic environment unless it also has, at least, the ability to evaluate expressions in a captured environment;[27] without that ability, the captured environment is simply an inert value.

Moreover, for the purposes of reflective Lisps, it is not always sufficient to supply just those operations on the captured object that would otherwise have been possible on uncaptured objects of its type. To endow the language with reflective capabilities, environments are given some particular representation, whose concrete operations can then be exploited in the captured object to achieve various reflective effects. The degree of reflection depends, in part, on the choice of representation. (On this choice of representation, see [Baw88].) Thus, to achieve reflection we have really made *two* changes to our treatment of environments: we have given them the status of objects; and we have also *broken their abstraction barrier*, replacing the abstract *environment* type of non-reflective Lisp with a more concrete type that supports additional reflective operations. This abstraction violation is analogous to the kind of type error that the philosophers were objecting to under the name *reification*.

With all the various background elements in place, we can now properly explain the chain of associations from fexprs, through reflection, to implicit evaluation.

[27] We could have said, the ability to look up symbols in a captured environment. Given the usual Lisp primitives `apply` etc., the expression-evaluation and symbol-lookup operations are equal in power.

1. In a climate where fexprs are supported only by reflective Lisps, fexprs are perceived as a reflective feature.

2. Standard usage in the reflective Lisp community refers to capturing, including operand capturing, as reification.

3. In viewing operand capturing as reification, one embraces the implicit-evaluation paradigm — because under explicit evaluation, it wouldn't be reification.

As we have seen, a careful analysis of the issues suggests that none of these three associations is necessary; but, absent a particular interest (such as ours) in the explicit-evaluation paradigm, there has been little motivation for such an analysis. Thus, the prevailing terminology for fexprs in reflective Lisp has promoted a paradigm that is orthogonal to both fexprs and reflection.

1.3 Thesis

This dissertation contends that the severe difficulties attendant on fexprs in the past are not essential, and can be overcome by judicious coordination with other elements of language design. In particular, fexprs can form the basis for a simple, well-behaved Scheme-like language, subsuming traditional abstractions without a multi-phase model of evaluation.

The thesis is supported by a new Scheme-like language called Kernel, created for this work, in which each Scheme-style procedure consists of a wrapper that induces evaluation of operands, around a fexpr that acts on the resulting arguments. This arrangement enables Kernel to use a simple direct style of selectively evaluating subexpressions (explicit evaluation), in place of most Lisps' indirect quasiquotation style of selectively suppressing subexpression evaluation (implicit evaluation). The semantics of Kernel are treated through a new family of formal calculi, introduced here, called *vau calculi*. Vau calculi use explicit-evaluation style to extend lambda calculus, eliminating a long-standing incompatibility between lambda calculus and fexprs that would otherwise trivialize their equational theories.

Chapter 2

Language preliminaries

2.1 About variables

In this dissertation, a *variable* is a mathematical symbol whose meaning is determined locally within a portion of the mathematical discussion, as opposed to a *reserved symbol*, whose meaning is fixed over the entire discussion. This is an inclusive sense of the term *variable*. Particular branches of mathematics may restrict their use of the term to symbols whose local interpretation is particularly of interest, as in high-school algebra, where variables denote numerical unknowns. Our inclusive sense of the term is used in programming and metamathematics, both of which are centrally concerned with how expressions are interpreted — so that *all* local interpretation of symbols is of particular interest.

Variables are notated differently here, depending on whether their meaning is to be interpreted by the human audience (*semantic* variables), or interpreted by the rules of a formal system (*syntactic* variables). In general mathematical discussion, a semantic variable is a single letter in italics, with or without subscripts or primes; thus, x, α, M, x_3, F', etc.; while a syntactic variable is a single unitalicized English letter; thus, x, M, etc.

Other variable notations will be introduced for more specialized discussions. For example, Lisp uses multi-character variable names; vau calculi will do likewise; and in discussion of those formal systems, semantic variable names may be multi-character as well. However, italics will never be used for syntactic variables.

When a variable occurs in an expression that locally defines its meaning, the variable-occurrence is said to be *bound* in the expression; otherwise, the occurrence is *free* in the expression. For example, in the mathematical statement

$$\forall x \in \mathbb{N}, \quad f(x) > 1\,, \tag{2.1}$$

the occurrence of x as an operand to f is bound by the universal quantifier, $\forall$, while the occurrence of f is free since the definition of f is not contained within the statement.

When a semantic variable occurs free in a mathematical expression surrounded by prose, the meaning of the variable must be determined from the prose context. When the prose context doesn't identify a specific value for the variable, the variable is by implication universally quantified over some domain of discourse. The domain of discourse for the variable must then be clear from the surrounding prose; sometimes there is a single domain for all variables, but often, variables are classified into domains of discourse by letter. Case in point: unless otherwise stated, throughout this dissertation semantic variables based on letters i through n are constrained to the nonnegative integers, $\mathbb{N}$.

In all of the formal systems considered here, variables may be bound by variants of λ-notation. The basic form of λ-notation is

$$\lambda x.M\,, \tag{2.2}$$

where x is some syntactic variable, and M is some mathematical expression of the formal system. Any free occurrences of x in M are bound by the λ. For example, given

$$\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y})\,, \tag{2.3}$$

the variables x and y are both free in the body, $(\mathrm{x}+\mathrm{y})$, while in the λ-expression as a whole, y is free and x is bound.

Semantic variables are bound here in mathematical expressions —as opposed to prose— only by quantifiers, such as $\forall$ in the earlier example, (2.1), or summation notation, such as $\sum_{k=1}^{m}$ (which binds the index of summation k). Thus, all λ-like expressions are formal; so that when a semantic variable occurs in such an expression, it stands for a portion of the expression to be substituted in before the expression is interpreted by the formal system. For example, in the above basic form for λ-notation, (2.2), x is understood by the human audience to stand for some syntactic variable, which must then be substituted in for x before the formal system can interpret the expression (as x is substituted for x in (2.3)).

When a semantic variable occurs in text discussion, usually it is a reference to the value of the semantic variable; but occasionally it is meant to refer to the semantic variable itself, rather than to its value (such as in the last sentence of the preceding paragraph). The distinction between the two kinds of reference is usually clear from context, so that no special notation is then needed to distinguish between them. The strongest contextual indication is an explicit statement of the domain of reference — hence, "syntactic variable x" (which can only refer to the value of the semantic variable, since the semantic variable itself is not a syntactic variable), "semantic variable x" (which can only refer to the semantic variable itself, since, at least in this case, the value of the semantic variable cannot be a semantic variable). When a passage contains a variety of references to semantic variables, contextual disambiguation may

be reinforced (and/or abbreviated) by placing explicit quotation marks ("") around the semantic variable when it is being named rather than invoked.[1]

A variable to be interpreted before the expression in which it occurs is called a *meta-variable*. Meta-variables constitute an alternative solution to the subexpression-evaluation problem of meta-programming that was discussed in §1.2.3; they are an implicit-evaluation device, less elaborate than quasiquotation because, in the terminology of §1.2.3.1, they address only object-evaluation, not object-examination. Our semantic/syntactic distinction between variables is often sufficient, without further complicating notation, to disambiguate the uses of meta-variables in the dissertation, because

- none of the formal systems of primary interest to us (Lisps, lambda calculi, and vau calculi) use meta-variables; if they address the subexpression-evaluation problem at all, they use some other strategy;

- semantic variables are used mostly to describe formal systems, only rarely to describe informal mathematics; and

- semantic variables aren't bound by any of the binding devices used in the formal systems.

Nevertheless, Part II of the dissertation, in its detailed exploration of formal calculi and their mathematical properties, will make use of rudimentary syntactic and semantic meta-variables. Syntactic meta-variables, with names based on "□", will occur in contexts, which are central to the modern study of calculi. Semantic meta-variables, with names based on "π", will be used to characterize classes of schemata in $\mathfrak{f}$-calculi. In both cases, the meta-variables are rudimentary in the sense that they cannot be targeted to different domains: there is only one universal quantification for syntactic meta-variables (namely, all syntactic expressions, including those involving meta-variables), and only one universal quantification for semantic meta-variables (namely, all semantic expressions, including those involving meta-variables).

[1] We resort to double-quotes in such stressful situations because they are a standard device in natural-language discussions and should therefore be readily understood by the human audience (notwithstanding the irony of using quotation to disambiguate discussion in a study of fexprs; use of fexprs in prose would belong to a different dissertation, probably in a different academic discipline). On the other hand, we more often elide these double-quotes, because most passages that could use them are just as practically unambiguous, and easier to read, without them. Technical simplicity would favor always using the double-quotes — but technical simplicity is a primary goal for expressions to be read by formal systems, not by the human audience. For the human audience our goal is accuracy of understanding, which is enhanced by ease of understanding but does not always correlate with technical simplicity, nor even with technical unambiguity. Distinguishing between semantic variables and syntactic variables helps to provide clean separation between situations that call for accuracy of understanding, and situations that call for technical simplicity.

2.2 Lisps

This section sketches basic concepts and terminology of Lisp (which are a cross-cutting concern, as they will apply to vau calculi nearly as much as to Kernel).

2.2.1 Values

Lisp values (also called *expressions*, eliding the distinction between data structures and source code) are partitioned into *atoms*, which are conceptually indivisible, and *pairs*, which are compound structures each made up of two values in specified order (more precisely, two references to values, in specified order).

In [McC60] there was only one kind of atom, the *symbol*, which was any sequence of letters, digits, and single embedded spaces, treated as an indivisible unit. The embedded-spaces idea was soon dropped, and other kinds of atomic values, such as numbers, were added; but Lisp began and remains more permissive than most programming languages in its symbols. In modern Lisps a symbol can be almost any sequence of letters, digits, and special characters, provided it doesn't contain certain punctuation characters, and doesn't start out looking like something else (as it would if, say, it started with a digit). The main types of atoms admitted in this dissertation are: symbols, integers (sequences of digits, not starting with zero, and optionally prefixed by a minus sign, `-`), booleans (which, following Scheme, we represent by `#t` for true, `#f` for false), strings (sequences of printable characters delimited by double-quotes, as `"hello, world"`[2]), nil (explained below), combiners (explained in the next subsection), and environments (explained in the next subsection).

The basic notation for representing a pair is `(`v_1 `.` v_2`)`, for expressions v_k; that is, a left-paren, first of two subexpressions, period, second of two subexpressions, and right-paren. This is sometimes called *dotted pair* notation. (Whitespace is a technicality we will usually assume without explanation; it only matters in that some may be needed to indicate where one lexeme ends and another begins.) The two values contained in a pair are called respectively its *car* and *cdr* (for historical reasons relating to the hardware architecture of the IBM 704 ([McC78])).

Pairs are meant to be particularly useful for building up lists, where the car of a pair is the first element of the list and its cdr is the rest of the list, i.e., the sublist of elements after the first. A special atom called *nil* stands for the empty list; nil is written as an empty set of parentheses, `()` (which some Lisps identify with the symbol `nil`, though we won't use that convention here). To facilitate the use of these list data structures, a shorthand notation is provided in which, when the cdr of a pair expression is itself a pair or nil expression, the dot of the outer pair and the parentheses of the cdr may be omitted together; thus,

$$\texttt{(foo bar quux)} \equiv \texttt{(foo . (bar . (quux . ())))}. \tag{2.4}$$

[2]We would need to introduce some special notation such as escape-sequences if we wanted to embed, say, newlines or double-quotes in our string literals; but we don't, so we don't.

(Don't mistake the prose period at the end of display (2.4), which terminates the sentence "To facilitate ...", for part of the displayed Lisp notation.)

2.2.2 Programs

Since all Lisp values were originally built up from symbols, they were called *S-expressions* (*S* for *Symbolic*); and despite the addition of non-symbol atoms, Lisp values are still sometimes called S-expressions. A strictly separate family of expressions, called *M-expressions*, was originally envisioned to denote programs that act on S-expressions (*M* for *Meta*); but then several man-years were unexpectedly shaved off the implementation of Lisp by coding an interpreter that processed S-expression representations of programs, and the precedent was established that Lisp programs are represented by the same kind of data structures as they manipulate (along with various other precedents that McCarthy had expected to have plenty of time to reconsider ([McC78])). Almost all Lisps represent programs by S-expressions.[3]

Throughout the dissertation, literal representations of S-expressions (and of fragments of S-expressions) are written in `monospace` lettering.

The act of interpreting an S-expression as a program is called *evaluation* — since Lisp is an expression-oriented language, in which every program produces a value as its result (if and when it terminates successfully). Lisp expression evaluation has three cases: non-symbol atoms, symbols, and pairs.

Non-symbol atoms self-evaluate; that is, they evaluate to themselves. The self-evaluating case will figure in §5.1 as a facilitator to good hygiene.

A symbol to be evaluated is called a *variable*. (In the terminology of §2.1, it is a *syntactic* variable, since it is interpreted by Lisp.) A meta-language structure specifying the value of a variable —i.e., the value that would result from evaluating the variable— is called a *binding*. We notate bindings by "$x \leftarrow v$", where x is the variable and v its value. We will also sometimes refer to the object bound to a Lisp variable (in whatever context we're considering, most often a Kernel standard environment) by the name of the variable written in ***italicized monospace*** lettering; thus, ***apply*** is the value named by symbol `apply`; by extension of the notation, ***cond*** is the value named (even if not formally bound) by symbol `cond`;[4] and so on.

Keeping track of variable bindings during Lisp computation is a critical problem that will recur, in various forms, throughout both parts of the dissertation; for an overview, see §§3.3.1–3.3.3. Often (but not always) binding maintenance is regulated using *environments*; an environment is a data structure representing a set of bindings,

[3] An exception is Robert Muller's M-LISP ([Mu92]).

[4] The symbol `cond` would not be formally bound if it is a reserved symbol rather than a variable; but in that case we may still wish to refer to what the symbol represents, even though what it represents is not a first-class value. The analogous value in Kernel *is* first-class, but is called `$cond` rather than `cond`; so the value bound to it would be ***$cond***.

usually the set of all bindings in effect at a particular point in a program. (Internals of the data structure won't matter till later, e.g. §3.3, Chapter 5.)

A pair to be evaluated is called a *combination*. This case is the most complex, having internal structure; and it is most central to the dissertation, since it is where fexprs come into play. So we provide detailed terminology for the various roles that Lisp values can play in relation to it. (About half of the following terminology is standard for Scheme, as set down in the Wizard Book, [AbSu96, §1.1]; the other half, for situations not covered by Scheme terminology, is remarked as it arises.)

The car of a combination is its *operator*; the cdr is an *operand tree*; and in the usual case that the operand tree is a list, any elements of that list are *operands*. (*Operand tree* is an extrapolation from the standard Scheme usage *operand list*. Scheme doesn't need the more general term because it doesn't permit evaluation of non-list pairs; but Kernel does permit evaluation of non-list pairs, because it treats evaluation as a right of all first-class objects.)

In the common case that the operands are evaluated, and all other actions use the results rather than the operands themselves, the results of evaluating the operands are *arguments*. This is exactly the ordering constraint of the applicative model that was discussed in §§1.2.2–1.2.3, so we call a combination of this kind an *applicative combination*. We call a non-applicative combination an *operative combination* (because its evaluation may, in general, depend directly on its operands).

The operator of a combination determines how its arguments (in the applicative case) or operands (in the operative case) will be used. In Kernel, this determination is made by evaluating the operator; the result of operator evaluation is, if type-correct, a *combiner*, which embodies an agenda for the rest of the combination evaluation. The combiner must specify first of all whether argument evaluation will take place, i.e., whether the combination is applicative or operative; and the adjectives *applicative* and *operative* are used so much more often for combiners than in any other capacity, that we usually just call the two kinds of combiners *applicatives* and *operatives*. (We avoid the ambiguous term *procedure*, which is used in Scheme for what we call an *applicative*, but often in the non-Scheme literature for what we call a *combiner*.)

A first-class combiner has the right to be invoked as the result of operator evaluation: if it can't be the result of operator evaluation, it's not first-class; if it doesn't determine combination evaluation when it's the result of operator evaluation, it's not a combiner. First-class applicatives (at least, first-class in this respect) are one of the defining characteristics of Lisp. First-class operatives, though, were not part of the original language design. They were introduced very early into the language implementation, in the form of *fexprs*, which specify computation from operands to combination-result in the same way that ordinary Lisp applicatives specify computation from arguments to combination-result; but first-class operatives (fexpr or otherwise[5]) present a practical difficulty. As long as it is possible for operator evaluation to produce either an operative or an applicative, there is no general way for a static pro-

[5]Macros are operatives, but specify computation differently from fexprs (§3.1.2ff).

gram analyzer —such as an optimizing compiler— to anticipate which operands are significant as data structures (as they might be in an operative combination), versus which operands can be replaced by any other expression that will induce an equivalent computation when evaluated (as would be the case in an applicative combination).

To avoid this difficulty, Scheme and most other modern Lisps do not allow operatives to be directly manipulated as values; hence, operator evaluation can only produce an applicative (or a non-combiner, which would cause evaluation to fail with a dynamic type error). A small set of symbols are then reserved for use as the operators of operative combinations. Combinations using these reserved operators are called *special forms*, because they are exceptions to the rule for combination evaluation:[6] the Lisp evaluator checks for any special-form operator first, and handles each according to its particular operative meaning, or goes on to evaluate the operator and operands if none of those special cases occurred.

Evaluation of the arguments to an applicative combination may be *eager* or *lazy*. Eager argument evaluation, which is the norm in Lisp, means that the operator and operands are all evaluated before further action (specified by the applicative combiner) is taken. Lazy argument evaluation, which is practiced in some other functional languages (such as Haskell), means that the argument evaluations may be postponed until some later time. The only logical constraint on lazy order of operations is that of the applicative model itself, that the arguments must be determined before they are used; and within that constraint there is room for variants lying partway between fully eager and fully lazy argument evaluation; but we have no need to consider them here.[7] Much of our discussion is orthogonal to the lazy/eager distinction; and in those cases where a position must be taken, either fully lazy or (more often, following Scheme) fully eager argument evaluation is used.

2.2.3 Core vocabulary

Most Lisps share a core vocabulary, inherited from their common ancestor; but even within that core, the combiner names, call syntax, and behaviors vary noticeably between dialects, and in particular there are some visible differences between Kernel and Scheme (most of them driven by the difference in combiner treatment). To avoid confusing shifts of vocabulary, Kernel usage will be preferred whenever context permits.[8] Here we briefly sketch the core Lisp vocabulary as it occurs in Kernel, noting its differences from Scheme.

[6] *Form* is an older term for what we're calling a *combination*.

[7] The principal partially-lazy argument-evaluation strategy is *call-by-need*, in which each argument is evaluated when its value is first needed (as in lazy evaluation), but the resulting value is memoized at that time so that the argument is not reevaluated if needed again later. In a language without side-effects this is merely a practical optimization of lazy evaluation, whereas in the presence of side-effects it may produce different results from fully lazy (a.k.a. *call-by-name*) evaluation.

[8] The largest concentration of exceptions, where Kernel usage would be nonsensical, is in the historical discussion of early Lisp in §3.3.2.

Additional details of Kernel will be introduced throughout the dissertation as they become relevant. The full current state of the Kernel language design is detailed in [Shu09].

2.2.3.1 Naming conventions

Kernel inherits from Scheme certain naming conventions, involving special characters, that indicate type information about its combiners. *Predicates* —combiners whose results are always boolean— conventionally have names ending with "`?`". *Mutators* —combiners that modify state of existing objects— conventionally have names ending with "`!`". (There is also a naming convention for type-conversion combiners, x`->`y for types x and y, but we will have no occasion here to do type conversions.)

Kernel applies these conventions more uniformly than does Scheme. This additional uniformity is visible in Kernel's use of the predicate suffix on numeric comparison operators and boolean operators, all of which are excepted by Scheme from the convention; thus, for example, Scheme `<=` (an applicative that tests whether all its arguments are in non-descending numerical order) becomes Kernel `<=?`, while Scheme *`not`* (an applicative that takes one boolean argument and negates it) becomes Kernel *`not?`*.

A *type predicate* is an applicative predicate that tests whether all its arguments belong to a certain type. By convention, the name of a type predicate is x`?`, where x is the name of the type. Nil is the single instance of type *null*, hence its type predicate is *`null?`*. Scheme type predicate *`procedure?`* is replaced in Kernel by type predicates *`combiner?`*, *`operative?`*, and *`applicative?`*. (The first of these three is the logical disjunction of the other two; thus, for example, `(combiner? x)` would be equivalent to `(or? (operative? x) (applicative? x))`.)

Scheme does nothing to distinguish operative names from applicative, so that the programmer must learn by rote to recognize the operative names. This rote memorization is somewhat mitigated by the small number of built-in Scheme operatives, and by the fact that user-defined Scheme operatives —which are hygienic macros— are usually limited in number by the difficulty of constructing them. Also, since Scheme operatives aren't first-class, the programmer *cannot* treat them as values, limiting the scope of their possible misuse (along with the scope of their possible use).

Kernel makes it easy to construct new operatives, and allows them to be used freely as values; so the factors that mitigated rote memorization in Scheme no longer pertain. To allow the programmer to distinguish between the two cases, Kernel uniformly applies the convention that operative names are prefixed by "`$`". (In practice, this convention has proven so effective in clarifying the abstract semantics of user-defined operatives, that one suspects Scheme could benefit from it even if nothing else in its treatment of combiners were changed.)

2.2.3.2 Particular combiners

Each combiner is introduced here by a template for its call syntax, with semantic variables wherever subexpressions may occur.

When a semantic variable occurs in a compound Lisp expression, it is always understood to represent a single Lisp subexpression (never an arbitrary syntactic fragment; for example, a subexpression can't have unbalanced parentheses). It may use the same notation for semantic variables as in general discussion, based on a single italicized letter; but alternatively, it may and usually will be based on a multi-character name, following the same rules as Lisp symbols, either in italics or delimited by angle brackets (but not both). Italics are preferred when describing operands of an applicative, angle brackets when describing operands of an operative; but the two styles are never mixed in a single Lisp expression.

If the name of the semantic variable is the name of a type, either the value of the variable (if in angle brackets) or the result of evaluating that value (if in italics) is constrained to the named type. Thus, in `($define!` ⟨symbol⟩ ⟨expression⟩`)` the first operand is constrained to be a symbol, whereas in `(set-car!` *pair expression*`)` the first operand is constrained to *evaluate* to a pair.

`(cons` *x y*`)`
`(list .` *arguments*`)`
`(car` *pair*`)`
`(cdr` *pair*`)`
`(set-car!` *pair expression*`)`
`(set-cdr!` *pair expression*`)`

cons returns a freshly allocated pair with given car and cdr. ***list*** returns a freshly allocated list of its arguments. ***car*** and ***cdr*** return the named component of a given pair. ***set-car!*** and ***set-cdr!*** mutate a given pair by making their second argument the new value for the named component of the pair.

Many Lisps provide special names for compositions of ***car***'s and ***cdr***'s, consisting of the names of the composed applicatives in the order they would occur in a nested expression, with the intermediate `r`'s and `c`'s left out. Thus,

$$(\textit{cadr}\ x) \equiv (\textit{car}\ (\textit{cdr}\ x)) \tag{2.5}$$

and so on. Common Lisp, Scheme, and Kernel provide names for all such compositions up to four deep (***caaaar*** through ***cddddr***).

`(read)`
`(write` *expression*`)`
`(display` *expression*`)`
`(newline)`

read reads and returns one expression from standard input; ***write*** writes its argu-

ment to standard output. *display* is similar to *write* except that, when outputting a string, *display* omits the delimiting double-quotes. *newline* sends a new-line to standard output. (In full Scheme or Kernel, these applicatives have optional syntax to specify ports other than standard input/output; but there is no need to override standard input/output here.)

`($if` ⟨test⟩ ⟨consequent⟩ ⟨alternative⟩`)`

⟨test⟩ is evaluated, and the result must be of type *boolean*; if that result is true, ⟨consequent⟩ is evaluated and its result returned, otherwise (the result of evaluating ⟨test⟩ is false, and) ⟨alternative⟩ is evaluated and its result returned. The semantics are slightly different in Scheme (where the operative is named `if`): Scheme does not require the result of evaluating ⟨test⟩ to be boolean (treating all non-`#f` values as "true"), and Scheme allows ⟨alternative⟩ to be omitted (in which case the result on false is implementation-dependent).

`($cond .` ⟨clauses⟩`)`

$cond is a more general Lisp conditional, for selecting one out of a series of cases (rather than out of exactly two). ⟨clauses⟩ must be a list of clause expressions, each of the form `(`⟨test⟩ `.` ⟨body⟩`)`, where each ⟨body⟩ is a list of expressions. The ⟨test⟩s are evaluated from left to right until one evaluates to true, then the expressions in the corresponding ⟨body⟩ are evaluated from left to right and the result of the last is returned.

In Kernel, it is good practice always to specify `#t` as the ⟨test⟩ of the last clause, so that the result of the conditional is always explicitly specified. Scheme allows the last ⟨test⟩ to be the symbol `else`, which is reserved in that setting to mean that the clause is always selected if it is reached — a specialized syntax that Kernel doesn't imitate, as Kernel prefers to avoid use of unevaluated keywords.

$cond and *$if* are equi-powerful, in that either could be defined in terms of the other if the other were already available. In formally defining Lisps from scratch, we will prefer to provide primitive *$if*, and usually omit *$cond* entirely, because *$if* is simpler; but in Lisp programming we will more often *use* *$cond*. *$cond* is derived from *$if* in Kernel, and *cond* from *if* in Scheme; however, traditionally in Lisp *cond* is primitive while *if* is derived, because that was the order in which they were introduced into the language (see [McC60]).

`($lambda` ⟨formals⟩ `.` ⟨body⟩`)`

$lambda is the usual constructor of applicatives in Lisp; primitive in most modern Lisps, derived in Kernel (though more often used in practice than the primitives from which it is derived). A great deal will be said later about the semantics of *$lambda* (notably in §3.3.1 and §4.3); for the moment, it suffices to give the general sense of it.

⟨formals⟩ is a *formal parameter tree*; in the most usual case, it is a list of symbols. ⟨body⟩ is a list of expressions.

The dynamic environment in which **$lambda** is called becomes the *static environment* of the constructed applicative. When the applicative is called, a *local environment* is constructed by extending the static environment, so that if a symbol is looked up locally but has no local binding, a binding for it will be sought in the static parent. (Because local lookups default to the static environment, the applicative is said to be *statically scoped*; more on that in §3.3.1.) The symbols in ⟨formals⟩ are locally bound to the corresponding parts of the argument list of the applicative call; then the expressions in ⟨body⟩ are evaluated left-to-right in the local environment, and the result of the last evaluation is returned as the result of the applicative call.

`(apply` *applicative object*`)`

apply provides a way to override the usual rules for argument evaluation when calling an applicative.

applicative is called using *object* in place of the list of arguments that would ordinarily be passed to *applicative*. In particular, a statement of the form

$$\texttt{(}\textbf{apply}\ \ appv\ \ \texttt{(}\textbf{list}\ \ arg_1\ \ \ldots\ \ arg_n\texttt{))} \tag{2.6}$$

is equivalent to

$$\texttt{(}appv\ \ arg_1\ \ \ldots\ \ arg_n\texttt{)}\,. \tag{2.7}$$

(Strictly, this equivalence is only up to order of argument evaluation. In (2.6), *appv* is evaluated either before all the arg_k or after all the arg_k; while in (2.7), Scheme would allow *appv* to be interleaved with the arg_k, and Kernel would always evaluate *appv* first, to determine whether to evaluate the operands.)

Kernel and Scheme both allow additional arguments to **apply**, but with different meanings; Kernel **apply** will be discussed in detail in §4.4.

`($let` ⟨bindings⟩ `.` ⟨body⟩`)`

$let is used to provide some specific local bindings for a local computation (whereas **$lambda** parameterizes a local computation by bindings whose values are to be specified later).

⟨bindings⟩ should be a list of definiend/expression pairings, each of the form `(`⟨definiend⟩ ⟨expression⟩`)`, while ⟨body⟩ should be a list of expressions. A statement of the form

$$\texttt{(}\textbf{\$let}\ \ \texttt{((}\langle \mathrm{sym}_1\rangle\ \ \langle \mathrm{exp}_1\rangle\texttt{)}\ \ \ldots\ \ \texttt{(}\langle \mathrm{sym}_n\rangle\ \ \langle \mathrm{exp}_n\rangle\texttt{))}\ \ \texttt{.}\ \ \langle \mathrm{body}\rangle\texttt{)} \tag{2.8}$$

is equivalent to

$$\texttt{((}\textbf{\$lambda}\ \ \texttt{(}\langle \mathrm{sym}_1\rangle\ \ \ldots\ \ \langle \mathrm{sym}_n\rangle\texttt{)}\ \ \texttt{.}\ \ \langle \mathrm{body}\rangle\texttt{)}\ \ \langle \mathrm{exp}_1\rangle\ \ \ldots\ \ \langle \mathrm{exp}_n\rangle\texttt{)}\,. \tag{2.9}$$

`($define!` ⟨definiend⟩ ⟨expression⟩`)`

The ⟨definiend⟩ is usually a symbol; ⟨expression⟩ is evaluated, and the symbol is then bound (in the dynamic environment of the call to ***$define!***) to the result of that evaluation.

Kernel and Scheme both allow compound ⟨definiend⟩s, but with drastically different semantics. This will matter only, for the moment, in that Kernel does not support Scheme's use of compound definiends to define procedures without an explicit `lambda`: in Kernel one would write

```
($define! square ($lambda (x) (* x x)))
```
, (2.10)

where in Scheme one could use the alternative form

```
(define (square x) (* x x))
```
. (2.11)

(Kernel's handling of compound definiends, which is incompatible with Scheme's shorthand (hence the omission), will figure prominently in the treatment of Kernel style in Chapter 7.)

Note that Kernel's ***$define!*** has a "`!`" suffix on its name because it mutates the dynamic environment from which it is called; strictly speaking, Scheme's ***define*** doesn't mutate the state of any first-class object, since environments in Scheme aren't first-class.

`(eval` *object environment*`)`

Evaluates *object* in *environment*.

The availability of ***eval*** enables meta-programming, and is a signature characteristic of Lisp. However, it is also entangled with the controversial technique of environment capturing (§5.2). ***eval*** was only added to standard Scheme in the *R5RS*, ([KeClRe98, Notes]), with a small, fixed set of "environment specifiers" for use as the second argument.[9]

2.3 Meta-languages for formal systems

Two principal techniques are used in the dissertation to specify the semantics of formal systems (that is, of Lisps and calculi): meta-circular evaluators, and reduction systems. Meta-circular evaluators are used only for Lisps, in Part I. Complete reduction systems are used only for calculi, in Part II; but fragments of reduction systems are also sometimes used to clarify points in Part I. This section outlines both techniques, sufficient for their uses in Part I, and discusses tradeoffs between the two (which are properly a cross-cutting concern).

[9]Common Lisp ***eval*** omits the environment argument entirely, always using a "blank" environment for the evaluation — but then also provides elaborate devices for overriding parts of the behavior of ***eval***, and thereby capturing the environments used within the evaluations it performs ([Ste90, Ch. 20]).

2.3.1 Meta-circular evaluators

A meta-circular evaluator for a programming language $\mathcal{L}$ is an interpreter I for $\mathcal{L}$ written in another programming language $\mathcal{L}'$ *in order to explain* $\mathcal{L}$. An $\mathcal{L}$-interpreter that is only meant to be executed isn't what we mean by a meta-circular evaluator: to merit characterization as a meta-circular evaluator, I should be meant for *two* audiences: the abstract machine of $\mathcal{L}'$, and human beings; and usually more the latter than the former. The human audience is supposed to better understand $\mathcal{L}$ by studying the source code of I — specifically, to understand from I certain (presumably advanced) features of $\mathcal{L}$ in terms of the (presumably simpler) features of $\mathcal{L}'$. Often, $\mathcal{L}'$ is a subset of $\mathcal{L}$.

Although meta-circular evaluators can reduce the size of the programming-language definition problem, they cannot eliminate it. This is the point that the term *meta-circular* was originally intended to convey (in [Rey72]): if $\mathcal{L} = \mathcal{L}'$, the definition would be *circular*, meaning that you couldn't understand the definition of $\mathcal{L}$ unless you already knew $\mathcal{L}$; but even without circularity, all such definitions are *meta*-circular, in that no matter how many such evaluators you have, you can't understand any of them unless you already know at least one programming language.

Meta-circular evaluators were the principal means for formally specifying programming-language semantics through the 1960s and early 1970s. In the mid-1970s, alternatives began to emerge that tie program semantics to non-program mathematical structures — denotational semantics, tying program semantics to extensionally defined mathematical functions; and small-step operational semantics (recently so called), tying program semantics to a term-reduction relation.[10] In competition with these more solidly founded alternatives, use of meta-circular evaluators for general-purpose semantic specification gradually declined.

While meta-circular evaluators have declined in general use, the Lisp community has maintained a vigorous tradition of constructing experimental Lisp interpreters (e.g., [SteSu78b, AbSu96]). S-expression Lisp is peculiarly suited to the meta-circular technique, both as an object-language for meta-circular evaluation, and as a meta-language for meta-circular evaluation *of Lisp*. As an object-language, S-expression Lisp is easier to interpret than most languages, because Lisp has a simple syntax, simple internal representation suitable for both data and programs (pairs and lists), and simple semantics assigned to the internal representation. As a meta-language, Lisp already has built-in facilities for handling the syntax and internal representation of Lisp. All the low-level tedium of reading S-expressions (at least for a simple prototype interpreter) are hidden by Lisp applicative `read`; of writing S-expressions, by `write`; and of handling internal representations, by `cons`, `car`, etc.; so that a Lisp meta-circular evaluator in Lisp can concern itself almost exclusively with semantics

[10] Big-step operational semantics emerged later, in the late 1980s ([To88]). The difference is, broadly, that small-step ties program semantics to the term-reduction relation itself, whereas big-step ties program semantics to complete reduction sequences. (Big- versus small-step semantics will be discussed in §8.3.2.)

(which is, recalling the purpose of the exercise, what one usually wants the meta-circular evaluator to explain).

The central logic for a Lisp meta-circular evaluator in Kernel would look something like

```
($define! interpreter
   ($lambda () (rep-loop (make-initial-env))))

($define! rep-loop
   ($lambda (env)
      (display ">>> ")
      (write (mceval (read) env))
      (newline)                                                    (2.12)
      (rep-loop env)))

($define! mceval
   ($lambda (expr env)
      ($cond ((symbol? expr) (lookup expr env))
             ((pair? expr) (mceval-combination expr env))
             (#t expr)))).
```

The centerpiece of the interpreter is the rep-loop, which prompts the user for an expression, evaluates it meta-circularly, prints the result, and repeats (hence the name `rep-loop`, short for `r`ead-`e`val-`p`rint-`loop`). The interpreter simply sets up an initial environment for evaluation and calls the rep-loop. The rep-loop itself contains none of the evaluation logic, and the main meta-circular evaluation applicative *`mceval`* contains (by our deliberate, but not unreasonable, choice) only logic that is likely to be the same for all Lisps. (Notwithstanding which, each of these three applicatives will be modified in two or more of the meta-circular evaluators in Chapter 6.)

2.3.2 Reduction systems

A *term-reduction system* is a syntactic domain of terms (data with hierarchical structure, a.k.a. term structure) together with a binary reduction relation on terms defined as the compatible closure of a set of reduction rules.

The reduction relation is conventionally named "$\longrightarrow$", with subscripts when necessary to distinguish between the reduction relations of different systems. The transitive closure of any relation is notated by superscripting it by a "+", its reflexive transitive closure, by a "*"; thus, in this case, "$\longrightarrow^{+}$", "$\longrightarrow^{*}$".

The full specification of a reduction system has three parts:

1. syntax, which describes the syntactic structure of terms, and also assigns semantic variables to particular syntactic domains;

2. auxiliary semantic functions, which are named by semantic variables, or use semantic notations, that didn't occur in (1); and

3. a set of reduction rule schemata, which are reduction relation assertions building on the semantic variables and notations from (1) and (2).

Here and in Part I, only reduction rule schemata are specified mathematically, while syntax and auxiliary functions are either implicit or at most informally described. (A detailed discussion of full reduction system specifications will be given in §8.2, preliminary to Part II.)

The λ-calculus, for example, has just one reduction rule schema:

$$(\lambda x.M)N \quad \longrightarrow \quad M[x \leftarrow N]\,, \tag{2.13}$$

where x is universally quantified over syntactic variables, M and N over terms, and notation $M[x \leftarrow N]$ invokes an auxiliary function that (hygienically) substitutes N for all free occurrences of x in M. Thus —assuming the availability of suitable syntactic variables— the schema asserts reductions such as

$$\begin{array}{rcl} (\lambda \mathrm{x.x})\mathrm{y} & \longrightarrow & \mathrm{y} \\ (\lambda \mathrm{x.}(\lambda \mathrm{y.x}))\mathrm{z} & \longrightarrow & \lambda \mathrm{y.z} \\ (\lambda \mathrm{x.}(\lambda \mathrm{y.y}))\mathrm{z} & \longrightarrow & \lambda \mathrm{y.y} \\ ((\lambda \mathrm{x.}(\lambda \mathrm{y.x}))\mathrm{z})\mathrm{w} & \longrightarrow^{+} & \mathrm{z} \\ ((\lambda \mathrm{x.}(\lambda \mathrm{y.y}))\mathrm{z})\mathrm{w} & \longrightarrow^{+} & \mathrm{w}\,. \end{array} \tag{2.14}$$

A relation $\sqsupset$ is *compatible* iff for all $M \sqsupset N$, whenever M occurs as a subterm of a larger term $C[M]$, replacing that subterm M with N in the larger term, $C[N]$, gives a relation $C[M] \sqsupset C[N]$. The compatible closure of a set of reduction rules is then the smallest compatible relation that implies all the rules. Building on the above example (2.14),

$$\begin{array}{rcl} (\lambda \mathrm{x.x})\mathrm{y} & \longrightarrow & \mathrm{y} \\ \lambda \mathrm{z.}((\lambda \mathrm{x.x})\mathrm{y}) & \longrightarrow & \lambda \mathrm{z.y} \\ ((\lambda \mathrm{x.x})\mathrm{y})\mathrm{z} & \longrightarrow & \mathrm{yz}\,, \end{array} \tag{2.15}$$

and so on.

When a reduction rule $M \longrightarrow N$ induces another rule $C[M] \longrightarrow C[N]$ by compatible closure, the incomplete C with a single subterm unspecified is called a *context* (often characterized as "a term with a hole in it"), while the subterm M of $C[M]$ is called a *redex* (a word formed by contraction of ***red****ucible* ***ex****pression*), and subterm N of $C[N]$ a *reduct*.

The distinction between naive substitution and hygienic substitution is in their treatment of $M[x \leftarrow N]$ when M binds a variable that occurred free in N. For the moment, we simply choose our examples so this doesn't happen. The particular form

of bad hygiene caused by naive substitution will figure prominently in the discussion of traditional macros in §3.3.3. Hygienic substitution will be defined in §3.3.3, and again in Chapter 8 (§8.2ff). (See also §3.3.1.)

Lambda calculus has only one class of compound terms; but the reduction systems used here for Lisps and vau calculi —all of which use explicit evaluation— will each distinguish two classes, one of *expressions* that are passive data, and a wider class of terms that also includes designators of expression evaluation. As a mnemonic notational convention, these explicit-evaluation reduction systems use three kinds of delimiters for compound constructions. Compound expressions that may be represented by source code are delimited by parentheses, "`()`". Compound expressions that cannot be represented by source code, but may be the final result of computation, are delimited by angle brackets, "$\langle\,\rangle$". Compound terms that are not expressions are delimited by square brackets, "[]".

Thus, for example, in a typical Lisp reduction system, the source expression for a function to square numbers would be written as

$$\texttt{(\$lambda (x) (* x x))}\,, \tag{2.16}$$

the term designating its evaluation in an environment e would be

$$[\text{eval } \texttt{(\$lambda (x) (* x x))}\ e]\,, \tag{2.17}$$

and the expression to which this term is eventually reduced (assuming e exhibits the usual binding $\texttt{\$lambda} \leftarrow \textit{\$lambda}$) would be

$$\langle\text{applicative } \texttt{(x) ((* x x))}\ e\rangle \tag{2.18}$$

(or, in Kernel, where an applicative is explicitly distinguished from its underlying combiner, one would have something of the form "$\langle$applicative $\langle$operative ...$\rangle\rangle$").

Also, terms that denote environments are delimited by doubled symbols; i.e., the usual delimiter is typeset twice with a small horizontal displacement. So if environments are first-class objects, as in Kernel or vau calculus, their delimiters would be "$\langle\!\langle\,\rangle\!\rangle$"; while, if environments are represented as terms but are not admissible as computation results, they would be delimited by "$[\![\,]\!]$".

2.3.3 Tradeoffs

The specification of a reduction system can be a lucid medium for articulating fine points of order of operations (which computational activities precede which others). The entire specification is geared toward its reduction rule schemata, and each schema is just a statement of one specific kind of permissible succession from computational state to computational state. Moreover, the schemata are understood separately from each other: whether a schema can be applied to a given term is determined entirely within that schema, regardless of any other schemata in the system. So each schema immediately expresses a specific point for the human reader to study in isolation.

The downside of this unremitting independence between schemata, for descriptive purposes, is that while detailed orderings are made explicit, any algorithmic structure larger than a single schema is made *im*plicit. The overall modular structure of the reduction algorithm may have been known to a human who wrote the description; but the description doesn't preserve that information, so the human reader has to reconstruct it.

An important consequence of the lack of overall structure is that there is nothing to prevent multiple schemata from being applicable to the same term. This tendency toward nondeterministic reduction is magnified by taking the compatible closure of the rules, which facilitates the construction of terms with multiple redexes.

If alternative reductions of the same term would result in different final answers (i.e., eventual reduction to different irreducible terms), then the nondeterministic choice between reductions would constitute semantic ambiguity. However, λ-calculus and each of its variants is equipped with a *Church–Rosser theorem*, which guarantees that all possible alternative reductions from a given term can arrive back at the same term later on. Formally, the Church–Rosser property says that if $L \longrightarrow^* M_1$ and $L \longrightarrow^* M_2$ then there exists N such that $M_1 \longrightarrow^* N$ and $M_2 \longrightarrow^* N$. Thus, from a given term L at most one irreducible term can be reached; so the system is semantically unambiguous, and nondeterministic choice between reductions constitutes *algorithmic flexibility* — multiple ways of doing the same thing, which can be exploited by meta-programs (such as optimizing compilers), and which manifests mathematically as strength of the equational theory.

While Church–Rosser nondeterminism is a desirable property for a meta-language, it comes at a price. Proving Church–Rosser theorems is a significant effort; theorists took half a century evolving better ways to prove Church–Rosser-ness for the pure λ-calculus ([Ros82, §4]), simple as it is. However, the very fact that reduction systems are usually used as a meta-language, not as a programming language,[11] means that one can afford to spend far more effort pursuing Church–Rosser-ness of a single reduction system than one could for each of many individual programs.

A general-purpose programming language can be a lucid medium for presenting the structure of an algorithm in a coherent and modular way.

As a pointedly relevant illustration, consider the central dispatching logic of a typical Lisp evaluator. In a reduction system specification, dispatching would appear as a set of schemata, something like

[11] Logic programming languages usually aren't compatible reduction systems, because they introduce global constraints in order to achieve well-behavedness (such as computational tractability).

$$
\begin{aligned}
[\text{eval } a\ e] &\longrightarrow a \\
[\text{eval } x\ e] &\longrightarrow \mathit{lookup}(x, e) \\
[\text{eval } (\texttt{\$if}\ v_1\ v_2\ v_3)\ e] &\longrightarrow [\text{if } [\text{eval } v_1\ e]\ v_2\ v_3\ e] \\
[\text{eval } (\texttt{\$lambda}\ v_1\ .\ v_2)\ e] &\longrightarrow \langle \text{applicative } v_1\ v_2\ e \rangle \\
\forall v_0 \notin \mathit{SpecialFormOperators}, & \\
[\text{eval } (v_0\ v_1\ \ldots\ v_n)\ e] &\longrightarrow [\text{apply } [\text{eval } v_0\ e] \\
& \qquad\qquad ([\text{eval } v_1\ e]\ \ldots \\
& \qquad\qquad\ [\text{eval } v_n\ e])]\,.
\end{aligned}
\tag{2.19}
$$

These schemata are intended to partition all possible subjects of evaluation; but for them to actually do so requires careful arrangement of syntactic domains elsewhere in the specification. Semantic variables with letters e, x, a, v are quantified respectively over environments, symbols, non-symbol atoms, and values; the domain of values is partitioned by construction into symbols, non-symbol atoms, and pairs; and *SpecialFormOperators* must be semantically bound to the set $\{\texttt{\$if}, \texttt{\$lambda}\}$. The use of semantic variable *SpecialFormOperators*, in particular, makes the reduction schema for application more readable at the cost of tighter coupling between syntax specification and dispatching logic: the elements of the set must match the special-form schemata. Even so, the application schema contains an awkward explicit universal quantifier (or equivalent notation) — which is why it is presented last in (2.19), to avoid misleading the reader into applying the explicit quantifier to any of the other schemata. The explicit quantifier could have been eliminated too, but only by further muddling the syntax specification with yet another semantic-variable-letter quantification, over the structurally unnatural domain of arbitrary values that aren't special-form operator symbols.

The same dispatching logic in a meta-circular evaluator (written in Kernel) would be

```
($define! mceval
   ($lambda (expr env)
      ($cond ((symbol? expr)  (lookup expr env))
             ((pair? expr)  (mceval-combination expr env))
             (#t  expr))))

($define! mceval-combination
   ($lambda ((operator . operands) env)
      ($cond ((if-operator? operator)
                 (mceval-if operands env))
             ((lambda-operator? operator)
                (mceval-lambda operands env))
             (#t  (mc-apply (mceval operator env)
                            (map-mceval operands env))))))).
```
(2.20)

The ***$cond*** operative, which dominates the logical structure of this code, was one of McCarthy's key original innovations in the Lisp language ([McC78, p. 218],

[SteSu78b, p. 4]). Its purpose was to encompass all the cases of a piecewise-defined function in a single expression, thus explicitly and succinctly expressing the mutual exclusion between cases (which is what gives (2.20) its modularity) — and specifically avoiding fragmentation of the cases into logically independent rules whose interrelation must be reconstructed by the reader.

Note, also, that the clarity of the code is enhanced by its modular division into two applicatives. No analogous modularization is possible for the reduction system of (2.19). One *could* extend (2.19) with a new schema,

$$[\text{eval } (v_1 \ . \ v_2) \ e] \longrightarrow [\text{eval-combination } v_1 \ v_2 \ e] \tag{2.21}$$

(presumably placed between the second and third schemata shown in (2.19)), and adjust the last three schemata to use 'eval-combination' instead of 'eval' on their left-hand sides; but then one would only have a slightly more verbose —hence, less readable— specification. Selectively subdividing the meta-circular evaluator code of (2.20) is only useful because the code has coherent structure to begin with.

Just as reduction system specifications pay for lucid order of operations by obscuring algorithmic structure, general-purpose programming languages pay for lucid algorithmic structure by obscuring order of operations. A notorious example is the problem of distinguishing between lazy and eager argument evaluation.[12] In extending the schemata of (2.19) to perform primitive applications, one might write

$$[\text{apply } p \ t] \longrightarrow \mathit{applyPrim}(p, t)\,, \tag{2.22}$$

where t is quantified over terms, so that semantic function *applyPrim* may be invoked without first reducing the argument-evaluation subterms within the argument list t; or

$$[\text{apply } p \ v] \longrightarrow \mathit{applyPrim}(p, v)\,, \tag{2.23}$$

so that, since v is quantified only over values, the argument-evaluation subterms have to be reduced to values before invoking *applyPrim*. However, the corresponding meta-circular evaluator code in (2.20) is the call to auxiliary applicative ***mc-apply***,

```
(mc-apply (mceval operator env) (map-mceval operands env)) ;
```  (2.24)

and there is simply nothing inherent in this syntax to indicate whether its second argument should be evaluated before its first argument is called. *If* the meta-language uses eager argument evaluation (which Kernel does), both arguments are evaluated before calling ***mc-apply***, and the object-language is eager; but if the semantics of the Kernel meta-language were modified to be lazy, the object-language would (barring oddities elsewhere in the code) become lazy too.

[12] This problem is described in §5 of [Rey72], the paper that introduced the term *meta-circular evaluator*.

Part II of the dissertation focuses exclusively on reduction systems; but it was made the latter half of the work so that it would be preceded by the informal explanation of the semantics in Part I. The informal explanation uses both kinds of meta-language, capitalizing on the strengths of each. It uses reduction rule schemata selectively, scattered through Chapters 3–4, to clarify specific points of order of operations; but also contains an in-depth exploration of the evaluator algorithm, in Chapter 6, that relies exclusively on meta-circular evaluators.

Part I

The Kernel programming language

Chapter 3

Preliminaries for Part I

3.0 Agenda for Part I

This chapter provides historical perspective and principles of language design preliminary to Part I. Chapter 4 explains Kernel-style operatives and applicatives. Chapter 5 considers a broad class of programming-language well-behavedness properties called *hygiene*, and tactics used by Kernel to promote hygiene in the presence of fexprs. Chapter 6 uses meta-circular evaluator code to compare the simplicity of Kernel's combiner support with that of other approaches to combiner support in Scheme-like languages. Chapter 7 explores how Kernel primitives `$vau`, `wrap`, and `unwrap` can be used effectively to achieve purposes traditionally addressed, in Scheme and other Lisps, by quasiquotation and macros.

3.1 Classes of constructed combiners

There are three major classes of constructed combiners in Lisp languages.

3.1.1 Applicatives

In most Lisp languages, there is an operative `$lambda` that is a universal constructor of applicatives, in the sense that it can implement all applicatives that are possible within the computational model of the Lisp in which it occurs. It is therefore unnecessary to have any other primitive combiner constructors, if one is willing to accept the limitation that all constructed combiners in the language will be applicative. `$lambda` was the only combiner constructor in the original description of Lisp, [McC60], and it was until recently the only combiner constructor in Scheme.[1]

[1]Macros were introduced to Scheme proper in the *R5RS* ([KeClRe98, Notes]), having appeared as a language extension appendix to the *R4RS*, [ClRe91b].

In practice, though, it's convenient to be able to construct new operatives — for modularity, or for simple abbreviation.[2] So, in the 1960s and 70s, Lisps developed two strategies for constructing compound operatives ([Pi80]): *macros* and *fexprs*.

3.1.2 Macros

A macro is an operative that uses its operands to derive a syntactic replacement for the entire combination of the macro call. The computation and substitution of this replacement for the original combination is called *macro expansion*.

All macro expansion is required to take place during a preprocessing phase, strictly prior to all non-macro expression-evaluation. This two-phase processing model is a natural property of syntactic transformation in assembly languages, where syntax is eliminated during assembly; and in conventionally compiled languages, where syntax is eliminated during compilation (cf. *syntactic abstraction*, §1.1); but not so in Lisp. Because of Lisp's meta-programming view of programs as data, syntax doesn't necessarily dissappear prior to evaluation, so that it would be entirely conceivable for macro expansion to be treated as an integral part of evaluation (a possibility further discussed below in §3.4.1). However, the macro-preprocessability requirement is prerequisite to the advantages of macros, discussed below, and so the requirement is maintained in Lisp.[3]

3.1.3 Fexprs

The construction of fexprs differs in only two fundamental respects from the ordinary construction of first-class applicatives via `$lambda`: first, a fexpr is passed its unevaluated operands rather than evaluated arguments (which is to say, it's operative rather than applicative); and second, it may also be passed the dynamic environment from which it was called. A typical constructor of fexprs therefore looks and behaves very similarly to `$lambda`, except that (first) its ordinary formal parameters will be bound to its unevaluated operands, and (second) it may have an additional formal parameter that will be bound to its dynamic environment.

The name FEXPR*s* dates back to LISP 1.5, which supported operatives of this kind in the early 1960s.[4] Later, when Lisp branched into multiple dialects, the feature

[2]The distinction intended is that modularity hides implementation, while abbreviation merely shortens expression. Other historical uses for constructed operatives (such as variable-arity procedures) were really orthogonal to operand evaluation; but these two uses have endured, because some abbreviation and some modularity are entangled with operand evaluation, so that operatives are genuinely needed. See [Pi80].

[3]Discussions of Lisp macros, especially early discussions, don't always state the requirement plainly, but when it matters they assume it. For example, [Pi80] presents macros as data-structure transformers written in ordinary Lisp, —using `car`, `cdr`, `list`, etc.— but then later attributes, as we will here, the key advantages of macros to the fact that they can be preprocessed.

[4]The memo that proposed addition of macros to LISP 1.5, [Ha63], uses *FEXPR* as a noun for

continued under the same name in MacLisp through the 1970s ([Pi83]), while a similar facility was provided by Interlisp —the other major Lisp camp of the seventies[5]— under the name NLAMBDAs. Some experimental languages in the eighties called them *reflective*, or *reifying*, procedures (see §1.2.4). Neither *reflective* nor *reifying* is appropriate to the current work; and recent literature, when it doesn't use either of these two recent terms, prefers the oldest term for the strategy, *fexprs* (even among researchers in the reflective Lisp tradition; see for example [Wa98, Mu91, Baw88]); so we also prefer *fexprs* here.

The fundamental distinction between fexprs and macros is one of explicit versus implicit evaluation. (See §1.2.3.) If the fexpr wishes any of its operands to be evaluated, it must explicitly evaluate them. The dynamic environment is provided to the fexpr so that operands, or parts of operands, can be evaluated *hygienically* (which by definition must occur in the dynamic environment; see §5.1).[6] In contrast, when a macro wishes any of its operands evaluated hygienically, it must arrange for their evaluation implicitly, through the form of the new expression that it constructs. Explicit evaluation of operands cannot occur hygienically during macro expansion because, since macros are required to be preprocessable, the dynamic environment does not necessarily exist yet during macro expansion.

3.2 The case against fexprs

Traditional (1960s–70s) Lisp macros had the limitation that, unlike most entities in Lisp, they were not first-class; and more seriously, they suffered from occasional misbehaviors called *variable capturing*. Both problems could be worked around, though; and up until about 1980, Lisp applicatives suffered from variants of the same two problems.

Moreover, both problems with macros (as both with applicatives) are relatively localized, in that the programmer can deal with them when and where he actually uses the feature. The misbehavior of fexprs is not localized in this sense. The mere possibility that fexprs *might* occur means that, in order to determine whether the operands of a combination will be evaluated, one has to know whether the operator will evaluate to a fexpr — which can only be known, in general, at evaluation time. Thus, a meta-program p examining an object-expression e cannot even tell

this kind of operative. On the other hand, the LISP I manual, [McC+60], mentions `FEXPR` as an internal tag for the operatives, but does not use `FEXPR` as a noun for them.

[5]On the overall shape of the Lisp community over the decades, see the Lisp paper in HOPL II, [SteGa93].

[6]Even in the age of dynamically scoped Lisps, it was necessary to explicitly pass in the dynamic environment. MacLisp supported "one-argument FEXPRs", whose one parameter was bound to the entire list of operands; but then the name of the one parameter could be captured if it happened to occur in an operand, so MacLisp also supported "two-argument FEXPRs", whose second parameter was bound to the unextended dynamic environment. [Pi83].

in the general case which subexpressions of e are program and which are data. An optimizing compiler cannot distinguish, in general, between correct transformation of code and incorrect mangling of data; and, in particular, a macro preprocessor cannot distinguish a macro call that should be expanded from a data structure that should be left alone.

The fundamental issues surrounding fexprs were carefully and clearly laid out by Kent M. Pitman in a 1980 paper, [Pi80]. After enumerating reasons for supporting constructed operatives, he discussed strengths and weaknesses of macros and fexprs and, ultimately, recommended omitting fexprs from future dialects of Lisp:

> It has become clear that such programming constructs as `NLAMBDA`'s and `FEXPR`'s are undesirable for reasons which extend beyond mere questions of aesthetics, for which they are forever under attack.

In other words, fexprs are (were) badly behaved and ugly.[7] The Lisp community mostly followed his recommendation (and, incidentally, accumulated quite a lot of citations of his paper), although, as noted, fexprs occurred under other names in the reflective Lisps of the 1980s; and the misbehavior of fexprs has continued to attract a modicum of attention, both in regard to the feature itself (as [Wa98]) and as a paradigmatic example of undesirable behavior (as [Mi93]).

3.3 Past evolution of combiner constructors

Prior to about 1980, Lisp applicatives and Lisp macros suffered from somewhat different forms of the same two problems: bad hygiene, and second-class status. *Bad hygiene* means, in this context, that variable bindings aren't strictly local to the region of the source code where binding occurred. (More general forms of bad hygiene will be discussed in Chapter 5.) *Second-class status* means that there are arbitrary restrictions on how certain kinds of objects can be employed. (First- and second-class objects were discussed in the Preface, and a more extensive treatment occurs in [Shu09, App. B (First-class objects)].)

3.3.1 Hygienic applicatives

The particular form of bad hygiene that affected applicatives in early Lisps is called *dynamic scope*.[8]

[7] When Pitman said this in 1980, mainstream Lisps were dynamically scoped —a notoriously poorly behaved scoping discipline that had been introduced into Lisp by a bug in the original language description, [McC60]— which, in retrospect, seems to add a certain extra sting to Pitman's aesthetic criticism of fexprs. (Dynamic scope will be discussed below in §§3.3.1–3.3.2.)

[8] The classic reference on the cause, consequences, and cure for dynamic scope is *The Art of the Interpreter*, [SteSu78b], which explores the historical development of Lisp through incremental modifications to a meta-circular evaluator. (Not quite tangentially, that paper makes repeated use

The λ-calculus is *statically scoped*. That is, each variable-instance is bound by the nearest same-named formal parameter of a λ-expression inside which the variable-instance occurs prior to evaluation. For example, in the λ-calculus function

$$\lambda x.(\lambda y.(x + y)), \tag{3.1}$$

variable y is bound locally by the inner λ; while variable x, which is a free variable of the inner λ, is bound by the outer λ. The upshot is that Function (3.1) *curries* binary addition, i.e., breaks it down into a succession of unary functions: it takes a single value x and returns a unary "add to x" function. Thus,

$$\begin{array}{lllll} (\lambda x.(\lambda y.(x + y)))3 & \longrightarrow & \lambda y.(3 + y) & & \\ ((\lambda x.(\lambda y.(x + y)))3)4 & \longrightarrow & (\lambda y.(3 + y))4 & \longrightarrow & 3 + 4 \quad \longrightarrow \quad 7\,. \end{array} \tag{3.2}$$

Some machinery or other is needed to keep track of variable-to-value binding information, and thus maintain the scoping discipline, during computation. λ-calculus accomplishes this by successively rewriting each expression to encode new binding information as the information becomes available. For example, in reduction sequences (3.2), the binding of x to 3 is preserved by rewriting $\lambda y.(x + y)$ as $\lambda y.(3 + y)$. (Despite this tame example, static scope maintenance by expression-rewriting does have potential pitfalls, which will be discussed in §3.3.3, below.)

In Lisp notation, curried-add-function expression (3.1) would be written as

```
($lambda (x) ($lambda (y) (+ x y)))
```
. (3.3)

However, the deliberate notational imitation of λ-calculus in the original Lisp did not extend to Lisp semantics, which were based neither on λ-calculus, nor on expression-rewriting. Lisp therefore needed a different, expression-preserving mechanism to maintain the scoping discipline; and to that end it adopted a mechanism based on *environment* data structures. An environment is, roughly, a list of variable-to-value bindings.[9] Each expression evaluation is provided with an environment in which to

of the term *abstractive power*.) Beyond that, scattered fragments of information for this section have been drawn from other sources. Joel Moses's *The Function of FUNCTION in LISP*, [Mose70], is entirely devoted to scoping, but its treatment predates the Lisp advent of static scope. The running example for this section, `(((lambda (x) (lambda (y) (+ x y))) 3) 4)`, is lifted from [SuSt75, §4]; but that is a (lucid) study of proper tail recursion, not scoping. The first edition of the Wizard Book has a short subsection on dynamic scope, [AbSu85, §4.2.2], that was dropped from the second edition available on the Web, [AbSu96]. Some history of scoping issues can be gleaned from the ACM HOPL and HOPL II papers on Lisp, [McC78] and [SteGa93]; but both papers self-consciously excuse themselves from directly covering it. ([McC78] pleads insufficient time, while [SteGa93] pleads insufficient space.)

[9] In a statically scoped Lisp that allows mutations to environments (even if just to the global environment, such as entering new top-level declarations), an environment is commonly represented as a *list of lists* of bindings. We simplify our discussion of environments in this section by disregarding environment mutation; interactions between mutation and environments are a substantial topic, accounting for much of the discussion in [SteSu78b]. (A still more elaborate *procedural* representation of environments is sometimes used in reflective Lisps; see [Baw88].)

look up the values of variables, and must then provide an appropriate environment to each subsidiary evaluation.

The first implementations of Lisp treated the environment as part of the state of the evaluator algorithm. In most subcases of evaluation, this traditional Lisp approach to scoping agrees with the expression-rewriting approach of λ-calculus: the Lisp evaluator recursively descends the structure of the expression, so that binding information passes from expression to subexpression just as it would in λ-calculus. For example, writing "[eval v e]" for evaluation of value v in environment e,

$$\begin{aligned} [\text{eval } \texttt{(\$if } v_1\ v_2\ v_3\texttt{)}\ e] &\longrightarrow [\text{if } [\text{eval } v_1\ e]\ v_2\ v_3\ e] \\ [\text{if } \texttt{\#t } v_2\ v_3\ e] &\longrightarrow [\text{eval } v_2\ e] \\ [\text{if } \texttt{\#f } v_2\ v_3\ e] &\longrightarrow [\text{eval } v_3\ e]\,. \end{aligned} \tag{3.4}$$

That is, to evaluate an ***$if***-expression in e, first evaluate the test clause in e; and then, if the result of the test is true, evaluate the consequent clause in e, or if the result is false, evaluate the alternative clause in e.

When an applicative is treated as data, it is (by common understanding of the notion of *data*) not part of the evaluator state; so under the traditional scoping strategy, since binding information is part of the evaluator state, an applicative should contain no binding information. Its information content is therefore just the content of the ***$lambda***-expression that specified it, and in traditional Lisps the applicative value is simply represented by the ***$lambda***-expression itself. Following this strategy,

$$[\text{eval } \texttt{(\$lambda (}x_1\ \ldots\ x_n\texttt{)}\ v\texttt{)}\ e] \longrightarrow \texttt{(\$lambda (}x_1\ \ldots\ x_n\texttt{)}\ v\texttt{)}\,. \tag{3.5}$$

(We're ignoring for the moment some aspects of the traditional Lisp handling of applicatives that, though important, would only confuse our explanation of dynamic/static scoping. Other aspects of the traditional treatment will be discussed in §3.3.2.)

Under the traditional strategy, scoping of an applicative combination follows the same general pattern as scoping of an ***$if***-expression, (3.4), with an environment propagated to subexpressions by recursive descent; except that in the compound-applicative case, local parameter-to-argument bindings are prefixed to the environment as it passes downward into the evaluation of the body of the applicative:

$$\begin{aligned} &\forall\ v_0 \notin \textit{SpecialFormOperators}, \\ &\quad [\text{eval } \texttt{(}v_0\ \ldots\ v_n\texttt{)}\ e] \\ &\qquad \longrightarrow [\text{apply } [\text{eval } v_0\ e]\ \texttt{(}[\text{eval } v_1\ e]\ \ldots\ [\text{eval } v_n\ e]\texttt{)}\ e] \\ &[\text{apply } \texttt{(\$lambda (}x_1\ \ldots\ x_n\texttt{)}\ v\texttt{)}\ \texttt{(}v_1\ \ldots\ v_n\texttt{)}\ e] \\ &\qquad\qquad \longrightarrow [\text{eval } v\ [\![x_1 \leftarrow v_1;\ \ldots\ x_n \leftarrow v_n]\!] \cdot e] \end{aligned} \tag{3.6}$$

(writing "$[\![x_1 \leftarrow v_1;\ \ldots\ x_n \leftarrow v_n]\!]$" for an environment binding variable x_1 to value v_1, then x_2 to v_2, etc., and "$\cdot$" for concatenation of environments).

Now, consider the behavior of the Lisp curried-add function (3.3). Under static scope, `(($lambda (x) ($lambda (y) (+ x y)) 3)` should evaluate to an "add to

3" function, and `(((`$`lambda (x) (`$`lambda (y) (+ x y))) 3) 4)` should evaluate to 7, as their λ-calculus analogs did in (3.2). Following the traditional strategy of Schemata (3.5) and (3.6), though,

$$\begin{aligned}
&[\text{eval } \texttt{((\$lambda (x) (\$lambda (y) (+ x y))) 3)}\ e] \\
&\quad\longrightarrow\quad [\text{apply } [\text{eval } \texttt{(\$lambda (x) (\$lambda (y) (+ x y)))}\ e] \\
&\qquad\qquad\qquad\quad ([\text{eval } \texttt{3}\ e]) \\
&\qquad\qquad\qquad\quad e] \\
&\quad\longrightarrow^{+}\quad [\text{apply } \texttt{(\$lambda (x) (\$lambda (y) (+ x y)))}\ \texttt{(3)}\ e] \\
&\quad\longrightarrow\quad [\text{eval } \texttt{(\$lambda (y) (+ x y))}\ [\![\texttt{x} \leftarrow \texttt{3}]\!] \cdot e] \\
&\quad\longrightarrow\quad \texttt{(\$lambda (y) (+ x y))}\,.
\end{aligned} \tag{3.7}$$

So the binding of `x` to `3` is simply *lost*; in fact, the entire subcomputation result `3` is lost, and

$$\begin{aligned}
&[\text{eval } \texttt{(((\$lambda (x) (\$lambda (y) (+ x y))) 3) 4)}\ [\![\texttt{+} \leftarrow \texttt{+}]\!]] \\
&\quad\longrightarrow\quad [\text{apply } [\text{eval } \texttt{((\$lambda (x) (\$lambda (y) (+ x y))) 3)} \\
&\qquad\qquad\qquad\quad\ [\![\texttt{+} \leftarrow \texttt{+}]\!]] \\
&\qquad\qquad\qquad\quad ([\text{eval } \texttt{4}\ [\![\texttt{+} \leftarrow \texttt{+}]\!]]) \\
&\qquad\qquad\qquad\quad [\![\texttt{+} \leftarrow \texttt{+}]\!]] \\
&\quad\longrightarrow^{+}\quad [\text{apply } \texttt{(\$lambda (y) (+ x y))}\ \texttt{(4)}\ [\![\texttt{+} \leftarrow \texttt{+}]\!]] \\
&\quad\longrightarrow\quad [\text{eval } \texttt{(+ x y)}\ [\![\texttt{y} \leftarrow \texttt{4};\ \texttt{+} \leftarrow \texttt{+}]\!]] \\
&\quad\longrightarrow^{+}\quad [\text{apply } \texttt{+}\ ([\text{eval } \texttt{x}\ [\![\texttt{y} \leftarrow \texttt{4};\ \texttt{+} \leftarrow \texttt{+}]\!]]\ \texttt{4})\ [\![\texttt{y} \leftarrow \texttt{4};\ \texttt{+} \leftarrow \texttt{+}]\!]]\,,
\end{aligned} \tag{3.8}$$

which then fails because there is no binding for `x` in environment $[\![\texttt{y} \leftarrow \texttt{4};\ \texttt{+} \leftarrow \texttt{+}]\!]$.

Under the scoping behavior of λ-calculus, local binding $[\![\texttt{y} \leftarrow \texttt{4}]\!]$ should have been prefixed not to $[\![\texttt{+} \leftarrow \texttt{+}]\!]$, but to $[\![\texttt{x} \leftarrow \texttt{3};\ \texttt{+} \leftarrow \texttt{+}]\!]$ — which was in effect only at the point where `(`$`lambda (y) (+ x y))` was evaluated. In modern terminology, the environment in effect where a compound combiner is first defined is its *static environment* (or, in an older usage, its *lexical environment*), while the environment in effect where the combiner is applied is its *dynamic environment*. The scoping discipline is then static or dynamic depending on which of these environments is suffixed to the local bindings in the rule for applying a compound applicative. The curried addition failed in (3.7)/(3.8) because the curried add function assumes static scope, but the traditional strategy of (3.5)/(3.6) implements dynamic scope.

In order to implement static scope, the schema for applying a compound applicative has to be able to extract the static environment from the compound applicative value — which means that the static environment has to have been bundled *into* the representation of the compound applicative at the point where the applicative was first defined. The dynamic *`$lambda`*-evaluation schema (3.5) is replaced by

$$[\text{eval } \texttt{(\$lambda (}x_1\ \ldots\ x_n\texttt{)}\ v\texttt{)}\ e] \longrightarrow \langle \text{applicative } \texttt{(}x_1\ \ldots\ x_n\texttt{)}\ v\ e\rangle \tag{3.9}$$

and the application schemata become

$$\begin{array}{l}\forall\, v_0 \notin \mathit{SpecialFormOperators},\\ \quad [\mathtt{eval}\ \ (v_0\ \ldots\ v_n)\ \ e]\\ \qquad\longrightarrow\ [\mathtt{apply}\ \ [\mathtt{eval}\ v_0\ e]\ \ ([\mathtt{eval}\ v_1\ e]\ \ldots\ [\mathtt{eval}\ v_n\ e])]\\ [\mathtt{apply}\ \ \langle \mathtt{applicative}\ \ (x_1\ \ldots\ x_n)\ \ v\ \ e\rangle\ \ (v_1\ \ldots\ v_n)]\\ \qquad\longrightarrow\ [\mathtt{eval}\ \ v\ \ [\![x_1 \leftarrow v_1;\ \ldots\ x_n \leftarrow v_n]\!]\cdot e]\,.\end{array} \tag{3.10}$$

Under these new rules,

$$\begin{array}{ll}[\mathtt{eval}\ \ \mathtt{((\$lambda\ (x)\ (\$lambda\ (y)\ (+\ x\ y)))\ 3)}\ \ e]\\ \quad\longrightarrow & [\mathtt{apply}\ [\mathtt{eval}\ \ \mathtt{(\$lambda\ (x)\ (\$lambda\ (y)\ (+\ x\ y)))}\ \ e]\\ & \qquad([\mathtt{eval}\ \ \mathtt{3}\ \ e])]\\ \quad\longrightarrow^{+} & [\mathtt{apply}\ \langle \mathtt{applicative}\ \ \mathtt{(x)\ \ (\$lambda\ (y)\ (+\ x\ y))}\ \ e\rangle\ \ \mathtt{(3)}]\\ \quad\longrightarrow & [\mathtt{eval}\ \ \mathtt{(\$lambda\ (y)\ (+\ x\ y))}\ \ [\![\mathtt{x} \leftarrow \mathtt{3}]\!]\cdot e]\\ \quad\longrightarrow & \langle \mathtt{applicative}\ \ \mathtt{(y)\ \ (+\ x\ y)}\ \ [\![\mathtt{x} \leftarrow \mathtt{3}]\!]\cdot e\rangle\end{array} \tag{3.11}$$

and

$$\begin{array}{ll}[\mathtt{eval}\ \ \mathtt{(((\$lambda\ (x)\ (\$lambda\ (y)\ (+\ x\ y)))\ 3)\ 4)}\ \ [\![\mathtt{+} \leftarrow \mathtt{+}]\!]]\\ \quad\longrightarrow^{+} & [\mathtt{apply}\ \langle \mathtt{applicative}\ \ \mathtt{(y)\ \ (+\ x\ y)}\ \ [\![\mathtt{x} \leftarrow \mathtt{3};\ \ \mathtt{+} \leftarrow \mathtt{+}]\!]\rangle\ \ \mathtt{(4)}]\\ \quad\longrightarrow & [\mathtt{eval}\ \ \mathtt{(+\ x\ y)}\ \ [\![\mathtt{y} \leftarrow \mathtt{4};\ \mathtt{x} \leftarrow \mathtt{3};\ \mathtt{+} \leftarrow \mathtt{+}]\!]]\\ \quad\longrightarrow^{+} & [\mathtt{apply}\ \ \mathtt{+}\ \ \mathtt{(3\ 4)}]\\ \quad\longrightarrow & \mathtt{7}\,.\end{array} \tag{3.12}$$

3.3.2 First-class applicatives

The *$lambda* constructor was included in the original design of Lisp, in 1960, to support first-class applicatives; yet, first-class applicatives were embraced by the Lisp community only with mainstream acceptance of Scheme, around 1980. The twenty-year incubation period of first-class applicatives is commonly explained (e.g., [Gra93, §5.1]) by observing that an applicative as a return value is only interesting if the applicative is statically scoped, and that Scheme was the first Lisp in which applicatives were statically scoped.

The former observation (about the importance of static scope) is borne out by the analysis of scoping issues in §3.3.1.

The latter observation, however, is false. By 1962, the extant dialect of Lisp (LISP 1.5, [McC+62]) allowed statically scoped applicatives to be returned as values, passed as arguments, stored in data structures, etc. The feature wasn't passed on to later dialects, though. How it failed is an instructive study in the nature of first-class-ness, and, in particular, of first-class treatment of combiners.

The evolution of applicative treatment in LISP 1.5 was largely a playing out of consequences from the original description of Lisp ([McC60]).

In its original form, Lisp syntax distinguished carefully between S-expressions, which could only specify constants, and M-expressions, which included the self-evaluating constants but could also specify arbitrary computations. The result of evaluating an M-expression was either an S-expression, or an "S-function" mapping S-expressions to S-expressions.[10] There were just six possible forms of M-expression:

0. An S-expression. Either a constant symbol, using upper-case letters and digits; or a dotted pair or list, delimited by parentheses, `()`, whose elements are S-expressions. ([McC60] separated list elements by commas, an archaism we won't imitate.)

1. A variable, using lower-case letters (thus distinguishing variables from constant symbols).

2. A combination, $f[e_1; \ldots; e_n]$, where f is an operator and the e_k are M-expressions.

3. A conditional expression, $[p_1 \rightarrow e_1; \ldots; p_n \rightarrow e_n]$, where the p_k and e_k are arbitrary M-expressions.

4. A lambda expression, $\lambda[(x_1 \ldots x_n); e]$, for constant symbols x_k and arbitrary M-expression e. (The x_k have to be constant symbols, rather than variables, because the entire first operand is a list — which is an S-expression.)

5. A label expression, $\mathrm{label}[a; m]$, where a is a variable and m is an M-expression. (This device was used to let a λ-expression m refer to itself recursively by the variable name a.)

McCarthy felt that Lisp was suited for both theoretical and practical purposes; and he sought to justify that belief, in [McC60], by demonstrating that a universal M-expression for evaluating M-expressions, analogous to a universal Turing machine for running Turing machines, is both possible (theoretically) and also less cumbersome than its Turing-machine equivalent (practically). M-expressions don't act on M-expressions, though; they act on S-expressions; so, just as a universal Turing machine

[10] This characterization of S-functions precludes encapsulated data structures. If S-functions map S-expressions to S-expressions, and S-functions are to be permitted both as arguments to S-functions and as results of S-functions, then S-functions must be represented by S-expressions. Identifying S-functions with the S-expressions that define them leads to dynamic scope, as discussed above in §3.3.1, while representing S-functions as S-expressions under static scope requires an S-expression representation of environments. LISP 1.5 ([McC+62]) did, in fact, represent environments as lists of symbol-value pairs. The lack of encapsulation in early Lisp won't matter for the discussion in this section; but elsewhere in the dissertation, encapsulation is a significant theme. It appears as a design goal in §1.1 (sometimes under the alias "information hiding"), a psychological factor in §1.2.4 (under the alias "abstraction barrier"), and a technical prerequisite to Kernel hygiene in §5.3.

acts on an encoding of a Turing machine as a string, a universal M-expression acts on an encoding of an M-expression as an S-expression. The encoding of an M-expression m as an S-expression m' was:

0. If m is an S-expression, m' is `(QUOTE` m`)`.

1. If m is a variable, m' is formed by converting the letters of m to upper case. (So `car` becomes `CAR`, etc.)

2. If m is $f[e_1;\ldots;e_n]$, m' is `(`f' e'_1 $\ldots$ e'_n`)`.

3. If m is $[p_1 \rightarrow e_1;\ldots;p_n \rightarrow e_n]$, m' is `(COND (`p'_1 e'_1`)` $\ldots$ `(`p'_n e'_n`))`.

4. If m is $\lambda[(x_1 \ldots x_n);e]$, m' is `(LAMBDA (`x'_1 $\ldots$ x'_n`)` e'`)`.

5. If m is label$[a;m_0]$, m' is `(LABEL` a' m'_0`)`.[11]

Rule 0 of the encoding introduced quotation into the language, creating a bias toward implicit evaluation that has persisted to the present day. Quotation and implicit evaluation were discussed at length in §1.2. (For a different analysis of Rule 0, see [Mu92].)

A critical property of the encoding as a whole is that S-expression m' has the same syntactic structure as M-expression m. On one hand, structural preservation made it easy for a universal M-expression to simulate evaluation of m by manipulating m', promoting McCarthy's contrast versus universal Turing machines.[12] On the other hand, structural preservation made it easy for the programmer to write programs directly as S-expressions rather than as M-expressions. The demonstration of a simple universal M-expression meant that an S-expression interpreter could be readily implemented; and the ease of S-expression programming meant that the interpreter would constitute a viable programming environment. The interpreter was implemented, programmers used it, and S-expression Lisp became the *de facto* form of the language.

The syntax of compound expressions in S-expression Lisp makes no distinction between applicative operators and operands: both are S-expressions. However, M-expression syntax assigns applicative operators to a different syntactic domain than operands, and since evaluation was still perceived (for several more years) as acting on encoded M-expressions, the uniform syntax of S-expressions did not create an expectation of uniform evaluation of operators versus operands.

The operator/operand asymmetry in Lisp was also more pronounced than it might have been because the M-expression language wasn't really finished. McCarthy had

[11] The modern Scheme/Kernel equivalent of *`label`* is called `$letrec`. See §7.1.1.

[12] Universal Turing machines have to simulate object Turing machine M by manipulating a structurally distorted string M'. The distortion is guaranteed, in general, because Turing machine M is essentially a directed graph, while string M' is... a string.

anticipated having more time to refine the design, during the gradual process of constructing a Lisp compiler; and once a programming community and body of existing code began to develop around the S-expression interpreter, further design changes that couldn't be avoided were complicated by the drive to maintain compatibility with the existing language.

Of the six forms of M-expression, Form 2 is the only one that uses applicative operators. (Forms 4 and 5 use operative operators; Forms 0 and 3 induce operative S-expression operators; and Form 1 is purely atomic.) However, [McC60] doesn't (directly) specify just which of the six forms of M-expression can be *used* as applicative operators. Based on the internal typing of the M-expression language, at least one case is clearly excluded from use: Form 0 (an S-expression) can't be an applicative operator, because S-expressions aren't S-functions. (Note, however, that this reasoning breaks down after the shift to S-expression Lisp, where S-expressions are not necessarily self-evaluating.) The proposed universal M-expression supported three of the six forms as applicative operators: variables (Form 1), lambda-expressions (Form 4), and label-expressions (Form 5).

The central combination-evaluation algorithm was:

```
($define! apply
   ($lambda (operator operands env)
      ($cond ((symbol? operator)
                 (apply-symbol operator operands env)
              ((lambda-operator? (car operator))
                 (apply-lambda operator
                    (map-eval operands env) env))
              ((label-operator? (car operator))
                 (apply-label operator operands env)))))

($define! apply-symbol
   ($lambda (operator operands env)
      ($cond ((quote-operator? operator)
                 (car operands))
              ((cond-operator? operator)
                 (eval-cond operands env))
              ((primitive-applicative? operator)
                 (apply-prim-appv operator
                    (map-eval operands env) env))
              (#t  (apply (lookup operator env)
                          (map-eval operands env)
                          env)))).
```

(3.13)

In keeping with the principle of minimality, this algorithm only checks for the `LAMBDA` and `LABEL` operators at a single point — in ***`apply`***, but not in ***`apply-symbol`***.

Consequently, the algorithm could evaluate an expression

```
((LAMBDA (X) (* X X)) 4)                                    (3.14)
```

(assuming primitive applicative operator `*`), but not

```
(LAMBDA (X) (* X X))                                        (3.15)
```

(which would be handled by ***apply-symbol***, where there is no provision for `LAMBDA`). There is no technical need to add evaluator logic to handle the latter, because whenever a lambda-expression is wanted in a non-operator position, one can specify it using `QUOTE`. For example, the expression

```
((LAMBDA (F X) (F (F X))) (QUOTE (LAMBDA (X) (* X X))) 4)   (3.16)
```

would apply the function `(LAMBDA (X) (* X X))` to value `4` twice, producing the result `256`.

The use of `QUOTE` to specify a function presupposes, however, that the function is fully represented by its lambda-expression — in which case, as discussed earlier (§3.3.1), the function will be dynamically scoped. This problem was discovered fairly soon once the interpreter was in use, and the interpreter was extended to support statically scoped functions. A new operative operator `FUNCTION` was added, that bundled its operand together with the current environment into a record structure called a *funarg*:

```
($define! apply-symbol
   ($lambda (operator operands env)
      ($cond ((quote-operator? operator)  (car operands))
             ((cond-operator? operator)
                (eval-cond operands env))
             ((function-operator? operator)           ; ***
                (list funarg-tag (car operands) env)) ; ***
             ((primitive-applicative? operator)
                (apply-prim-appv operator
                   (map-eval operands env) env))
             (#t  (apply (lookup operator env)
                         (map-eval operands env)  env))))).
```
(3.17)

An additional clause in ***apply*** would handle the funarg according to static scope, applying the stored operator in the stored environment. Also, by this time a default clause had been added to ***apply*** to evaluate arbitrary compound operators that didn't match any other clause:

```
($define! apply
   ($lambda (operator operands env)
      ($cond ((symbol? operator)
                 (apply-symbol operator operands env)
              ((funarg-tag? (car operator))            ; ***
                 (apply (cadr operator) operands       ; ***
                        (caddr operator)))             ; ***
              ((lambda-operator? (car operator))                (3.18)
                 (apply-lambda operator
                    (map-eval operands env) env))
              ((label-operator? (car operator))
                 (apply-label operator operands env))
              (#t  (apply (eval operator env) operands ; +++
                          env))                        ; +++
               ))).
```

Under this extended algorithm, the curried-add example from §3.3.1 could be coded as

```
(((LAMBDA (X) (FUNCTION (LAMBDA (Y) (+ X Y)))) 3) 4),          (3.19)
```

which would then evaluate correctly to 7.

The question arises of whether applicatives under this evaluation algorithm are first-class objects. They do have the four rights of first-class objects standardly cited for Scheme (by the Wizard Book, [AbSu96, §1.3.4]) — to be named by variables, passed as arguments, returned as values, and stored in data structures. Yet, they can only appear as themselves in an operator position. When they appear in any other capacity, they have to display special tags on their fronts (`QUOTE` or `FUNCTION`), announcing to the world that they are traveling outside their native community (in which non-combiners aren't welcome).

In fact, the segregation of combiners in LISP 1.5 was a stage more pronounced than shown in (3.17) and (3.18), because thus far we have ignored the complicating LISP 1.5 concept of *property lists*. Rather than an environment simply mapping each variable to a single value, it would map each variable to a mapping from property tags to values. The evaluator used five property tags: `APVAL` (used to store values for general use), `SUBR` (used to store the addresses of compiled applicatives), `EXPR` (used to store S-expressions representing compound applicatives), `FSUBR` (used to store the addresses of compiled operatives), and `FEXPR` (used to store S-expressions representing compound operatives). When a symbol was evaluated in a non-operator position, its APVAL value was used; when in an operator position, its FSUBR or FEXPR value was used if present (passing the operands unevaluated), or failing that its SUBR or EXPR value (evaluating and passing the arguments). In effect, each environment was partitioned into separate neighborhoods, with barriers of administrative overhead between them that discouraged mixing.

In LISP 1.5, because environments were represented as S-expressions (specifically, *alists — Association LISTs*), an environment would be allocated on the heap and remain there until deallocated by the garbage collector. After about 1965, the evolution of Lisp branched into multiple paths ([SteGa93, §2.1]), and descendant dialects used alternative representations of environments to achieve faster evaluation. Stacks, in particular, were commonly used as (or at least in) these representations, because allocation and deallocation of space on a stack can be performed rapidly. However, a statically scoped combiner has to preserve the environment in which it was constructed, and if it is then to be first-class, that environment has to persist arbitrarily long after combiner construction — until, in particular, the combiner becomes garbage. So an environment representation that isn't on the heap, i.e., isn't within the purview of the garbage collector, can't support first-class statically scoped combiners. Thus the funarg feature from LISP 1.5 was dropped.

Under the right circumstances, heap allocation of environments can be fairly time-efficient. *If* the Lisp dialect is purely statically scoped, then the relative position of a variable's storage location on the alist can always be calculated at compile-time (for combiners whose definition is explicit in the source code, thus known at compile time); and *if* environments are also encapsulated such that their content is only used to find the storage location for any given variable, they can be represented with arrays (i.e., contiguous memory blocks rather than linked lists of pairs); so that, given both assumptions, symbol lookups can be implemented using fast array-element accesses instead of linear searches. However, both assumptions were false in LISP 1.5: dynamically scoped combiners were not only possible, but were the rule, or at least were perceived to be the rule; and environments were unencapsulated.

The later Lisp dialects again encountered the problems of dynamic scope, and classified them into two kinds of problem with different implementation properties ([Mose70]), called the *downward funarg problem* and the *upward funarg problem* ([AbSu85, §4.2.2]).

The downward funarg problem occurs when a dynamically scoped function f is passed as an argument into another function where some of the free variables of f have local bindings that override those that existed at f's definition. For example, one might write (returning now to Kernel notation[13])

```
($define! y 3)
($define! f ($lambda (x) (+ x y)))
($define! g ($lambda (y) (+ 1 (f y))))
(g 5),
```
(3.20)

[13]This is an opportune moment to revert to Kernel notation, because otherwise we'd now be introducing still more syntax, in order to cope with definitions in an environment partitioned by property lists; and the complication would only distract from the discussion, without contributing anything useful.

in which free variable `y` is bound to `3` at the point where *`f`* is defined, but locally bound to the argument of *`g`* at the point where *`f`* is called. Under dynamic scope, `(g 5)` would evaluate to `11`; while under static scope, it would evaluate to `9`.

The downward funarg problem can be resolved by a limited form of `$function` operator, while still using stack-based allocation of environments, as long as the statically scoped applicatives constructed by *`$function`* are only passed *down* the stack.[14] Once an applicative call returns, its environment is deallocated from the stack, so any statically scoped applicative created within that environment is no longer valid. (Hence the term *downward funarg*.) In fact, Algol, which was a statically scoped stack-based language, allowed procedures (applicatives) to be passed as arguments to other procedures — but did not allow procedures to be returned as results. (As remarked here in the Preface, procedures in Algol were Strachey's paradigmatic example of second-class objects.)

The *upward* funarg problem concerns the consequences of dynamically scoping an applicative that is returned as the result of an applicative call. The running example from §3.3.1,

```
(($lambda (x) ($lambda (y) (+ x y))) 3) 4)
```
, (3.21)

was of this kind. The static environment of the applicative would have to be passed up the call stack, and persist after the death and presumed deallocation of the call-stack frame in which that environment was captured. A stack-based environment-allocation strategy would require extraordinary measures —both complex and likely time-expensive— to reconcile with this case of static scope. The historical perception of upward funargs as a low priority is inherent in the name that was given to the static-scoping feature when it was first introduced — *funarg*, short for FUNctional *ARGument*. (Outside the literature of traditional Lisp, such bundlings of functions with environments are usually called by the non-biasing term *closures*.[15])

Several design biases against upward funargs were simultaneously removed in 1975 by an experimental Lisp dialect called *Scheme* ([SuSt75]). In Scheme, all applicatives

[14]This assumes that the stack grows downward, i.e., when *`g`* calls *`f`* the stack frame for *`g`* is above the stack frame for *`f`*. Arbitrary directional conventions like this find their way into technical terminology with great regularity (left adjoints versus right adjoints in category theory come to mind). In fairness, this particular directional convention may have more mnemonic value when one recognizes that stack growth is being related to syntax traversal from term to subterm, which is traditionally called recursive *descent* as syntax trees are traditionally oriented with the root at the top and subterms below — also a somewhat arbitrary directional convention, but a very well-established one. The upward/downward terminology will relate directly to syntax, without the intervening notion of stack, in the term-calculus treatment of §13.3.

[15]The term *closure* is due to P.J. Landin. Joel Moses ([Mose70, p. 12]) suggests a metaphorical interpretation for the term:

> Think of `QUOTE` as a porous or an open covering of the function since free variables escape to the current environment. `FUNCTION` acts as a closed or nonporous covering.

were statically scoped, so it was not possible for dynamically scoped applicatives to divert attention from the static case; applicatives, and therefore effectively environments, were encapsulated (at least in principle), opening the door for rapid compiled lookups; there were no property lists, eliminating the administrative segregation of environments; and operators were evaluated using the same general-purpose evaluation algorithm as operands (which, together with exclusively static scope, also eliminated special tags on lambda-expressions in non-operator positions). This wholesale removal of obstacles resulted in recognition, over the next several years, of the programming and implementation advantages of statically scoped first-class applicatives that had previously failed to garner attention (e.g. [SteSu78b, Ste78], also [SteSu76, Ste76]). The Scheme authors, by their own account, hadn't intended Scheme as an improved Lisp: it was an exploration, within the general framework of Lisp, of ideas in the Actors model of computation. Actors, being of a considerably later vintage than Lisp, had been designed with smoothnesses such as operator/operand symmetry from the outset; and, as the Scheme authors discovered that actors were entirely isomorphic to first-class statically scoped applicatives, Scheme applicatives inherited the entire gamut of smoothness properties intact.

As the merits of statically scoped Lisp became evident, static scope became a uniform property of subsequent Lisp dialects.

In a curious twist of fate (or committees), Common Lisp is statically scoped, but preserves both the pre-Scheme operator/operand evaluation asymmetry, and segregation of environments between applicatives and general values.

3.3.3 Hygienic macros

A traditional macro, in the most usual and simplest case, is specified by a list of parameters and a template; the operands of the macro call are matched with the parameters for substitution into the template, and the resulting expression replaces the macro call in the source code. For example, one might specify a macro ***$max*** that takes two parameters, compares them numerically, and returns the larger of the two, using some syntax such as

```
($define-macro ($max a b) -> ($if (>=? a b) a b)) .
```
(3.22)

This operative combination specifies a reduction schema

($max v_1 v_2) $\longrightarrow$ ($if (>=? v_1 v_2) v_1 v_2)
(3.23)

to be applied during a preprocessing phase, prior to ordinary evaluation. ***$max***, and ***$define-macro***, fall short of first-class status exactly because they are required to be preprocessable: they do not have the right to be employed in any way that would entangle them with ordinary evaluation such that they couldn't be preprocessed.

It's clear that ***$define-macro*** is an operative, since its first two of three operands are never evaluated (hence the $ prefix). The operative status of ***$max*** is subtler. The

distinguishing characteristic of an applicative, as defined in §2.2.2, is that it doesn't *use* the operands themselves, only the results of their evaluations. But a macro, by its very nature, uses the unevaluated operands by copying them into the template. This gives the macro complete control over when, and even whether, the operands are evaluated. Just because an operative chooses to evaluate its operands doesn't make it an applicative — because the choice was its to make.

The particular macro definition (3.22) has two problems. One problem is that, depending on the result of the comparison, one or the other operand will be evaluated a second time. (To be precise, two *copies* of one or the other operand will both be separately evaluated — a distinction that is only a difference in the pathological, but entirely possible, case that the operand evaluation has side-effects.) Such multiple evaluation isn't usually what one wants to happen; when one writes an operand once, one tends to expect that it will be evaluated at most once, unless one is invoking an operative whose overt purpose is repetition. Macros are always prone to the problem of unintended multiple evaluations. In any particular case, it can be prevented by some contortion of the template; here, one could write

```
($define-macro ($max a b) ->
               ($let ((x  a)
                      (y  b))
                  ($if (>=? x y) x y)));
```
(3.24)

but the problem must be addressed separately for each macro, because it concerns what each particular macro chooses to do. The reason it's a *problem* —the reason why the error is very easy to commit— is that macros use implicit evaluation, so that any programmer reading the macro (including the programmer who writes it) has to deduce when and where operand evaluations occur, rather than being told when and where explicitly.

The second problem with (3.22) is one of *hygiene*; that is, separation of interpretation concerns according to lexical position in the source code. Because macro expansion produces unevaluated source code, free variables in the macro template will be interpreted *at the source-code position where the macro is called*. The runtime behavior of the macro therefore depends on how its free variables are bound at that point, rather than how they were bound where the macro was defined. In the case of (3.22), one could arbitrarily change its comparison criterion simply by locally binding symbol `>=?`; thus,

```
($let ((>=?  <=?)) ($max 2 3))
```
(3.25)

would evaluate to `2`.

The literature on macros calls this *variable capturing*: variable `>=?` in the macro body is captured by `$let` at the point of call. Relating it to the preceding discussion, though, it is essentially a downward funarg problem, in which the macro is defined at the top level of the program and passed downward into a subexpression where its free

variable `>=?` is locally bound. In fact, traditional macros exhibit either two or four distinct kinds of variable capturing, depending on how one breaks them down (four according to [ClRe91a]), *all* of which are varieties of downward funarg problem.

On the other hand, there is usually no upward funarg problem for macros. In part, this is because the situation that precipitates it (involving a macro defined within a local binding construct) isn't usually allowed to happen. Most macro preprocessing languages have required macro definitions to occur at the top level of the source code, so that the parameters of the macro are the only bound variables in its body — which greatly simplifies the preprocessor algorithm, by allowing it to completely ignore run-time binding constructs.[16] Even if macros were allowed to be nested within local binding constructs, though, the use of substitution to maintain bindings has an intrinsic bias against the upward funarg problem. The situation is thus rather inverted from that of the environment approach to binding maintenance, where the upward funarg problem was the harder of the two.

To see how substitution gives rise to this alternative landscape of hygiene problems, consider the λ-calculus, whose sole reduction schema is

$$(\lambda x.M)N \quad \longrightarrow \quad M[x \leftarrow N]\,. \tag{3.26}$$

The detailed handling of bindings, and therefore hygiene, is contained in the semantic function $M[x \leftarrow N]$, which substitutes term N for variable x in term M. Moreover, the rules of substitution are self-evident when M is a constant, variable, or application,

$$\begin{aligned} c[x \leftarrow N] &= c \\ x_1[x_2 \leftarrow N] &= \begin{cases} N & \text{if } x_1 = x_2 \\ x_1 & \text{otherwise} \end{cases} \\ (M_1 M_2)[x \leftarrow N] &= (M_1[x \leftarrow N]\ M_2[x \leftarrow N])\,, \end{aligned} \tag{3.27}$$

so any plausible hygiene problem must hinge on the one remaining case, substitution into a λ-expression.

An upward funarg in λ-calculus cannot be passed upward, from inside a binding construct $(\lambda x.\,\Box)$ to outside it, without being subjected to a substitution $\Box[x \leftarrow N]$. Therefore, in order for the one remaining case of substitution to permit an upward funarg problem, that case would have to *discard* the binding applied to the upward funarg. For example,

$$(\lambda x_1.M)[x_2 \leftarrow N] \quad = \quad \lambda x_1.M\,. \tag{3.28}$$

The immediate effect of this rule on binding maintenance is substantially identical to that of the self-evaluation rule for Lisp `$lambda`-expressions, (3.5): applicative

[16] In fact, some macro preprocessors for non-Lisp languages (e.g., [KeRi78]) have ignored the syntactic and even lexical structure of the run-time language, with potentially startling (and therefore error-prone) effects. See [ClRe91a].

objects are prevented from retaining binding information. However, upward-funarg substitution Rule (3.28) is not likely to be proposed *by accident*. It was possible, when proposing self-evaluation Schema (3.5), to imagine that things would somehow work out, because one knew that there was an external data structure —the environment— systematically preserving binding information such as what the self-evaluation schema discarded; but here there is no such external data structure, so the discard in the upward-funarg substitution rule really looks like loss of information, and therefore doesn't look like behavior expected of λ-calculus.

As a serious attempt to produce the expected behavior of λ-calculus, one might write

$$(\lambda x_1.M)[x_2 \leftarrow N] \;=\; \begin{cases} (\lambda x_1.M) & \text{if } x_1 = x_2 \\ (\lambda x_1.\, M[x_2 \leftarrow N]) & \text{otherwise}\,. \end{cases} \tag{3.29}$$

Substitution under this rule (together with the other cases in (3.27)) is sometimes called *naive* or *polynomial* substitution. The separate provision here for $x_1 = x_2$ prevents any substitution $M[x \leftarrow N]$ from rewriting bound occurrences of x in M. Also, there is no upward funarg problem, since bindings of free variables are propagated into the body of the λ-expression. However, there is a downward funarg problem.

Recall the downward-funarg example from §3.3.2, (3.20), which renders into λ-calculus as

$$\begin{array}{l} (\lambda \mathrm{y}.((\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3\,. \\ \uparrow \quad \uparrow \quad \uparrow \qquad\qquad\qquad\qquad \uparrow \\ (1) \quad (2) \quad (3) \qquad\qquad\qquad\quad (4) \end{array} \tag{3.30}$$

In reducing this expression, there are three possible choices for what to do first: one could reduce the application of (1), whose argument is 3; (2), whose argument is $(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))$; or (3), whose argument is 5. ((4) is the downward funarg, so isn't yet being applied.) Here are the three alternative reduction steps:

$$\begin{array}{ll} & (\lambda \mathrm{y}.((\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3 \\ \longrightarrow & ((\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y})))[\mathrm{y} \leftarrow 3] \\ = & (\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+3)) \end{array} \tag{3.31(1)}$$

$$\begin{array}{ll} & (\lambda \mathrm{y}.((\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3 \\ \longrightarrow & (\lambda \mathrm{y}.(\ ((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5)[\mathrm{f} \leftarrow (\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))]\ ))3 \\ = & (\lambda \mathrm{y}.(\ ((\lambda \mathrm{y}.(1+((\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))\mathrm{y})))5)\ ))3 \end{array} \tag{3.31(2)}$$

$$\begin{array}{ll} & (\lambda \mathrm{y}.((\lambda \mathrm{f}.((\lambda \mathrm{y}.(1+(\mathrm{f\,y})))5))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3 \\ \longrightarrow & (\lambda \mathrm{y}.((\lambda \mathrm{f}.(\ (1+(\mathrm{f\,y}))[\mathrm{y} \leftarrow 5]\ ))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3 \\ = & (\lambda \mathrm{y}.((\lambda \mathrm{f}.(\ (1+(\mathrm{f}5))\ ))(\lambda \mathrm{x}.(\mathrm{x}+\mathrm{y}))))3\,. \end{array} \tag{3.31(3)}$$

Two out of the three alternatives have no problem. In (3.31(1)), the free variable in the funarg is eliminated before the funarg is passed downward. In (3.31(3)), the local

binding construct that could capture the free variable in the funarg is eliminated before the funarg is passed downward. Only in (3.31(2)) is the variable captured, because the funarg is passed downward while it still has a free variable, and the local binding construct to capture that variable is still in place.

So the λ-calculus as we've defined it, with Schema (3.26) and naive substitution, isn't Church–Rosser: a single term can be reduced to multiple results. If we were really only interested in the calculus, we might restore both Church–Rosser-ness and static scope under naive substitution by restricting the schema so that it only applies when naive substitution would not cause variable capture. This would forcibly perturb the order of reductions just exactly enough to avoid exercising the defective case in naive substitution; in the example, it would prevent application of (2) as long as f remained free in both its argument and its body, which would hold just until after application of either (1) or (3). Unfortunately, it would also undermine the soundness of the calculus as a model of the expected behavior of λ-calculus, by sometimes preventing completion of computations that should complete under call-by-name static scope.[17]

Moreover, our interest is in macros, whose preprocessing requirement perturbs the reduction order in ways that can mandate exercising the defective case in naive substitution. To model this complication, we introduce a "macro" variant of λ into the calculus, λ_m, with corresponding naive substitution rule

$$(\lambda_m\, x_1.M)[x_2 \leftarrow N] \;=\; \begin{cases} (\lambda_m\, x_1.M) & \text{if } x_1 = x_2 \\ (\lambda_m\, x_1.\, M[x_2 \leftarrow N]) & \text{otherwise}\,; \end{cases} \tag{3.33}$$

and distinguish two reduction relations, $\longrightarrow$ that involves only macro-free terms via the usual Schema (3.26), and $\longrightarrow_m$ that does minimal reduction necessary to eliminate λ_m's. The preprocessing requirement is that all $\longrightarrow_m^+$ reduction be completed before $\longrightarrow^+$ reduction can proceed. Significantly, *two* schemata are needed to define $\longrightarrow_m$:

$$(\lambda_m\, x.M)N \;\longrightarrow_m\; M[x \leftarrow N] \tag{3.34}$$

$$(\lambda x.M)(\lambda_m\, y.N) \;\longrightarrow_m\; M[x \leftarrow (\lambda_m\, y.N)]\,. \tag{3.35}$$

The first of these schemata does actual elimination of a macro, by applying it. The second is a necessary prerequisite to later application of a macro that occurs in an operand position (which, in λ-calculus, is how one gives a macro a symbolic name).

[17] Because we allow a λ-expression to ignore its parameter, we expect that an expression containing free variables may nevertheless reduce to an integer, as in $(\lambda \mathrm{x}.5)\mathrm{y} \longrightarrow 5$. Particularly, we expect that a free variable that is ignored will not prevent reduction — but under the restricted schema, a reduction may be prevented by capture of an ignored free variable. This peculiar phenomenon occurs in the (rather contorted) expression

$$((\lambda \mathrm{f}.(\mathrm{f}\,(\lambda \mathrm{y}.(\mathrm{f}\,(\lambda \mathrm{z}.3)))))(\lambda \mathrm{x}.(\mathrm{xy}))\,, \tag{3.32}$$

which under call-by-name static scope ought to reduce to 3, and which actually *does* reduce to 3 via the unrestricted schema with naive substitution, but which is irreducible under the restriction.

The property of a calculus that it does everything we expect it to do is technically called *operational completeness*, and will be treated in Part II, especially Chapter 13.

In the particular example of downward-funarg Expression (3.30), suppose that either λ (2) or (4) is changed to λ_m, while the other λ's are left as-is. Then, of the three possible reduction orders (3.31(1–3)), only (3.31(2)) can be performed during preprocessing — by macro-operator Schema (3.34) if (2) was changed, or macro-operand Schema (3.35) if (4) was changed. That is,

$$\begin{array}{rcl}(\lambda y.((\lambda_m f.((\lambda y.(1+(f y)))5))(\lambda x.(x+y))))3 & & \\ \quad \longrightarrow_m & (\lambda y.(\ ((\lambda y.(1+(f y)))5)[f \leftarrow (\lambda x.(x+y))]\ ))3 & (3.36(2)) \\ \quad = & (\lambda y.(\ ((\lambda y.(1+((\lambda x.(x+y))y)))5)\ ))3 & \end{array}$$

or

$$\begin{array}{rcl}(\lambda y.((\lambda f.((\lambda y.(1+(f y)))5))(\lambda_m x.(x+y))))3 & & \\ \quad \longrightarrow_m & (\lambda y.(\ ((\lambda y.(1+(f y)))5)[f \leftarrow (\lambda_m x.(x+y))]\ ))3 & (3.36(4)) \\ \quad = & (\lambda y.(\ ((\lambda y.(1+((\lambda_m x.(x+y))y)))5)\ ))3\,. & \end{array}$$

So in either case, preprocessing of naive macros has forced variable capturing.

From the macro's point of view, the two cases of variable capturing are capture of a free variable in the body of the macro by a local binding construct at the point of call, (3.36(4)); and capture of a free variable in an operand of the macro by a local binding construct in the body of the macro, (3.36(2)).

The first case, capture of a variable in a macro body, is possible when the macro itself is a downward funarg. In our variant λ-calculus, this happens through the auspices of macro-operand Schema (3.35); a Lisp example is the first definition of macro ***$max*** above, (3.22).

The second case, capture of a variable in a macro operand, is possible when the *operand* to the macro is, in effect, a downward funarg. (The operand to a macro isn't often explicitly presented as a lambda-expression; but it has the essential property of a lambda-expression, deferred evaluation, because macros are preprocessed.) In our macro extension of λ-calculus, this happens through the auspices of macro-operator Schema (3.34). As a Lisp example, here is a binary version of "short-circuit or", which evaluates its operands left-to-right only until a result is true.

```
($define-macro ($or? x y) ->
               ($let ((temp  x))                    (3.37)
                  ($if temp temp y))).
```

This implementation carefully evaluates its first operand just once, storing the result in local variable `temp` for possibly multiple use (using the stored result once in the test, and perhaps a second time in the consequent clause of the ***$if***). However, if it evaluates its second operand, it does so inside the local scope of `temp`, capturing any free occurrence of `temp` in that operand. So

```
($or? foo temp)                                     (3.38)
```

would expand to

```
($let ((temp  foo)) ($if temp temp temp)).
```
(3.39)

For those languages sophisticated enough to support declarations of non-global macros (such as our macro extension of λ-calculus), each of the two cases of variable capturing may be further divided into two subcases, depending on whether the captured variable is bound at preprocessing time or run-time ([ClRe91a]).

In Lisp, traditional macros are actually not defined by templates. Instead, a macro is defined by a general Lisp function, specified via ***$lambda***, to be run at preprocessing time with its operands as parameters, whose result is the expansion of the macro call. For example, the non-hygienic version of ***$max***, (3.22), could be written as

```
($define-macro $max
   ($lambda (a b)
      (list ($quote $if) (list ($quote >=?) a b) a b))).
```
(3.40)

(A shorter form could be achieved using the standard syntactic sugar for quasiquotation, which will be discussed in §7.3.) For purposes of hygiene analysis, though, procedural macro definitions offer nothing new over template macro definitions. A procedural macro still gets all its input information through its operands, and determines only what source expression will replace the calling combination, and under these constraints the procedural macro has no way to induce any other kind of variable capturing than the four kinds already identified for template macros.

The hygiene problems of naive substitution can be eliminated by renaming the parameter of a λ-expression when substituting into it, so as to avoid name collisions:

$$\begin{aligned}(\lambda x_1.M)[x_2 \leftarrow N] \;\; &= \;\; \lambda x_3.((M[x_1 \leftarrow x_3])[x_2 \leftarrow N]),\\ &\qquad \text{where } x_3 \neq x_2 \text{ and } x_3 \text{ doesn't occur free in } M \text{ or } N\,.\end{aligned} \tag{3.41}$$

The same rule works for λ_m in the extended calculus (changing the λ's to λ_m's); and when solutions to the macro hygiene problem began surfacing in the late 1980s,[18] hygienic substitution was one of them ([KoFrFeDu86]).

However, a straightforward implementation of hygienic substitution takes asymptotically longer than naive substitution. Renaming for a single substitution, in its obvious implementation, traverses the entire body of M; and to assure hygienic substitution, renaming must be repeated with each λ-reduction, as new opportunities for naming conflicts arise. It *is* possible to avoid this combinatorial time explosion, at least for pure λ-calculus, by careful use of internally cross-referenced data structures to keep track of the variable occurrences without having to repeatedly search for them ([ShiWa05]); but this technique is a recent development, a decade and a half later. In the late 1980s, the remedy proposed for the time explosion was to switch from macro

[18] The history of hygienic macro technology is summarized in [ClRe91a, §5].

binding maintenance by substitution to macro binding maintenance by environments ([ClRe91a]).

In the environment approach to hygienic macro expansion, the two-phase model of processing induces a segregation between run-time environments, which manage bindings to first-class objects, and preprocessing-time (a.k.a. *syntactic*[19]) environments, which manage bindings to second-class objects. This is in contrast to the substitution approach, where a single substitution function can be used to manage bindings in both phases of processing; but hygienic substitution *can* be orthogonal to the order in which applications are reduced, exactly because it handles just one variable at a time, ignoring what else has or hasn't been done yet. An environment processes all variables in a single term-traversal — which is why it tends to lower time complexity, but which also means that it must contain a complete description of what is being done; so since the two processing phases do different things, they need different environments. A run-time environment contains bindings to run-time values, which in general don't exist at preprocessing time; a preprocessing-time environment contains tables of macro definitions and symbol renamings, which cease to exist before run-time.

One feature that the environment and substitution approaches share is that, in order to maintain hygiene, they must specifically detect run-time binding constructs (λ in our calculus examples). If they didn't do so, they couldn't orchestrate symbol renamings to prevent capture of run-time variables (two out of the four kinds of variable capturing). From this observation it follows that, *in a language with first-class operatives* (which make run-time binding constructs impossible to identify in general before run-time), *preprocessable macros cannot be hygienic.*

3.3.4 First-class macros

Because macros are required to be preprocessable, they can't be first-class objects: they can't be used in any way that would prevent their preprocessing. If macros *could* be both preprocessable and first-class (which they can't), they then wouldn't be hygienic, because preprocessable macros can't be hygienic in the presence of first-class operatives, and the macros themselves would *be* first-class operatives (though they wouldn't be fexprs, since fexprs use explicit evaluation while macros use implicit evaluation).

Alan Bawden has recently proposed a device called *first-class macros* ([Baw00]). His macros are preprocessable and, potentially at least, hygienic. Although they aren't first-class, they do come closer than other forms of Lisp hygienic macros, in that they can be employed in some ways that other hygienic macros can't. These new employment opportunities are enabled by introducing a moderate dose of explicit *static typing* (i.e., type declarations enforced at preprocessing time) into the language.

[19]The term *syntactic environment* is used consistently in [ClRe91a], and appears sporadically (without definition) in the subsequent Scheme reports ([ClRe91b, KeClRe98, Sp+07]).

The particular limitation addressed by Bawden's proposal is that, although an ordinary macro declaration can be nested within a run-time binding construct, the macro cannot then be used outside that run-time construct. That is, a macro can't be an upward funarg from a run-time combination. A simple form of this problem occurs in the macro-extended λ-calculus of §3.3.3: an expression such as

$$((\lambda \mathrm{x}.(\lambda_m \mathrm{y}.(\mathrm{x}+\mathrm{y})))3)4 \tag{3.42}$$

cannot be reduced properly, because the preprocessing reduction step $\longrightarrow_m$ cannot occur until after a run-time reduction step $\longrightarrow$. If the $\longrightarrow$ step *were* applied first, the macro expression $(\lambda_m \mathrm{y}.(\mathrm{x}+\mathrm{y}))$ would be required to absorb run-time binding $[\mathrm{x} \leftarrow 3]$; in terms of the environment approach to binding maintenance, the macro would have to retain its run-time environment rather than merely its syntactic environment.

The Lisp example in [Baw00] concerns a macro

```
($define-macro ($delay expression) ->
               (make-promise ($lambda () expression)))
```
(3.43)

which is meant to be used in conjunction with an applicative ***make-promise*** to implement explicitly lazy expression-evaluations, called *promises*, in Scheme.[20] Bawden points out that there are at least two different ways that a programmer might reasonably want ***make-promise*** to behave,[21] and therefore two different ways that one might want ***$delay*** to behave. Another applicative that uses promises —he describes one called `lazy-map`— could be made to work with both implementations of promises, by taking the appropriate ***make-promise*** as a parameter; but as a matter of good modular encapsulation, one ought to hide ***make-promise*** within a local scope where the promises facility is defined, and require external clients to use ***$delay***. One would therefore have to export ***$delay*** as an upward funarg out of the local scope where the facility is defined, and then pass it into ***lazy-map*** as a parameter.[22]

It isn't possible for ***lazy-map*** to take an *arbitrary* macro as a parameter at run-time, because the actual macro call within ***lazy-map*** has to be expanded during preprocessing. However, if the definition of ***lazy-map*** explicitly specifies that the parameter is a ***$delay*** macro, using a macro definition exactly as in (3.43) but in some unknown run-time environment, it would then be possible to perform the macro expansion at preprocessing time. Bawden describes a static type system to do this. The run-time object actually passed to ***lazy-map*** is a representation of the run-time environment in which to look up symbol `make-promise`; and when a combination within ***lazy-map*** "calls" that run-time object by specifying it in the operator position, the combination disappears during macro expansion; but the static type system

[20] Bawden borrowed this example from the *R5RS*, [KeClRe98, §6.4]. Bawden's macro-definition syntax is different from the *R5RS*'s, and both are different from the notation used here.

[21] For the reader versed in promises, the two behaviors are caching and non-caching.

[22] This is a manifestation of the anti-encapsulatory nature of macros that was noted while discussing the history of abstraction in §1.1.1.

creates an illusion for the programmer that the run-time object is an operative, and that `lazy-map` never interacts directly with applicative `make-promise`.

However, first-class status achieved through static typing is *intrinsically* illusory, because static typing is by nature a restriction on how objects can be used. Sometimes, making the restrictions on a second-class object explicit can mitigate the impact of the restrictions —as with Bawden's device— by enabling the language interpreter to recognize additional situations where the restrictions aren't really being violated; but there must have been some restrictions to begin with, and the type system works by enforcing them. (One would expect a language without type declarations, such as Lisp, to have a propensity for first-class objects, since the programmer is then under no illusions about fundamental restrictions on objects.)

So if one really wants to smooth out the design roughness of second-class operatives (per §1.1.2), type declarations aren't the answer.

3.4 The case for first-class operatives

The Smoothness Conjecture, §1.1.2, suggests that second-class object status limits abstractive power. Therefore, we want our operatives to be first-class.

To make this a credible choice, we need to answer the fundamental objections to first-class operatives that led to their deletion from Lisp. The root of the objections was that, in the presence of first-class operatives, whether or not the operands of a combination are evaluated depends on the result of evaluating the operator — which is a general computation and so may be undecidable. This undecidability is not an unintended side-effect of giving operatives first-class status: it is *part* of giving them first-class status. First-class objects have the right to be results of general, hence potentially undecidable, computations. To vindicate first-class operatives, we need to show that the undecidability can be tolerated — specifically, that it doesn't cripple static program analysis.

Therefore, we seek to maximize the incidence of things that can be decided. Our foremost strategy to that end is *hygiene*, a systematic partitioning of interpretation concerns by location in the source code (in other words, a divide-and-conquer strategy). Tactics for hygiene, and related well-behavedness properties, in the presence of first-class operatives will be discussed in Chapter 5.

That said, a gap still remains to be closed between justifying first-class operatives, and justifying *fepxrs*.

3.4.1 Single-phase macros

As an alternative form of first-class operatives, one could drop the preprocessing requirement from macros, and perform macro expansion when the macro-call combination is evaluated. Call these *single-phase macros*. Since expression processing no

longer takes two passes, there is no longer a need for two different kinds of environments; and the two cases of macro variable-capture aren't further subdivided into four cases by a distinction between capture of syntactic bindings and capture of run-time bindings. Further, the two cases of variable capture can be largely defanged, just by stipulating that the explicit macro-transformation algorithm is interpreted in the macro's static environment, while the expanded code is interpreted in the dynamic environment of the macro call. Variables destined for the static environment are looked up during expansion, so are never in jeopardy of capture by the dynamic environment. Bindings inserted into the expanded code (as in *$or?*, (3.37)) can readily avoid capturing free variables from the operands by using a unique-symbol-generator device to select names for their variables.

To make this concrete, we posit a template-based constructor of single-phase macros called *$macro*, using a similar call syntax to *$lambda*. It takes three operands: a formal parameter tree, a list of symbols we'll call *meta-names*, and a template expression. When the macro is called, a local environment is constructed by extending the static environment of the macro (where *$macro* was called). Symbols in the formal parameter tree are locally bound to the corresponding parts of the operand list of the call to the macro; and meta-names are bound to unique symbols that are newly generated for each call to the macro. A new expression is constructed by replacing each symbol in the template with its value in the local environment; and finally, this new expression is evaluated in the dynamic environment of the call to the macro, producing the result of the call.

Here is a hygienic single-phase version of the binary *$or?* macro, (3.37):

```
($define! $or?
   ($macro (x y) (temp)
      ($let ((temp  x))
         ($if temp temp y)))).
```
(3.44)

Symbols `$let` and `$if` can't be captured by the dynamic environment of the macro call because they are locally looked up and replaced during macro expansion (a hygiene tactic discussed in Chapter 5). The *$let* in the expanded expression can't capture free variables in the operands *x* and *y* because its bound variable *temp* is unique to the particular call to *$or?*.

Alternatively, following the precedent of traditional Lisp macros, one might wish to specify arbitrary macro-expansion algorithms, rather than merely templates. For this purpose, we posit a procedural variant *$macro** of the template-based *$macro* constructor. *$macro** takes just two operands: a formal parameter tree, and an expression called the *body*. As before, when the macro is called, a local environment extends the static environment, and parameters are locally bound to parts of the operand list. Instead of merely looking up symbols, though, the entire body is evaluated in the local environment; and the result of that evaluation is evaluated in the dynamic environment. For binary *$or?*, one would write

```
($define! $or?
   ($macro* (x y)
      ($let ((temp  (gensym)))                                    (3.45)
         (list $let (list (list temp x))
            (list $if temp temp y)))))
```

or, using quasiquotation with standard syntactic sugar,[23]

```
($define! $or?
   ($macro* (x y)
      ($let ((temp  (gensym)))                                    (3.46)
         `(,$let ((,temp  ,x))
             (,$if ,temp ,temp ,y))))).
```

3.4.2 The case for fexprs

Fexprs have an inherent clarity advantage over macros.

In part, this is because a fexpr specifies its entire computation explicitly, whereas a macro specifies only the front (and usually lesser) half of its computation, arranging the back half indirectly. Naturally, an algorithm is easier to understand when it is actually stated.

However, there is also a deeper issue involved, relating to smooth language design. The behavior of fexprs is mostly compatible with that of ordinary Lisp applicatives, with the argument evaluation factored out. The behavior of macros, though, is at an oblique angle to ordinary applicatives. Macros retain the local environment and parameter matching from applicatives, and omit argument evaluation; but they also *add* a second evaluation using the result of the explicitly specified algorithm, and in doing so they fundamentally alter the significance of the body of a combiner: the body of a *$lambda*-expression specifies how to compute a value, but the body of a macro specifies how to derive an algorithm. Corresponding to the latter distinction, variable references can have a new kind of meaning in a macro body, denoting not the determination of a value, but the performance of a computation (which is why macros are vulnerable to accidental redundant operand-evaluation, as noted in §3.3.3).

The particular form of fexprs proposed in this dissertation pivots on the smooth factorization of applicatives into fexprs and argument evaluation.

As a study in the clarity/obscurity of single-phase macros (and also, for the curi-

[23] Under the standard syntactic sugar, a *$quasiquote*d expression is prefixed by backquote (`), and an *$unquote*d subexpression, i.e. one that should be evaluated, is prefixed by comma (,). In a typical procedural macro body, the entire template-expression is quasiquoted, and just a few subexpressions are unquoted; in this case, the whole is quasiquoted and exactly all of the symbols are unquoted.

ous, their expressive power), here is an implementation of *$macro** using *$macro*:[24]

```
($define! $macro*
   ($macro (ptree body) (xform x expanded)
      ($let ((xform  ($lambda ptree body)))
         ($macro x (expanded)
            ($let ((expanded  (apply xform ($quote x))))
               (($macro () () expanded)))))))
```
(3.47)

3.5 Kernel

The purpose of the Kernel programming language is compatible with, but distinct from, the purpose of the dissertation.

The dissertation is concerned with the feasibility of fexprs as a language feature. It claims that fexprs can form the basis for a simple, well-behaved language, and uses Kernel as a practical demonstration. The ulterior motive for studying fexprs is abstractive power, by way of the Smoothness Conjecture (§1.1.2).

The Kernel language project is concerned with the feasibility of pure stylistic design principles as the basis for a practical language. For this approach to succeed, the pure style must be reconciled with practical concerns without compromising either. The project claims that the Kernel language model is a pure style that *can* be reconciled without compromise. The ulterior motive for pursuing purity of style is abstractive power, by way of the Smoothness Conjecture — which is why the objectives of Kernel and the dissertation are compatible.

The design goals of the Kernel project are discussed in detail in [Shu09, §0.1]. The pure style is embodied by a series of (relatively) specific guidelines, refined from a philosophical principle in the Scheme reports:[25]

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Of the seven guidelines enumerated in the Kernel report, five are at least peripherally relevant to the dissertation:

G1 [uniformity] Special cases should be eliminated from as many aspects of the language as the inherent semantics allow.

G1a [object status] All manipulable entities should be first-class objects.

[24] Another curious property of single-phase macros is that they cannot be used to implement quotation if it wasn't already in the language (which is expected for macros, but unexpected for first-class operatives). Further, the derivation of *$macro** from *$macro* uses quotation.

[25] This passage first appeared in the *R3RS*, [ReCl86, p. 2], and has been replicated by each revision since ([ClRe91b, p. 2], [KeClRe98, p. 2]).

G1b [extensibility] Programmer-defined facilities should be able to duplicate all the capabilities and properties of built-in facilities.

G3 [usability] Dangerous computation behaviors (such as hygiene violations), while permitted on general principle, should be difficult to program *by accident*.

G4 [encapsulation] The degree of encapsulation of a type should be at the discretion of its designer.

Guideline *G3*, often abbreviated to *dangerous things should be difficult to do by accident*, was devised as an articulation of how a programming language can remain sane in the absence of a strong type system.[26] The motivation for Guideline *G4* is deeply entangled with *G1a* (promoting smoothness) and *G3* (promoting well-behavedness), with the added observation that the designer of a type will be held responsible for its behavior, and so should have the power to address that responsibility. *G3* and *G4* will bear on the treatment of hygiene in Chapter 5, and of Kernel style in Chapter 7.

The reader should keep in mind that, while the dissertation discusses the rationale for details of Kernel only when it relates to the thesis, very little about the Kernel design is arbitrary. About a third to half of the Kernel report, interleaved with the descriptions of language features, consists of detailed rationale discussion of the features, clarifying how each design choice relates to the guidelines.

[26]While formulated with latently typed languages in mind, this guideline is equally valid for strongly typed languages.

Chapter 4

The factorization of applicatives

> `LAMBDA` definitions should remain trivially separable from argument evaluation information.
>
> — [Pi80, p. 183].

4.0 Introduction

Evaluation of a call to an applicative involves two logically separable parts: evaluation of the operands, and action dependent on the resulting arguments. This is not a language-specific statement; it is a paraphrase of the definition of the term *applicative*, from §2.2.2. If the combiner call is dependent on the operands in any way other than through the values of the arguments, the combiner isn't applicative.

Therefore, an applicative can be factored into two parts: a front end specifying simply that the operands are to be evaluated into arguments, and a back end specifying how to perform a computation dependent on the results from the front end. The key strategem of the dissertation, by which to support fexprs smoothly, is to view the back end of each applicative as a fexpr. Primitive tools are provided for affixing and removing front ends to coordinate argument evaluation (applicatives ***wrap*** and ***unwrap***), leaving the task of constructing back ends to a primitive fexpr-constructor, ***$vau***, using a call syntax similar to ***$lambda***. Constructor ***$lambda*** is then defined as a compound operative —constructed via ***$vau***— that constructs an applicative by composing the two orthogonal constructors ***$vau*** and ***wrap***.

Breaking the classical ***$lambda*** constructor into two parts is a deep change to the Lisp computation model, and cannot be accomplished smoothly as a small localized amendment to the language. ***$lambda*** is almost the entire core of the language; the only other operatives in the minimal semantics of Scheme ([KeClRe98, §7.2]) are ***$if*** and ***$set!*** — and each of the others handles just one task, whereas ***$lambda*** covers *everything else*.[1] In order to make the modified computation model work out cleanly

[1] A quick list of roles covered by ***$lambda*** would include variable binding, compound control

—and especially, to allow the programmer to manage the inherent volatility of first-class operatives— some sweeping global changes were made to the Scheme design. Most of these changes (excepting the "$" on operative names (§2.2.3.1)) are related to hygiene, and will be discussed in Chapter 5.

4.1 Applicatives

The primitive constructor for Kernel's *applicative* data type is ***wrap***; it simply takes any combiner c, and returns an applicative that evaluates its operands and passes the list of results on to c. The following equivalence holds (i.e., evaluating either expression in the same environment would have the same consequences), up to order of argument evaluation:

$$\begin{array}{rl} & (\texttt{(}\textit{wrap}\ \textit{combiner}\texttt{)}\ x_1\ \ldots\ x_n\texttt{)} \\ \equiv & (\textit{eval}\ (\textit{list}\ \textit{combiner}\ x_1\ \ldots\ x_n) \\ & \qquad (\textit{get-current-environment}))\ . \end{array} \tag{4.1}$$

That is, to evaluate a combination with an applicative, build a combination that passes the *arguments* to the underlying combiner, and evaluate the new combination in the current environment.

Equivalence (4.1) would be much less generally valid if it contained symbolic names of applicatives (`wrap`, `eval`, `list`, `get-current-environment`), rather than the applicatives themselves. As the equivalence is stated, operators ***wrap***, ***eval***, ***list***, and ***get-current-environment*** are non-symbol atoms, and will therefore self-evaluate regardless of what, if anything, their standard symbolic names are bound to in the environment where one or the other expression is to be evaluated.

We can now be more specific about our claim, in §2.2.2, that the operative/applicative distinction is orthogonal to lazy/eager argument evaluation. Equivalence (4.1) would be unchanged if applicatives constructed with ***wrap*** used lazy rather than eager argument evaluation, provided that standard applicative ***list*** used lazy argument evaluation too; most other equivalences in this chapter will be similarly invariant under choice of lazy/eager policy.

Note that the argument to ***wrap*** is required only to be a combiner, not necessarily an operative. It is entirely possible in Kernel to wrap an operative, say, twice, in which case the equivalence dictates that the resulting combiner will evaluate its operands, then evaluate the results of those first evaluations, and pass the results of the second evaluations to the operative. This increases the range of free use of applicatives, and therefore (in principle, at least) the smoothness of the language design (§1.1.2).

The standard Lisp applicative ***apply*** implicitly accesses the underlying combiner of its applicative argument. Kernel provides a simpler and more direct (therefore

construction, recursion, and encapsulation.

smoother) primitive for the purpose, ***unwrap***, which extracts the underlying combiner of a given applicative. It affords equivalence

$$(\textit{unwrap}\ (\textit{wrap}\ \textit{combiner})) \equiv \textit{combiner}\,. \tag{4.2}$$

Given ***unwrap***, ***apply*** can be constructed as a compound applicative, and this will be done below in §4.4. (However, the reverse is not quite true: ***unwrap*** can only be imperfectly approximated using ***apply***.[2])

4.2 Operatives

The general form of a ***$vau*** expression is

$$(\textit{\$vau}\ \langle\text{ptree}\rangle\ \langle\text{eparm}\rangle\ \langle\text{x}_1\rangle\ \ldots\ \langle\text{x}_n\rangle)\,. \tag{4.3}$$

Evaluating the expression constructs and returns a compound operative. As noted (of fexprs generally) in §3.1.3, it works almost the same way as classical ***$lambda***:

- ⟨ptree⟩ is the formal parameter tree, a generalization of the formal parameter list used in Scheme.
- When the constructed operative is called, the formal parameters in the tree will be bound to the corresponding parts of the operand tree of the call.
- ⟨eparm⟩ is an additional formal parameter, called the *environment parameter*, that will be bound to the dynamic environment of the call to the constructed operative.

In all other respects (notably static scoping), ***$vau*** works as does Scheme ***$lambda***.

Kernel's smooth treatment of formal parameter trees (a.k.a. definiends) will figure prominently in the Kernel addressing of traditional abstractions in Chapter 7; it facilitates structured data flow in Kernel code, observable in several instances in meta-circular evaluator code in the dissertation (one of which has already occurred, in (2.20) of §2.3.3). In technical detail: A formal parameter tree is either a symbol,

[2]The essential difference is that, where ***unwrap*** gives its client actual access to the underlying combiner, ***apply*** only gives its client the limited power of passing an arbitrary operand tree to that combiner. A straightforward approximation would be

```
($define! unwrap
   ($lambda (appv)
      ($vau object #ignore
         (apply appv object)))) ;
```

but then, (***unwrap*** *combiner*) would always return an operative. If *combiner* were not an applicative, no error would occur until the resulting "unwrapped" operative was called; and "unwrapping" a combiner twice would never produce correct behavior.

nil, special atomic value `#ignore`, or a pair whose car and cdr are formal parameter trees. Duplicate symbols in the tree aren't allowed. When a symbol is matched against a part of the operand tree, it is locally bound to that part. When nil is matched against a part of the operand tree, that part must be nil (otherwise it's a run-time error). When `#ignore` is matched against a part of the operand tree, no action is taken. When a pair is matched against a part of the operand tree, that part must be a pair, and their corresponding elements are matched (car to car, cdr to cdr).

The environment parameter is either a symbol (not duplicated in the formal parameter tree) or `#ignore`, and is matched against the dynamic environment using the same matching algorithm as above.

Use of special atom `#ignore` prevents accidental use of data that was meant to be ignored (*dangerous things should be difficult to do by accident*, §3.5), and is therefore particularly critical for the environment parameter. Ignoring the dynamic environment promotes both hygiene and *proper tail recursion*, on the latter of which see Appendix B. Note that these advantages accumulate particularly by maintaining applicative wrappers separate from their underlying operatives (rather than simply making all combiners operatives, and requiring them to evaluate their own arguments if they want it to happen at all): the vast majority of opportunities for an operative to ignore its dynamic environment only occur because the task of argument evaluation is left to an applicative wrapper.

Here are some simple examples; the behavior of ***$vau*** will be shown in detail for a more sophisticated example in §4.3.1.

```
($define! $quote
   ($vau (x) #ignore x))
```
(4.4)

When this compound operative is called, a local environment is created whose parent environment is the static environment in which the ***$vau*** expression was evaluated (a child environment is an extension of its parent that is distinct for purposes of mutation; environment mutation will be addressed in Chapter 5); then the local environment is populated by matching the formal parameter tree, `(x)`, against the operand tree of the call. In this case, there must be exactly one operand, which is locally bound by symbol `x`. Since the environment parameter is `#ignore`, the dynamic environment is not given a local name. There is one expression in the body of the compound operative, `x`, which is evaluated in the local environment, where it has been bound to the operand of the call, and so the unevaluated operand of the call is returned as the result of the call — just the behavior one expects of the quotation combiner.

$quote will be considered several times in §5.2, in relation to various undesirable behaviors. Gathering, from these and similar instances, that quotation and quasiquotation do not interact well with the explicit-evaluation orientation of the Kernel language design, they have been omitted from the standard tools of the language. (As

just demonstrated for *$quote*, Kernel is easily expressive enough for the programmer to construct these tools if desired (which would probably be a bad idea); however, the syntactic sugar usually used for quotation and quasiquotation would be much more difficult to add, since Kernel is not designed for notational extensions).

```
($define! get-current-environment
   (wrap ($vau () e e)))
```
(4.5)

In this definition, the *wrap* expression constructs an applicative from an underlying compound operative. The operative takes zero operands —that is, the cdr of the calling combination must be nil— so the applicative takes zero arguments.[3] The first instance of `e` is the environment parameter, which is locally bound to the dynamic environment from which the call was made. The second instance of `e` is evaluated in the local environment, where it has just been bound to the dynamic environment; so the dynamic environment is returned as the result of the call. Standard applicative *get-current-environment* was used in Equivalence (4.1), §4.1.

```
($define! list
   (wrap ($vau x #ignore x))
```
(4.6)

Here, the underlying compound operative locally binds `x` to the entire list of operands to the operative — which are themselves the results of evaluating the operands to the enclosing applicative. So the overall effect of the constructed applicative is to evaluate its operands, and return a list of the resulting arguments.

4.3 $lambda

In the *list* example above, (4.6), Lisp programmers will have noted that standard applicative *list* is much easier to construct using *$lambda*; instead of

```
(wrap ($vau x #ignore x))
```
, (4.7)

one could simply write

```
($lambda x x)
```
. (4.8)

$lambda is a standard operative in Kernel, but it isn't *primitive*: it can be constructed from standard primitives, using *$vau*.

[3]One might ask, if there are no operands to be evaluated or not evaluated, why we bother to wrap *get-current-environment*. We prefer to make our combiners applicative unless there's a specific need to make them operative. It's a matter of not crying wolf: anyone reading a program should know to pay special attention when encountering an explicit operative. Operative definitions will stand out more once we introduce *$lambda* in the next subsection (§4.3).

The intended behavior for *$lambda* is expressed by equivalence

$$\begin{array}{ll} & \texttt{(\$lambda}\ \langle\text{ptree}\rangle\ \langle\text{x}_1\rangle\ \ldots\ \langle\text{x}_n\rangle\texttt{)} \\ \equiv & \texttt{(wrap (\$vau}\ \langle\text{ptree}\rangle\ \texttt{\#ignore}\ \langle\text{x}_1\rangle\ \ldots\ \langle\text{x}_n\rangle\texttt{))}\,. \end{array} \tag{4.9}$$

In other words, a *$lambda* expression is just a *wrap*ped *$vau* expression that ignores its dynamic environment. The equivalence can be converted straightforwardly into an implementation of *$lambda* as a compound operative:

```
($define! $lambda
   ($vau (ptree . body) static-env
      (wrap (eval (list* $vau ptree #ignore body)
                  static-env))))
```
(4.10)

We describe how this compound operative *$lambda* works, first in brief, and then in detail through an extended example.

When this *$lambda* is called, the first operand in the call is locally bound by symbol `ptree`, and the list of remaining operands (the cddr of the combination) is locally bound by symbol `body`. The dynamic environment of the call to *$lambda* is locally bound by symbol `static-env`; that environment will become the static environment of the applicative constructed by *$lambda*. The body of *$lambda* is then evaluated in the local environment.

*list** is a standard applicative that works almost like *list*, returning a list of its arguments except that the last argument becomes the rest of the list rather than the last element of the list.[4] In this case it constructs a combination whose operator is operative *$vau* (not symbol `$vau`), whose cadr is the intended parameter tree, whose caddr is `#ignore`, and whose cdddr is the intended body. The constructed combination is then evaluated in the intended environment *static-env*, so that *$vau* makes *static-env* the static environment of the compound operative that it constructs. That compound operative is then wrapped, and the resulting applicative is returned as the result of the call to *$lambda*.

4.3.1 An extended example

To show how this works in detail, we trace through an extended example — defining *$lambda*, using *$lambda* to define an applicative *square*, and using *square* to square a number. For precision, we use reduction schemata.

Structured combiner terms have the forms

$$\begin{array}{l} \langle\text{operative } \textit{ptree } \textit{eparm } \textit{body } \textit{env}\rangle \\ \langle\text{applicative } \textit{combiner}\rangle \end{array} \tag{4.11}$$

[4] *list** isn't standard in Scheme, but it was in Scheme's predecessor MacLisp ([Pi83]), and is in Common Lisp ([Ste90]).

where the first form denotes a compound operative with given parameter tree, environment parameter, body, and static environment; and the second form denotes an applicative with given underlying combiner. Non-object terms have the forms

$$[\text{eval } t_1\ t_2]\,, \tag{4.12}$$

where t_2 should reduce to an environment, and

$$[\text{combine } t_1\ t_2\ t_3]\,, \tag{4.13}$$

where t_1 should reduce to a combiner and t_3 to an environment.

The schemata for combinations are

$$\begin{array}{l}[\text{eval } (\textit{operator}\ .\ \textit{operands})\ e] \\ \longrightarrow\ [\text{combine } [\text{eval } \textit{operator}\ e]\ \textit{operands}\ e\,]\end{array} \tag{4.14}$$

$$\begin{array}{l}[\text{combine } \langle\text{applicative } v_0\rangle\ (v_1\ \ldots\ v_n)\ e] \\ \longrightarrow\ [\text{combine } v_0\ ([\text{eval } v_1\ e]\ \ldots\ [\text{eval } v_n\ e])\ e]\end{array} \tag{4.15}$$

$$\begin{array}{l}[\text{combine } \langle\text{operative } \textit{ptree}\ \textit{eparm}\ v\ e_1\rangle\ \textit{operands}\ e_2] \\ \longrightarrow\ [\text{eval } v\ \ \textit{match}(\textit{ptree}, \textit{operands}) \cdot \textit{match}(\textit{eparm}, e_2) \cdot e_1]\,,\end{array} \tag{4.16}$$

where semantic function *match* implements the matching algorithm, and semantic function $\cdot$ concatenates environments. For this example, we ignore the complicated issue of environment identity, which will be exploited for hygiene in §5.3.2; largely ignore the issue of evaluation order, which must be attended to in the presence of side-effects (§10.2); and assume that the body of a compound operative contains just one expression, again avoiding the issue of evaluation order.

For each primitive standard operative `$foo`, there must also be a rule for reducing [combine `$foo` x e]; we omit these here, as they are self-evident where they occur in the reductions.

As a guide and supplement to the reductions, the entire arrangement of environments and (constructed) combiners through all three phases of the example is illustrated by Figure 4.1. An environment is depicted as a boxed set of bindings, a constructed operative is an oval, and a constructed applicative is a boxed reference to its underlying combiner. Unused bindings in each environment are omitted; in particular, the figure omits the bindings *created* by the definitions — those of `$lambda` in e_0 and `square` in e_1.

Suppose our compound definition for `$lambda` (which we repeat here),

```
($define! $lambda
   ($vau (ptree . body) static-env
      (wrap (eval (list* $vau ptree #ignore body)
                  static-env))))
```
(4.17)

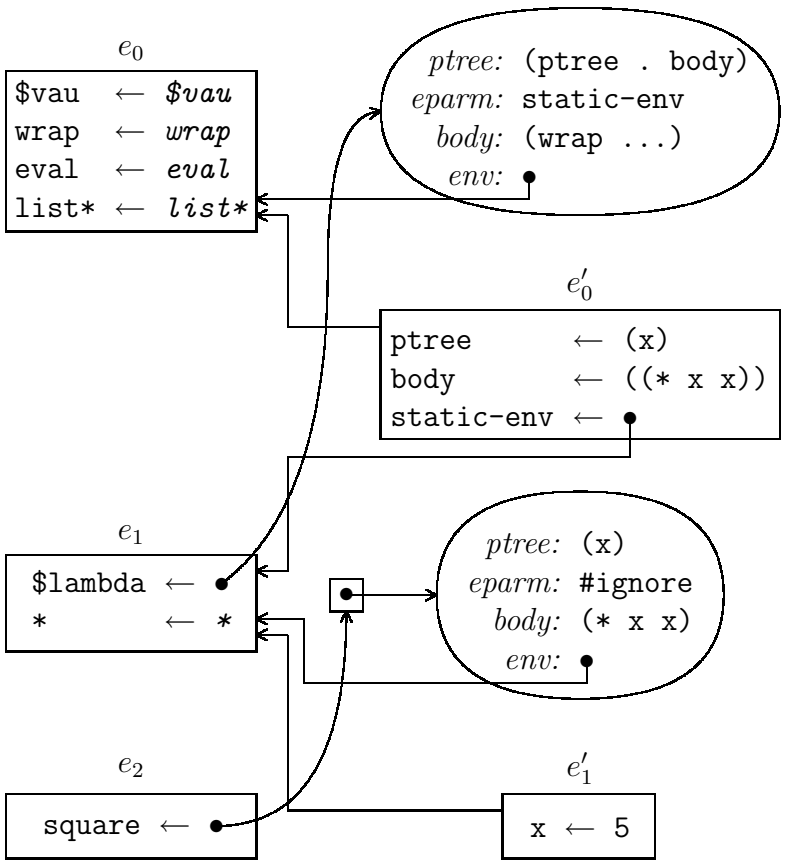


Figure 4.1: Objects of the extended example.

is evaluated in an environment e_0, in which we will assume the default bindings for all the standard combiners used in the definition — ***$vau***, ***wrap***, ***eval***, and ***list****.[5] We have

$$\begin{aligned}
&[\texttt{eval (\$vau (ptree . body) static-env ...)}\ e_0] \\
\longrightarrow\ &[\texttt{combine}\ [\texttt{eval \$vau}\ e_0]\ \texttt{((ptree . body) static-env ...)}\ e_0] \\
\longrightarrow\ &[\texttt{combine}\ \textbf{\textit{\$vau}}\ \texttt{((ptree . body) static-env ...)}\ e_0] \\
\longrightarrow\ &\langle\texttt{operative (ptree . body) static-env ...}\ e_0\rangle\,.
\end{aligned} \tag{4.18}$$

Now, suppose we evaluate another definition,

```
($define! square
   ($lambda (x) (* x x)))
```
, (4.19)

in environment e_1. It doesn't matter to us whether e_1 is related to e_0 (they could even be the same environment), so long as symbol `$lambda` is bound in e_1 to the compound operative ***$lambda*** we just constructed, and symbol `*` to standard combiner *******.

For any standard applicative ***foo***, we'll name its underlying operative ***$foo***; that is, ***foo*** = ⟨applicative ***$foo***⟩.

$$\begin{aligned}
&[\texttt{eval (\$lambda (x) (* x x))}\ e_1] \\
\longrightarrow\ &[\texttt{combine}\ [\texttt{eval \$lambda}\ e_1]\ \texttt{((x) (* x x))}\ e_1] \\
\longrightarrow\ &[\texttt{combine}\ \langle\texttt{operative (ptree . body) static-env (wrap ...)}\ e_0\rangle\ \texttt{((x) (* x x))}\ e_1] \\
\longrightarrow\ &[\texttt{eval (wrap ...)}\ e_0'] \qquad \text{where } e_0' \text{ extends } e_0 \text{ with bindings} \\
&\qquad\qquad \texttt{ptree} \leftarrow \texttt{(x)} \\
&\qquad\qquad \texttt{body} \leftarrow \texttt{((* x x))} \\
&\qquad\qquad \texttt{static-env} \leftarrow e_1 \\
\longrightarrow\ &[\texttt{combine}\ [\texttt{eval wrap}\ e_0']\ \texttt{((eval (...) static-env))}\ e_0'] \\
\longrightarrow\ &[\texttt{combine}\ \textbf{\textit{wrap}}\ \texttt{((eval (...) static-env))}\ e_0'] \\
\longrightarrow\ &[\texttt{combine}\ \textbf{\textit{\$wrap}}\ ([\texttt{eval (eval (...) static-env)}\ e_0'])\ e_0']\,.
\end{aligned} \tag{4.20}$$

From this point, it would be needlessly cumbersome to carry along the continuation [combine ***$wrap*** (□) e_0'] through the entire subsidiary evaluation of `(eval ...)`, so

[5] We don't bother to assume a binding for ***$define!*** since, to avoid tangling with environment mutation, we will only describe the evaluation of the body of each definition — that is, evaluation of the second operand passed to ***$define!***.

we follow the subsidiary evaluation separately.

$$
\begin{array}{ll}
 & [\texttt{eval (eval (list* ...) static-env)}\ e_0'] \\
\longrightarrow & [\texttt{combine}\ [\texttt{eval eval}\ e_0']\ \texttt{((list* ...) static-env)}\ e_0'] \\
\longrightarrow & [\texttt{combine}\ \textit{\texttt{eval}}\ \texttt{((list* ...) static-env)}\ e_0'] \\
\longrightarrow & [\texttt{combine}\ \textit{\texttt{\$eval}}\ ([\texttt{eval (list* ...)}\ e_0']\ [\texttt{eval static-env}\ e_0'])\ e_0'] \\
\vdots & \\
\longrightarrow & [\texttt{combine}\ \textit{\texttt{\$eval}}\ ((\textit{\texttt{\$vau}}\ \texttt{(x) \#ignore (* x x)})\ e_1)\ e_0'] \\
\longrightarrow & [\texttt{eval}\ (\textit{\texttt{\$vau}}\ \texttt{(x) \#ignore (* x x)})\ e_1] \\
\longrightarrow & [\texttt{combine}\ [\texttt{eval}\ \textit{\texttt{\$vau}}\ e_1]\ \texttt{((x) \#ignore (* x x))}\ e_1] \\
\longrightarrow & [\texttt{combine}\ \textit{\texttt{\$vau}}\ \texttt{((x) \#ignore (* x x))}\ e_1] \\
\longrightarrow & \langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle\,.
\end{array}
\tag{4.21}
$$

Although the details of evaluating `(list* ...)` were omitted above, note that, since it was evaluated in e_0', that is also the environment in which symbol `$vau` was looked up, so that the operator later evaluated in e_1 was the self-evaluating object ***$vau***.

Splicing this subsidiary work (4.21) back into the main reduction (4.20),

$$
\begin{array}{ll}
 & [\texttt{eval ($lambda (x) (* x x))}\ e_1] \\
\longrightarrow^+ & [\texttt{combine}\ \textit{\texttt{\$wrap}}\ ([\texttt{eval (eval (...) static-env)}\ e_0'])\ e_0'] \\
\longrightarrow^+ & [\texttt{combine}\ \textit{\texttt{\$wrap}}\ (\langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle)\ e_0'] \\
\longrightarrow & \langle \texttt{applicative} \langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle \rangle\,.
\end{array}
\tag{4.22}
$$

To round out the example, suppose e_2 is an environment where symbol `square` is bound to the applicative we just constructed. (We don't need to assume *any* other bindings in e_2.)

$$
\begin{array}{ll}
 & [\texttt{eval (square 5)}\ e_2] \\
\longrightarrow & [\texttt{combine}\ [\texttt{eval square}\ e_2]\ \texttt{(5)}\ e_2] \\
\longrightarrow & [\texttt{combine}\ \langle \texttt{applicative} \langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle \rangle\ \texttt{(5)}\ e_2] \\
\longrightarrow & [\texttt{combine}\ \langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle\ ([\texttt{eval 5}\ e_2])\ e_2] \\
\longrightarrow & [\texttt{combine}\ \langle \texttt{operative (x) \#ignore (* x x)}\ e_1 \rangle\ \texttt{(5)}\ e_2] \\
\longrightarrow & [\texttt{eval (* x x)}\ e_1'] \qquad \text{where } e_1' \text{ extends } e_1 \text{ with binding } \texttt{x} \leftarrow \texttt{5} \\
\longrightarrow & [\texttt{combine}\ [\texttt{eval *}\ e_1']\ \texttt{(x x)}\ e_1'] \\
\longrightarrow & [\texttt{combine * (x x)}\ e_1'] \\
\longrightarrow & [\texttt{combine}\ \textit{\texttt{\$*}}\ ([\texttt{eval x}\ e_1']\ [\texttt{eval x}\ e_1'])\ e_1'] \\
\longrightarrow^+ & [\texttt{combine}\ \textit{\texttt{\$*}}\ \texttt{(5 5)}\ e_1'] \\
\longrightarrow & \texttt{25}\,.
\end{array}
\tag{4.23}
$$

4.4 apply

Standard applicative ***apply*** is used to replace the usual process of argument evaluation with an arbitrary computation to produce the "argument list" to be passed to the underlying combiner of an applicative. The fixpoint of this replacement —where the usual process is replaced by itself— is expressed by equivalence

$$\begin{array}{ll} & (\textit{apply}\ v_0\ (\textit{list}\ v_1\ \ldots\ v_n)\ (\textit{get-current-environment})) \\ \equiv & (v_0\ v_1\ \ldots\ v_n)\,. \end{array} \tag{4.24}$$

This equivalence is taken to be basic to the purpose of ***apply***; but it only makes sense if v_0 evaluates to an applicative, because if v_0 evaluated to an operative then the $v_{k\geq 1}$ would be evaluated in the first form but not the second. Kernel therefore signals a dynamic type error if the first argument to ***apply*** is not an applicative. (In MacLisp, where generality tended to be pursued for its own sake, it was permissible to ***apply*** a fexpr ([Pi83]). The resulting situation was in keeping with the old reputation of fexprs as aesthetically unpleasant.)

In full generality, the behavior of Kernel's ***apply*** is expressed by equivalence

$$\begin{array}{ll} & (\textit{apply}\ (\textit{wrap}\ \textit{combiner})\ v\ \textit{environment}) \\ \equiv & (\textit{eval}\ (\textit{cons}\ \textit{combiner}\ v)\ \textit{environment})\,. \end{array} \tag{4.25}$$

(Cf. the identity for ***wrap***, (4.1).)

As a matter of free uniform treatment, i.e., smoothness, Kernel pointedly does not require that argument v in the equivalence be a list. Scheme requires a list argument to ***apply***; but in Scheme, all constructed combiners are applicative — and automatic argument evaluation presumes a list of operands. In Kernel, one can explicitly construct operatives that have no inherent commitment to a proper list of operands, such as `($vau x #ignore x)` (which underlies applicative ***list***, (4.6)); so restricting the second argument of ***apply*** to lists would be an arbitrary restriction on underlying operatives. Thus, for example, in Kernel,

$$[\text{eval}\ (\textit{apply}\ \textit{list}\ 2)\ e] \longrightarrow^{+} 2\,. \tag{4.26}$$

The environment argument to ***apply*** does not occur in Scheme because all Scheme applicative calls are independent of their dynamic environment.[6] The environment argument in Kernel is optional; if it's omitted, an empty environment is manufactured for the call. That is,

$$\begin{array}{ll} & (\textit{apply}\ (\textit{wrap}\ c)\ x) \\ \equiv & (\textit{eval}\ (\textit{cons}\ c\ x)\ (\textit{make-environment}))\,. \end{array} \tag{4.27}$$

[6]Scheme supports an extended syntax for ***apply***, taking three or more arguments, in which all arguments after the first are consed into a list as by Kernel's ***list**** applicative. This would seem to defy a natural orthogonality between application and improper list construction; at any rate, in Kernel the ***list**** applicative tends to arise in compound operative constructions where ***apply*** would have to be artificially imposed (such as the construction of ***$lambda*** in (4.10) of §4.3).

Defaulting to an empty environment favors good hygiene (Chapter 5) by requiring the programmer to explicitly specify any dynamic environment dependency in the call, as in the 'fixpoint' equivalence for ***apply***, (4.24).

As with ***$lambda*** in §4.3, the general equivalence for ***apply***, (4.25), translates straightforwardly into a compound operative implementation (where for simplicity of exposition we ignore the optionality of the third argument):

```
($define! apply
   ($lambda (c x e)                                          (4.28)
      (eval (cons (unwrap c) x) e))).
```

Chapter 5

Hygiene

5.0 Introduction

Hygiene is the systematic separation of interpretation concerns between different parts of a program, enabling the programmer (or meta-program) to make deductions about the meaning of a particular program element based on the source-code context in which it occurs.[1] To deduce *all* of its meaning locally would require the uninteresting case of a completely isolated program element; but if too little is locally deducible, reasoning about large software systems becomes intractable. A set of hygiene conditions must therefore be chosen to balance local deduction against non-local interaction, tailored to the overall paradigm of computation (which shapes both deductions and interactions). In a purely applicative setting, such as λ-calculus, the classical hygiene conditions are that

1. the only dependence of the behavior of a combiner on the context from which it is called is through the arguments supplied to the call, and

2. the only dependence of a potentially calling context on a combiner is through the behavior of the combiner when called.

Most programming languages exempt certain imperative features from hygiene, conceding that they do not meet the conditions — usually (for example), a small fixed set of operatives, whose unhygienic behavior is carefully limited and well-understood; and often, mutable variables, which however can introduce module interactions that are disorganized and difficult to track (hence the usual remonstrations against global variables).

[1] [KoFrFeDu86] cites Barendregt for the informal term *hygiene*: H.P. Barendregt, "Introduction to the lambda calculus", *Nieuw Archief voor Wisenkunde* 2 4 (1984), pp. 337–372; it also notes the formal property of being-free-for in [Kle52] (where it occurs in Kleene's §34 as an auxiliary to his definition of *free substitution*).

Macros cannot be subjected to the applicative hygiene conditions, because the macro paradigm of evaluation is two-tiered where the applicative paradigm is one-tiered; but traditional macros are subject to substitutional misbehaviors closely akin to applicative bad hygiene, and, moreover, techniques now exist to reliably correct the misbehaviors (§3.3.3, §6.4). So a modified set of hygiene conditions has been formulated for macros ([ClRe91a, §2]).

Fexprs, though, require a different approach, and this is the subject of the current chapter. Certain blatant violations of hygiene (operand capturing and environment capturing) are intrinsic to the concept of fexprs; so that, in the presence of fexprs, no universally guaranteed hygiene conditions would be strong enough to be useful.[2] This chapter addresses hygiene as a relative, rather than absolute, property, with the applicative hygiene conditions as the ideal. Further, merely observing how far fexprs fail the conditions is inadequate. The influence of fexprs on the language semantics is ubiquitous; in the most general case, one is left unable to deduce anything at all. So, if useful local deductions are to be achieved, one must

1. identify special cases in which fexpr misbehavior is provably bounded; and
2. show that those cases are likely, and especially that the programmer is unlikely to deviate from them *by accident*. (Cf. Kernel design Guideline *G3*, given here in §3.5.)

Hygienic variable bindings were discussed in depth in §3.3. This chapter surveys hygiene problems more broadly, and discusses tactics used in Kernel to foster special cases in which the intrinsic hygiene problems of first-class operatives generally (operand capturing) and fexprs particularly (environment capturing) are manageable.

5.1 Variable capturing

Macros are liable to two kinds of hygiene violations, called *variable capturing*, of which fexprs are (of course) also capable. Phrased generally to cover both macros and fexprs, they are:

1. A symbol in the body of the compound combiner is looked up in an environment that isn't local to the point where the body occurs. (Most common is the environment at the point from which the combiner was called.)
2. A symbol in an operand of the call is looked up in an environment that isn't local to the point where the operands occur. (Most common is the local environment of the body of the compound combiner.)

[2]Fexprs in this dissertation are protected by some absolute guarantees; the guarantees simply don't appear to be *hygiene* in any straightforward, absolute sense. Some consequences of these guarantees for properties of the object language as a whole (contrasting with [Wa98], whose object language has no guarantees analogous to them) will be discussed in Chapter 15.

These were discussed for macros in §3.3.3. For Kernel-style fexprs, though, both problems are unlikely to occur by accident, because Kernel fexprs are statically scoped, so that both all interpretation subtasks that override the local environment of the body, and all interpretation of symbols from the operands, are minimized and (thanks to the explicit-evaluation nature of fexprs) explicitly flagged out.

To illustrate these violations, and Kernel's resistance to them, suppose that Kernel were equipped with a naive template-macro facility, using the syntax described in §3.3.3. (Such a facility could be implemented using *$vau*.[3]) One might then attempt to implement *$lambda* as a macro, by

```
($define-macro! ($lambda ptree . body) ->
                (wrap ($vau ptree #ignore . body))).
```
(5.1)

Naive macro expansion means that, when the macro is called, template `(wrap ($vau ptree #ignore . body))` is transcribed with parameters `ptree` and `body` replaced by the corresponding parts of the operand tree, but symbols `wrap` and `$vau` copied verbatim. The transcription is then evaluated in the dynamic environment of the macro call — so that if symbol `wrap` or `$vau` has a nonstandard binding in the dynamic environment, the call to *$lambda* will not have its intended meaning.

However, when a similar situation arises using *$vau*, in which a new expression is explicitly constructed and then evaluated in the dynamic environment, the tools ordinarily used to construct the new expression are all *applicatives*, such as *list**. Symbols introduced during the construction —in this case, `wrap`, `$vau`, `ptree`, and `body`— are evaluated locally to determine arguments for the applicative constructors; and the ones that aren't parameters —in this case, `wrap` and `$vau`— usually evaluate locally to combiners, which are non-symbol atoms and therefore self-evaluate regardless of any bindings in the dynamic environment. Here, for example, one would write

```
($define! $lambda
   ($vau (ptree . body) env
      (eval (list wrap (list* $vau ptree #ignore body))
            env))),
```
(5.2)

and the expression evaluated in the dynamic environment of the call to *$lambda* would be

```
(wrap ($vau ptree #ignore . body)),
```
(5.3)

[3]No magical introduction of a preprocessing phase is implied; that would require a meta-circular evaluator, and the facility would be implemented in a distinct object-language rather than in the same language where the facility is implemented. Here, the macro definition is processed during evaluation, an operative is constructed that performs the naive transformation on its operands, and that operative is bound in the dynamic environment from which the constructor was called; therefore, a run-time-mutator suffix "`!`" is added to the name of the constructor, `$define-macro!` rather than `$define-macro`.

whose only symbols are any that occur in *ptree* and *body*.

This hygienic outcome is stylistically encouraged in Kernel, by

- omitting quotation and quasiquotation from the standard language, so not to facilitate the construction of expressions that introduce new unevaluated symbols. The programmer could reintroduce general quotation and quasiquotation tools using *$vau* (though better Kernel style would use specialized tools and techniques for specific situations, avoiding the general devices; see §7.3.3); but even after reintroducing general tools, the programmer would not achieve the facility of (quasi)quotation in implicit-evaluation Lisps, because Kernel doesn't provide syntactic sugar for the purpose.

- avoiding use of keywords in call syntax. A prominent example is the default clause at the end of a *$cond*-expression: in Scheme, the default clause uses keyword `else` in stead of a test-expression; but in Kernel, the same effect is accomplished by specifying `#t` as the test-expression of the last clause, which (besides being more uniform) eliminates a possible motive to construct expressions containing unevaluated symbols.

While the compound construction of *$lambda* neatly illustrates Kernel's resistance to capture of local variables by the dynamic environment, it is not otherwise a simple example — because it not only isn't hygienic (under the operative hygiene conditions we've set down), but it isn't *meant* to be. It is a binding construct, that captures occurrences of its parameters (the symbols in its first operand) in its body (any operands after the first). New binding constructs are readily defined in Kernel —as this example shows— but, since Kernel discourages introduction of unevaluated symbols into a constructed expression, the names of the variables bound by the new construct are usually specified explicitly in the operands of the calling expression, decreasing the likelihood that the programmer will be surprised by the binding (hence, decreasing the likelihood of accidents).[4]

Binding constructs will be discussed further in Chapter 7.

Note that the non-hygiene of operative *$lambda* does not compromise the applicative hygiene of λ-calculus only because λ-calculus classifies "λ" as administrative syntax, rather than as an operator. The view of this as an exception to hygiene is easier to see in Lisps, where the treatment of `$lambda` (or `lambda`) as administrative syntax —i.e., as an unevaluated special-form operator— is obviously nonuniform.

The second kind of variable capture, in which a symbol in an operand is looked up in the local environment of the compound combiner, is most likely to occur *by accident* in the use of naive macros when the macro represents a computation for

[4]This might be thought of as a sort of hygiene condition for binding constructs. Cf. Footnote 9 of this chapter; and Appendix B, especially its Footnote 1.

which an intermediate result must be temporarily stored. For example, the binary short-circuit *or* macro

```
($define-macro! ($or? x y) ->
                ($let ((temp  x)) ($if temp temp y)))
```
(5.4)

would capture any free variable `temp` in operand ***y***, only because it stored the result of hygienically evaluating its first operand so that the result could be both test and consequent of the ***$if***-expression in the expansion of the call.

In operatives that require temporary storage of a dynamic-environment evaluation, the Kernel programmer is naturally guided toward storing the result locally, because arranging indirectly for its dynamic storage would require a significant willful effort (which is, again, not something one would do by accident). Here, the indirect approach in Kernel would be

```
($define! $or?
   ($vau (x y) env
      ($let ((temp  ($quote temp)))
         (eval (list $let (list (list temp x))
                          (list $if temp temp y))
               env)))
```
(5.5)

—note the use of non-standard operative ***$quote*** (≡ `($vau (x) #ignore x)`) to embed unevaluated symbol `temp` in the constructed expression— while the direct approach would be

```
($define! $or?
   ($vau (x y) env
      ($let ((temp  (eval x env)))
         ($if temp temp (eval y env)))))
```
(5.6)

which, besides being evidently much simpler and more readable, cannot possibly capture occurrences of `temp` in its operands ***x*** and ***y***, because it binds `temp` in its local environment but evaluates ***x*** and ***y*** in its dynamic environment.

5.2 Context capturing

Not all hygiene violations are readily classified in terms of mis-selection of bindings for variables. A more inclusive approach classifies violations according to the nature of non-argument information accessed by a combiner. We will call violations of this kind *context capturing*. In Kernel, combiners can access three kinds of non-argument contextual information, hence there are three classes of context capturing.

5.2.1 Operand capturing

Any Lisp that supports first-class operatives is subject to *operand capturing*, wherein a combiner that was thought to be applicative may actually be operative and so unexpectedly access its operands. Consider the following minimalist example[5]:

```
($define! call ($lambda (f x) (f x))).
```
(5.7)

Applicative ***call*** is apparently intended to take a function f and an arbitrary object x, and call f with argument x; we expect equivalence

(***call*** f x) $\equiv$ (f x).
(5.8)

This equivalence holds for a tame argument f, as in

(***call*** ***cos*** 0) $\Longrightarrow$ 1
(5.9)

(using "$\Longrightarrow$" for "evaluates to"), but, recalling non-standard operative ***$quote***,

(***call*** ***$quote*** 0) $\Longrightarrow$ x.
(5.10)

This behavior violates the encapsulation of ***call***, by capturing and publicizing an operand, x, that occurs only in the body of ***call***, and that was presumably intended to remain strictly private to that definition.

The misbehavior is, in essence, a poorly managed consequence of a type error. Combiner ***call*** expected an applicative as its first argument, but got an operative instead; the intended non-operativity of the argument is suggested (though not, alas, guaranteed) by the absence of a $ prefix on the parameter name, f.[6]

We could replace the misbehavior with an error report by using ***apply*** to call ***f***:

```
($define! call ($lambda (f . x) (apply f x))).
```
(5.11)

Now (***call*** ***$quote*** 0) explicitly fails, when ***apply*** attempts to unwrap operative ***$quote***. This is a slightly less than pleasing resolution, because it seems to defy our principle that dangerous things be difficult to do by accident; as another partially mitigating measure, it may be appropriate for Kernel compilers to generate a warning message in some problematic situations, such as —for a conservative example— when a parameter without a $ prefix occurs directly as an operator.[7]

[5]Exactly this example (modulo trivialities of Kernel syntax) was used as a criticism of the behavior of first-class operatives in [Baw88].

[6]The type constraint is not only "not guaranteed" in the sense that it isn't enforced, but also in the sense that it might not be intended: the naming convention distinguishes variables that should *always* be operative, but does not distinguish between those meant to be strictly non-operative and those meant to range over both operative and non-operative values.

[7]More interventionist arrangements might be made to prohibit this type of mistake. Some such arrangements may even be rather straightforward to implement, using sophisticated Kernel features such as keyed static variables ([Shu09, §11]) that are otherwise outside the purview of this dissertation; others may be more involved, such as environment guarding that will be described in Footnote 9 of this chapter. However, since even straightforward techniques of this sort seem to involve meta-circular evaluation, it is unclear that their existence bears directly on the thesis of this dissertation, which concerns the existence of a *simple* well-behaved language.

Most any software construct (such as *call*) is apt to behave in unexpected and even bizarre ways when given an input outside its design specs. That said, two aspects of this situation are especially troublesome, and Kernel takes measures to mitigate both.

- There is no way in general to distinguish statically (i.e., prior to evaluation) between operands that must be treated as syntax, and operands that will affect the computation only through the results of evaluating them. This was the key objection to fexprs in [Pi80], because it sabotages meta-programs that manipulate programs as data. Such higher-order programs prominently include compilers, as well as forming an entire genre of custom programs within the Lisp tradition.

  Kernel mitigates this problem by means of environment stabilization tactics, which will be discussed in §5.3.

- The possibility of operand capture leads to the prospect that any vulnerable, non-atomic operand might be *mutated*. Especially messy would be operand mutation within the body of a compound operative, where it could alter the behavior of the operative on subsequent calls.

  Kernel precludes common opportunities for accidental operand mutation by imposing immutability on selected data structures. In particular, when *$vau* constructs a compound operative, it stores an immutable copy of the operand tree to *$vau*; and applicative *load* (analogous to C `#include`), rather than simply reading S-expressions and evaluating them, makes immutable copies of the S-expressions it reads, and evaluates the copies.[8]

5.2.2 Environment capturing

Any Lisp that supports fexprs will also support *environment capturing*, wherein a combiner accesses its dynamic environment; fexprs wouldn't be useful without this feature, since without it they would have no way to induce hygienic argument evaluation. (This is true even of dynamically scoped Lisps, since the local environment still has parameter bindings that could capture variables in the operands.)

Moreover, in Kernel, the basic equivalence relating *wrap* and *eval*, (4.1), implies that applicatives too can capture their dynamic environments.

To continue the above example of (the first, unsafe version of) *call*, (5.7),

$$\texttt{(}\textit{call}\ \texttt{(}\textit{\$vau}\ \texttt{\#ignore e e) 0)} \Longrightarrow \langle\text{an environment}\rangle\,. \tag{5.12}$$

Here, the value returned is actually *the local environment created for the call*, and thus a child of the static environment of combiner *call*. This is especially troublesome because Kernel/Scheme programmers commonly rely, for module encapsulation,

[8] To be precise, Kernel's *$vau* and *load* make immutable copies of the *evaluation structures* of objects; an object's evaluation structure is the part of the object that presumably designates an algorithm when the object is evaluated. For details, see [Shu09, §4.7.2 (`copy-es-immutable`)].

on the inability of clients to directly extract the static environment of an exported applicative (as below in §5.3.2, and later in §7.2).

Two simple measures in Kernel discourage accidental environment capturing.

1. The most convenient way to specify an applicative algorithm, *$lambda*, only constructs combiners that `#ignore` their dynamic environments; the programmer has to use a more circuitous locution to specify an environment-capturing applicative (explicitly composing *wrap* with *$vau*).

2. The names of operatives are conventionally prefixed with "$", so that the programmer will not usually specify an operative without realizing its type. This is evidently more effective in reducing accidental operand capturing than in reducing environment capturing, since even knowing for certain that a combiner is applicative only absolutely precludes operand capturing; but it should also discourage accidental environment capturing, exactly because the behavior of *$lambda* reduces the frequency of environment-capturing applicatives.

While these measures reduce the likelihood of accidents, they do not contribute much to any special cases in which environment capturing provably cannot happen. As with operand capturing, environment stabilization tactics (§5.3) can provide special cases with provable hygiene.

5.2.3 Continuation capturing

Because Kernel, and Scheme, include the standard applicative *call-with-current-continuation*, both languages are subject to *continuation capturing*, wherein a combiner acquires an (encapsulated) object representing all computation that will follow once the capturing combiner returns a result, parameterized by what result will be returned. Continuation capture can be used to implement non-sequential control patterns, analogously to GOTOs in Fortran- and Algol-style languages (in fact, continuations were devised in the 1970s as a mathematical device to treat GOTOs).

Continuation capturing is not a fexpr-related issue (although, historically, it was entangled with fexprs for a time in the treatment of reflective Lisps; see [Baw88]); but it is clearly a violation of applicative hygiene, as *call-with-current-continuation* does not derive its continuation from its argument. Kernel does, in fact, provide a facility to mitigate continuation capturing, by intercepting abnormal transfer of data similarly to the exception-handling facilities of stack-based modern languages such as Java (except that Kernel, not being limited by a sequential control stack, allows interception of abnormal *entrances* as well as abnormal exits; details are in [Shu09, §7 (Continuations)]).[9]

[9]One might ask whether a symmetric facility could be devised to intercept uses of captured environments. The most systemic challenge for such a facility is in choosing interceptors for a given use. For continuation-use, interceptors are determined by the act of capturing a continuation (the

5.3 Stabilizing environments

A source expression has, by definition, an explicit input representation. This means that all of its atoms are either symbols or literal constants; and since there are no literals that denote combiners, all source-expression atomic operators have to be symbols. The upshot is that a Kernel source expression has substantially *no behavioral meaning* independent of the environment where it is evaluated.

This phenomenon differs Kernel from Scheme only in degree. Consider Scheme source expression

```
(define square (lambda (x) (* x x)))
```
. (5.13)

The resulting compound applicative ***square*** ceases to have its original meaning if the binding of symbol `*` in its static environment is later changed. In the general case, a Scheme processor would have to look up symbol `*` every single time ***square*** is called. These repeated lookups could be eliminated, and other optimizations might also become possible, if it could be proven that the relevant binding of `*` is *stable* (i.e., will never change).

The corresponding Kernel source expression,

```
($define! square ($lambda (x) (* x x)))
```
, (5.14)

is potentially even worse off, since the definition itself will misfire unless variables `$define!` and `$lambda` have their standard bindings when the definition is evaluated. If the definition occurs at the top level of the program, at least it will only be evaluated once; but local definitions won't even have that reassurance. It is therefore of great importance in Kernel to cultivate circumstances under which binding stability can be guaranteed.

If an environment is only accessible from a fixed finite set of source code regions, it will usually be straightforward (if tedious) to prove that most of its bindings are stable. The key to stability is therefore to avoid open-ended access to environments. Open-ended access can occur in two ways: *unbounded lexical extent*, and *environment capturing*. Kernel measures to discourage environment capturing were described in §5.2.2; following, §5.3.1 discusses techniques to avoid unbounded lexical extents, and §5.3.2 describes measures to provably bound the possible damage from environment capturing when it occurs.

---

destination of the use), and the act of invoking it (the source of the use); so the determination is all about continuations. For environment-use, though, interceptors are determined by the act of capturing an environment (destination) and the act of capturing an *operand* that will later be evaluated (the "source" being the environment at the moment of capture); so the determination is not only about environments. The needed information might be maintained through a "weak closure" device that would attach the dynamic environment of an operative call to symbols that it captures. The development of such a device seems worthy of investigation (though it might turn out to be intractable, or at least un-smooth in the sense of §1.1.2), but is beyond the scope of the current dissertation.

5.3.1 Isolating environments

The lexical extent of an environment is the set of all source expressions that are evaluated in it without an explicit call to *eval*.[10] Whether it is possible for such an extent to be unbounded depends on how the language processor is arranged; the usual unbounded case is a global environment used by a read-eval-print loop to evaluate the entire (unbounded) sequence of input expressions. If virtually all environments are descended from the global environment, as is commonly the case in Scheme systems, then mutating standard bindings in the global environment will cause most compound combiners to malfunction.

The flexibility with which Kernel handles environments can be brought to bear on this problem, by making it easy to isolate software elements in non-descendants of the global environment.

Isolating software elements from their surrounding environment is facilitated in Kernel by introducing a (derived) variant of *$let*, called `$let-redirect`, in which the parent of the local environment for the body of the construct is not a child of the surrounding environment, but instead is explicitly specified.

Ordinary *$let* obeys equivalence

$$\begin{aligned} &(\textit{\$let}\ ((\langle \mathrm{sym}_1 \rangle\ \langle \mathrm{exp}_1 \rangle)\ \ldots\ (\langle \mathrm{sym}_n \rangle\ \langle \mathrm{exp}_n \rangle))\ .\ \langle \mathrm{body} \rangle) \\ \equiv\ &((\textit{\$lambda}\ (\langle \mathrm{sym}_1 \rangle\ \ldots\ \langle \mathrm{sym}_n \rangle)\ .\ \langle \mathrm{body} \rangle) \\ &\ \langle \mathrm{exp}_1 \rangle\ \ldots\ \langle \mathrm{exp}_n \rangle), \end{aligned} \tag{5.15}$$

and could be implemented by

```
($define! $let
   ($vau (bindings . body) dynamic
      (eval (cons (list* $lambda (map car bindings) body)
                  (map cadr bindings))
            dynamic)))
```
(5.16)

(where *map* applies a given applicative to all the elements of a given list, and returns a list of the results[11]).

The general instability-resistant version, called `$let-redirect`, takes the intended parent environment as its first operand (evaluated in the dynamic environment), fol-

[10] There are only four cases: the body of a compound combiner is in the lexical extent of the local environment created by each call to the combiner; the operator of a combination is in any lexical extent that the combination is in; the operands of an applicative combination are in any lexical extent that the combination is in; and an expression evaluated at the top level of the language processor is in the lexical extent of whatever environment the processor evaluates it in.

[11] Full *map* in Kernel not only allows multiple list arguments, as in Scheme, but also thoroughly supports cyclic lists, on the grounds that cyclic lists wouldn't be fully first-class if the standard tools couldn't handle them; details are in [Shu09, §5.9.1 (`map`)].

lowed by the same operands as ordinary *$let*. The constructed *$lambda* expression is then evaluated in the specified parent, so that

$$\begin{aligned}&(\textit{\$let-redirect}\ \langle\text{env}\rangle\\&\qquad((\langle\text{sym}_1\rangle\ \langle\text{exp}_1\rangle)\ \ldots\ (\langle\text{sym}_n\rangle\ \langle\text{exp}_n\rangle))\\&\qquad.\ \langle\text{body}\rangle)\\\equiv\ &((\textit{\$eval}\ (\textit{\$lambda}\ (\langle\text{sym}_1\rangle\ \ldots\ \langle\text{sym}_n\rangle)\ .\ \langle\text{body}\rangle)\ \langle\text{env}\rangle)\\&\ \langle\text{exp}_1\rangle\ \ldots\ \langle\text{exp}_n\rangle)\,.\end{aligned}\tag{5.17}$$

$let-redirect could be implemented by

```
($define! $let-redirect
   ($vau (static bindings . body) dynamic
      (eval (cons (eval (list* $lambda (map car bindings)
                                       body)
                          (eval static dynamic))
                  (map cadr bindings))
            dynamic))).
```
(5.18)

Kernel supports the construction of interesting environments not descended from the global environment, by means of a primitive *make-kernel-standard-environment* that takes no arguments and, each time it is called, returns a freshly constructed environment in which all the standard Kernel bindings are visible. (This is actually a very inexpensive facility to provide, as will emerge below in §5.3.2.) The combination

```
($let-redirect (make-kernel-standard-environment)
   ((foo  foo)
    (quux  quux))
   ...)
```
(5.19)

would then evaluate the body, "...", in a local environment with all the Kernel standard bindings, and also bindings of symbols `foo` and `quux` to whatever those symbols were bound to in the surrounding environment when the local environment was set up. Once the local environment has been set up, mutations to the outside environment are not directly visible in it.

To facilitate this particular case, Kernel provides a standard operative *$let-safe*, with

$$\begin{aligned}&(\textit{\$let-safe}\ \langle\text{bindings}\rangle\ .\ \langle\text{body}\rangle)\\\equiv\ &(\textit{\$let-redirect}\ (\textit{make-kernel-standard-environment})\\&\qquad\langle\text{bindings}\rangle\ .\ \langle\text{body}\rangle)\,,\end{aligned}\tag{5.20}$$

which could be implemented by[12]

```
($define! $let-safe
   ($vau x dynamic
      (eval (list* $let-redirect
                   (make-kernel-standard-environment)
                   x)
            dynamic))).
```
(5.21)

5.3.2 Restricting environment mutation

Given that environment capturing cannot always be avoided, Kernel curtails its destructive potential by encapsulating the *environment* type such that

1. the programmer cannot determine the ancestors of an arbitrary environment, and

2. an environment can only be mutated if the environment is, or can be, captured.

To illustrate how this works, consider the following Scheme code for an encapsulated counter:

```
(define count
  (let ((counter  0))
    (lambda ()
      (set! counter (+ counter 1))
      counter))).
```
(5.22)

The first time *count* is called it returns 1, the second time 2, and so on. The internal counter can't be accessed except through *count* because it's stored in an environment reachable only through the static-environment reference from *count* — and standard Scheme, like Kernel, provides no general way to extract the static environment of an applicative.[13]

Scheme's *set!* operative, although it seems innocuous in this example, is an indiscriminately overpowered tool in general. Whereas Kernel *$define!* creates or modifies a binding in the immediate dynamic environment, Scheme *set!* finds the visible binding for the specified symbol, and mutates *that binding*, even if the binding is non-local. Consequently, there is no way in Scheme to make a binding visible without also granting the observer the right to mutate it.

[12] *$let-safe* and *make-kernel-standard-environment* are equi-powerful, in that either could be derived from the other. The derivation of the latter from the former would be

```
($define! make-kernel-standard-environment
   ($lambda () ($let-safe () (get-current-environment)))).
```

[13] Some dialects of Scheme don't respect this encapsulation, as MIT/GNU Scheme's *procedure-environment* that extracts the static environment of a compound procedure. ([MitGnuScheme].)

Kernel *$define!* only mutates its immediate dynamic environment; non-local bindings are unaffected (although they might be locally overridden by a local binding for the same symbol). Given an explicit reference to any environment e, one could mutate e by assembling and evaluating a *$define!* combination in e; but since *$define!* is Kernel's only primitive mutator of environments, there is no way to mutate e without an explicit reference to it. So, with no general operation to extract the parent of a given environment, bindings in Kernel can be made visible without granting the right to mutate them.

In addition to primitive *$define!*, Kernel also provides (derived) environment mutators *$set!* and *$provide!* (both of which are equi-powerful with *$define!* in that any two of the three mutators could be derived from the third). As a practical tool, Kernel *$set!* is the most versatile of the three, so is developed here in order to illustrate the restrictions on environment mutation. *$provide!* is specifically suited for programmer-defined modules, and will be developed in §7.1.2.

The following implements operative *$set!*:[14]

```
($define! $set!
   ($vau (e d v) dynamic-env
      (eval (list $define! d                                    (5.23)
                  (list (unwrap eval) v dynamic-env))
            (eval e dynamic-env)))).
```

The key to this implementation is that *$define!* will evaluate its second operand in the target environment determined by evaluating *e* in *dynamic-env*, but we need *v* to be evaluated in *dynamic-env*; so we make the second operand to *$define!* an operative combination, whose operands therefore *won't* be evaluated before calling it. Here we've used the underlying operative of *eval*, which will take its unevaluated first operand (*v*, which is the unevaluated third operand of *$set!*) and evaluate it in the environment that is its second operand (*dynamic-env*, the dynamic environment from which *$set!* was called).[15]

[14]The implementation of *$define!* using *$set!* is simpler than that of *$set!* using *$define!*:

```
($set! (get-current-environment)
       $define!
       ($vau (d v) e (eval (list $set! e d v) e))).
```

[15]Alternatively, we could have evaluated *v* in *dynamic-env* before we constructed the *$define!* expression, and embedded the result in the constructed expression as an operand to non-standard operative *$quote*. For illustrative purposes, here is an implementation using this approach:

```
($define! $set!
   ($vau (e d v) dynamic-env
      (eval (list $define! d
                  (list $quote (eval v dynamic-env)))
            (eval e dynamic-env))).
```

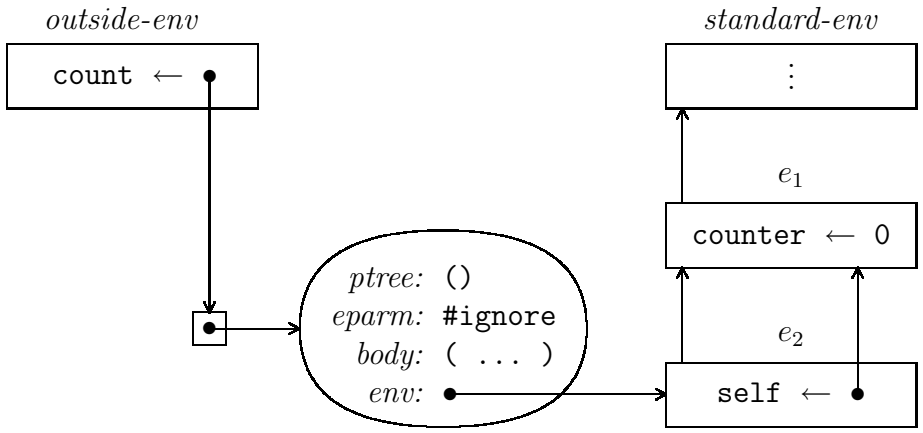


Figure 5.1: Objects in the Kernel ***$let-safe*** / ***$let*** version of ***count***.

Using ***$set!***, here is a translation of ***count*** into Kernel:

```
($define! count
   ($let-safe ((counter  0))
      ($let ((self  (get-current-environment)))
         ($lambda ()
            ($set! self counter (+ counter 1))
            counter)))).
```
(5.24)

The ***$let-safe*** creates a local environment, call it e_1, with binding `counter` ← `0`, whose parent environment exhibits the standard Kernel bindings. The ***$let*** then creates a local environment e_2 whose parent is e_1, with binding `self` ← e_1 in e_2; the latter binding arises because ***$let*** evaluates value expressions for its bindings in the surrounding environment: the surrounding environment is e_1, and the value expression is `(get-current-environment)`, so the value to be bound is e_1. Finally, ***$lambda*** constructs an applicative with static environment e_2. The arrangement of objects is illustrated by Figure 5.1.

Note that the need for additional code in the Kernel version of ***count*** —code for binding and accessing variable `self`— is consistent with the language design principles stated in §3.5. Non-local environment mutation (which is a potentially dangerous activity) must have an explicitly specified target (so the programmer has to do it deliberately); and *permission* for non-local environment mutation (dangerous) must be explicitly supported by providing a name for the environment to be mutated (deliberate).

Lest the above Kernel implementation of ***count*** be taken dogmatically, here is an

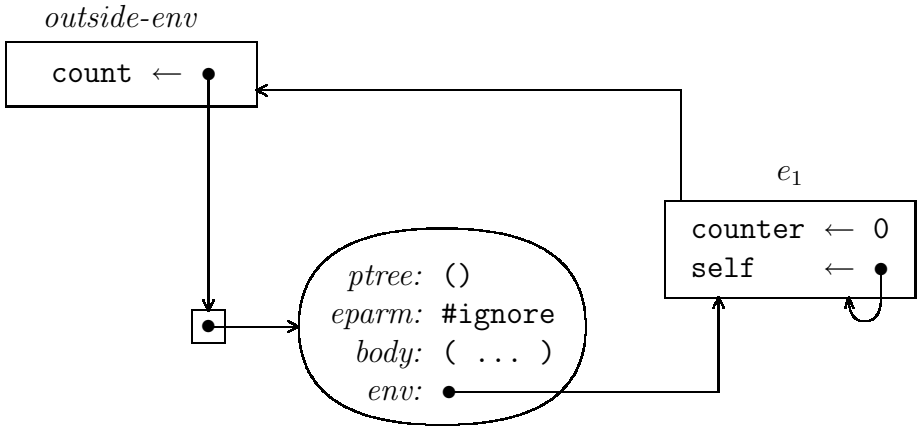


Figure 5.2: Objects in the Kernel *$letrec* version of *count*.

alternative Kernel implementation that more closely parallels the Scheme version.

```
($define! count
   ($letrec ((self     (get-current-environment))
             (counter  0))
      ($lambda ()
         ($set! self counter (+ counter 1))
         counter))).
```
(5.25)

The call to *$let-safe* has been omitted, reducing the nesting depth to that of the Scheme implementation (and, of course, leaving the local environment vulnerable to mutations of the surrounding environment — which may make the earlier version, (5.24), better style if the surrounding environment is vulnerable). The call to *$let* has been replaced with *$letrec*, a variant also available in Scheme (and discussed here in §7.1.1), that evaluates its binding expressions in the constructed local environment rather than in its surrounding parent environment; thus, the local environment is returned by *get-current-environment*, and locally bound by variable `self`. The arrangement of objects is illustrated by Figure 5.2.

The encapsulation of type *environment* affords an inexpensive means for constructing fresh standard environments, as for *make-kernel-standard-environment* (or *$let-safe*). Rather than create fresh copies of all the standard bindings for each standard environment,[16] one can assume the existence of a *ground environment* that is the parent of all standard environments, exhibiting all the standard bindings of

[16]The current draft of the Kernel report, [Shu09], specifies more than a hundred standard bindings, and does not yet elaborate most non-core data types, such as *number*, *character*, *string*, and *port*. Those types should roughly double the number of standard bindings.

the language, but with no means to capture or mutate it; so one can construct fresh standard environments as easily as one would construct the local environment for any compound-combiner call.

There is also another form of environment stability in standard Scheme that Kernel lacks in general (as do some dialects of Scheme): the set of symbols that are bound in a local Scheme environment is fixed at the time the environment is constructed. The Scheme global environment is exempted from this form of stability, so that global definitions can be added over time; but local Scheme ***define***s are required to be placed at the beginning of the local block so that they can be treated as syntactic sugar for ***letrec***s. In contrast, Kernel ***$define!*** can imperatively create a local binding at any time for any symbol.

This facet of Kernel is not necessary to its combiner-handling;[17] it was chosen as a simple and uniform device compatible with strictly local environment mutation (and, therefore, supportive of Kernel's resistance to non-local environment mutation).

The standard Scheme policy toward mutation of local environments is shaped by the Scheme standard's treatment of *environment* as an incidental term for the set of all bindings visible at some point in a program. Bindings are perceived to be the only underlying reality; so it would be unnatural to speak of "adding a binding to the local environment," as if environments had prior existence. Kernel environments, though, are first-class objects, which is a natural consequence of their capturability (key for fexprs) and their encapsulation (key for binding stability); so "adding a binding to a local environment" is an entirely conceivable act. Kernel *could* have restricted local environment mutation to local bindings established at environment construction; but that restriction is superfluous to Kernel's primary binding-stability measure (limiting non-local environment mutation), and contrary to Kernel's removal-of-restrictions design philosophy. The restriction will also emerge, in §7.1.1, as a source of nonuniformity in the semantics of Scheme ***letrec***.

[17] As the Kernel design is currently written, [Shu09], ***$define!*** is an optional feature (along with all other explicit environment mutators — though ***$letrec*** is required).

Chapter 6

The evaluator

6.0 Introduction

This chapter addresses the algorithmic simplicity of Kernel's combiner support, in comparison to that of other approaches to programmer-defined operatives in Scheme-like languages — that is, simplicity of the evaluator algorithm; as opposed to simplicity of use, which is addressed mainly by demonstration throughout Part I. As simplicity is a structural property of the algorithms (rather than a fine-grained mechanical detail), the algorithms are presented here as meta-circular evaluators (rather than as reduction systems; cf. §2.3.3).

For each algorithm, only those parts of a meta-circular evaluator are presented that materially contribute to combiner-handling. Low-level mechanics, such as simple constructors/accessors/mutators for different types of data structures, are reserved to Appendix A. The starting point for the algorithms is the top-level code

```
($define! interpreter
   ($lambda () (rep-loop (make-initial-env))))

($define! rep-loop
   ($lambda (env)
      (display ">>> ")
      (write (mceval (read) env))
      (newline)
      (rep-loop env)))

($define! mceval
   ($lambda (expr env)
      ($cond ((symbol? expr)  (lookup expr env))
             ((pair? expr)  (mceval-combination expr env))
             (#t  expr)))).
```
(6.1)

Applicatives *interpreter* and *rep-loop* are adjusted slightly for preprocessed macros, in §§6.2–6.4, to insert a preprocessing phase between *rep-loop* and *mceval*; and applicative *mceval*, slightly for first-class operatives in §§6.5–6.6, to eagerly evaluate the operator of a combination.

For the illustrative purposes of the chapter, only a few elements of the object language are needed. No attempt is made to handle errors usefully. The only combiners supported are a few arithmetic applicatives and *cons* (convenient for testing the object-languages); operatives *$if* and *$define!* (vital for testing the object-languages); and whatever combiner-handling primitives are appropriate to the particular evaluator, as Scheme *$lambda* and *apply*, or Kernel *$vau wrap* and *unwrap*. Compound combiner bodies are assumed to consist of a single expression (rather than a sequence of expressions to be evaluated left-to-right, which would add nothing to a comparison of combiner-handling strategies).

In meta-circular evaluators for Scheme, it is common to name the central meta-circular evaluation applicative `eval`, and similarly to name its companion applicative `apply`, overriding the standard Scheme bindings for those symbols. The practice is workable in Scheme because Scheme programs almost never use *eval*, and don't often use *apply*.[1] However, this would work quite poorly if the meta-circular evaluator were written in Kernel, because Kernel compound operatives practice explicit evaluation, in which *eval* and *apply* are used routinely; and the meta-circular evaluators are actually written here in Kernel (although Kernel code that exploits first-class operatives is reserved to Appendix A); therefore, several of the meta-circular combiners have "`mc`" in their names.

Meta-circular evaluators are sometimes written with an additional *continuation* argument to *mceval*, that explicitly represents all computation that will follow the evaluation, as a function of the result of the evaluation. Continuations are appropriate for a meta-circular evaluator that explains control flow, where what will be done next is of primary concern; proper tail recursion ([Shu09, §3.10]) and non-linear control flow are usually explained in terms of continuations. A continuation-passing meta-circular Kernel evaluator would be worthwhile for a study of Kernel (cf. §3.5), because Kernel's non-linear control flow support differs from Scheme's ([Shu09, §7 (Continuations)]). However, for this chapter a continuation argument to *mceval* is an unnecessary complication, since continuations have no interesting interactions with ordinary combiners under any of the evaluator variations considered here.

[1] The meta-circular evaluator in the Wizard Book ([AbSu96, Ch. 4]) never uses Scheme *eval* (in fact, standard Scheme didn't yet have *eval* when the second edition of the Wizard Book came out), and uses Scheme *apply* in just one place, within meta-circular *apply-primitive-procedure*. To enable this one use, it saves Scheme *apply* under the name `apply-in-underlying-scheme`.

6.1 Vanilla Scheme

The baseline for comparisons will be the "vanilla Scheme" strategy: a fixed set of operatives designated by special-form operators, without any facilities for programmer-defined operatives (neither macros nor fexprs). Most of the other combiner-handling strategies in the chapter will be embellishments of the vanilla strategy.

The combination dispatching logic for this strategy is

```
($define! mceval-combination
   ($lambda ((operator . operands) env)
      ($cond ((if-operator? operator)
                 (mceval-if operands env))
             ((define-operator? operator)
                 (mceval-define! operands env))
             ((lambda-operator? operator)
                 (mceval-lambda operands env))                (6.2)
             (#t  (mc-apply (mceval operator env)
                            (map-mceval operands env))))))

($define! map-mceval
   ($lambda (operands env)
      (map ($lambda (expr) (mceval expr env))
           operands))).
```

Predicates *if-operator?*, *define-operator?*, and *lambda-operator?* are defined by

```
($define! if-operator?     ($make-tag-predicate $if))
($define! define-operator? ($make-tag-predicate $define!))    (6.3)
($define! lambda-operator? ($make-tag-predicate $lambda)),
```

where *$make-tag-predicate* constructs an applicative predicate to test its argument for equality to a specified symbol. Operative *$make-tag-predicate* is one of several constructed meta-language operatives, scattered through the chapter, part of whose purpose is to avoid explicit quotation. Avoiding quotation is good Kernel programming style, as quotation was repeatedly found to proselytize bad hygiene in Chapter 5; and it also serves to reduce dependence in this chapter on Kernel semantics that differ from Scheme. (The implementation of these compound meta-language operatives is, of course, at the heart of the Kernel/Scheme semantic difference; for example, *$make-tag-predicate* would be a Scheme macro that explicitly quotes its operand, or a Kernel fexpr that simply uses its operand.)

The handling of object-language *$if* is orchestrated by meta-language *$if*:

```
($define! mceval-if
   ($lambda ((test consequent alternative) env)
      ($if (mceval test env)                                    (6.4)
           (mceval consequent env)
           (mceval alternative env)))).
```

To facilitate populating the initial object-language environments constructed by *make-initial-env*, we posit a meta-language operative *$mc-define!*, which works identically to meta-language *$define!* except that, after meta-language-evaluating its second operand, it deposits its bindings in an object-language ground environment (rather than in its meta-language dynamic environment, as *$define!* would). Initial object-language environments are then fresh children of the ground environment:

```
($define! ground-environment (make-mc-environment))

($define! make-initial-env                                      (6.5)
   ($lambda ()
      (make-mc-environment ground-environment))),
```

where *make-mc-environment* constructs an object-language environment with given parent. (Conditions under which this strategy is safe were discussed in §5.3.1.) Like *$make-tag-predicate* (introduced in (6.3)), *$mc-define!* allows the chapter to remain independent of the implicit/explicit-evaluation differences between Scheme and Kernel.

Object-language applicatives are constructed by meta-language *make-applicative*, which converts a meta-language applicative to an object-language applicative; that is, *mc-apply*ing the object-language applicative has the effect of *apply*ing the meta-language applicative from which it was constructed. The population of the object-language ground environment (for this and the next several sections) is then

```
($mc-define! <?    (make-applicative <? ))
($mc-define! <=?   (make-applicative <=?))
($mc-define! =?    (make-applicative =? ))
($mc-define! >=?   (make-applicative >=?))
($mc-define! >?    (make-applicative >? ))
($mc-define! +     (make-applicative +  ))                       (6.6)
($mc-define! -     (make-applicative -  ))
($mc-define! *     (make-applicative *  ))
($mc-define! /     (make-applicative /  ))
($mc-define! cons  (make-applicative cons))
($mc-define! apply (make-applicative mc-apply)).
```

The construction of *mceval-lambda* further assumes a meta-language applicative *match!*, which takes an object-language environment, parameter tree, and operand tree, and binds parameters in the object-language environment accordingly. Then,

```
($define! mceval-lambda
   ($lambda ((ptree body) env)
      (make-applicative
         ($lambda arguments
            ($let ((env  (make-mc-environment env)))
               (match! env ptree arguments)
               (mceval body env))))))).
```
(6.7)

This construction is somewhat atypical of meta-circular evaluators: meta-circular evaluators usually avoid blatantly exploiting the first-class status of applicatives in the meta-language, because they are trying to explain as much of the object-language as possible, and first-class applicatives are a likely subject for explanation. However, in this chapter we are trying to explain as *little* of the object-languages as needed to compare their combiner-handling strategies, so it makes sense to simplify our constructions by using first-class meta-language applicatives to orchestrate object-language combiners.

Given meta-language *match!*, object-language *$define!* is straightforward (assuming Kernel-style compound definiends, rather than Scheme-style; cf. §2.2.3.2):

```
($define! mceval-define!
   ($lambda ((definiend definition) env)
      (match! env definiend (mceval definition env)))).
```
(6.8)

At the end of each section of the chapter, the size of that section's evaluator will be compared to the vanilla evaluation phase based on line count. Since line count is somewhat style-dependent,[2] usually only rough percentages (to the nearest 5%) will be given. In absolute size, the vanilla evaluation phase comprises 49 lines of high-level executable code (the parts shown in this chapter); but 10 of them (20%) are to populate the ground environment with arithmetic applicatives and *cons*, which are conveniences for testing and somewhat arbitrary in number. The comparisons in later sections will exclude those ten lines. Vanilla evaluation also comprises 69 lines of low-level executable code (the parts reserved to the appendix; which, in contrast to the high-level code presented here, heavily exploit features of the Kernel meta-language that differ from Scheme).

[2] A reasonably consistent style is attempted. Lines are counted in the original source files, which are subject to a 79-character margin (though maximizing readability usually prevents lines from getting nearly that long); narrower effective margins in the typeset dissertation force some code to occupy additional lines. Also, the line counts exclude comment lines, blank lines (though one might defensibly include blanks, since their contribution is very similar to that of linebreaks — readability through formatting), and *load* commands (equivalent to C `#include`s).

6.2 Naive template macros

Naive macro definitions occur at the top level of interpretation, i.e., they can't be embedded in any other expression. In this case, they must be entered directly at the `>>>` prompt of ***rep-loop***. The essential elements for a template macro definition are the name for the macro, parameter tree, and template in which the parameters are to be substituted for. The syntax used here (from §3.3.3) is

`($define-macro (`*name* `.` *parameters*`) -> `*template*`)`. (6.9)

Nothing about the behavior of ***mceval*** is changed. The top-level code is tweaked so that, rather than pass the input expression (from `(read)`) directly to ***mceval***, we first run it through an applicative ***preprocess***, thus:

```
($define! interpreter
   ($lambda () (rep-loop (make-initial-macro-env)
                         (make-initial-env))))

($define! rep-loop
   ($lambda (macro-env env)
      (display ">>> ")
      (write (mceval (preprocess (read) macro-env) env))
      (newline)
      (rep-loop macro-env env))).
```
(6.10)

macro-env remembers what macro definitions have been entered. Macro-environments require a different lookup primitive than run-time environments: when a symbol is looked up in a macro environment, failure to find a binding for the symbol *isn't an error*. This is natural to macro preprocessing, which just leaves things alone (rather than complain) if it doesn't know what to do with them. So macro environments are a conceptually distinct data type.

It doesn't much matter, for the initial macro environment, just what ***macro-lookup*** returns when it doesn't find a binding, as long as it allows the failure to be handled easily. However, it will matter for later uses of macro environments, and different —even individually customized— behaviors will eventually be desired (the latter for hygienic macros in §6.4), so we specify the unbound-symbol behavior flexibly by a meta-language applicative argument to ***make-empty-macro-env***; when a symbol lookup fails, the symbol is passed to this applicative argument to determine the result. Here (arbitrarily for now, but compatibly with the hygienic-macro treatment to come),

```
($define! make-initial-macro-env
   ($lambda ()
      (make-empty-macro-env ($lambda (x) x)))).
```
(6.11)

preprocess simply checks for a macro definition, and either handles the definition, or scans through the source expression looking for macro calls to transform.

```
($define! preprocess
   ($lambda (expr macro-env)
      ($if ($and? (pair? expr)
                  (define-macro-operator? (car expr)))
           (preprocess-define-macro! (cdr expr) macro-env)
           (expand expr macro-env))))

($define! define-macro-operator?
          ($make-tag-predicate $define-macro)).
```
(6.12)

(Meta-language operative ***$and?*** is a "short-circuit" version of logical *and*, that doesn't bother to evaluate its second operand if its first operand evaluated to false.)

expand scans through a source expression, performing macro transformations.

```
($define! expand
   ($lambda (expr macro-env)
      ($if (pair? expr)
           (check-for-macro-call
              (map ($lambda (expr) (expand expr macro-env))
                   expr)
              macro-env)
           expr)))

($define! check-for-macro-call
   ($lambda (expr macro-env)
      ($if (symbol? (car expr))
           ($let ((x  (macro-lookup (car expr) macro-env)))
              ($if (symbol? x)
                   expr
                   (expand (apply-macro x (cdr expr))
                           macro-env)))
           expr))).
```
(6.13)

(This implementation actually supports a sort of second-class "upward funarg" macro, in that, since the operator of a combination is ***expand***ed before being checked to see whether it names a macro, it would be possible for a macro-calling operator to expand to the name of a macro and thereby cause that macro to be called.)

preprocess-define-macro! binds the given macro name to a meta-language applicative that performs the appropriate template-substitution transformation on its arguments. (There is no difficulty in binding the macro name to a non-object-

language object, since the macro environment will never be directly accessible from the object language.)

```
($define! preprocess-define-macro!
   ($lambda (((name . parameters) #ignore template)
             macro-env)
      (macro-match! macro-env name
         (make-macro parameters template))))

($define! make-macro
   ($lambda (parameters template)
      ($lambda operands
         ($let ((macro-env  (make-empty-macro-env
                               ($lambda (x) x))))
            (macro-match! macro-env parameters operands)
            (transcribe template macro-env)))))
```

(6.14)

```
($define! transcribe
   ($lambda (template macro-env)
      ($cond ((symbol? template)
                  (macro-lookup template macro-env))
             ((pair? template)
                  (cons (transcribe (car template)
                                    macro-env)
                        (transcribe (cdr template)
                                    macro-env)))
             (#t  template))))

($define! apply-macro apply).
```

The `#ignore` parameter to *process-macro-define!* skips over the "`->`" in the definition syntax. A properly paranoid implementation of *transcribe* would make a structural copy of each result returned by *macro-lookup*, so that during evaluation (by *mceval*), if one occurrence of a macro operand were mutated, other occurrences of the same operand would be unaffected.

Note the resemblance between *make-macro* here, and *mceval-lambda* in (6.7).

The naive template-based preprocessing phase takes about 120% as many lines of high-level executable code as the vanilla evaluation phase. Low-level support code for preprocessing is about 60% as long as for evaluation, because evaluation support involves two object-language data types (environments, applicatives), while preprocessing support involves just one (macro-environments).

6.3 Naive procedural macros

To do full justice to the preprocessing constraint on procedural macros, one would need to prohibit macro expansions from containing objects that can't be specified directly through source-code — such as first-class applicatives. We approximate procedural macros here; but within the framework of our interpreter the prohibition is artificial, and we do not enforce it. (Recall that embedded first-class combiners were one of the hygiene tactics discussed in §5.1.)

The procedural macro definition syntax assumed here is (borrowing again from §3.3.3)

```
($define-macro name expr) ,
```
(6.15)

where *expr* must evaluate (in the object-language) to an applicative.[3]

The change itself is almost entirely encapsulated within ***preprocess-define-macro!***:

```
($define! preprocess-define-macro!
   ($lambda ((name definition) macro-env)
      (macro-match! macro-env name
         (mceval definition (make-initial-env)))))

($define! apply-macro mc-apply) .
```
(6.16)

There is no ***make-macro***, since the argument to ***apply-macro*** is now an object-language applicative supplied by ***mceval***; and no ***transcribe***, since expansion is now the responsibility of the procedural macros themselves.

However, in order to make a procedural macro facility useful, two other (implementationally orthogonal) supporting object-language features are usually included with it.

One of these is *quasiquotation*. Because the output of a macro cannot (usually) contain embedded combiners, some means is needed for the macro to build structures that intersperse unevaluated symbols with computed subexpressions (the unevaluated symbols being used to designate the evaluation-time combiners that can't be directly embedded in the expansion). For template-based macros, any symbol in the template that isn't a parameter name is copied into the expansion; but for procedural macros, unevaluated symbols must be produced as a result of computation. Therefore quotation is needed, at a minimum, and quasiquotation greatly simplifies

[3]Some Lisp meta-circular Lisp evaluators take advantage of the similarity between the object- and meta-languages by evaluating procedural macros in the meta-language. However, one weakness of the (otherwise quite versatile) practice of writing Lisp evaluators in Lisp is that one is vulnerable to confusion between the object- and meta-languages; and evaluating macros in the meta-language greatly exacerbates the problem. It would therefore seem a poor choice for a meta-circular evaluator, whose *raison d'être* is to aid understanding.

most tasks. Of all the approaches to operative construction considered in this chapter (or this dissertation), procedural macros are the *only* one that specifically motivates quasiquotation.

There is no need to provide quotation here because, having failed to enforce the prohibition against embedding combiners in a macro expansion, we don't need to embed their unevaluated names in the expansion.

The second feature usually included with procedural macros is *gensyms*, generated symbols that are guaranteed to be absolutely unique — that is, each time the symbol generator is called, it produces a symbol that cannot possibly occur in any other way. The symbol generator is traditionally called `gensym`. Symbols generated during a macro call cannot possibly occur free in its operands; and they therefore prevent one of the two kinds of variable capturing, *if* the macro uses them properly. The technique was illustrated for a macro `$or?` in §3.4.1.

`gensym` will be essential for both hygienic and single-phase macros (§§6.4–6.5), so its implementation is relevant. The meta-language might or might not support gensyms (standard Scheme and standard Kernel don't); and if it doesn't, then, to guarantee uniqueness, object-language gensyms will have to be disjoint from the meta-language *symbol* type. To keep the chapter independent of this detail (reserving it to the appendix), we assume meta-language applicatives `gensym` (which might or might not be standard), and `mc-symbol?` (which might or might not do the same thing as meta-language `symbol?`). The only changes to high-level code are the replacement of `symbol?` by `mc-symbol?` in `mceval`, and exportation of `gensym` to the object-language:

```
($define! mceval
   ($lambda (expr env)
      ($cond ((mc-symbol? expr)  (lookup expr env))
             ((pair? expr)  (mceval-combination expr env))
             (#t expr))))

($mc-define! gensym (make-applicative gensym)).
```
(6.17)

There are a few other uses of `symbol?` in the high-level code, but they all test either for macro names or for macro parameter names; and since at this point we don't allow nested macro definitions, neither of those could possibly be a gensym.

(The use of `mc-symbol?` in `mceval` will continue through the remaining sections of the chapter —it could have been done earlier if there'd been any apparent reason— but the `mc-define!` of `gensym` is unique to the procedural macro algorithm of this section. Later sections that use `gensym` will use it internally, with no reason to export it to the object language.)

The change from template to procedural macros has actually produced a net savings of about 30% of the high-level preprocessing code (85% rather than 120% of vanilla evaluation phase). On the other hand, introducing quasiquotation would

more than make up the difference. Also, low-level code to implement gensyms adds, at it happens, exactly one more line than was saved in high-level preprocessing.

6.4 Hygienic macros

Hygienic macro transformations are template-based (as §6.2), rather than procedural (§6.3). This section shows how template-based macro transformations are implemented hygienically.

Although a single template-based macro transformation is computationally trivial, Scheme's hygienic macro facility as a whole is Turing-powerful. Rather than selecting a macro transformation based solely on the operator symbol of the combination (the traditional selection strategy of Lisp), Scheme uses the operator symbol to select a table of patterns, then matches the patterns against the entire structure of the calling combination to select a transformation. The individual pattern/transformation pairs are analogous to the syntax productions of a Chomsky type 0 grammar — hence, Turing power.

The introduction of a second computational model into macro expansion offers no interesting characteristics for the current discussion, as it is a substantial step away from uniformity (hence, smoothness) and is largely orthogonal to binding maintenance. (Re orthogonality to binding maintenance, note that the same tactic could be applied as readily to the naive template-macros of §6.2.) Therefore, this section doesn't implement pattern-based invocation for its transformations.

For consistency, `$define-macro` will use the same syntax as for naive template-based macros in §6.2:

(`$define-macro` (*name* . *parameters*) `->` *template*) . (6.18)

However, hygienic macro definitions don't have to be global — they can be nested, using a `$let`-like syntax. Moreover, the nestable syntax is fundamental to demonstrating hygiene. Just as a hygienic applicative remembers the environment at its construction, so a hygienic macro remembers the macro-environment at its construction; but if hygienic macros could only be defined globally, they would all be remembering the same macro-environment, and there would be no way to exercise the inter-macro aspects of the hygiene. (Recall from §3.3.3 that inter-macro interference accounts for two out of four cases of variable capturing.)

The syntax used here for local macro declarations is

(`$let-macro` ((*name* . *parameters*) `->` *template*) *expr*) , (6.19)

where the declared macro is only visible in a local macro-environment used to preprocess *expr*, and the macro's static macro-environment is the macro-environment provided for the preprocessing of the `$let-macro`-expression.

The only high-level preprocessing code retained here from the naive treatments (§§6.2–6.3) is *preprocess* itself, (6.12).

The constructor for hygienic macros, *make-macro*, takes three arguments, parallel to vanilla *mceval-lambda* in (6.7): parameter tree, body, and macro-environment. *make-macro* is also the most complicated (or *subtle*, as [ClRe91a] puts it) element of the hygiene maintenance, so will be explained last in the section, when the reader no longer has any other unknowns to worry about. Meanwhile, knowing the call format for *make-macro*, *preprocess-define-macro!* is simply

```
($define! preprocess-define-macro!
   ($lambda (((name . parameters) #ignore template)
             macro-env)
      (macro-match! macro-env name
         (make-macro parameters template macro-env)))).
```
(6.20)

Macro-environments will map each symbol to one of four kinds of values:

- a symbol, indicating that the looked-up symbol is renamed.
- a macro.
- *$lambda*, the second-class combiner initially designated by operator `$lambda`.
- *$let-macro*, the second-class combiner initially designated by operator `$let-macro`.

Explicit bindings to *$lambda* and *$let-macro* are needed because, in the process of symbol-renaming to maintain hygiene, the algorithm might temporarily rename symbols `$lambda` and `$let-macro`.

To streamline the handling of these cases in *expand*, macros and *$lambda* and *$let-macro* are all given a single representation: a meta-language applicative of two arguments, that takes its operand-tree and macro-environment, and performs *all further preprocessing* of the calling combination. Thus,

```
($define! macro? applicative?)

($define! apply-macro
   ($lambda (macro operands macro-env)
      (macro operands macro-env))).
```
(6.21)

It then suffices for *expand* to handle variable renaming and (what it believes to be)

macro calls:

```
($define! expand
   ($lambda (expr macro-env)
      ($cond ((pair? expr)
                 ($let ((x  (macro-lookup (car expr)
                                          macro-env)))
                    ($if (macro? x)
                         (apply-macro
                            x (cdr expr) macro-env)
                         (map ($lambda (x)
                                 (expand x macro-env))
                              expr))))
             ((mc-symbol? expr)
                 (macro-lookup expr macro-env))
             (#t  expr)))).
```
(6.22)

(This implementation doesn't support expanding a compound operator to `$lambda` or `$let-macro`, so there are no "second-class upward funargs" as in §6.2. There, the feature arose naturally, even somewhat simplifying the implementation of ***expand***; whereas here, it would complicate things.[4])

Bindings to ***$lambda*** and ***$let-macro*** are placed in a ground macro-environment by means of an operative ***$mc-macro-define!***, analogously to the population of the evaluation-time ground environment via ***$mc-define!*** (in §6.1, surrounding (6.5)):

```
($define! ground-macro-env (make-empty-macro-env
                              ($lambda (x) x)))

($define! make-initial-macro-env
   ($lambda ()
      (make-child-macro-env ground-macro-env))).
```
(6.23)

Meta-language applicatives ***$lambda*** and ***$let-macro*** will each return a pair whose car is an unevaluated symbol. General quotation is avoided, in this case, by means of a meta-language operative ***$make-tag-prefixer***, which takes one operand, ⟨prefix⟩, and returns an applicative of one argument that ***cons***es ⟨prefix⟩ with its argument. `(($make-tag-prefixer foo) (+ 2 3))` would evaluate to `(foo . 5)`. Then,

```
($define! make-lambda    ($make-tag-prefixer $lambda))
($define! make-let-macro ($make-tag-prefixer $let-macro)).
```
(6.24)

[4] [ClRe91a, §2] mentions that the authors had implemented variants of their hygienic algorithm with and without expansion of compound operators into syntactic keywords.

$let-macro is straightforward: construct the new macro using *make-macro*, bind it in a child of the surrounding macro-environment, and preprocess the body of the *$let-macro*-expression in the child macro-environment:

```
($mc-macro-define! $let-macro
   ($lambda ((((name . ptree) #ignore template) body)
             macro-env)
      ($let ((macro  (make-macro ptree template macro-env))
             (macro-env  (make-child-macro-env macro-env)))
         (macro-match! macro-env name macro)
         (expand body macro-env)))).
```
(6.25)

$lambda is more involved. It renames all the parameters of the expression; but to do so, it must recursively traverse the parameter tree (its first operand) to find each parameter name, generate a gensym to rename it to, and register the renaming in a local child of the surrounding macro-environment. Then it *expand*s the body of the expression (its second operand) in the local macro-environment, so that the renamings performed on the parameter tree will be imposed on free variables in the body as well. Thus:

```
($mc-macro-define! $lambda
   ($lambda ((ptree body) macro-env)
      ($let ((macro-env  (make-child-macro-env macro-env)))
         ($let ((ptree  (rename-ptree! ptree macro-env)))
            (make-lambda ptree (expand body macro-env))))))

($define! rename-ptree!
   ($lambda (ptree macro-env)
      ($cond ((mc-symbol? ptree)
                  ($let ((gs  (gensym)))
                     (macro-match! macro-env ptree gs)
                     gs))
             ((pair? ptree)
                  (cons (rename-ptree!
                           (car ptree) macro-env)
                        (rename-ptree!
                           (cdr ptree) macro-env)))
             (#t  ptree)))).
```
(6.26)

Finally, *make-macro* takes the parameter tree, template, and static macro-environment for a macro; and constructs an applicative that uses all those elements, an operand-tree, and the dynamic macro-environment from which the macro is called, to perform expansion of a call to the macro. Expansion of the call is a two-phase process: first, *transcribe* the template of the macro with appropriate substitutions

(of operands for parameters) and renamings (of non-parameter symbols in the template), and second, *expand* the transcribed expression in a local child of the dynamic macro-environment.

Transcription uses a special "substitution" macro-environment, constructed via *make-empty-macro-env* and then populated with bindings of parameters to operands (reminiscent of the use of a macro-environment in the naive template *make-macro*, in (6.14)). The subtle heart of the hygienic algorithm is in what happens when a symbol in the template is found to be *unbound* in the substitution macro-environment. The unbound symbol is renamed to a gensym; the renaming is registered in the substitution macro-environment, so that other occurrences of the same symbol in the template will be renamed to the same gensym; and a binding is registered in the *local* macro-environment from the gensym back to whatever value the original symbol was mapped to in the *static macro-environment of the macro*.

If a non-parameter symbol ⟨symbol⟩ occurs free in the template, it won't be captured by the dynamic macro-environment because it has been temporarily renamed to a gensym that, being unique, cannot possibly be bound by the dynamic macro-environment; and after expansion of the transcription it will revert to whatever it was in the static macro-environment, regardless of whether ⟨symbol⟩ had a different binding in the dynamic macro-environment. If ⟨symbol⟩ occurs *bound* in the template (via *$lambda*), its bound occurrences in the template will be renamed again during expansion of the transcription, to still another gensym that won't capture any symbols in the operands — and, in particular, won't capture ⟨symbol⟩ if it occurs in the operands.

Here is hygienic *make-macro*:

```
($define! make-macro
   ($lambda (parameters template static-menv)
      ($lambda (operands dynamic-menv)

         ($define! new-menv
                   (make-child-macro-env dynamic-menv))

         ($define! subst-menv
            (make-empty-macro-env
               ($lambda (s)
                  ($let ((gs  (gensym)))
                     (macro-match! subst-menv s gs)
                     (macro-match! new-menv
                        gs (macro-lookup s static-menv))
                     gs))))

         (macro-match! subst-menv parameters operands)
         ($let ((expr  (transcribe template subst-menv)))
            (expand expr new-menv)))))

($define! transcribe
   ($lambda (template subst-menv)
      ($cond ((mc-symbol? template)
                 (macro-lookup template subst-menv))
             ((pair? template)
                 (cons (transcribe (car template)
                                   subst-menv)
                       (transcribe (cdr template)
                                   subst-menv)))
             (#t  template))))
```

(6.27)

The high-level preprocessing code for hygienic macros, as presented here *sans* pattern-matching, is about 65% longer than for the naive template algorithm (195% versus 120% of vanilla evaluation phase). The low-level preprocessing code is about 15% longer (70% versus 60% of vanilla evaluation phase).

6.5 Single-phase macros

Single-phase macros were described in §3.4.1. Although their indirection and lack of commonality with applicatives (the latter being a sort of non-smoothness) are

undesirable, they are an interesting case for a study of hygiene, because if no other means is provided in the language for producing symbols as the result of computation —such as quotation[5]— there seems to be no way for single-phase macros to cause variable capture.

The object-language constructor of single-phase macros is an object-language operative *$macro*, whose call syntax is

(*$macro* ⟨parameters⟩ ⟨meta-names⟩ ⟨template⟩) . (6.28)

⟨parameters⟩ is the usual parameter tree, ⟨meta-names⟩ is a list of symbols, and ⟨template⟩ is an arbitrary expression. When the macro is called, a local child of its *static* environment is created; local bindings are made of parameters to operands, and of meta-names to freshly created gensyms; the template is transcribed, replacing every symbol in the template by the value it is bound to in the local environment; and the transcribed expression is then evaluated in the *dynamic* environment of the macro call.

Here, operatives are first-class objects, so the operator of a combination is always evaluated. This could be accomplished within *mceval-combination*, but it will be convenient to introduce the operator evaluation directly into *mceval*; to maintain clarity in comparing different evaluators, the applicative that receives the combiner, operand tree, and dynamic environment is then called `combine` rather than `mceval-combination`. The modified *mceval* is

```
($define! mceval
   ($lambda (expr env)
      ($cond ((mc-symbol? expr)  (lookup expr env))
             ((pair? expr)
                 (combine (mceval (car expr) env)
                          (cdr expr)
                          env))
             (#t expr)))).
```
(6.29)

combine distinguishes between object-language operatives and object-language applicatives, and invokes the appropriate low-level tool for either:

```
($define! combine
   ($lambda (combiner operands env)
      ($if (mc-operative? combiner)
           (mc-operate combiner operands env)
           (mc-apply combiner
                     (map-mceval operands env))))).
```
(6.30)

[5]This constraint would rule out more than just operand capturing. For example, the standard Scheme applicative *string->symbol* would violate it, as would *gensym*.

The meta-language constructor for object-language operatives converts a meta-language applicative of two arguments —operand-tree and dynamic environment— into an object-language operative with that behavior. Thus,

`(mc-operate (make-operative` *meta-appv*`) operands env)` (6.31)

would be equivalent to

(*meta-appv* `operands env`). (6.32)

The ***mc-define!***s for ***$if***, ***$define!***, and ***$lambda*** then use exactly the same meta-language ***$lambda***-expressions as for vanilla Scheme (from (6.4), (6.7), and (6.8)):

```
($mc-define! $if
   (make-operative
      ($lambda ((test consequent alternative) env)
         ($if (mceval test env)
              (mceval consequent env)
              (mceval alternative env)))))

($mc-define! $define!
   (make-operative
      ($lambda ((definiend definition) env)
         (match! env definiend (mceval definition env)))))

($mc-define! $lambda
   (make-operative
      ($lambda ((ptree body) env)
         (make-applicative
            ($lambda arguments
               ($let ((env  (make-mc-environment env)))
                  (match!  env ptree arguments)
                  (mceval body env)))))))).
```
(6.33)

The only other element needed is *$macro*:

```
($mc-define! $macro
   (make-operative
      ($lambda ((parameters names body) static-env)
         (make-operative
            ($lambda (operands dynamic-env)
               (mceval
                  ($let ((local-env  (make-mc-environment
                                        static-env)))
                     (match! local-env parameters operands)
                     (gensyms! local-env names)
                     (transcribe body local-env))
                  dynamic-env))))))

($define! gensyms!
   ($lambda (env names)
      ($cond ((mc-symbol? names)
                 (match! env names (gensym)))
             ((pair? names)
                 (gensyms! env (car names))
                 (gensyms! env (cdr names))))))

($define! transcribe
   ($lambda (body env)
      ($cond ((mc-symbol? body)  (lookup body env))
             ((pair? body)
                 (cons (transcribe (car body) env)
                       (transcribe (cdr body) env)))
             (#t  body)))).
```

(6.34)

The evaluation phase of this algorithm takes about 60% more lines of high-level code than that of vanilla Scheme. In comparison, the smallest of the preprocessing phases in the preceding sections —that of procedural macros— was about 85% as long as the vanilla evaluation phase. The low-level evaluation code has increased by about 30% of the vanilla low-level code, of which 20% is gensyms, and the remaining 10% is the object-language *operative* type.

6.6 Kernel

The Kernel meta-circular evaluator shares elements of the preceding section that are generic to first-class operatives: the operator-evaluating version of *mceval*, (6.29);

meta-language tools *make-operative* and *mc-operate*; and, based on the latter tools, the *mc-define!*s for *$if* and *$define!* (from (6.33)).

The basic meta-language tools for object-language applicatives are *mc-wrap* and *mc-unwrap* (rather than *make-applicative* and *mc-apply* that were used in all the previous sections of the chapter). There is no need for an *mc-apply* in the meta-language, since its only use by the evaluator was in handling applicative combinations, and *combine* now handles that case by calling itself recursively:

```
($define! combine
   ($lambda (combiner operands env)
      ($if (mc-operative? combiner)
           (mc-operate combiner operands env)
           (combine (mc-unwrap combiner)
                    (map-mceval operands env) env)))).
```
(6.35)

However, *make-applicative* facilitates population of the object-language ground environment, so is constructed from the more basic tools, by

```
($define! make-applicative
   ($lambda (meta-appv)
      (mc-wrap
         (make-operative
            ($lambda (operands #ignore)
               (apply meta-appv operands))))))).
```
(6.36)

Meta-language *mc-wrap*, *mc-unwrap*, and *mceval* are all needed in the object-language as primitives, so

```
($mc-define! wrap   (make-applicative mc-wrap))
($mc-define! unwrap (make-applicative mc-unwrap))
($mc-define! eval   (make-applicative mceval)).
```
(6.37)

Object-language *apply* and *$lambda* are no longer object-language primitives. The only other object-language primitive needed is *$vau*:

```
($mc-define! $vau
   (make-operative
      ($lambda ((ptree eparam body) static-env)
         (make-operative
            ($lambda (operands dynamic-env)
               ($let ((local-env  (make-mc-environment
                                     static-env)))
                  (match! local-env ptree operands)
                  (match! local-env eparam dynamic-env)
                  (mceval body local-env)))))))).
```
(6.38)

The Kernel evaluation phase takes about 25% more high-level code than vanilla evaluation. (The absolute net difference is 9 lines.) The low-level code for Kernel is actually two lines shorter than for vanilla Scheme.

6.7 Line-count summary

Table 6.1 gives the net change in the combined evaluation and (when present) preprocessing phases of each algorithm relative to vanilla Scheme. The percentages of low-level code represent larger absolute numbers.

algorithm	high-level	low-level
naive template macros	120%	60%
naive procedural macros	85%	85%
hygienic macros	195%	95%
single-phase macros	60%	30%
Kernel fexprs	25%	-5%

Table 6.1: Line-count increase for each algorithm, vs. vanilla Scheme.

Chapter 7

Programming in Kernel

7.0 Introduction

If Kernel is to satisfy the spirit of the thesis, it has to satisfy the claims of the thesis simultaneously, not serially: Kernel must subsume Lisp's traditional syntactic abstractions —macros— *together with* its own peculiarly fexpr-based abstractions, and be well-behaved at the same time. Complicating this task, Kernel hygiene is not absolute but (as observed in Chapter 5) depends on the *style* of programming used; and, moreover, the introduction of general quotation facilities into Kernel —*together with* fexprs— was found to be antagonistic to good hygiene.

This chapter considers how Kernel can support a coherent programming style that uses Kernel's native fexpr support, fosters good hygiene, and achieves the purposes traditionally addressed in Lisp by quotation and macros. Neither general quotation nor macro-style constructors (such as Scheme's `$syntax-rules`) are introduced.

7.1 Binding

7.1.1 Declarative binding

Scheme and Kernel both support block declarations through a family of standard `let` operatives, all derivable from more primitive facilities (using Scheme's hygienic macro facility, or Kernel's `$vau`).[1] Each operative takes a list of bindings and a list of expressions (called the *body*), and evaluates the expressions in a local environment with the given bindings; the operatives in the family differ from each other in details of how they set up the local environment.

[1] While the *R5RS* ([KeClRe98]) treats these hygienic-macro derivations as primary definitions, the *R6RS* ([Sp+07]) presents some of the more awkward hygienic-macro definitions as merely "approximate", relying instead on formal semantics for the primary definitions. In effect, the *R6RS* has conceded that some of its `let` operatives cannot be derived using its abstraction facilities.

R5R Scheme provides three standard `let` operatives —*`let`*, *`let*`*, and *`letrec`*— to which *R6R* Scheme adds *`letrec*`*.[2]

Kernel has four basic `$let`s —*`$let`*, *`$let*`*, *`$letrec`*, and *`$letrec*`*— and an additional two `$let`s tailored to insulate their local environments from the environments that surround them — *`$let-redirect`* and *`$let-safe`*.

This subsection reviews all six Kernel `$let`s, both as preparation for their use in more advanced techniques, and as a demonstration of the facility of constructing new binding tools afforded by Kernel's simultaneous smooth treatment of operatives, environments, and compound definiends.

Tools

The most basic operative in the facility is *`$let`*, which is merely a shorthand for calling an explicit *`$lambda`*-expression:

$$\begin{aligned} &(\texttt{\$let}\ ((\langle \mathrm{p}_1\rangle\ \langle \mathrm{v}_1\rangle)\ \cdots\ (\langle \mathrm{p}_n\rangle\ \langle \mathrm{v}_n\rangle))\ .\ \langle \mathrm{body}\rangle) \\ \equiv\ &((\texttt{\$lambda}\ (\langle \mathrm{p}_1\rangle\ \cdots \langle \mathrm{p}_n\rangle)\ .\ \langle \mathrm{body}\rangle)\ \langle \mathrm{v}_1\rangle\ \cdots\ \langle \mathrm{v}_n\rangle)\,, \end{aligned} \tag{7.1}$$

derivable in Kernel by

```
($define! $let
   ($vau (bindings . body) env
      (eval (cons (list* $lambda (map car bindings) body)
                  (map cadr bindings))
            env))).
```
(7.2)

A critical difference between Scheme *`let`* and Kernel *`$let`* is that, in Kernel, the definiends —the $\langle \mathrm{p}_k\rangle$ in (7.1)— can be compound. This allows any of the $\langle \mathrm{v}_k\rangle$ to specify a structured result that is then automatically destructured for separate binding of its parts (whereas in Scheme the whole would have to be bound first, and then broken down into its parts). Kernel treats definiends uniformly: any definiend may be compound, and is matched structurally against pairs and nil as specified in §4.2. Kernel's smooth handling of definiends greatly streamlines the use of Kernel environments in managing complex patterns of data flow; and, more specifically, streamlines various combiner constructions throughout the dissertation (e.g., *`$letrec`* later in this subsection, *`mceval-combination`* et al. in Chapter 6), and the handling of encapsulation below in §7.2.[3]

The "`*`" variant `$let`s process bindings in order from left to right, each binding in the local environment provided by the preceding binding, so facilitating specification

[2] *R6R* Scheme also has two other `let` operatives, *`let-values`* and *`let*-values`*. However, those two exist to support a Scheme misfeature called *multiple-value return*. In Kernel, "multiple-value return" is a mundane special case of single-value return; see Footnote 3, below.

[3] Rationale for Kernel's definiend handling —including how it interacts with Kernel's treatment of continuations to de-exceptionalize multiple-value return— is discussed in [Shu09, §4.9.1 (`$define!`)].

of a series of bindings where each may depend on its predecessors. In the case of *$let**,

```
   ($let* ((⟨p1⟩ ⟨v1⟩) ··· (⟨pn⟩ ⟨vn⟩)) . ⟨body⟩)
≡ ($let ((⟨p1⟩ ⟨v1⟩))
     ($let ((⟨p2⟩ ⟨v2⟩))
        ···
           ($let ((⟨pn⟩ ⟨vn⟩))
              ($let () . ⟨body⟩))···)),
```
(7.3)

which behavior is derivable in Kernel by

```
($define! $let*
   ($vau (bindings . body) env
      (eval ($if (null? bindings)
                 (list* $let bindings body)
                 (list $let
                       (list (car bindings))
                       (list* $let* (cdr bindings) body)))
            env)).
```
(7.4)

The (*$let* () . ⟨body⟩) at the bottom of the nesting discourages accidents when $n = 0$, by guaranteeing that the body will always be evaluated in a local environment.

The "`rec`" variant `$let`s compute the value for each binding within the local environment where the binding will be created. The primary reason to do so is recursion: if a combiner is bound in its own static environment, then it can see its own name in order to call itself.[4] *$define!* facilitates recursion too, but uses imperative style for its environment mutation (more general, but therefore more error-prone); and does not introduce a local block, so is non-global only when a local block has been introduced by another construct.

Standard Scheme requires that the set of symbols to be bound in a local environment must be fixed at environment construction (see also §5.3.2). The status of Scheme *letrec*'s local bindings is therefore problematic during the time that the values for those bindings are being locally computed: the bindings "exist" but have not been given values, so that it is an error (meaning it's wrong but the interpreter isn't required to notice[5]) for any of those value computations to use any of the local bindings. The implementation of Scheme *letrec* as a hygienic macro initializes the bindings to a hypothetical "undefined" value that, when discovered as the result of a symbol lookup, causes an error; the values for the bindings are then locally computed, and the bindings are mutated using Scheme *set!*.

[4] In principle, one can implement recursion using just *$lambda*, but the constructions to do so are cumbersome; see [AbSu96, Exercise 4.21].

[5] *R6R* Scheme *is* required to notice.

Because Kernel ***$define!*** can locally bind symbols that had been locally unbound, there is no need to introduce a new concept (that of "undefined value" or "uninitialized binding") to explain ***$letrec***. Its behavior follows equivalence

$$\begin{aligned}
&(\textbf{\$letrec}\ ((\langle \mathrm{p}_1\rangle\ \langle \mathrm{v}_1\rangle)\ \cdots\ (\langle \mathrm{p}_n\rangle\ \langle \mathrm{v}_n\rangle))\ .\ \langle \mathrm{body}\rangle)\\
\equiv\ &(\textbf{\$let}\ ()\\
&\quad (\textbf{\$define!}\ (\langle \mathrm{p}_1\rangle\ \cdots\ \langle \mathrm{p}_n\rangle)\ (\textbf{list}\ \langle \mathrm{v}_1\rangle\ \cdots\ \langle \mathrm{v}_n\rangle))\\
&\quad .\ \langle \mathrm{body}\rangle)\,.
\end{aligned} \tag{7.5}$$

While this equivalence is possible because ***$define!*** can bind previously unbound symbols, it is *simple* because of ***$define!***'s uniform treatment of compound definiends, which allows all the bindings to be made in parallel after all the values have been computed (guaranteeing that none of them will be able to see any of the other local bindings).

Kernel ***$letrec*** is derivable by

```
($define! $letrec
   ($vau (bindings . body) env
      (eval (list* $let ()
                   (list $define!
                         (map car bindings)
                         (list* list (map cadr bindings)))
                   body)
            env))).
```
(7.6)

Kernel ***$letrec**** is equivalent to a nesting of ***$letrec***s, just as ***$let**** is equivalent to a nesting of ***$let***s:

$$\begin{aligned}
&(\textbf{\$letrec*}\ ((\langle \mathrm{p}_1\rangle\ \langle \mathrm{v}_1\rangle)\ \cdots\ (\langle \mathrm{p}_n\rangle\ \langle \mathrm{v}_n\rangle))\ .\ \langle \mathrm{body}\rangle)\\
\equiv\ &(\textbf{\$letrec}\ ((\langle \mathrm{p}_1\rangle\ \langle \mathrm{v}_1\rangle))\\
&\quad (\textbf{\$letrec}\ ((\langle \mathrm{p}_2\rangle\ \langle \mathrm{v}_2\rangle))\\
&\qquad \cdots\\
&\qquad\quad (\textbf{\$letrec}\ ((\langle \mathrm{p}_n\rangle\ \langle \mathrm{v}_n\rangle))\\
&\qquad\qquad (\textbf{\$letrec}\ ()\ .\ \langle \mathrm{body}\rangle))\cdots))\,.
\end{aligned} \tag{7.7}$$

Each binding value is evaluated in the environment where its binding(s) will be created, and each successive value computation can see the bindings from previous clauses. The (***$letrec*** () . ⟨body⟩) at the bottom of the nesting discourages accidents when $n = 0$ by guaranteeing that the body will always be evaluated in a local environment.

Because each binding clause is processed in a different environment, there cannot be mutual recursion between combiners constructed by different clauses; however,

Kernel's treatment of compound definiends affords the flexibility to construct and separately bind mutually recursive combiners within a single clause, as in

```
($letrec* ((foo  ($lambda () 1))
           ((bar
             baz)  (list ($lambda () (baz))
                         ($lambda () (bar))))
           (quux  ($lambda () (bar))))
   (quux))
```
(7.8)

(which won't produce a symbol-not-found error, though it won't do anything else useful either).

*$letrec** is derivable by

```
($define! $letrec*
   ($vau (bindings . body) env
      (eval ($if (null? bindings)
                 (list* $letrec bindings body)
                 (list $letrec
                       (list (car bindings))
                       (list* $letrec* (cdr bindings) body)))
            env))).
```
(7.9)

The two remaining members of Kernel's `$let` family, *$let-redirect* and *$let-safe*, were discussed in §5.3.1 as means to promote stable bindings. They isolate their local block from the surrounding environment, by using some other environment as the parent for the local environment — either a specified environment (*$let-redirect*), or a freshly constructed standard environment (*$let-safe*).

For these two operatives, the processing of binding clauses is auxiliary to the primary purpose of insulating the local block. Therefore, only one binding strategy is supported: computing the binding values in unspecified order in the surrounding environment (as for *$let*), which complements the local insulation by allowing controlled importation of information from the surrounding environment.

Usage

The environment-isolating `$let`s —*$let-redirect* and *$let-safe*— could become a *cause* of accidents if they were used casually; the programmer ordinarily expects symbol meanings to be drawn from surrounding context, and exceptions to that rule should be rare and well-marked. Use of isolated blocks should therefore be reserved for large modular units of a software system. Since the isolating `$let`s aren't to be used often, and to clearly mark the rare occasions that they are used, they have

comparatively long, descriptive names.

The four Scheme-like members of the family have two uses in Scheme, both of which continue in Kernel.

The simpler Scheme use is for temporary storage of auxiliary computation results that are only needed in a certain block. One might write

```
($let ((temp  (foo x y)))
   (+ temp (* temp temp)))
```
(7.10)

in which side-calculation `(foo x y)` (whatever it is) is performed once and the result stored for repeated use in the ensuing formula.

The more advanced use in Scheme is to provide local bindings that are only accessible to compound combiners constructed in the local block, but that persist between calls to those combiners. A classic example (which was discussed in §5.3.2, and which arises in practice in §A.5) is an applicative that returns a larger integer each time it is called, using a persistent local variable to keep track of what to return next. The Kernel implementation of this example requires *two* local bindings: one for the integer, and another referencing the local environment itself. The latter binding gives the applicative means to mutate the persistent local environment.

```
($define! get-next-integer
   ($letrec ((self  (get-current-environment))
             (n     0))
      ($lambda ()
         ($set! self n (+ n 1))
         n))).
```
(7.11)

In Kernel, persistent local bindings are commonly shared by multiple compound combiners, as in (hypothetically; a better way to do this will be developed in §7.1.2)

```
($letrec ((n  0))

   ($define! add-to-count
      ($let ((self  (get-current-environment)))
         ($lambda () ($set! self n (+ n 1)))))

   ($define! get-count
      ($lambda () n))
   ⋮
   ).
```
(7.12)

Permission to mutate the persistent local environment —conferred here by the binding of `self`— is potentially dangerous, and so should not be distributed any more widely than necessary. Whenever a locally constructed combiner is given permission

to mutate the persistent local environment, any third-party combiner that captures *its* local environment will then also acquire that permission. Here, `$set!` acquires (and uses) permission to mutate the persistent local environment, and if `$set!` behaves as expected, `+` can acquire that permission, too (underlining the importance of stabilizing the bindings of `$set!` and `+`). On the other hand, once a local environment has been constructed for `get-count`, capturing that local environment would only allow a third party to *see* the persistent local bindings, not to modify them.

It is therefore good practice to limit local mutation-permissions to those locally constructed combiners that need it.

While a local block in Scheme might contain multiple local combiners, as in this example, Scheme makes it awkward to export more than one object to the surrounding environment; even if a list of multiple objects were returned as the value of the block, Scheme doesn't provide declarative means to separately bind the elements of the returned list in the surrounding environment (since it doesn't support Kernel-style compound definiends). However, Kernel affords convenient multiple exportation, as seen in the next subsection (§7.1.2).

7.1.2 Imperative binding

Standard Scheme supports just two cases of environment mutation: top-level `define`, which adds bindings to the global environment; and `set!`, which cannot create bindings, but which can mutate any visible binding non-locally (undermining binding stability, as discussed in §5.3.2). Scheme restricts local `define` to the beginnings of blocks, so that it is effectively not a mutator, but an alternative notation for declarative `letrec`.

Kernel's three standard environment-mutators —`$define!`, `$set!`, and `$provide!`— are technically equi-powerful, in that any of them can be derived from either of the others; but they are practically suited to play different roles. `$define!` is streamlined for mutating the current environment (a task it performs more versatilely than standard Scheme supports, as discussed above in §7.1.1). `$set!` is better suited than either of the others for general forms of environment mutation — with an explicitly computed target environment, explicitly computed value to bind to, and general definiend; it is commonly used to mutate an ancestor of the current environment, as illustrated in §5.3.2 and §7.1.1.

`$provide!` is suited to the specialized task of constructing several bindings in a local block and then exporting them back to the surrounding environment.

The first operand to `$provide!` is a list of symbols to be bound, and the remaining operands are the body of a local block. A local child of the surrounding environment is constructed; the expressions in the body are locally evaluated left-to-right; and then each symbol is bound in the surrounding environment to whatever it is bound to in the local environment. This behavior follows equivalence

```
   ($provide! ⟨symbols⟩ . ⟨body⟩)
≡ ($define! ⟨symbols⟩
     ($let ()                                                    (7.13)
        ($sequence . ⟨body⟩)
        (list . ⟨symbols⟩)))
```

(where *$sequence* evaluates its operands left-to-right in its dynamic environment and returns the result of the last evaluation), derivable by

```
($define! $provide!
   ($vau (symbols . body) env
      (eval (list $define! symbols
                  (list $let ()                                  (7.14)
                     (list* $sequence body)
                     (list* list symbols)))
            env))).
```

Using *$provide!*, one might then write (adapting the multiple-export fragment from (7.12))

```
($provide! (add-to-count  get-count)

   ($define! n 0)

   ($define! add-to-count                                         (7.15)
      ($let ((self  (get-current-environment)))
         ($lambda () ($set! self n (+ n 1)))))

   ($define! get-count ($lambda () n))).
```

Technically, the same thing could be accomplished using *$define!* or *$set!*. One could write

```
($define! (add-to-count  get-count)
   ($let ((n  0))
      ⋮                                                           (7.16)
      (list add-to-count get-count))),
```

returning the exported combiners in a structure to be broken down for binding to the compound definiend; but then the programmer would have to manually maintain coordination between the definiend at the top of the block, and the list of exported objects at the bottom of the block — which might be very remote from each other in

a large module. Alternatively, one could write

```
($let ((outside  (get-current-environment))
       (n        0))
   ⋮                                                          (7.17)
   ($set! outside (add-to-count get-count)
             (list add-to-count get-count))),
```

so that the lists to be coordinated are adjacent; but then the surrounding environment captured for use by ***$set!*** has also been made available to all of the tools in the local block, none of which need it, multiplying its potential binding instability.

Use of ***$provide!***, (7.15), in contrast to both alternative approaches, encapsulates mechanical details of the intended bad hygiene —the exportation— so that the client can more reliably avoid *un*intended bad hygiene.

7.2 Encapsulation

In modern programming languages, when a local block is encapsulated and exports multiple combiners, the exports usually constitute one or more encapsulated data types. Standard Scheme does not support programmer-defined encapsulated data types; but Kernel, in accordance with its design Guideline *G4* (*The degree of encapsulation of a type should be at the discretion of its designer*, §3.5 and [Shu09, §0.1.2]), does provide a simple device for generating encapsulated types. Kernel's type-generation device figures prominently in Kernel style, complementing the multiple-exportation facility of Kernel's ***provide!*** (§7.1.2) and exploiting Kernel's uniform treatment of compound definiends.

Kernel's type-generation facility consists of a single standard applicative, ***make-encapsulation-type*** ([Shu09, §8 (Encapsulations)]). Each time it is called, *make-encapsulation-type* returns a list of three freshly allocated applicatives, `(`*e p? d*`)` where

- *e*, called an *encapsulator*, takes one argument, and returns a freshly allocated *encapsulation* object. The argument to *e* is called the *content* of the encapsulation.
- *p?* takes zero or more arguments, and returns true iff all of them are encapsulations generated by *e*.
- *d*, called a *decapsulator*, takes one argument, which must be an encapsulation generated by *e*, and returns its content.

The three applicatives form a matched set of tools for a newly constructed data type (that exists for the duration of the Kernel interpreter session). The content of an encapsulation can only be accessed via its matching decapsulator, so that limiting

visibility of the decapsulator limits access to the content of matching encapsulations. A common Kernel idiom is to call *make-encapsulation-type* inside a *$provide!* block, so assigning local names to its three results through a compound definiend, then use the encapsulator and decapsulator as private tools from which to construct public tools that are exported from the block. Schematically,

```
($provide! (public-constructor
            type-predicate?
            public-accessor)

   ($define! (encapsulator
              type-predicate?
              decapsulator)  (make-encapsulation-type))      (7.18)

   ($define! public-constructor
      ($lambda ···))

   ($define! public-accessor
      ($lambda ···))).
```

This idiom is used repeatedly in Appendix A (§A.5–§A.6), where it encompasses most of the low-level code for the meta-circular evaluators. Another example is the implementation of promises in [Shu09, §9 (Promises)].

7.3 Avoiding macros and quotation

Although general quotation facilities (*$quote*, *$quasiquote*, etc.) could be constructed in Kernel, their coexistence with first-class operative support would provoke bad hygiene (as found repeatedly in Chapter 5 on Kernel hygiene, and also noted of single-phase macros in §3.4.1). Macro constructors —template or procedural, naive or hygienic— could also be constructed, but to do so would be of merely computational, not abstractional, interest. This section discusses how Scheme tasks involving general quotation and macros can be accomplished via a Kernel programming style that involves neither.

7.3.1 Macros

The elimination of macros and quotation is essentially a shift from implicit to explicit evaluation. Starting from Scheme-style hygienic macros, the first step in the shift is to convert the macros from template form, to procedural form using quasiquotation. This step is straightforward, because quasiquotation is still a template-based implicit-evaluation device; but the template in a quasiquoted expression, unlike that in a template-macro, can be eliminated gradually — as will be done below in §7.3.2.

Put another way, all macros implicitly evaluate a constructed expression, but template macros also perform the *expression construction* implicitly; procedural macros by nature perform expression construction explicitly, so the residual use of quasiquotation is only piecewise-implicit, not systemically implicit.

For now, we ignore hygiene issues in the conversion, to focus on the more global algorithmic changes; hygiene will be restored later, during quotation elimination in §7.3.2.

Scheme hygienic macros also use pattern-directed invocation, each macro selecting from amongst multiple call-syntax–to–template clauses. In principle, the clauses are (as noted in §6.4) analogous to syntax rules of a Chomsky type 0 grammar, hence can be used to perform Turing-powerful syntax transformations; under the Smoothness Conjecture (§1.1.2), however, Turing-powerful pattern-directed macro invocation is an abstractional liability, since it heightens the mismatch between computations in the two mutually segregated phases of processing. Procedural macros —and fexprs— use ordinary Lisp facilities to express syntax transformations, obviating the need for a separate syntax-transformation sublanguage.

If clause-selection were the only control structure in Scheme's macro sublanguage, the equivalent procedural macro could always make do with a top-level conditional (***$if*** or ***$cond***) to choose between the alternatives, and quasiquotation to handle each alternative when chosen. However, clause selection only achieves Turing power if one is willing to construct auxiliary macros to handle subcomputations; and these auxiliary macros cannot be made local to the macro definition that they assist.[6] Therefore, to reduce clutter of the client's namespace with auxiliary macros that modularly ought to be hidden, Scheme increases the expressive power of each individual clause by means of an *ellipsis* facility for specifying and rearranging repeated subpatterns. The ellipsis symbol (`...`) in a pattern indicates zero or more repetitions of the subpattern it follows; and ellipsis can be distributed (in the algebraic sense) in the template over the parts of a compound subpattern. For example, a Scheme pattern-macro for simple ***$let***,[7]

```
($define-syntax $let
   ($syntax-rules ()
      (($let ((name val) ...) exp ...)
       (($lambda (name ...) exp ...) val ...))),
```
(7.19)

distributes ellipsis of the binding-clause pattern `(name val)` over the parts of the pattern. Distribution of ellipsis in a template-clause can be implemented in a procedural

[6]That is, one cannot declare the auxiliary macros in a local syntactic environment and then export the primary macro from that local syntactic environment as an *upward funarg*; cf. §3.3.4.

[7]Full Scheme ***$let*** requires at least one expression in the body (because Scheme ***$lambda*** does), and provides an extended syntax allowing the constructed ***$lambda***-expression to call itself recursively ([KeClRe98, §4.2.4]).

macro using *map*; here,

```
($define-macro $let
   ($lambda (bindings .  body)
      `((($lambda ,(map car bindings) . ,body)
        . ,(map cadr bindings)))).
```
(7.20)

(As footnoted in §3.4.1, standard syntactic sugar for quasiquotation uses prefix backquote (`) to designate a quasiquoted expression, and prefix comma (,) to designate its unquoted, i.e. evaluated, subexpressions.)

Once macros are in procedural form, their remaining systemic commitment to implicit evaluation can be eliminated by rewriting them as fexprs that explicitly evaluate the same expression used in the procedural macro; that is,

```
   ($define-macro ⟨name⟩
      ($lambda ⟨ptree⟩
         ⟨exp⟩))
⟶                                                    (7.21)
   ($define! ⟨name⟩
      ($vau ⟨ptree⟩ ⟨eparm⟩
         (eval ⟨exp⟩ ⟨eparm⟩))).
```

In the above case of simple *$let*, (7.20), one would have

```
($define! $let
   ($vau (bindings . body) env
      (eval `((($lambda ,(map car bindings) . ,body)          (7.22)
              . ,(map cadr bindings))
            env))).
```

The fact that expression construction now occurs during evaluation rather than before evaluation has no effect on the outcome of computation, because macro expression construction has no side-effects.[8]

7.3.2 Quasiquotation

The remaining implicit evaluation in our converted code consists entirely of quotation, which is distributed rather than systemic; that is, it can be eliminated gradually, whereas the template-macro/procedural-macro and macro/fexpr distinctions were all-or-nothing.

[8]Since macro expansion is Turing-powerful, it can have the side-effect of nontermination; but we take nonterminating macro expansion to be axiomatically an error, since the purpose of macro expansion is to provide syntax for subsequent evaluation, so that nonterminating macro expansion has literally no purpose.

We have also neglected, thus far, to preserve the hygiene of the original Scheme macros we were converting; for example, in the above quasiquoting fexpr, (7.22), symbol `$lambda` is introduced by the construction and could be captured by the dynamic environment *`env`*. The usual measures to restore hygiene, from §5.1, also further reduce the amount of quotation used (consonant with the quotation/bad-hygiene correlation). Capture of variables introduced by the construction is eliminated by *unquoting the symbols*, so that the constructed expression contains self-evaluating combiners rather than unevaluated symbols; simple *`$let`* would become

```
($define! $let
   ($vau (bindings . body) env
      (eval `((,$lambda ,(map car bindings) . ,body)
              . ,(map cadr bindings))
            env)))
```
(7.23)

(the difference from (7.22) being the comma prefix on `$lambda`). Capture of variables in the operands, by bindings introduced into the construction, is eliminated by pulling the binding and its bound variable occurrences out of the constructed expression, hence unquoting them; in the case of binary *`$or?`* from §5.1, naive fexpr

```
($define! $or?
   ($vau (x y) e
      (eval `(,$let ((temp ,x)) (,$if temp temp ,y)) e)))
```
(7.24)

would become hygienic

```
($define! $or?
   ($vau (x y) e
      ($let ((temp  (eval x e)))
         ($if temp temp (eval y e))))).
```
(7.25)

Above and beyond the needs of hygiene, there are often further actions of the constructed expression that could just as well have been performed directly in the local environment, outside the constructed expression; and these usually *should* be performed locally, as source code is clearer when it specifies what to do than when it specifies how to specify what to do (i.e., direct code is clearer than indirect code). This was done, for example, in the compound construction of *`$lambda`* in §4.3, (4.10), where the call to *`wrap`* was performed locally rather than add a second layer to the expression-construction.

If there is any quasiquotation left by now (binary *`$or?`*, for example, lost the last of its quotation when it was made hygienic in (7.25)), the remaining quasiquotation should be converted to explicit pair constructions using *`list`*, *`list*`*, and (when occasionally clearer than *`list*`*) *`cons`*. Experience with Kernel to date suggests that, by this point in the conversion, any remaining expression-construction is fairly simple, so that the calls to *`list`* etc. will not themselves become a significant obstacle

to programmers reading the source code; note that overall smoothness of the language should contribute to simplicity of expression-constructions (cf. the role of compound definiends in the derivation of `$letrec` in §7.1.1, (7.6)). In the event that a large complicated expression-construction is still needed, though, the Kernel programmer has the option of modularizing the construction by means of local auxiliary combiners — because in Kernel, the auxiliary combiners can be declared locally, and so not clutter the client's namespace.

7.3.3 Quoted symbols

With macros and quasiquotation eliminated, the only possible remaining vestige of implicit-evaluation style is the quotation of individual symbols. The vast majority of quoted symbols in macro templates will have been eliminated during conversion, by one or the other hygiene measure; and the Kernel design further reduces the need to embed unevaluated symbols in constructed expressions by avoiding keywords, such as the `else` in the call-syntax of Scheme `$cond` (§2.2.3.2).

Explicitly embedded symbols can often be avoided stylistically, by designing compound operatives so that all the unevaluated symbols for a constructed expression are provided by the operands. All the derived binding constructs in §7.1 worked this way, each extracting its definiends and body from its operand tree. The practice is relatively hygienic, because the constructed expression will usually be evaluated in the dynamic environment, which is where its captured parts were originally specified. Free variables in the constructed expression are interpreted in the context where they appeared (that surrounds the calling combination); bound variables in the constructed expression are bound by declarative definiends that are specified in the same context as the variables (that of the calling combination), as with the `$let` constructs of §7.1.1; and even imperative definiends, as to `$provide!`, are apparent in the context where the bindings will take effect. Also, aside from hygiene, an operative that draws its variables and definiends from its operands is a *more general tool* than if these things were hardwired into the operative, which is also good programming style by promoting reusability.

Besides the construction of expressions to be evaluated, two other uses for an unevaluated symbol are: to serve as a tag in a data structure, and to be used for comparison with another object that might be the same symbol. The latter arises, e.g., when *checking* a tag on a data structure; also, when looking up a symbol in a table, as in the implementation of environments for the meta-circular evaluator, §A.5; and when analyzing a value that has been read, as in the special-form handling of the Scheme meta-circular evaluator, §6.1.

In these cases, the programmer can avoid introducing general quotation facilities by combining the specification-by-operands technique with *currying* (§3.3.1). That is, the requisite unevaluated symbol is provided to an operative constructor, which then returns an applicative specialized to perform the relevant operation for that symbol

by drawing the symbol from its static environment (which is the local environment of the constructor). Schematically,

```
($define! $constructor
   ($vau (symbol) #ignore                                    (7.26)
      ($lambda (···) ··· symbol ···))) .
```

From the vanilla Scheme meta-circular evaluator (§A.5, §A.3),

```
($define! $make-tag-predicate
   ($vau (tag) #ignore
      ($lambda (x) (eq? x tag))))
                                                             (7.27)
($define! if-operator?     ($make-tag-predicate $if))
($define! define-operator? ($make-tag-predicate $define!))
($define! lambda-operator? ($make-tag-predicate $lambda)) .
```

Part II

The vau calculus

Chapter 8

Preliminaries for Part II

8.0 Agenda for Part II

This chapter provides historical perspective, concepts, and conventions preliminary to Part II. Chapter 9 develops pure vau calculi, clarifying that fexprs are not an inherently imperative (side-effect-ful) feature. Chapters 10–12 develop impure vau calculi, establishing that our fexpr strategy remains tractable in an imperative setting. Chapters 13–14 prove basic well-behavedness results for vau calculi — Church–Rosser-ness, standardization, and operational soundness. Chapter 15 relates the non-reflective fexpr facility of vau calculi to the reflective fexpr facility of Mitchell Wand's well-known paper [Wa98].

8.1 Some history

8.1.1 Logic and mathematics

> Progress in mathematics almost demands a complete disregard of logical scruples.
>
> — Morris Kline, [Kli72, §18.4].

> Top-down design is a great way to redesign a program you already know how to write.
>
> — P.J. Plauger, [Pla86].

We selectively review the historical interplay between mathematics and logic.

Confusion may be avoided later by recognizing, at the outset, that there are no universally agreed-upon boundaries for *logic* and *mathematics*. Rather, each of the two subjects is pursued by a living community, and so ultimately each is defined by its community: logic is what logicians study, and mathematics is what mathematicians

study. Historically, logic is the study of inference, and mathematics is the study of number; and this might still be claimed, since *study of inference* can be stretched to include nearly any reasoning, while *number* can be stretched to include nearly anything reasoned about; but in practice, rapid technological expansion of mathematics over the past four and a half centuries, and of logic over the past one and a half, has provided great scope for disagreement over the boundaries of both disciplines.

The modern entanglement of mathematics with logic was precipitated by the scientific revolution.

The ancient Greeks had imposed on mathematics such a high standard of rigorous justification that, for about two thousand years, it prevented generalizations of number beyond the positive rationals. The scientific revolution called for more powerful mathematical tools. Mathematicians first responded by abandoning rigor in order to exploit extended forms of number — negative, complex, and infinite and infinitesimal.[1] By the early nineteenth century, though, the trend toward still more general numbers led back into rigor. Driven by the ubiquity of directed magnitudes in physics, and guided by the correspondence between complex arithmetic and plane geometry, mathematicians were looking for a generalization of complex numbers that would correspond to three-dimensional geometry ([Kli72, §32.2]); and there is such a generalization, called *quaternions*; but general quaternion multiplication is non-commutative.[2] Mathematicians taking a non-rigorous approach had managed the earlier kinds of numbers by assuming —incautiously but, in the event, correctly— that the basic laws of arithmetic for positive integers would continue to hold for the extensions; but when the familiar laws ceased to hold, a rigorous approach was needed to work out what modified laws should apply.[3]

In the new paradigm of mathematical rigor, elements of a formal system would be constructed mechanically, and their properties then deduced by logic. Once made respectable by the success of quaternions, the new paradigm was used on an expanding

[1]The prolific mathematical advances of the eighteenth century (largely delimited, in both time and content, by the life of Leonard Euler) were fueled by these extended forms of number. Irrational numbers are less strongly associated with that century; by 1700, they had already been in wide —though by no means universal— use in Europe for about one and a half centuries. Historically, irrational numbers have been much more widespread than negative numbers: irrationals had to be deliberately rejected (which they usually were), but negative roots of equations simply weren't conceived of. (See [Kli72, §13.2].)

[2]Arithmetically, the three products of the later vector analysis —scalar, dot, and cross— are fragments of the quaternion product; and the cross product isn't commutative, so the quaternion product isn't either. Geometrically, the quaternion product corresponds to rotation in three-space (and four-space); and permuting a sequence of general rotations produces different results, so permuting a sequence of quaternion multiplications produces different results.

[3]Scientific paradigm shifts, such as the ones in mathematics that we're describing, may look like a reasoned progression of ideas when viewed from a comfortable distance, yet in detail be more population replacement than conversion of individuals. Sir William Rowan Hamilton, who discovered quaternions in 1843, already belonged to the minority of mathematicians at the time who favored a rigorous approach — as did George Boole, who treated logic algebraically a few years later.

range of formal systems. At first, researchers pursued formal systems in the numerical vein, notably leading them beyond quaternions to linear algebra. Even the early work of George Boole and Augustus De Morgan in the mathematical treatment of logic has a numerical flavor to it. However, in the last decades of the century, Gottlob Frege attempted a radically different formal treatment of logic —the *logistic* program— in which formal logic is founded on its own small set of formal principles, and *all of mathematics* is then derived from it.

Frege began his program by constructing a new symbolic language for logic, replacing verbal syllogisms (in use since the ancient Greeks — "Some A is all B" and the like[4]) with a calculus of propositions (substantially modern, though using his own idiosyncratic notation). In doing so, he introduced so many of the staple concepts of modern logic that he may fairly be considered its founder. Among his contributions, of particular relevance here are the rigorous treatment of variables; decomposition of propositions into propositional function and argument; and distinction between a proposition p and the assertion that p is true. (See [Chu71, §iv.15], [Kli72, §51.4].) Frege's magnum opus was the two-volume *Grundgesetze der Arithmetik* (The Fundamental Laws of Arithmetic), of which Volume 1 came out in 1893 and Volume 2 in 1903.

Meanwhile, from about 1895 Georg Cantor's investigation of infinite sets began to uncover antinomies[5] ([Kli72, §41.9]). In ordinary logic, once an antinomy is proven, every expressible proposition can be proven, and proof itself becomes a worthless exercise. Further, a foundational problem in Cantor's set theory had wider implications, because uncountably infinite sets are used in *analysis* (the theory of real numbers, i.e., including irrationals). Then Bertrand Russell adapted one of the earlier antinomies to Frege's logic (latterly called Russell's Paradox), communicating it to Frege in 1902, just as the final volume of Frege's *Grundgesetze* was going to press. Further logical antinomies were discovered in the next few years ([Kle52, §11]). Thus the foundational crisis spread from analysis (underlain by uncountably infinite sets) to all of mathematics (underlain by logic).

In evident response to the crisis, three schools of thought emerged on the proper foundations for mathematics, called *logicism* (Frege's approach), *intuitionism*, and *formalism*.[6] ([Kle52, Ch. iii], [Kli72, Ch. 51].)

[4]If you aren't quite sure what "Some A is all B" means, the problem isn't you; the correct interpretation of syllogisms was sometimes a topic for debate in its own right. Today, we would conduct such a debate in terms of existential and universal quantifiers, set membership ($\in$), and subset ($\subseteq$) — all of which in their modern forms are due to Frege.

Incidentally, Aristotle would not have used a quantifier (here, "all") on the predicate B; quantification of the predicate was a refinement introduced in the early nineteenth century by another Sir William Hamilton, this one a Scottish philosopher.

[5]An antinomy is a pair of mutually contradictory statements that are both provable. The term *antinomy* is therefore more precise than the more commonly used *paradox*, which has been used colloquially for any logical conclusion that seems to defy common sense (such as Frederic's predicament in Act II of *The Pirates of Penzance,* [GiSu1880]).

[6]These names were assigned to the schools later.

Russell collaborated with Alfred North Whitehead in a second attempt at Frege's logistic program, deriving mathematics from logic; their attempt culminated in the three-volume *Principia Mathematica* of 1910–13. They attributed the antinomies to *impredicative* definitions, i.e., definitions that are not strictly prior to what they define (such as the set of all sets, impredicative since the set contains itself; or the set of all sets that do not contain themselves, viciously impredicative since its self-containment leads immediately to Russell's Paradox). To eliminate impredicativity, they stratified properties (i.e., propositional functions) in the *Principia* by nonnegative integer *type*, restricting each function to take arguments of strictly lower type.[7] However, having carefully arranged this stratification, they also introduced an axiom to collapse the hierarchy (for every proposition involving properties of type ≥ 1 there is an equivalent proposition with properties all of type 0), for which the justification, by their own admission, was only that it was what was needed to found analysis. Even the authors considered this axiom a stopgap.

Some mathematicians objected that infinity cannot be achieved, only perpetually approached, and therefore it is nonsensical to treat any infinite structure (such as an infinite set) as a completed object; this position had occasional proponents through the nineteenth century.[8] From 1907, it was incorporated into a detailed philosophy of mathematics, *intuitionism*, by L.E.J. Brouwer. In the intuitionist view, mathematics is exact thought, while logic is an application of thought to the experience of language; so logic is based on mathematics, rather than mathematics on logic as in logicism. Mathematics determines the validity of logical principles; logical consistency or inconsistency is irrelevant to the validity of mathematical principles. When deriving logic from mathematics, intuitionists reject the Law of the Excluded Middle (for every proposition p, $p \vee \neg p$), on the grounds that it only applies to completed, i.e. finite, structures (and one can only perpetually approach "every proposition"). In effect, they thus allow for the possibility of undecidable propositions; and, in the process, also just happen to eliminate the known antinomies whose existence is of no concern to mathematics. Without the Law of the Excluded Middle, parts of classical mathematics apparently cannot be salvaged, and others require much more elaborate devices than in their classical treatments. (But even Brouwer contributed to areas of mathematics outside those justified by his philosophy.)

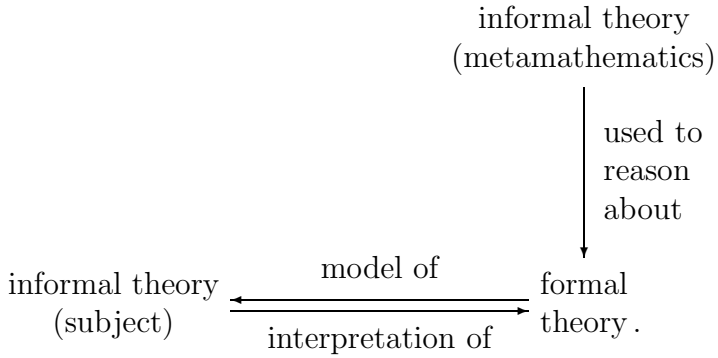
The nineteenth-century mathematicians supposed that each of their formal theories was *about* some reality outside the formalism (e.g., Frege's logic was meant to model correct reasoning, making it that much more appalling when antinomies were derived from it). At the end of the century, though, David Hilbert developed a dif-

[7] Ironically, and perhaps insightfully, the properties *predicative* and *impredicative* cannot be addressed in Russell and Whitehead's logic, because these properties pertain to properties of every type $n \in \mathbb{N}$.

[8] That is, they objected to *reifying* the infinite (§1.2.4). The objection was expressed by Gauss in 1831; by Kronecker in the 1870s and 80s; and later by Poincaré, whose criticisms tended toward the snide, made more obnoxious by his accompanying tendency to be right. (For an acute mind, philosophies of mathematics make good target practice.)

ferent approach. An axiomatic theory was to be based entirely on formal axioms and rules of deduction, and the formal consequences of the theory would be studied *independent* of whether any system of objects conforms to the theory. The existence, or multiplicity, of systems conforming to the theory was then a separate question.

When the logical antinomies surfaced, Hilbert agreed that infinite structures are not completed objects; but he did not consider difficulties with the previously proposed formalisms sufficient cause to abandon either classical mathematics (notably, analysis) or the Law of the Excluded Middle, both of which he judged too useful to give up without a fight. Instead, his impulse was to study the consequences of alternative formalizations of mathematics. He proposed specifically (in 1904) that one could *prove* the consistency of a formal theory that correctly models classical mathematics. About fifteen years later, he undertook to drive this program forward himself,[9] and a school of *formalists* developed around him. In the formalist program, a formal theory is set up that models an informal theory of interest (typically some portion of classical mathematics), and an informal theory —called the *metamathematics*— is used to reason about the formal theory and, hopefully, prove that it is consistent:



(8.1)

Conclusions about the subject theory are valid only insofar as the metamathematics uses principles whose validity is accepted beforehand; so Hilbert used substantially intuitionistic metamathematics.

If mathematics is 'what mathematicians do', then its future must be dominated by those who advocate wider mathematical investigations. Hilbert's overarching strategy retained existing mathematics and opened new frontiers for exploration, while the logicists' and intuitionists' instinct was to fall back to a secure perimeter; so the future could only belong, in this general and aphilosophical sense, to Hilbert.[10] Moreover,

[9]The evolution of Hilbert's program is described in some depth in [Za03].

[10]That is, *of the three foundational approaches* the future belonged to Hilbert; most mathematicians didn't have to take sides in the foundational dispute to get on with their work. Also keep in mind that the advantage to Hilbert lay in proliferation, not suppression: the other two philosophical positions have continued since, as have their research agendas — with metamathematics added to their toolkits. Regarding the longevity of the philosophical positions, Quine (himself a logicist) observes in "On What There Is" ([Qu61, p. 14 of 1–19]) that the three positions on the founda-

his new frontier of metamathematical investigation turned out to be so powerful that, by an acute use of it, Kurt Gödel was able to prove in 1931 that Hilbert's specific proposal (to prove classical mathematics consistent via intuitionistic metamathematics) is impossible. Gödel's Second Theorem says that for any sufficiently powerful formal mathematics $\mathcal{M}$, if $\mathcal{M}$ is consistent then $\mathcal{M}$ cannot prove itself consistent (or, put in its more confrontational form, if $\mathcal{M}$ is able to prove itself consistent then $\mathcal{M}$ is actually not consistent);[11] therefore a consistent formalization of intuitionistic mathematics cannot be used to prove its own consistency, let alone that of a classical superset.

Just at this point, with Gödel's Theorems about to make mathematical foundations a much more overtly treacherous subject, the history of lambda calculus splits off as a distinct historical thread.[12] Alonzo Church suggested in 1932 ([Chu32/33]) an alternative set of axioms for logic, with the two intents to (1) prohibit free variables in propositions, in connection with which he introduced λ-notation to explicitly bind the parameter of a propositional function;[13] and (2) avoid antinomies without completely abandoning the Law of the Excluded Middle, for which he abandoned instead *reductio ad absurdum* (if assuming p leads to an antinomy, then $\neg p$; which Brouwer considered intuitive and therefore beyond reproach). In unremittingly formalist style, he expressed the hope that if his axioms weren't consistent one might be able to fix the problem by tweaking them slightly, and also remarked that since the formal system has in itself no meaning at all, it might turn out to be useful as something other than a logic. He did tweak his axioms slightly the next year (1933); but by 1935 it had emerged that there were antinomies in the system that were not readily avoided, arising from basic elements of its structure — facilitated, in fact, by the first-class status of functions in his system.[14]

---

tions of mathematics correspond to the three medieval positions on the existence of universals (i.e., roughly, on the existence of abstractions). Quine's analogy would group Plato with Frege, Russell, and Whitehead; John Locke with Brouwer; and William of Ockham with Hilbert.

[11] For sufficiently powerful $\mathcal{M}$ (essentially, $\mathcal{M}$ encompassing integer arithmetic), one can construct a proposition A of $\mathcal{M}$ that amounts to "this proposition is unprovable". If A is proven, that would show that A is false, thus would constitute a proof of $\neg A$; if $\neg A$ is proven, that would constitute a proof of A; so if $\mathcal{M}$ is consistent, both A and $\neg A$ are unprovable. (This is Gödel's Theorem, that $\mathcal{M}$ must be either incomplete or inconsistent.) But then, a *proof* that $\mathcal{M}$ is consistent would constitute proof that A and $\neg A$ are unprovable; and a proof that "A is unprovable" is a proof of A, and a proof of A means that $\mathcal{M}$ is inconsistent. (This, however, is scarcely more than a demonstration of plausibility; for a sound informal explanation of the proof (and for the proof itself), see [Kle52, §42].)

[12] General sources for the history of lambda calculus up to about 1980 are [Ros82], [Bare84, §1.1].

[13] In fact, λ was the *only* variable-binding device in Church's logic; rather than introduce multiple binding devices, he contrived higher-order propositional functions to do universal and existential quantification independent of binding.

[14] The antinomy demonstrated in [KleRo35] is a form of the Richard Paradox, which concerns the use of an expression in some class to designate an object that, by definition, cannot be designated by expressions of that class. (A version due to G.G. Berry concerns the twenty-one syllable English expression "the least natural number not nameable in fewer than twenty-two syllables".) Naturally,

Church's remark on non-logical applications was more fruitful. While the part of the system involving only functions fostered antinomies when combined with the logical operators, he and J.B. Rosser did prove that the functional part in isolation was consistent ([ChuRo36]); and at the same time Church proposed that the functional part of the system, in its own right, is sufficient to specify all effectively computable functions ([Chu36]), broadly the Church–Turing Thesis.[15] Church then systematically developed the functional part as a model of computation in its own right, collecting his treatment into a (highly compact) 1941 book titled *The Calculi of Lambda-Conversion* ([Chu41]).

8.1.2 Logic and lambda calculus

The traditional notion of logical consistency as freedom from contradictions is useless for reasoning about the formal system of lambda calculus, because lambda calculus doesn't have a negation operator, so there is no way to formulate a contradiction. Instead, one uses a more general notion of formal consistency as the property that not every expressible proposition is provable. Consistency in this general sense is prerequisite for the lambda calculus to mean anything useful (as inconsistency would entail that all expressions mean the same thing), and also bears directly on whether lambda calculus could ever be embedded in a larger formal system —such as Church's 1932 logic— that *does* include negation, without instantly producing antinomies.

Church's 1932 system had provided five general postulates for rewriting any subterm of a proposition to produce an equivalent proposition, three of which are retained in the lambda-calculus subsystem:[16]

I. Renaming of bound variables (α-renaming). That is, $\lambda x_1.T \longrightarrow \lambda x_2.(T[x_1 \leftarrow x_2])$, where x_2 doesn't occur free in T.

granting the designators first-class status aids formulation of the antinomy. A different antinomy in combinatory logic based on Russell's Paradox was later (1942) developed by Curry, involving far fewer logical postulates than Kleene and Rosser's, and relying much more heavily on first-class functions (specifically, the Fixed Point Theorem); see [Ros82, §2].

[15]Church's 1936 paper showed that the λ-definable functions and the recursive functions are the same. Alan Turing had independently developed a mechanical notion of calculability that he also considered universal, which he subsequently showed ([Tu37]) to be equivalent to Church's functional notion. (See [Sho95].) The Church–Turing Thesis, that the class defined in all these ways is just the effectively calculable functions, may be called Church's Thesis, or Turing's Thesis, by those who believe that one or the other of them had the idea first. Turing ended up as Church's doctoral student; in fact, most of the people we're talking about at this point were Church's students, with the notable exception of Curry (who was three years older than Church, and was a student of Hilbert). Church's students included Kleene, Rosser, Turing, and later John Kemeny (co-inventor of the programming language BASIC), Hartley Rogers, Jr. (author of [Rog87], the bible of recursive function theory), and Dana Scott (co-inventor of denotational semantics, a.k.a. Scott-Strachey semantics). Regarding names for the Church–Turing thesis, we prefer any doubt in our choice to lean toward Richard Feynman's rule, "Always give them more credit than they deserve." ([Dy92].)

[16]The other two rewriting postulates concerned Church's versions of existential and universal quantification (Σ and Π).

II. Replacement of a lambda-application by substitution (β-reduction). That is, $(\lambda x.T)T' \longrightarrow T[x \leftarrow T']$.

III. Replacement of a substitution by a lambda-application (β-expansion). That is, $T[x \leftarrow T'] \longrightarrow (\lambda x.T)T'$.

A pair of terms rewritable as each other under these rules are said to be *convertible*. The basic Church–Rosser result was that there exist pairs of terms in the lambda calculus that aren't convertible; thus, if the system were extended with basis postulates making some lambda-terms provable, they wouldn't necessarily *all* become provable. (The lambda calculus itself, viewed as a subsystem of Church's 1932 logic, contains no provable propositions since all thirty seven of his basis postulates involve non-lambda operators.)

To prove their result, Church and Rosser considered just the effect of β-reduction (Rule II) on equivalence classes of terms under α-renaming (Rule I), thus replacing the problem of bounding an undirected relation (conversion) with the more structured problem of bounding a directed relation (reduction). Studying the reflexive transitive closure of the directed relation, $\longrightarrow^*$, they showed that for any T there is at most one irreducible T' such that $T \longrightarrow^* T'$, called a *normal form* T' of T; therefore, any two normal forms cannot be convertible to each other. Besides consistency, Church and Rosser's treatment also provides a suitable view of lambda calculus as a model of computation, i.e., as directing the relation from a computational query (T) to its answer (normal form T' with $T \longrightarrow^* T'$).

The Church–Rosser strategy for studying directed rewriting systems grew over the ensuing decades into a substantial subject in its own right; for a broad overview, see [Ros82].

Parallel to Church's development of lambda calculus (for which, remember, one of Church's goals had been to avoid free variables in propositions), H.B. Curry had been developing a combinatorial logic in which there were no variables at all. (E.g., [Cu29].) In the framework of lambda calculus, a *combinator* is any term with no free variables; and, as it turns out, all possible combinators can be built up (up to behavioral equivalence) out of just the two combinators $\mathsf{K} = \lambda\mathrm{x}.(\lambda\mathrm{y}.\mathrm{x})$ and $\mathsf{S} = \lambda\mathrm{x}.(\lambda\mathrm{y}.(\lambda\mathrm{z}.((\mathrm{xz})(\mathrm{yz}))))$. (For example, the identity combinator $\mathsf{I} = \lambda\mathrm{x}.\mathrm{x}$ is equivalent to $(\mathsf{SK})\mathsf{K}$.)[17] Curry's principal focus was metamathematical, studying the combinators as abstract functions, rather than logical, studying the combinators as propositional functions; so he chose to work with propositions of the form $T = T'$ for combinatorial terms T, T', rather than treating the terms themselves as potentially provable.

[17] As footnoted in §1.2, Church preferred a restricted lambda calculus, called *λI-calculus*, with the syntactic constraint that in any term $\lambda x.T$, T must contain at least one free occurrence of x. Although λI-calculus is Turing-powerful, it cannot express the K combinator, $\lambda\mathrm{x}.(\lambda\mathrm{y}.\mathrm{x})$ — which is why the calculus more commonly used today, lacking the syntactic constraint and therefore able to express K, is called *λK-calculus*.

Curry's equational approach was later absorbed into the study of lambda calculi, with $T = T'$ iff T, T' are convertible. This approach gives lambda calculus an *equational theory*, in which the propositions are the possible equations between terms, and formal equality is required to be a congruence on terms (that is, reflexive symmetric transitive and compatible). Where the Church–Rosser result, in its original presentation, only bore indirectly on consistency by guaranteeing that if one term were postulated (designated as provable) it wouldn't necessarily imply all the others, in the setting of an equational theory the Church–Rosser result says directly that the theory is consistent, i.e., that not every equation is provable.

8.2 Term-reduction systems

Although fragmentary term-reduction systems were used in Part I as an informal explanatory aid, in this part we will study complete reduction systems in depth; and this will require their precise specification, including precise notation for keeping straight the associated structures of multiple systems.[18] The totality of structures associated with a reduction system will be called, somewhat informally, a *calculus*.

Several of the main structures of a given calculus are named by regularly varied forms of a general name for the calculus, consisting of a standard base letter qualified by suffixes or subscripts. The base letter is generally chosen to identify the central operator of the calculus: λ (lambda) for calculi based on that traditional constructor, $\wr$ (vau) for calculi based on the explicit-evaluation constructor of operatives. Heuristically, suffixes are used to distinguish variation by adding something, while subscripts are used to distinguish variation by modifying something. Thus, λK-calculus adds permissible syntax to the syntactically restricted λI-calculus; $\lambda\delta$-calculi add reduction rules of a certain class called *δ-rules*; λ_v-calculus modifies the rule for applying a λ-expression. Many authors designate one particular calculus to be named without suffixes or subscripts; in [Chu41], "λ-calculus" meant "λI-calculus", while most modern authors, including the current one, use "λ-calculus" for "λK-calculus".

As stated, but not fully elaborated, in §2.3.2, the specification of a reduction system has three parts:

1. syntax, which describes the syntactic structure of terms, and assigns semantic variables to particular syntactic domains;

2. auxiliary semantic functions, which are named by semantic variables, or use semantic notations, that didn't occur in (1); and

[18] The uniform conventions presented here are designed particularly to facilitate keeping simultaneous track of many different reduction systems of widely varying kinds, including all those discussed in this chapter (even though some of them aren't treated formally in their discussion) as well as all those developed or considered in later chapters. Elements of the conventions were drawn eclectically from various sources, including [Bare84], [FeHi92], [WrFe94].

3. a set of reduction rule schemata, which are reduction relation assertions building on the semantic variables and notations from (1) and (2).

The set of terms of a calculus is named by setting its base letter in upper-case non-italic, and its suffixes in non-italic; thus, Λ for λ-calculus, Ɨ for ɨ-calculus, ΛI for λI-calculus, etc. (However, if a suffix of the calculus name is a lower-case greek letter, it is boldfaced rather than non-italic; $\lambda\delta$-calculus would have term set $\Lambda\boldsymbol{\delta}$.) The set of *closed* terms, i.e., terms with no free variables, is named by superscripting the term set name with "0"; thus, Λ^0, Ɨ^0, etc. The set of free variables of a term T is denoted $\text{FV}(T)$.

The term set is generated (usually, though not always, freely) by a context-free grammar. In specifying the grammar, each syntactic domain is given a base letter for use in naming semantic variables quantified over that domain, which doubles as the nonterminal for that domain in syntax production rules; and a verbal name, using one or more words (possibly abbreviated). The syntactic domain of terms is always called "Terms". There are two modes for specifying the domains.

- A domain may be specified by an EBNF (Extended Backus-Naur Form) production rule. The production operator is "::=", alternatives are separated with "|", and superscript "$*$" indicates zero or more repetitions of an element of a rule. The nonterminal symbol used for the domain is the base letter for semantic variables over that domain. The verbal name of the domain is given in parentheses to the right of the rule.

- A primitive domain, i.e., a domain whose members are atomic objects, usually isn't explicitly enumerated. Instead, the base letter and verbal name of the domain are designated by asserting membership of the former in the latter via "$\in$". In this symbolic-notation setting, the verbal name is merged into a single compound word, with capitalization marking the beginnings of the constituent words.

For example, the syntax for the pure λ-calculus would be

$$\begin{array}{l}\text{Syntax:}\\ \quad c \in \textit{Constants}\\ \quad x \in \textit{Variables}\\ \quad T ::= c \mid x \mid (TT) \mid (\lambda x.T) \qquad (\text{Terms})\,.\end{array} \tag{8.2}$$

Outside the specification of the syntax rules themselves, by convention when parentheses, "()", occur at the outside of a term they may be omitted; so for terms in Λ one may write "T_1T_2", "$\lambda x.T$". (However, other delimiters such as "[]" are never elided.)

To simplify the treatment of diverse calculi in this part of the dissertation, the semantic variable letter for terms is always chosen to be "T" (whereas in common

practice for λ-calculus, including practice in this dissertation other than Part II, terms use semantic variable letters M, N, and sometimes others).

A derivative syntactic domain in any calculus is that of *contexts*. Informally, a context is "a term with a hole in it". More precisely, a context has the syntactic structure of a term except that, at exactly one point in its syntax tree where a term could occur, syntactic meta-variable "$\Box$" occurs instead. The semantic variable letter for contexts is always chosen to be "C". Notation "$C[T]$" signifies the term resulting from syntactic replacement of $\Box$ by T in C.

A variable x is *captured* by a context C if for any term T with $x \in \mathrm{FV}(T)$, the free occurrences of x in T are bound in $C[T]$. (It is still possible that $x \in \mathrm{FV}(C[T])$ due to free occurrences of x in C.) The set of variables captured by C is denoted $\mathrm{CV}(C)$.

For λ-calculus,

$$C \ ::= \ \Box \mid (CT) \mid (TC) \mid (\lambda x.C) \qquad \text{(Contexts)}\,. \tag{8.3}$$

In principle, the syntax of contexts could always be left implicit, since it can be generated in a purely automatic way from the syntax of terms: the production rule for C parallels that of T, with $\Box$ as an alternative, non-recursive alternatives omitted, and recursive alternatives modified so that exactly one subexpression is a context. For clarity, though, we will often provide an explicit syntax of contexts.

The derived syntactic domain of *poly-contexts* generalizes that of contexts by allowing multiple meta-variables $\Box_k$, and multiple occurrences of each meta-variable. An m-ary poly-context (poly-context with arity m) has the syntactic structure of a term except that, at zero or more points in its syntax tree where a term could occur, some syntactic meta-variable $\Box_k$, with $1 \leq k \leq m$, occurs instead. The semantic variable letter for poly-contexts is always chosen to be "P". Notation "$P[\vec{T}]$" signifies the term resulting from syntactic replacement of each occurrence of $\Box_k$ in P by $\vec{T}(k)$, the kth component of $\vec{T}$. Notation "$ar(P)$" signifies the arity m of P, "$ar(\vec{T})$" the arity m of $\vec{T}$.

By convention, each poly-context has a fixed arity, i.e., can only be used with one certain number of operands. This arity has to be fixed by the discussion in which the poly-context occurs, since it cannot be reconstructed from the syntax of the poly-context, in which there may be zero occurrences of some meta-variables allowed by its arity. Often, the arity of a poly-context is fixed indirectly, by requiring that it be applicable to some particular vector of operands.

To facilitate the use of poly-contexts, vectors may be specified algebraically via a summation notation. For semantic variable k over integers, and semantic expression $p(k)$ in variable k, notation "$\sum_k p(k)$" signifies the term vector whose kth element is $p(k)$, for k ranging from 1 to the largest consecutive integer for which $p(k)$ is defined. For example, given vector of functions $\vec{f} \subseteq (\mathit{Terms} \to \mathit{Terms})$ and vector of terms

$\vec{T}$, the vector constructed by applying elements of $\vec{f}$ to corresponding elements of $\vec{T}$ would be $\sum_k (\vec{f}(k))(\vec{T}(k))$ with arity $min(ar(\vec{f}), ar(\vec{T}))$.

The word *Variable* will be used in a syntactic domain name only when that domain is managed via substitution. Syntactic domain name *Symbols*, and semantic variable letter s, are used in $\mathfrak{l}$-calculi for syntactic variables (in the sense of §2.1) that are managed via environments. Variable substitution is provided as an auxiliary semantic function using some variant of notation "$(T_1[x \leftarrow T_2])$". In λ-calculus,

$$\begin{aligned} c[x \leftarrow T] &= c \\ x_1[x_2 \leftarrow T] &= \begin{cases} T & \text{if } x_1, x_2 \text{ are syntactically identical} \\ x_1 & \text{otherwise} \end{cases} \\ (T_1 T_2)[x \leftarrow T_3] &= (T_1[x \leftarrow T_3])(T_2[x \leftarrow T_3]) \\ (\lambda x_1.T_1)[x_2 \leftarrow T_2] &= \lambda x_3.((T_1[x_1 \leftarrow x_3])[x_2 \leftarrow T_2]) \\ &\quad \text{where } x_3 \notin \{x_2\} \cup \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)\,. \end{aligned} \tag{8.4}$$

Also, binary reduction relations are understood to be relations between equivalence classes under α-renaming of substitutionally managed variables; that is, equivalence classes of the congruence generated, in the case of λ-calculus, by

$$\lambda x_1.T \;\sim\; \lambda x_2.(T[x_1 \leftarrow x_2]) \qquad \text{where } x_2 \notin \mathrm{FV}(T)\,. \tag{8.5}$$

In effect, terms are treated as syntactically identical when they are congruent under this relation — a treatment that must be respected by any auxiliary semantic functions, including substitution. Note that the congruence only relates terms: contexts are not terms and so are not subject to α-renaming (though a term generated by replacement into a context is subject to α-renaming).

Syntactic equivalence (up to however much α-renaming the formal system supports) is designated by operator "$\equiv_\alpha$", to avoid confusion with the various uses of "$=$" associated with equational theories (which will be discussed momentarily).

The binary relation directly enumerated by the reduction rule schemata of a calculus (that is, the set of reduction rules enumerated by systematically filling in all permissible combinations of syntax for the semantic variables in each schema[19]) is given a name using a different base letter than the calculus itself. The relation is named by this base letter in boldface, often using the same qualifiers as its associated calculus. In λ-calculus, the base letter for the reduction relation is β, so the name of the relation is $\boldsymbol{\beta}$; while the λ_v-calculus has relation $\boldsymbol{\beta}_v$. By convention, the enumerated reduction relation of a calculus uses base letter β even if the calculus itself

[19]This use of the term *enumerated* emphasizes that, under reasonable assumptions, the binary relation can be enumerated by diagonalization. The reasonable assumptions are that each semantic variable is universally quantified over an enumerable (a.k.a. countable) syntactic domain, and that any additional constraints are decidable.

doesn't use base letter λ; $\mathfrak{l}$-calculi qualify it with a subscript $\mathfrak{l}$, so the reduction relation of pure $\mathfrak{l}$-calculus would be $\boldsymbol{\beta}_{\mathfrak{l}}$. (There will be little occasion here for reduction relations using other base letters.)

When stating that the enumerated reduction relation holds between terms, infix operator "$\longrightarrow$" is superscripted to designate the relation. If the name of the relation doesn't itself involve subscripts, infix "$\longrightarrow$" is superscripted by the name of the relation (not boldfaced). When the relation name has subscripts, some infix operator superscript is chosen on a case-by-case basis; typically, the relation's base letter is β, and the infix superscript omits the β and concatenates its subscripts, as $\longrightarrow^{v}$ for $\boldsymbol{\beta}_{v}$, or $\longrightarrow^{\mathfrak{l}}$ for $\boldsymbol{\beta}_{\mathfrak{l}}$.

Reduction rule schemata are specified using the unqualified reduction operator "$\longrightarrow$". For example, the schema for pure λ-calculus is

$$(\lambda x.T_1)T_2 \quad \longrightarrow \quad T_1[x \leftarrow T_2]\,. \tag{8.6}$$

This particular schema is traditionally called the *β-rule* (related to the names *β-reduction* and *β-expansion*). A few other schemata have their own traditional names, such as

$$\lambda x.(Tx) \quad \longrightarrow \quad T \qquad \text{if } x \notin \mathrm{FV}(T)\,, \tag{8.7}$$

called the *η-rule*, whose enumerated reduction relation is called $\boldsymbol{\eta}$.

For the principal reduction relation (or, step relation) of a calculus, the infix operator of the enumerated binary relation is changed by shifting its superscript to a subscript; thus, $\longrightarrow^{v}$ becomes $\longrightarrow_{v}$, etc. In most calculi, the step relation is the compatible closure of the enumerated relation, and this will be assumed unless otherwise stated for a particular calculus. The reflexive closure of the step relation is then designated by a superscript "?", the transitive closure by a superscript "+", and the reflexive transitive closure by a superscript "$*$"; thus, $\longrightarrow_{v}^{?}$, $\longrightarrow_{v}^{+}$, $\longrightarrow_{v}^{*}$.

In general discussion, occasionally one needs to distinguish an anonymous enumerated reduction relation from its compatible closure, without naming an associated calculus. The infix operator is then super/subscripted by "$\bullet$"; thus, $\longrightarrow^{\bullet}$ for the anonymous enumerated relation, $\longrightarrow_{\bullet}$ for its compatible closure.

The equational theory of a calculus varies the name of the calculus by boldfacing the base letter and suffixes. Thus, the equational theory of λI-calculus is $\boldsymbol{\lambda I}$, of λ_{v}-calculus is $\boldsymbol{\lambda}_{v}$, etc. In calculi where the step relation is compatible, the equational theory is simply the congruence generated by the step relation; if the step relation isn't compatible, the equational theory is explicitly specified. The equality relation determined by the theory uses infix operator "$=$" with the same subscripts as the step relation; thus, $=_{v}$, $=_{\mathfrak{l}}$.[20]

[20]We are defining $=_{\bullet}$ to be a semantic symbol, not a syntactic one; thus, $T_1 =_{\bullet} T_2$ is a metamathematical assertion, not a formal one. Another important metamathematical notation in the literature is $\mathcal{T} \vdash T_1 = T_2$, asserting that $T_1 = T_2$ belongs to theory $\mathcal{T}$ (an instance of metamathemat-

When a formal system —be it an equational theory or a reduction relation— must be specified by an inductive process other than compatibility, its inductive postulates may be stated via metamathematical notation

$$\frac{A_1 \ldots A_n}{B}\,, \tag{8.8}$$

where A_k, B are propositions of the system, denoting that if all the A_k are provable, then B is provable. For example, one might have a parallel reduction relation with

$$\frac{T_1 \longrightarrow T_1' \quad T_2 \longrightarrow T_2'}{T_1 T_2 \longrightarrow T_1' T_2'}\,. \tag{8.9}$$

Note that the A_k are formal propositions, not semantic assertions; if any semantic restrictions must be placed on the postulate, they are presented separately, maintaining a crisp distinction between object theory and metamathematics. Thus, rather than mixed (and therefore, by our conventions, invalid) notation

$$\frac{x \notin \mathrm{FV}(T)}{\lambda x.(Tx) \longrightarrow T}\,, \tag{8.10}$$

one would write

$$\lambda x.(Tx) \;\longrightarrow\; T \qquad \text{if } x \notin \mathrm{FV}(T)\,. \tag{8.11}$$

Here is a complete reduction system specification for λ-calculus:

λ-calculus.

Syntax:

$c \in$ *Constants*
$x \in$ *Variables*

$$\begin{array}{lll} T & ::= & c \mid x \mid (TT) \mid (\lambda x.T) \qquad \text{(Terms)} \\ C & ::= & \Box \mid (CT) \mid (TC) \mid (\lambda x.C) \qquad \text{(Contexts)} \end{array}$$

Auxiliary functions:

$$\begin{array}{rcl} c[x \leftarrow T] & = & c \\ x_1[x_2 \leftarrow T] & = & \begin{cases} T & \text{if } x_1 = x_2 \\ x_1 & \text{otherwise} \end{cases} \\ (T_1 T_2)[x \leftarrow T_3] & = & (T_1[x \leftarrow T_3])(T_2[x \leftarrow T_3]) \\ (\lambda x_1.T_1)[x_2 \leftarrow T_2] & = & \lambda x_3.((T_1[x_1 \leftarrow x_3])[x_2 \leftarrow T_2]) \\ & & \quad \text{where } x_3 \notin \{x_2\} \cup \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2) \end{array} \tag{8.12}$$

Schemata:

$$(\lambda x.T_1)T_2 \;\longrightarrow\; T_1[x \leftarrow T_2]\,.$$

ical notation $\mathcal{F} \vdash F$ asserting that formula F is provable in formal system $\mathcal{F}$). Our notation is more succinct, e.g. "$T_1 =_v T_2$" rather than "$\boldsymbol{\lambda}_v \vdash T_1 = T_2$". Occasionally, authors may write $=_\bullet$ for the formal operator, thus "$\boldsymbol{\lambda}_v \vdash T_1 =_v T_2$" (e.g., [FeFrKoDu87]); but under our notational conventions this would be a confusion between formal theory and metamathematics.

A frequently useful class of calculi are the *λδ-calculi*. These are formed from λ-calculus —or from most variants of λ-calculus— by adding so-called *δ-rules*, which are reduction rules of the form $(T_1 T_2) \longrightarrow T'$ where T' is closed and the T_k are closed normal forms. (See [Bare84, §15.3], [Plo75, §3].) Rules of this form are convenient when studying programming languages because they model the action of primitive functions; and they are admissible theoretically because, as long as no two δ-rules have the same left side and different right sides, their addition to a λ-like calculus preserves Church–Rosser-ness. (That is to say, in essence, that a δ-redex in a larger term T never disrupts, nor is disrupted by, another redex in T.) Since most results about the calculus are relative to whatever δ-rules are chosen, it is usually convenient to encapsulate the individual input-output pairs into a single semantic function which is not specified in detail, and invoke the semantic function in a single schema. The syntactic domain of constants may also be partitioned at the same time into several subdomains. For example, one might amend λ-calculus Specification (8.12) by

Amendments for a λδ-calculus.
Syntax:
$n \in \mathit{Numbers}$
$p \in \mathit{PrimFns}$
$c ::= n \mid p$ (Constants) (8.13)
Auxiliary functions:
$\delta\colon \mathit{PrimFns} \times \mathit{Constants} \xrightarrow[p]{} \Lambda^0$
Schemata:
$(pc) \longrightarrow \delta(p,c)$ if $\delta(p,c)$ is defined

(where operator $\xrightarrow[p]{}$ signifies a partial function). The schema limits δ-rule operands to constants, not just to closed normal forms. The programming languages we consider often do not have quite the same notion of halting as the calculi that model them, so that a non-constant may be irreducible in the programming language but reducible in the calculus. (E.g., SECD versus λ_v-calculus, below in §8.3.2.) It is therefore convenient for modeling by λ-calculus that we restrict δ-rule operands to the syntactic domain of constants, which by nature are irreducible in both systems. (This is also convenient for specification since the syntactic domains are equipped with semantic-variable letters.)

8.3 Computation and lambda calculi

8.3.1 General considerations

When a calculus is used to study some aspect of functions, syntactically distinct terms represent intensionally distinct algorithms. Each equation means that two intensionally distinct algorithms are extensionally indistinguishable, i.e., they have the same effects. For the calculus to fully model the domain, there should be at

least one term representing each computable extensional function of interest; and for the calculus to be a useful tool for study, there should be many equations between intensionally distinct algorithms for each function.[21]

Neither Church–Rosser-ness nor compatibility of the step relation guarantees a strong theory. Church–Rosser-ness guarantees that all alternative reductions of a term T (that is, terms T' such that $T \longrightarrow_\bullet T'$) are equal; but it is entirely possible to set up a reduction relation in which each T is reducible in at most one way, in which case Church–Rosser-ness is trivial. Compatibility says that every nontrivial context C engenders new reduction rules, hence new equations ($T \longrightarrow_\bullet T'$ implies $C[T] \longrightarrow_\bullet C[T']$, $T =_\bullet T'$ implies $C[T] =_\bullet C[T']$); but it is entirely possible to set up a context-free syntax in which terms are non-recursive, i.e. a term cannot occur within a larger term, so that the only context is the trivial one, $C = \Box$. However, both properties *facilitate* strong theory. If there are nontrivial contexts, compatibility induces equational strength; if a term can have nonoverlapping subterms, compatibility induces alternative reductions; and if a term has alternative reductions, Church–Rosser-ness induces equational strength.

When the particular focus of study is simply *computation*, there is no need for functions to interact syntactically with their operands, which is to say, no need for explicit evaluation. So, one may reasonably design the calculus around implicit evaluation, decoupling operand processing from the rest of computation and thereby permitting a stronger theory. On the other hand, when the particular focus of study is *abstraction* (per §1.1), the objects to be modeled are abstractive languages; and where a computational function maps a semantic object to a semantic object (say, an integer to an integer), an abstractive language maps a program text to another abstractive language. So, the syntax of each function operand (being part of the program text) potentially matters, which is to say that the function is operative.[22] The portion of the theory dealing with operatives is necessarily weaker than that

[21] Maximally, the equational theory would be *Hilbert Post complete*, meaning that, for every equation e that isn't in the theory (isn't provable), postulating e would render the theory inconsistent. The theory $\boldsymbol{\lambda\eta}$ (closure of relation $\boldsymbol{\beta} \cup \boldsymbol{\eta}$), restricted to the terms in Λ that have normal forms, is Hilbert Post complete ([Bare84, §2.1]).

[22] Although abstractive languages have just made a cameo appearance, the discussion in which they have appeared is about abstractive behavior of *functions*, that is, parametric artifacts within a program. The objects of our calculi —terms, and subexpressions within terms— are naturally understood to model programs and program objects (such as functions); calculi of the kinds considered in this dissertation do not directly model abstractive languages, and therefore are not very well suited to support a theory of abstractive power. As an attempt to shift the focus toward abstractive languages, one might try to model an abstractive language as a *context*, which takes a term as "input", and does interact with that input syntactically; and this has in fact been done, in [Mi93]; but the corresponding "output" is a term, not a context, so at best this approach models a single abstractive step, with no possibility of further steps to follow. Alternatively, [Shu08] proposes to address abstractive power through a different class of formal systems, resembling abstract state machines in which each state models an abstractive language, and each transition between states is labeled by a program text.

dealing with applicatives — although, if the theory is set up so that applicatives are clearly identified (as by a *wrap* device), the purely applicative subtheory may retain its strength, and lend additional strength to partially constrained impure cases.

Although compatible Church–Rosser calculi may be highly useful models for studying various computational facilities, they have not been historically successful as primary definitions of those facilities. This difficulty showed itself immediately: Although Church came to believe that λ-definability characterized effective calculability, he was unable for several years to convince Kleene (whose research into their capabilities had led Church to his belief), nor Gödel. By contrast, Turing's automaton candidate for effective calculability was readily persuasive to Gödel, and to others; λ-calculus gained greatly in credibility from being proved equi-powerful to Turing machines. ([Sho95].)

8.3.2 Semantics

In 1960, Christopher Strachey hired a technical assistant, Peter Landin, whom he encouraged to spend part of his time on a side project of investigating the theory of programming languages (as a result of which, the job Landin was hired to do received too little of Landin's time ([Cam85, §4], [La00])). From his theoretical study, Landin proposed in 1964 an automaton, the *SECD machine*,[23] for performing reduction of terms in a λ-calculus (extended somewhat for data and primitive operations). The computational facilities defined by the SECD machine are typical of many programming languages; they are not, however, consistent with the facilities of Church's λ-calculus. Particularly, SECD doesn't reduce the body of an applicative until and unless the applicative is called (so SECD halts on $\lambda x.T$ even if it has no normal form), and SECD practices eager argument evaluation (call-by-value), fully reducing the argument of an applicative combination before performing the call (so SECD doesn't halt on $(\lambda x.T)T'$, even if it has a normal form, unless it also halts on T').[24]

In 1975 ([Plo75]), Gordon Plotkin proposed a general strategy to redress the definition-versus-study problem, applying his strategy to the particular case of the SECD machine. In broad outline, his approach would begin with an intuitively validated *operational semantics* as primary formal definition of a subject facility, then custom-build a compatible calculus sufficiently similar to the operational semantics that one could prove suitable correspondences in both directions (semantics to calculus, calculus to semantics).

[23]The original paper on the SECD machine is [La64]. *SECD* is an initialism for the four components of each configuration of the machine, called a *dump*: *Stack, Environment, Control, Dump.*

[24]Remember that in λ-calculus (but not λI-calculus), $\lambda x.T$ may ignore its argument. In λI-calculus, there would be no difference between call-by-value and call-by-name unless side-effects were introduced. This could be construed to imply that the K combinator is side-effect-ful. However, our treatment of impure $\mathfrak{l}$-calculi in Chapter 10 will favor a more conventional view, that call-by-value causes argument nontermination to become a quasi-side-effect.

For the operational semantics, Plotkin proposed to use a binary relation between computational configurations of an abstract machine. He refined his views on this point in [Plo81]. Although one *could* use as operational semantics the step relation of a low-level mechanical automaton (such as the SECD machine that he necessarily started with in his 1975 paper), he argued that this approach would retard intuitive validation of the formal definition (which is critical since, as the *primary* formal definition, it could only be validated intuitively). Instead, he recommended that low-level automaton bookkeeping features such as states be omitted, and computation on large configuration structures be built up inductively from computation on smaller structures. One might use this technique to define a binary relation directly from initial configuration to final configuration of the abstract machine, and in fact Plotkin did this in the 1975 paper; but as later refined, his strategy was to define instead a computation step relation, so that the relation would remain primitive.

The two variants of his strategy, in which the computation relation either maps initial to final configuration, or takes primitive steps, are modernly called *big-step operational semantics* and *small-step operational semantics*. Big-step operational semantics was later developed as a strategy for type theory by Mads Tofte (in his doctoral dissertation, [To88]). The small-step approach affords a closer correspondence with the reduction step of a calculus, and we will prefer it here.

The totality of structures associated with an operational semantics will be named $\bullet$-*semantics* to distinguish it from the corresponding $\bullet$-*calculus*; here, λ_v-semantics corresponding to λ_v-calculus. The computation step relation in operational semantics will be designated by infix operator "$\longmapsto$", subscripted similarly to the step relation of the corresponding calculus; in this case, $\longmapsto_v$. In small-step semantics, notation $\mathrm{eval}_\bullet(T_1) = T_2$ means that $T_1 \longmapsto_\bullet^* T_2$ and T_2 is irreducible; in big-step semantics, $\mathrm{eval}_\bullet(T_1) = T_2$ is synonymous with $T_1 \longmapsto_\bullet T_2$. For all the semantics in this dissertation, $\longmapsto_\bullet$ will be unambiguous, i.e., to each term T_1 there will be at most one term T_2 with $T_1 \longmapsto_\bullet T_2$; therefore, $\mathrm{eval}_\bullet$ will always be a partial function.

For perspective, following are the postulates of Plotkin's λ_v-semantics, as he recast it from its low-level automaton version (itself a slight simplification of Landin's automaton). He assigned an integer "time" to each relation, for later use in his treatment of the correspondence with his calculus, providing in effect a clue to the otherwise absent small-step structure of the computation; here we denote the time by stacking it above the relation operator.

λ_v-semantics.
Postulates:

$$c \overset{1}{\longmapsto} c$$

$$\lambda x.T \overset{1}{\longmapsto} \lambda x.T$$

$$\frac{T_1 \overset{t}{\longmapsto} \lambda x.T_2 \quad T_1' \overset{t'}{\longmapsto} T_2' \quad T_2[x \leftarrow T_2'] \overset{t''}{\longmapsto} T_3}{T_1 T_1' \overset{t+t'+t''+1}{\longmapsto} T_3} \tag{8.14}$$

$$\frac{T \overset{t}{\longmapsto} c \quad T' \overset{t'}{\longmapsto} c'}{TT' \overset{t+t'+1}{\longmapsto} \delta(c,c')} \quad \text{if } \delta(c,c') \text{ is defined}.$$

The operational semantics and calculus can't be isomorphic (or why bother with the calculus at all?), but need to correspond closely enough that one can learn about the semantics by studying the calculus. Plotkin proved that

- every computation in the semantics is a reduction in the calculus; and
- every equation in the calculus implies that the terms are *operationally equivalent* in the semantics, i.e., that interchanging them as subterms of a larger term won't change halting behavior, nor the result if it is an observable.

The operational equivalence relation uses infix operator "$\simeq$" with the subscripts of the equality of the calculus. Precisely, $T_1 \simeq_\bullet T_2$ iff for all closed $C[T_1]$, closed $C[T_2]$, and observable T', $\mathrm{eval}_\bullet(C[T_1])$ is defined iff $\mathrm{eval}_\bullet(C[T_2])$ is defined, and $\mathrm{eval}_\bullet(C[T_1]) = T'$ iff $\mathrm{eval}_\bullet(C[T_2]) = T'$. A suitable definition of "observable" is provided as a parameter of the correspondence; typically, the observables are a syntactic domain called *Constants* (as above in (8.12) and (8.13)). Plotkin used the constants as observables, and his second correspondence result says exactly that $=_v \subseteq \simeq_v$.

These general correspondence criteria allow that, as long as the calculus does everything that the semantics does, and doesn't violate the semantics *publicly* (i.e., observably), it can do as it likes in private. In particular, the semantics only has to complete a reduction when the result is observable — which, in the case of Plotkin's semantics for SECD, (8.14), means that computation can return an unnormalized $\lambda x.T$ provided $\lambda x.T$ will itself behave appropriately when applied.[25]

[25]Observable behavior is not only of interest when studying programming languages. Essentially the same notion occurs in the theoretical pursuit of λ-calculus, under the name *solvability*: a term $T \in \Lambda^0$ is solvable if there exist $T_1, \ldots, T_n \in \Lambda$ such that $(\ldots((TT_1)T_2)\ldots T_n) = \mathsf{I}$. ($\mathsf{I} = \lambda x.x.$) Solvability is key to Barendregt's advocacy of λK-calculus over Church's λI-calculus. Church wanted to treat every unnormalizable term as being meaningless ([Chu41, §17]), but setting all such terms equal in $\boldsymbol{\lambda K}$ would render the theory inconsistent. Barendregt argued that unsolvability is the proper criterion for meaninglessness in $\boldsymbol{\lambda K}$ ([Bare84, §2.2]).

Plotkin's λ_v-calculus differs from ordinary λ-calculus only by adding a syntactic constraint on the operand in the β-rule:

$$\begin{array}{l}\text{Amendment for } \lambda_v\text{-calculus.}\\ \text{Schemata:}\\ \qquad (\lambda x.T_1)T_2 \;\longrightarrow\; T_1[x \leftarrow T_2] \qquad \text{if } T_2 \text{ is not of the form } T_3T_4\,.\end{array} \tag{8.15}$$

Note that, although one might casually refer to λ_v-calculus as "call-by-value" or "eager-argument-evaluation" λ-calculus, whether the syntactic constraint in (8.15) really qualifies as eager argument evaluation depends on one's notion of evaluation. It does qualify when correlated with Plotkin's λ_v-semantics, (8.14), exactly because that semantics doesn't reduce the body of an applicative; but one might imagine an operational semantics in which applicative bodies are reduced as far as possible, and under that semantics λ_v-calculus would be only intermediate between eager and lazy. A call-by-value calculus under that stronger notion of evaluation would further restrict the β-rule so that T_2 could have the form $\lambda x.T_3$ only if T_3 is irreducible; and that calculus would have a weaker equational theory than $\boldsymbol{\lambda}_v$, since its reduction relation would have less flexibility to choose alternative orders of operations.[26]

Plotkin's strategy also has one other major element: he proposes that the correspondences between semantics and calculus should be *mediated* by a standardization theorem.

The Standardization Theorem for λ-calculus, due to Curry and Feys,[27] says that if $T_1 \longrightarrow_\beta^* T_2$, then there is a reduction from T_1 to T_2 that performs its individual reduction steps in a canonical order (a.k.a. *normal* order, a.k.a. *standard* order; see [Bare84, §11.4]). Because there is a standard-order reduction iff there is any reduction at all, when theorem-proving one can often restrict one's attention to standard-order reductions without (thanks to the Standardization Theorem) loss of generality, considerably simplifying the task of proof. Plotkin realized that, in attempting to prove the correspondences between an operational semantics and calculus, the task would

[26]If one further (or instead) restricted β-reduction to irreducible arguments, one would also have to resolve a technical difficulty: naively restricting the β-rule in this way destroys Church–Rosser-ness. As noted in [Plo75, §4], the problem arises because substitution does not preserve the property of irreducibility; i.e., $T_1[x \leftarrow T_2]$ may be reducible even though T_1, T_2 are both irreducible. Hence, a combination $(\lambda x'.T_3)T_1$ subject to β-reduction may cease to be β-reducible following a substitution $((\lambda x'.T_3)T_1)[x \leftarrow T_2]$. Church–Rosser-ness would then fail on the following example (adapted from Plotkin):

$$\begin{array}{lll}(\lambda\text{x.}((\lambda\text{y.z})(\lambda\text{w.}(\text{x}(\lambda\text{x.}(\text{xx})))))(\lambda\text{x.}(\text{xx})) & \longrightarrow_\bullet & (\lambda\text{x.z})(\lambda\text{x.}(\text{xx})) \;\longrightarrow_\bullet\; \text{z}\\ (\lambda\text{x.}((\lambda\text{y.z})(\lambda\text{w.}(\text{x}(\lambda\text{x.}(\text{xx})))))(\lambda\text{x.}(\text{xx})) & \longrightarrow_\bullet & (\lambda\text{y.z})(\lambda\text{w.}((\lambda\text{x.}(\text{xx}))(\lambda\text{x.}(\text{xx}))))\,.\end{array}$$

The latter reduct cannot be reduced to z under the restricted β-rule, because its operand λw.((λx.(xx))(λx.(xx))) has no normal form.

[27][Bare84, §11.4] cites H.B. Curry and R. Feys, *Combinatory Logic, Vol. I*, Studies in Logic 65 (North-Holland, Amsterdam), 1958. The latter volume, "Curry and Feys", was the standard reference on λ-calculus in the 1960s and 70s, prior to the 1981 appearance of Barendregt's *The Lambda Calculus* (of which [Bare84] is the revised edition).

be tremendously simplified if one had first identified, and proven a standardization theorem for, a standard order of reduction coinciding closely with the operational semantics.[28] A small-step operational semantics has an evident advantage in this regard, since it should already largely determine the corresponding standard order of reduction.

8.3.3 Imperative semantics

In the late 1980s, Matthias Felleisen applied Plotkin's strategy to two language facilities that are traditionally considered imperative: first-class continuations (*sequential control*), and mutable data structures (*sequential state*). Just as Plotkin had introduced flexibility into the semantics–calculus correspondences in modeling SECD by a calculus, so Felleisen further weakened Plotkin's criteria to accommodate the imperative behaviors of his calculi for imperative control/state. In fact, after successfully constructing a variant λ-calculus by weakening one criterion,[29] he went on to try two alternative ways of modifying Plotkin's strategy to model the same facilities.

The common problem of imperative control and state is that whenever the imperative facility is invoked (by transferring control, or by mutating state), the consequences of the invocation aren't local to the immediate redex: they may have, in general, global consequences distributed across the entire term in which the redex occurs; and Felleisen constructed his calculi on the assumption that the consequences were in fact global. The notion of global consequences is, on the face of it, non-compatible.

8.3.3.1 Imperative control

We first consider Felleisen's 1987 operational semantics of first-class continuations, $\lambda_v C$-semantics (per [FeFrKoDu87]). Syntactically, he added to Λ two new operators, $\mathcal{C}$ and $\mathcal{A}$ (mnemonically, *Capture continuation* and *Abort*):

$$T \ ::= \ c \mid x \mid (TT) \mid (\lambda x.T) \mid (\mathcal{C}T) \mid (\mathcal{A}T) \qquad \text{(Terms)}\,. \tag{8.16}$$

The treatment of this imperative language involves repeated inductions over a selected subset of the syntax structure; so Felleisen contrived to specify this subset just once,

[28] Standardization in Plotkin's treatment constrains order of reduction, so that the terminology "standard order of reduction" offers useful intuition — but with the caveat that, unlike Curry and Feys' notion of standardization, Plotkin's does not always *uniquely* determine order of reduction. The same will be true of the generalization of his approach here, late in Chapter 13.

[29] The basic work occurred in his doctoral dissertation, [Fe87]. The treatments here are based on [FeFrKoDu87] and [FeFr89].

in the form of a restricted syntactic domain of contexts:

$$\begin{array}{lll} V & ::= & c \mid x \mid (\lambda x.T) \qquad\qquad\qquad\qquad\quad \text{(Values)} \\ T & ::= & V \mid (TT) \mid (\mathcal{C}T) \mid (\mathcal{A}T) \qquad\qquad\quad \text{(Terms)} \\ C & ::= & \Box \mid (CT) \mid (TC) \mid (\lambda x.C) \\ & & \qquad\qquad \mid (\mathcal{C}C) \mid (\mathcal{A}C) \qquad\quad \text{(Contexts)} \\ E & ::= & \Box \mid (ET) \mid (VE) \qquad\qquad\qquad\quad \text{(Evaluation contexts)}\,. \end{array} \tag{8.17}$$

Using evaluation contexts, the small-step semantics for SECD-style λ become simply

$$E[(\lambda x.T)V] \;\longmapsto\; E[T[x \leftarrow V]]\,, \tag{8.18}$$

while those for $\mathcal{A}$ and $\mathcal{C}$ are

$$\begin{array}{lll} E[\mathcal{A}T] & \longmapsto & T \\ E[\mathcal{C}T] & \longmapsto & T(\lambda x.(\mathcal{A}(E[x]))) \qquad \text{where } x \text{ doesn't occur in } E\,. \end{array} \tag{8.19}$$

Both $\mathcal{A}$ and $\mathcal{C}$ remove the context E from around the redex, while $\mathcal{C}$ also packages E into a combiner and passes it to T.

These semantics are based on a network of mutually supporting assumptions, entirely plausible when viewed from within, but (like the reflection/implicit-evaluation association discussed in §1.2.4) on careful analysis, unnecessary. Here, the supporting network consists largely of the following four assumptions:

(1) All continuations are bounded by termination; that is, they specify future computation just until arrival at a final result. Continuations were originally conceived as a way to represent the future of a computation *as a function* ([Rey93]), and within its domain a function always terminates. However, the purpose of computation is not necessarily to terminate with a result (although traditionally this has been assumed; see [GoSmAtSo04]); and continuations in practical programming are unlike functions in that continuations have input but, from the program's perspective, no output. (Captured continuations in Kernel are, in fact, not combiners ([Shu09, §7 (Continuations)]).)

(2) Continuation capture gives the capturer complete control over all future computation up to termination. This is evidently predicated on the existence of a unique termination event. (But, as mentioned in §5.2.3, continuation capture in Kernel does not necessarily give the capturer complete control over the future of computation.)

(3) Continuation capture is performed *eagerly*. That is, redex $\mathcal{C}T$ is exercised at the first moment its context becomes an evaluation context (the moment $C[\mathcal{C}T]$ has the form $E[\mathcal{C}T]$). This is not how Scheme continuation-capture works; there the analog of $\mathcal{C}$ is a procedure `call/cc`, which, being a procedure, is call-by-value so takes no action until after argument evaluation. Superficially, call-by-value behavior would appear in the operational semantics as making $\mathcal{C}E$ an evaluation context and restricting the $\mathcal{C}$ schema to value operands:

$$\begin{array}{lll} E & ::= & \Box \mid (ET) \mid (VE) \mid (\mathcal{C}E) \qquad \text{(Evaluation contexts)} \\ E[\mathcal{C}V] & \longmapsto & V(\lambda x.(\mathcal{A}(E[x]))) \qquad \text{where } x \text{ doesn't occur in } E\,. \end{array} \tag{8.20}$$

(Call-by-value $\mathcal{A}$ could be articulated similarly.) However, even if continuation capture is thus postponed, when it does occur it will still be eager in the sense that the entire evaluation context E is packaged into a function all at once; there is nothing to suggest lazily packaging up just that part of the context that will actually have to be restored when the continuation is invoked, because evidently *all* of the context will have to be restored when the continuation is invoked. Lazy packaging of this kind might be suggested if $\mathcal{C}$ did not remove its evaluation context; that is, if

$$E[\mathcal{C}T] \;\longmapsto\; E[T(\lambda x.(\mathcal{A}(E[x])))] \qquad \text{where } x \text{ doesn't occur in } E \tag{8.21}$$

(which is, in fact, how `call/cc` works). One might then imagine a further variant of the formalism in which, rather than putting all of E into the captured continuation at once, one would lazily add surrounding context to it as it is passed outward through that context (as an upward funarg; cf. §3.3.2ff), keeping the continuation-handling as local as possible within the overall term and avoiding reference to a unique outermost continuation (i.e., termination event).[30] However, since $\mathcal{C}$ does remove its context E, there is no redundancy between captured and surrounding contexts to suggest lazy context capture.

(4) Continuation-capture is a *global* event. This assumption is implicit in much of the above (termination, complete control, removal of context), and is the particular point with which Felleisen's three alternative treatments of continuations attempted to cope.

Felleisen's first approach was to formulate a compatible calculus for *almost* all of the semantics, and then add a minimal number of non-compatible "computation rule" schemata to provide the base case for outward propagation of information from the point of call ($\mathcal{C}T$ or $\mathcal{A}T$). To distinguish the non-compatible overall relation from its compatible subset, he used a different infix operator, $\triangleright$. Since $\triangleright_{vc}$ isn't compatible, it can't be Church–Rosser (which by definition requires compatibility); but, postulating $\longrightarrow_{vc}^{*} \subseteq \triangleright_{vc}$, he required of $\triangleright_{vc}$ that for all T_1, T_2, T_3, if $T_1 \triangleright_{vc} T_2$ and $T_1 \triangleright_{vc} T_3$ then there exists T_4 such that $T_2 \triangleright_{vc} T_4$ and $T_3 \triangleright_{vc} T_4$ (the so-called *diamond property*, just the condition that $\longrightarrow_{\bullet}^{*}$ has to satisfy in order for $\longrightarrow^{\bullet}$ to be Church–Rosser). He could therefore reason most of the time using traditionally well-behaved $\longrightarrow_{vc}^{*}$, and the rest of the time with the incompatible but still somewhat well-behaved $\triangleright_{vc}^{*}$. The smallest equivalence containing $\triangleright_{vc}$ he denoted $=_{vc}^{\triangleright}$; but $=_{vc}^{\triangleright}$ isn't compatible, so its theory $\boldsymbol{\lambda}_v\boldsymbol{C}^{\triangleright}$ (unlike the theory $\boldsymbol{\lambda}_v\boldsymbol{C}$ of $=_{vc}$) isn't really what one understands as an "equational" theory.

The rule schemata for the inductive handling of continuations are

[30] Lazy capture of surrounding context will be supported by $\wr C$-calculus, in Chapter 11.

$$
\begin{array}{rcl}
(\mathcal{A}T_1)T_2 & \longrightarrow & \mathcal{A}T_1 \\
V(\mathcal{A}T) & \longrightarrow & \mathcal{A}T \\
(\mathcal{C}T_1)T_2 & \longrightarrow & \mathcal{C}(\lambda x_1.(T_1(\lambda x_2.(\mathcal{A}(x_1(x_2T_2)))))) \\
 & & \quad \text{where } x_1 \notin \mathrm{FV}(T_1) \cup \{x_2\} \text{ and } x_1, x_2 \notin \mathrm{FV}(T_2) \\
V(\mathcal{C}T) & \longrightarrow & \mathcal{C}(\lambda x_1.(T(\lambda x_2.(\mathcal{A}(x_1(Vx_2)))))) \\
 & & \quad \text{where } x_1 \notin \mathrm{FV}(T) \cup \{x_2\} \text{ and } x_1, x_2 \notin \mathrm{FV}(V)\,,
\end{array}
\tag{8.22}
$$

and the computation rule schemata for the base cases are

$$
\begin{array}{rcl}
\mathcal{A}T & \triangleright & T \\
\mathcal{C}T & \triangleright & T(\lambda x.(\mathcal{A}x))\,.
\end{array}
\tag{8.23}
$$

Felleisen couldn't meet Plotkin's calculus-to-semantics correspondence criterion, that the full theory of the calculus should imply operational equivalence, because the relations of Schemata (8.23) clearly *don't* imply operational equivalence: one doesn't expect in general that $C[\mathcal{A}T]$ will have the same observable effect as $C[T]$, nor $C[\mathcal{C}T]$ as $C[T(\lambda x.(\mathcal{A}x))]$. So $=_{vc}^{\triangleright} \not\subseteq \simeq_{vc}$. Weakening the criterion slightly, he proved that

(1) $=_{vc} \subseteq \simeq_{vc}$, and

(2) for all T_1, T_2, if, for all E, $E[T_1] =_{vc}^{\triangleright} E[T_2]$, then $T_1 \simeq_{vc} T_2$.

The latter result, though stronger than the former, is difficult to use, since its would-be user must first show —presumably by induction— that $E[T_1] =_{vc}^{\triangleright} E[T_2]$ for all E; but it does relieve the user of checking non-evaluation contexts, which operational equivalence would otherwise engage.

Felleisen's second approach to modeling continuations, in [Fe88], was to introduce an enclosing syntactic frame, which he called a *prompt-application*, that would act as an explicit bound on continuations — in place of the implicit bound of termination from his earlier $\lambda_v C^{\triangleright}$-calculus. His prompt-application syntax was "$(\#T)$"; adding this to the $\lambda_v C$-calculus syntax of (8.17),[31]

$$
\begin{array}{rcll}
T & ::= & V \mid (TT) \mid (\mathcal{C}T) \mid (\mathcal{A}T) \mid (\#T) & \text{(Terms)} \\
C & ::= & \Box \mid (CT) \mid (TC) \mid (\lambda x.C) & \\
 & & \quad \mid (\mathcal{C}C) \mid (\mathcal{A}C) \mid (\#C) & \text{(Contexts)} \\
E & ::= & \Box \mid (ET) \mid (VE) \mid (\#E) & \text{(Evaluation contexts)}\,.
\end{array}
\tag{8.24}
$$

[31] We have chosen to present here, for clarity of exposition, the $\lambda_v C^{\triangleright}$-calculus from Felleisen's early paper [FeFrKoDu87]. His later control calculi, in [Fe88] and [FeHi92], are based on a different version of $\lambda_v C^{\triangleright}$-calculus, with no primitive abort operator $\mathcal{A}$. For continuity of exposition, we prefer to construct hybrid modified calculi (in this case, a hybrid $\lambda_v \#C$-calculus), by applying the modifications from the later papers to the version of $\lambda_v C^{\triangleright}$-calculus from the earlier paper.

The base-case schemata for $\mathcal{C}$ and $\mathcal{A}$, (8.23), can then be converted from computation rules using $\triangleright$ to ordinary rules using $\longrightarrow$:

$$\begin{array}{rcl} \#(\mathcal{A}T) & \longrightarrow & \#T \\ \#(\mathcal{C}T) & \longrightarrow & \#(T(\lambda x.(\mathcal{A}x))) \\ \#V & \longrightarrow & V\,, \end{array} \tag{8.25}$$

where the third schema, $\#V \longrightarrow V$, removes the prompt-application operator at the end of evaluation (akin to the self-evaluation base case for Lisp `eval`). The problematic weakened calculus-to-semantics correspondence is restored to Plotkin's simpler form, $=_{v\#c} \subseteq \simeq_{v\#c}$.

Because prompt-application was made a compatible feature of the syntax (compatibility being the point of the exercise), it can be embedded at will within larger terms, $C[\#T]$. In that capacity it is an additional facility of the programming language, serving to limit how much control over the future is granted by continuation capture (i.e., weakening the complete-control assumption). The facility is a rudimentary form of the exception-handling supported by languages such as Java and (more so) Kernel.

Note that the introduction of an explicit bounding frame for evaluation is suggestive of the explicit-evaluation paradigm of the current work.

Felleisen's third approach, with Robert Hieb ([FeHi92]), was to base the semantics/calculus correspondences on non-identity mappings from reduction sequences of either system to the other. Recall that Plotkin had been looking, to begin with, at configuration-reduction semantics (SECD) that had very different syntax than their intended term-reduction calculi; he was just conveniently able to establish an isomorphism from the automaton to an operational semantics whose syntax coincided exactly with the calculus. Moreover, as long as the mappings are sufficiently straightforward, they should not obstruct using the calculus to study the semantics.

Consider the compatible reduction relation $\longrightarrow_{vc}$ of $\lambda_v C^{\triangleright}$-calculus, without the computation step relation $\triangleright_{vc}$. The inductive Schemata (8.22) will cause $\mathcal{A}$s and $\mathcal{C}$s to bubble up to the top level of the term, where they will accumulate irreducibly because there is no base-case device to eliminate them. If the semantics/calculus syntax were required to correspond identically, the calculus would have to eliminate all the $\mathcal{A}$s and $\mathcal{C}$s because the semantics eliminates them; and this *cannot* be done compatibly — but if the correspondences can be approximate, then it suffices to add some simplification schemata so that the accumulated $\mathcal{A}$s and $\mathcal{C}$s are reduced to a single operator.

The primary foci for simplification are terms of the form $\mathcal{C}(\lambda x.T)$, which are created by the inductive $\mathcal{C}$-handling schemata in (8.22). Within a term of this form, T can be reduced as if it has no surrounding context, because any surrounding context will be removed by the outer $\mathcal{C}$; the context $\mathcal{C}(\lambda x.\Box)$ approximates $\#\Box$ of $\lambda_v \#C$-

calculus. Paralleling computation rule Schemata (8.23),

$$\begin{array}{rcl} \mathcal{C}(\lambda x.(\mathcal{A}T)) & \longrightarrow & \mathcal{C}(\lambda x.T) \\ \mathcal{C}(\lambda x.(\mathcal{C}T)) & \longrightarrow & \mathcal{C}(\lambda x.(T(\lambda x'.(\mathcal{A}x')))) \qquad \text{where } x' \neq x\,. \end{array} \tag{8.26}$$

Call this system $\lambda_v C'$*-calculus*. For any terms T, T' and $x \notin \mathrm{FV}(TT')$, if $T \longmapsto_{vc} T'$ then $\mathcal{C}(\lambda x.T) \longrightarrow^*_{vc'} \mathcal{C}(\lambda x.T')$.

Also, $=_{vc'} \subseteq \simeq_{vc}$. Of the two correspondence results Felleisen proved from $\lambda_v C^{\triangleright}$-calculus to $\lambda_v C$-semantics, this is stronger than the first result but weaker than the second: $T_1 =_{vc} T_2$ implies $T_1 =_{vc'} T_2$, since $\longrightarrow^{vc'}$ has all the schemata of $\longrightarrow^{vc}$ plus simplification Schemata (8.26); but $E[T_1] =^{\triangleright}_{vc} E[T_2]$ for all E does not necessarily imply $T_1 =_{vc'} T_2$. The difficulty arises because simplification Schemata (8.26) can only eliminate a control operator inside another if the outer operator is a $\mathcal{C}$ with a λ just inside it — which is a common pattern since it is created by the inductive $\mathcal{C}$-handling schemata in (8.22), but is not a necessary pattern. (For example, let $T_1 = \mathcal{C}(\lambda x.c)$ and $T_2 = \mathcal{C}(\mathcal{C}(\lambda x.c))$.)

To shore up the equational theory, [FeHi92] introduced additional schemata, so that arbitrary control-operator applications could be converted into the form needed for simplification:

$$\begin{array}{rcl} \mathcal{A}T & \longrightarrow & \mathcal{C}(\lambda x.T) \qquad \text{where } x \notin \mathrm{FV}(T) \\ \mathcal{C}T & \longrightarrow & \mathcal{C}(\lambda x_1.(T(\lambda x_2.(\mathcal{A}(x_1 x_2))))) \\ & & \qquad\qquad \text{where } x_1 \neq x_2 \text{ and } x_1 \notin \mathrm{FV}(T)\,. \end{array} \tag{8.27}$$

Call this system $\lambda_v Cd$*-calculus* (the somewhat arbitrary name used in [FeHi92]). With the additional schemata, $=_{vcd} \subseteq \simeq_{vc}$ turns out to be exactly as powerful as the clumsier correspondence from Felleisen's first approach; that is, $T_1 =_{vcd} T_2$ iff for all E, $E[T_1] =^{\triangleright}_{vc} E[T_2]$.

The formulation of Schemata (8.27) requires some care, against both cyclic reductions (which become a problem *if* they interfere with existence of normal forms) and violations of Church–Rosser-ness. Note that the difficulty of untangling these problems in (8.27) results from cascading complexity of non-orthogonal interaction between $\mathcal{C}$ and λ, visible first in (8.22), and then in (8.26).

8.3.3.2 Imperative state

To handle sequential state, Felleisen partitioned the syntactic domain of variables into assignable and non-assignable,

$$\begin{array}{rcl} x_\lambda & \in & \mathit{NonAssignableVars} \\ x_\sigma & \in & \mathit{AssignableVars} \\ x & ::= & x_\lambda \mid x_\sigma \qquad \text{(Variables)}\,, \end{array} \tag{8.28}$$

and introduced a new construct called a *sigma-capability* to perform assignment, with notation "$(\sigma x_\sigma.T)$":

$$\begin{array}{lll} V & ::= & c \mid x_\lambda \mid (\lambda x.T) \mid (\sigma x_\sigma.T) \qquad \text{(Values)} \\ T & ::= & V \mid (TT) \mid x_\sigma \qquad\qquad\qquad\quad \text{(Terms)}\,. \end{array} \tag{8.29}$$

When a sigma-capability $\sigma x_\sigma.T$ is applied to a value V, $(\sigma x_\sigma.T)V$, the language evaluator assigns V to x_σ and then evaluates T. Operator σ does not perform binding; x_σ must be bound by some enclosing λ. The assignment to x_σ persists until the next assignment to x_σ, if any (possibly, but not necessarily, after evaluation of T).

Placing the capability syntactic frame $\sigma x_\sigma.\Box$ around T avoids the awkwardness of an expression that returns no value (type *void* in Java); but it is also convenient for making the assignment "bubble up" to the top level of the term, as control operators $\mathcal{A}$ and $\mathcal{C}$ did in the $\lambda_v C$-calculi. One might imagine bubbling-up schemata for a $\lambda_v S$-calculus such as

$$\begin{array}{lcl} ((\sigma x_\sigma.T_1)V)T_2 & \longrightarrow & (\sigma x_\sigma.(T_1T_2))V \\ V_1((\sigma x_\sigma.T)V_2) & \longrightarrow & (\sigma x_\sigma.(V_1T))V_2\,. \end{array} \tag{8.30}$$

This simple arrangement is complicated, however, by the need to track the assigned values of variables over time. The traditional technique is to maintain a global *store*, which is a mapping from locations to values ([Stra00]). An abstract-machine configuration in the semantics consists of a term paired with a store. When a subterm $(\lambda x_\sigma.T)V$ is β-reduced, rather than substituting V for x_σ in T, we map a fresh location l to V in the store, and substitute l for x_σ in T. Evaluating a subterm l consists of replacing it by its value in the current store (which depends on when the subterm is evaluated).

$\lambda_v S'$-semantics.

Syntax (amending λ_v-semantics):

$$\begin{array}{rcll} x_\lambda & \in & \mathit{NonAssignableVars} & \\ x_\sigma & \in & \mathit{AssignableVars} & \\ l & \in & \mathit{Locations} & \\ \\ x & ::= & x_\lambda \mid x_\sigma & \text{(Variables)} \\ V & ::= & c \mid x_\lambda \mid (\lambda x.T) \mid (\sigma x_\sigma.T) \mid (\sigma l.T) & \text{(Values)} \\ T & ::= & V \mid (TT) \mid x_\sigma \mid l & \text{(Terms)} \\ \\ B & ::= & l \leftarrow V & \text{(Bindings)} \\ S & ::= & \{\!\{B^*\}\!\} & \text{(Stores)} \end{array}$$

Auxiliary functions (amending λ_v-semantics):

$$\begin{array}{rcl} \{\!\{B_1 \dots B_m\}\!\} \cdot \{\!\{B'_1 \dots B'_{m'}\}\!\} & = & \{\!\{B_1 \dots B_m\, B'_1 \dots B'_{m'}\}\!\} \\ \mathit{lookup}(l,\ \{\!\{l' \leftarrow V\}\!\} \cdot S) & = & \begin{cases} V & \text{if } l = l' \\ \mathit{lookup}(l, S) & \text{otherwise} \end{cases} \\ \mathrm{dom}(\{\!\{l_1 \leftarrow V_1\ \dots\ l_m \leftarrow V_m\}\!\}) & = & \{l_1 \dots l_m\} \end{array} \tag{8.31}$$

Schemata:

$$\begin{array}{rcll} \langle E[(pc)], S\rangle & \longmapsto & \langle E[\delta(p,c)], S\rangle & \text{if } \delta(p,c) \text{ is defined} \\ \langle E[(\lambda x_\lambda.T)V], S\rangle & \longmapsto & \langle E[T[x_\lambda \leftarrow V]], S\rangle & \\ \langle E[(\lambda x_\sigma.T)V], S\rangle & \longmapsto & \langle E[T[x_\sigma \leftarrow l]], \{\!\{l \leftarrow V\}\!\} \cdot S\rangle & \\ & & & \text{where } l \notin \mathrm{dom}(S) \\ \langle E[l], S\rangle & \longmapsto & \langle E[\mathit{lookup}(l, S)], S\rangle & \text{if } l \in \mathrm{dom}(S) \\ \langle E[(\sigma l.T)V], S\rangle & \longmapsto & \langle E[T], \{\!\{l \leftarrow V\}\!\} \cdot S\rangle\,. & \end{array}$$

Locations are essentially just another kind of variable, with global scope and no possibility of local shadowing. Two configurations are considered equivalent when they differ only by an isomorphism of locations, just as terms are considered equivalent when they differ only by an isomorphism of bound variables. However, stores present a new kind of challenge, because they are fundamentally alien to a term-reduction system (whereas contexts, the basis for continuations in $\lambda_v C$-calculi, are native to terms). Before constructing a calculus, we want to eliminate the stores. Moreover, setting up direct correspondences between a semantics with stores and a calculus without would be problematic at best. So Felleisen devised first an isomorphic semantics without stores, and *then* constructed a calculus corresponding to the store-free semantics.

In eliminating the store, one has to decide where to keep the value assigned to a location. Felleisen abandoned the idea of keeping the value for a location in a single place; instead, he proposed to maintain copies of the value at *all* points in the term where the location occurs as a subterm. Wherever $\Lambda S'$ would use a subterm l, with a visible binding $l \leftarrow V_1$ in the store, the store-free syntax ΛS has a labeled term V_1^l.

When l is assigned a new value V_2, each subterm V_1^l is replaced by V_2^l; and when V^l is evaluated, the label is removed so that subsequent assignments to l won't replace that subterm. The labeled-value substitution semantic function, that replaces values of label l, has notation "$(T_1[\bullet^l \leftarrow T_2])$".

This arrangement is complicated by the need to handle *self-referencing* terms. Technically, self-reference arises in $\lambda_v S$-semantics when an assignable variable x_σ occurs in the body of a λ- or σ-expression that, in turn, occurs in a value assigned to x_σ; e.g., $(\sigma x_\sigma.x_\sigma)(\lambda x_\lambda.x_\sigma)$. To represent such circular structures by finite terms, when an l-labeled value has an l-labeled subterm, the subterm must be the special anonymous l-labeled value, "$\bullet^l$"; so $(\sigma x_\sigma.x_\sigma)(\lambda x_\lambda.x_\sigma)$ would reduce (in a context that binds x_σ) to $(\lambda x_\lambda.\bullet^l)^l$. The anonymous $\bullet^l$ is also used, as a notational convenience, when the variable in a capability frame $\sigma x_\sigma.\Box$ is given a location; that is,

$$(\sigma x_\sigma.T)[x_\sigma \leftarrow L^l] \;=\; \sigma\bullet^l.(T[x_\sigma \leftarrow L^l])\,. \tag{8.32}$$

When a labeled subterm V^l is delabeled by evaluation, its internal self-references $\bullet^l$ are all expanded to V^l, reducing the subterm to $V[\bullet^l \leftarrow V^l]$; in the running example, $(\lambda x_\lambda.\bullet^l)^l$ becomes $(\lambda x_\lambda.(\lambda x_\lambda.\bullet^l)^l)$. The internal copies of V^l are pending algorithmic recursive calls, rather than self-references to the identity l of a data structure (which would presumably be fixed at the time pre-existing V^l is evaluated), since the semantics practices implicit evaluation (so that all expressions —including references to assignable variables— are algorithms rather than data structures). Consequently, evaluation of the subterm stops after this one level of expansion, as all the internal self-references are embedded within λ- or σ-expressions, from which recursive calls are algorithmically deferred.

Distributing the store into the term has the fringe benefit of eliminating garbage collection as a formal consideration (which would have required configurations to be considered equivalent not only under trivial isomorphism of locations, but under non-trivial elimination of unreachable bindings): when the last reference to a location disappears from the term, no explicit binding for the location lingers after it. The price for this simplification is paid at the meta-level (metamathematics, language interpreter). Metamathematically, the syntax of terms is no longer context-free, because all the subterms of a term must now maintain consistency with the same implicit store. This in turn weakens the metamathematical notion of compatibility, because some terms are syntactically prohibited from some contexts. For a naive language interpreter, each term may be much larger than an equivalent configuration with store, since each referenced value of an assignable variable is repeated at each reference point; while, if an interpreter seeks to conserve space by using pointers to a single copy of each subterm, it takes on an additional administrative burden (which may amount to simulating a store, hence garbage collection).

$\lambda_v S$-semantics.
Syntax (amending λ_v-semantics):

$$
\begin{array}{rcll}
x_\lambda & \in & \mathit{NonAssignableVars} & \\
x_\sigma & \in & \mathit{AssignableVars} & \\
l & \in & \mathit{Labels} & \\
x & ::= & x_\lambda \mid x_\sigma & \text{(Variables)} \\
V & ::= & c \mid x_\lambda \mid (\lambda x.T) \mid (\sigma x_\sigma.T) \mid (\sigma \bullet^l.T) & \text{(Values)} \\
T & ::= & V \mid (TT) \mid x_\sigma \mid \bullet^l \mid V^l & \text{(Terms)} \\
A & ::= & x_\sigma \mid V^l & \text{(Assignable values)} \\
X & ::= & x_\sigma \mid \bullet^l & \text{(Capability parameters)} \\
L & ::= & \bullet \mid V & \text{(Label subjects)}
\end{array}
\tag{8.33}
$$

where
a term L^l has the form $\bullet^l$ iff it occurs within a larger term V^l;
a term $\lambda x.T$ cannot contain a subterm V^l with $x \in \mathrm{FV}(V^l)$; and
in a term T_1T_2, if V_1^l is a subterm of T_1, and V_2^l of T_2, then V_1, V_2 must be identical up to content of labeled subterms.

(Rigorously, maintaining the first context-sensitive constraint requires active measures in both the schemata and the definition of substitution, to anonymize labeled subterms as they are introduced into contexts with the same label; we will elide these measures, for clarity of exposition.)

The qualification *up to content of labeled subterms* on the third context-sensitive constraint allows for a self-referencing structure that is referenced from multiple points within the structure. For example, a subterm $(\sigma x_\sigma.((\sigma x'_\sigma.(x_\sigma x'_\sigma))(\lambda x_\lambda.x_\sigma)))(\lambda x_\lambda.x'_\sigma)$ should reduce in suitable context to $(\lambda x_\lambda.(\lambda x_\lambda.\bullet^l)^{l'})^l(\lambda x_\lambda.(\lambda x_\lambda.\bullet^{l'})^l)^{l'}$ (and thence to final evaluation result $\lambda x_\lambda.(\lambda x_\lambda.(\lambda x_\lambda.\bullet^l)^{l'})^l$).

$\lambda_v S$-semantics.
Schemata:

$$
\begin{array}{rcll}
E[(pc)] & \longmapsto & E[\delta(p,c)] & \text{if } \delta(p,c) \text{ is defined} \\
E[(\lambda x_\lambda.T)V] & \longmapsto & E[T[x_\lambda \leftarrow V]] & \\
E[(\lambda x_\sigma.T)V] & \longmapsto & E[T[x_\sigma \leftarrow V^l]] & \\
 & & \text{where } l \text{ doesn't occur in } E,T,V & \\
E[V^l] & \longmapsto & E[V[\bullet^l \leftarrow V^l]] & \\
E[(\sigma \bullet^l.T)V] & \longmapsto & E[T][\bullet^l \leftarrow V^l]\,. &
\end{array}
\tag{8.34}
$$

Definitions of the substitution functions (for non-assignable variables, assignable variables, and labeled values) are tedious but straightforward, except for the case of substitution for the parameter in a capability frame $(\sigma X.\Box)$, where X can only be replaced by an anonymous labeled value:

$$
\begin{aligned}
(\sigma X.T)[x_\lambda \leftarrow V] &= \sigma X.(T[x_\lambda \leftarrow V]) \\
(\sigma X.T)[x_\sigma \leftarrow L^l] &= \begin{cases} \sigma \bullet^l.(T[x_\sigma \leftarrow L^l]) & \text{if } X = x_\sigma \\ \sigma X.(T[x_\sigma \leftarrow L^l]) & \text{otherwise} \end{cases} \\
(\sigma X.T)[\bullet^l \leftarrow L^l] &= \sigma X.(T[\bullet^l \leftarrow L^l])\,.
\end{aligned}
\tag{8.35}
$$

(The one subcase that replaces the parameter was given earlier, as (8.32). For the complete labeled-value substitution function, see [FeFr89, §4].)

Despite the administrative complexity incurred by self-reference in $\lambda_v S$-semantics, the self-reference is of a restrained form: references can only consist (as cautiously observed earlier) of algorithmically deferred lookup of a symbol to determine the value result of a previous evaluation. The restraint would be more apparent if bindings were managed by environments rather than by substitution; then λ- and σ-expressions would evaluate to closures (as in §3.3.2), each capturing its static environment, and any assignable-variable instances in the body of the resulting applicative would not be touched until the applicative was actually applied to an argument. (Sic: a capability would evaluate to an applicative, which when applied would assign its argument to its parameter in its static environment, instead of creating a local environment for the binding.) A more immediate form of self-reference is supported by Scheme's (and Kernel's) data-mutation applicatives ***set-car!*** and ***set-cdr!***, in that a self-referencing algorithm can cause non-termination only if it is applied, but a self-referencing data structure can cause non-termination merely by being evaluated.

The non-compatible $\lambda_v S^{\triangleright}$-calculus corresponding to $\lambda_v S$-semantics bubbles assignments up to the top level, and then distributes the assigned value to the entire term by substitution in a computation rule:

$\lambda_v S^{\triangleright}$-calculus.
Schemata (assignment):

$$
\begin{aligned}
((\lambda x_\sigma.T_1)V)T_2 &\longrightarrow (\lambda x_\sigma.(T_1T_2))V && \text{if } x_\sigma \notin \mathrm{FV}(T_2) \\
V_1((\lambda x_\sigma.T)V_2) &\longrightarrow (\lambda x_\sigma.(V_1T))V_2 && \text{if } x_\sigma \notin \mathrm{FV}(V_1) \\
((\sigma X.T_1)V)T_2 &\longrightarrow (\sigma X.(T_1T_2))V \\
V_1((\sigma X.T)V_2) &\longrightarrow (\sigma X.(V_1T))V_2 \\
(\lambda x_\sigma.T)V &\triangleright T[x_\sigma \leftarrow V^l] && \text{where } l \text{ doesn't occur in } T, V \\
(\sigma \bullet^l.T)V &\triangleright T[\bullet^l \leftarrow V^l]\,.
\end{aligned}
\tag{8.36}
$$

Delabeling (i.e, evaluation of a labeled value) is also side-effect-ful, so also has to be globally coordinated. Labeled values are therefore also bubbled up to the top level, where delabeling is performed by a computation rule; but bubbling-up of labeled values can't alter the labeled value (as bubbling up a capability alters the capability), so instead additional λ's are introduced to shift the labeled value upward:

$\lambda_v S^{\triangleright}$-calculus.
Schemata (delabeling):

$$\begin{array}{rcl} V_1 V_2^l & \triangleright & V_1(V_2[\bullet^l \leftarrow V^l]) \\ V^l & \triangleright & V[\bullet^l \leftarrow V^l] \end{array} \tag{8.37}$$

$$\begin{array}{rcll} V^l T & \longrightarrow & (\lambda x_\lambda.(x_\lambda T))V^l & \text{where } x_\lambda \notin \mathrm{FV}(T) \\ (VA)T & \longrightarrow & (\lambda x_\lambda.((V x_\lambda)T))A & \text{where } x_\lambda \notin \mathrm{FV}(VT) \\ V_1(V_2 A) & \longrightarrow & (\lambda x_\lambda.(V_1(V_2 x_\lambda)))A & \text{where } x_\lambda \notin \mathrm{FV}(V_1 V_2)\,. \end{array}$$

In the same paper where Felleisen and Hieb describe their compatible control calculus, $\lambda_v Cd$-calculus, they also describe a compatible state calculus (in our nomenclature, $\lambda_v S\rho$-calculus; [FeHi92, §4]). Where the compatible control calculus uses contexts $\mathcal{C}(\lambda x.\Box)$ as bounding frames, the compatible state calculus uses admixtures of λ with σ to construct bounding frames providing a partial environment in effect over a subterm. In their treatment, an environment is a set of assignable-variable bindings (versus the location bindings of a store), linearized by an ordering of the variables, and with at most one binding for each variable. Because the binding frames are extremely cumbersome in the unsweetened notation of ΛS, they introduce syntactic sugar "$\rho e.T$" for effecting environment e over subterm T:[32]

$$\begin{array}{l} \rho\{x_{\sigma,1} \leftarrow V_1,\ \ldots\ x_{\sigma,m} \leftarrow V_m\}.T \\ \quad = \quad (\lambda x_{\sigma,1} \ldots x_{\sigma,m}.((\sigma x_{\sigma,1} \ldots x_{\sigma,m}.T)V_1 \ldots V_m))(\lambda x_\lambda.x_\lambda) \ldots (\lambda x_\lambda.x_\lambda) \end{array} \tag{8.38}$$

(which depends, in turn, on abbreviation

$$((\lambda x_1 x_2 \cdots.T)V_1 V_2 \cdots) \quad = \quad (\cdots(((\lambda x_1.(\lambda x_2.\cdots))V_1)V_2)\cdots) \tag{8.39}$$

and similarly for σ).[33]

ρ-frames must involve both λ and σ because otherwise there would be no way for ρ to express self-reference: the much simpler $\rho\{x_\sigma \leftarrow V\}.T = (\lambda x_\sigma.T)V$ would put any free occurrence of x_σ in V outside the scope of the specified binding $x_\sigma \leftarrow V$.[34]

[32] We are using our own internally consistent base letters for semantic variables. Whereas we use e for environments, in [FeHi92] they used θ; and they consistently used e for *expressions*, which we call *terms*, T.

[33] Shorthand (8.39), without which the right-hand side of (8.38) would be effectively unreadable, is standard in studies of λ-calculus. Such conventional abbreviations of λ-calculus are mostly avoided in this dissertation, on the principle that they remove from sight technical details that cannot safely be put out of mind.

[34] This difficulty arises because Felleisen's binding construct for assignable variables, $\lambda x_\sigma.\Box$, is non-orthogonal to applicative combination (presumably in structural imitation of his other binding construct, $\lambda x_\lambda.\Box$, whose *sole* purpose is applicative combination). The corresponding $\natural$ calculus (Chapter 12) will use an assignable-variable binding construct orthogonal to combination.

The basic $\lambda_v S\rho$-calculus simplification schemata, analogous to the $\lambda_v S^{\triangleright}$-calculus computation rule schemata of (8.36) and (8.37), are

$$\begin{array}{rcl} \rho e.((\lambda x_\sigma.T)V) & \longrightarrow & \rho e \cup \{x_\sigma \leftarrow V\}.T \\ & & \qquad \text{where } x_\sigma \notin \mathrm{dom}(e) \cup \mathrm{FV}(V) \\ \rho e \cup \{x_\sigma \leftarrow V_1\}.(V_2 x_\sigma) & \longrightarrow & \rho e \cup \{x_\sigma \leftarrow V_1\}.(V_2 V_1) \\ \rho e \cup \{x_\sigma \leftarrow V_1\}.((\sigma x_\sigma.T)V_2) & \longrightarrow & \rho e \cup \{x_\sigma \leftarrow V_2\}.T\,. \end{array} \tag{8.40}$$

Without the computation rules, though, explicit bindings accumulate at the top level, *even if the bound variables aren't used*. It is therefore necessary to introduce a garbage-collection schema:

$$\rho e \cup \{x_\sigma \leftarrow V\}.T \longrightarrow \rho e.T \qquad \text{if } x_\sigma \notin \mathrm{FV}(\rho e.T)\,. \tag{8.41}$$

Note that, by keeping each assignable-variable binding in just one place, the $\lambda_v S\rho$-calculus entirely eliminates labeled subterms, and with them all the convoluted provisions for algorithmic self-reference. Each assignable-variable subterm remains in place until it is evaluated (as would a symbol under an explicit-evaluation discipline); and further, when it finally comes time to replace the variable with its currently assigned value, the replacement is done *without substitution*.

8.4 Meta-programming

8.4.1 Trivialization of theory

It was observed in §1.2.3 that adding object-examination to an implicit-evaluation calculus trivializes its equational theory. To see this in detail, consider the addition of a quotation device to λ-calculus. Denote quotation of a term T by $(\mathcal{Q}T)$. The normal form of $\mathcal{Q}T$ ought to contain essentially the same information as unevaluated T itself; so assume, for simplicity, that $\mathcal{Q}T$ is irreducible.

The root of the problem is that equality should be compatible. With the introduction of quotation context $\mathcal{Q}\Box$, this means that $T_1 =_\bullet T_2$ should imply $\mathcal{Q}T_1 =_\bullet \mathcal{Q}T_2$.

Our expectation for the behavior of quotation is that $\mathcal{Q}T_1$ means the same thing as $\mathcal{Q}T_2$ only when T_1 and T_2 denote the same evaluable structure; so for compatibility, $T_1 =_\bullet T_2$ only when T_1 and T_2 denote the same evaluable structure. Under implicit evaluation, though, the only difference between an evaluable structure and the result of evaluating it is whether it has been reduced yet; so two terms denote the same evaluable structure just when they are syntactically identical. Thus, the only equations are the reflexive ones $T = T$ that are required of every equational theory by definition (equality being reflexive symmetric transitive and compatible). An equational theory that contains only reflexive equations is *trivial*, a property that is dual to inconsistency since a trivial theory is perfectly uninformative by admitting nothing whereas an inconsistent theory is perfectly uninformative by admitting everything.

A trivial equational theory, besides being useless, also flies in the face of the usual practice in calculi of generating the equational theory from the reduction relation (unless the reduction relation is empty, in which case reduction is useless too). That is, we ordinarily expect $T_1 \longrightarrow_\bullet^* T_2$ to imply $T_1 =_\bullet T_2$, and $T_1 =_\bullet T_2$ to imply the existence of a term T_3 with $T_1 \longrightarrow_\bullet^* T_3$ and $T_2 \longrightarrow_\bullet^* T_3$. Suppose that, in order to retain the use of the equational theory in studying the reduction relation, we are willing to adjust our understanding of "quotation" by admitting $\mathcal{Q}T_1 =_\bullet \mathcal{Q}T_2$ whenever T_1 and T_2 have a common reduct.

To further support object-examination, suppose operators $\mathcal{E}$, $\mathcal{L}$, and $\mathcal{R}$ (mnemonic for *Eval*, *Left*, and *Right*), and schemata

$$\begin{array}{rcl} \mathcal{E}(\mathcal{Q}T) & \longrightarrow & T \\ \mathcal{L}(\mathcal{Q}(T_1T_2)) & \longrightarrow & \mathcal{Q}T_1 \\ \mathcal{R}(\mathcal{Q}(T_1T_2)) & \longrightarrow & \mathcal{Q}T_2\,. \end{array} \tag{8.42}$$

If we were still trying for the traditional semantics of quotation, these schemata would be complicated by the need to prevent reduction of a quoted subterm; but now we are deliberately allowing that subcase. Call this system $\lambda\mathcal{Q}$-*calculus*. Since reduction implies equality,

$$\begin{array}{rcl} \mathcal{E}(\mathcal{Q}T_1) & =_\mathcal{Q} & T_1 \\ \mathcal{L}(\mathcal{Q}(T_1T_2)) & =_\mathcal{Q} & \mathcal{Q}T_1 \\ \mathcal{R}(\mathcal{Q}(T_1T_2)) & =_\mathcal{Q} & \mathcal{Q}T_2\,. \end{array} \tag{8.43}$$

Writing K for the combinator λx.(λy.x), we have for all terms T, $(\mathsf{KK})T \longrightarrow_\beta^* \mathsf{K}$; therefore, for all terms T_1 and T_2,

$$(\mathsf{KK})T_1 \;=_\mathcal{Q}\; (\mathsf{KK})T_2\,. \tag{8.44}$$

By compatibility,

$$\mathcal{E}(\mathcal{R}(\mathcal{Q}((\mathsf{KK})T_1))) \;=_\mathcal{Q}\; \mathcal{E}(\mathcal{R}(\mathcal{Q}((\mathsf{KK})T_2)))\,; \tag{8.45}$$

and by the behaviors of the meta-programming operators, (8.43),

$$\begin{array}{rcl} \mathcal{E}(\mathcal{R}(\mathcal{Q}((\mathsf{KK})T_1))) & =_\mathcal{Q} & T_1 \\ \mathcal{E}(\mathcal{R}(\mathcal{Q}((\mathsf{KK})T_2))) & =_\mathcal{Q} & T_2\,. \end{array} \tag{8.46}$$

So $T_1 =_\mathcal{Q} T_2$, for every possible T_1 and T_2, which is to say that the theory $\boldsymbol{\lambda\mathcal{Q}}$ is inconsistent.

In proving inconsistency, we used compatibility but never actually reduced a quoted subterm. A single reduction step on a quoted subterm allows us to further prove that $\longrightarrow^\mathcal{Q}$ isn't Church–Rosser ($\longrightarrow_\mathcal{Q}^*$ doesn't have the diamond property). Suppose T_1, T_2 have no common reduct, and $x \notin \mathrm{FV}(T_1)$; then

$$\begin{array}{rcl} \mathcal{R}(\mathcal{Q}((\lambda x.(T_1T_1))T_2)) & \longrightarrow_\mathcal{Q} & \mathcal{Q}T_2 \\ \mathcal{R}(\mathcal{Q}((\lambda x.(T_1T_1))T_2)) & \longrightarrow_\mathcal{Q} & \mathcal{R}(\mathcal{Q}(T_1T_1)) \;\longrightarrow_\mathcal{Q}\; \mathcal{Q}T_1\,. \end{array} \tag{8.47}$$

(It is possible to choose T_1, T_2 with no common reduct despite equational inconsistency exactly *because* reduction is not Church–Rosser.)

The explicit-evaluation solution to this predicament is to explicitly distinguish between a combination T_1T_2 and a term designating evaluation of T_1T_2. We use operator $\mathcal{E}$ to designate evaluation of its operand; so T_1T_2 can only be reduced by compatible reduction of T_1 or T_2; β-reduction requires an evaluation context, $\mathcal{E}(T_1T_2)$.

We'll call this new system *$\lambda\mathcal{E}$-calculus*.

Of the three meta-programming Schemata (8.42), the first is unchanged,

$$\mathcal{E}(\mathcal{Q}T) \;\longrightarrow\; T\,. \tag{8.48}$$

There are several ways one might adapt the object-examination schemata, the simplest of which is

$$\begin{array}{ll} \mathcal{L}(T_1T_2) & \longrightarrow\; T_1 \\ \mathcal{R}(T_1T_2) & \longrightarrow\; T_2\,. \end{array} \tag{8.49}$$

Here, each examination operator ($\mathcal{L}$ or $\mathcal{R}$) eagerly extracts the appropriate part of its operand, but does not initiate evaluation of its operand. The programmer can induce operand evaluation before examination by putting an $\mathcal{E}$ inside the examination operator ($\mathcal{L}(\mathcal{E}\Box)$ or $\mathcal{R}(\mathcal{E}\Box)$); and can exempt the result of examination from a surrounding evaluation context by putting a $\mathcal{Q}$ outside the examination operator ($\mathcal{Q}(\mathcal{L}\Box)$ or $\mathcal{Q}(\mathcal{R}\Box)$). One *could* modify the schemata so that the examination operator automatically initiates operand evaluation before examination, but that would require new syntax to distinguish the operator before initiation from the operator after initiation (which, as the schemata stand, we can distinguish with existing syntax). One *could* modify the schemata so that examination only takes place in the presence of a surrounding evaluation context ($\mathcal{E}(\mathcal{L}\Box)$ or $\mathcal{E}(\mathcal{R}\Box)$), but there is no need to wait for an evaluation context since examination terms $\mathcal{L}T$ or $\mathcal{R}T$ are not themselves subject to examination before reduction (so reducing them eagerly doesn't cause inconsistency), and eager reduction promotes strong theory.

Constants self-evaluate,

$$\mathcal{E}c \;\longrightarrow\; c\,. \tag{8.50}$$

Variable evaluations, $\mathcal{E}x$, are irreducible, because the calculus uses variable *substitution*, so $\mathcal{E}x$ represents an incomplete evaluation that should proceed once substitution replaces x with some evaluable term T (via $(\mathcal{E}x)[x \leftarrow T]$).

As a first attempt at handling application evaluation, one might have

$$\mathcal{E}((\lambda x.T_1)T_2) \;\longrightarrow\; \mathcal{E}\,T_1[x \leftarrow T_2]\,. \tag{8.51}$$

However, the operator of a combination might need to be reduced first in order to achieve the form $\lambda x.T_1$, and for strength of theory we would also like to be able to

reduce the operand before β-reduction. So, as a second attempt, we split the schema into two stages, the first to initiate subterm evaluation and the second to β-reduce:

$$\begin{array}{rcl} \mathcal{E}(T_1T_2) & \longrightarrow & \mathcal{E}'((\mathcal{E}T_1)(\mathcal{Q}(\mathcal{E}T_2))) \\ \mathcal{E}'((\lambda x.T_1)T_2) & \longrightarrow & \mathcal{E}\,T_1[x \leftarrow T_2]\,. \end{array} \tag{8.52}$$

New operator $\mathcal{E}'$ keeps track of the fact that operand evaluation has already been initiated[35] (so it isn't initiated multiple times, which would reintroduce an erratic form of implicit evaluation). The quotation context $\mathcal{Q}\Box$ around the operand evaluation $\mathcal{E}T_2$ is needed to prevent the operand from being evaluated a second time after being substituted into the evaluation context of the body.

As a first attempt at a schema for evaluating λ-expressions, one might have

$$\mathcal{E}(\lambda x.T) \longrightarrow (\lambda x.(\mathcal{Q}(\mathcal{E}T)))\,; \tag{8.53}$$

but since λ-expressions, like examination terms $\mathcal{L}T$ and $\mathcal{R}T$, aren't subject to examination, there is no need to wait for a surrounding evaluation context before initiating evaluation of the body, and for strength of theory we would rather *not* wait. So we introduce new syntax $\langle\lambda x.T\rangle$ to indicate that body evaluation has already been initiated, and schemata

$$\begin{array}{rcl} (\lambda x.T) & \longrightarrow & \langle\lambda x.(\mathcal{E}T)\rangle \\ \mathcal{E}\langle\lambda x.T\rangle & \longrightarrow & \langle\lambda x.T\rangle\,. \end{array} \tag{8.54}$$

The second application-evaluation schema in (8.52) is adjusted to use $\langle\lambda x.T\rangle$ rather than $(\lambda x.T)$, and not to initiate a second evaluation of T.

In all,

$\lambda\mathcal{E}$-calculus.

Syntax (amending λ-calculus):

$$\begin{array}{rcl} T & ::= & c \mid x \mid (TT) \mid (\lambda x.T) \mid \langle\lambda x.T\rangle \\ & & \mid (\mathcal{L}T) \mid (\mathcal{R}T) \mid (\mathcal{E}T) \mid (\mathcal{Q}T) \mid (\mathcal{E}'T) \qquad \text{(Terms)} \end{array}$$

Schemata:

$$\begin{array}{rcll} \mathcal{E}(\mathcal{Q}T) & \longrightarrow & T & \\ \mathcal{L}(T_1T_2) & \longrightarrow & T_1 & \\ \mathcal{R}(T_1T_2) & \longrightarrow & T_2 & \\ \mathcal{E}c & \longrightarrow & c & \\ \mathcal{Q}c & \longrightarrow & c & \\ \mathcal{E}(T_1T_2) & \longrightarrow & \mathcal{E}'((\mathcal{E}T_1)(\mathcal{Q}(\mathcal{E}T_2))) & \\ \mathcal{E}'(\langle\lambda x.T_1\rangle T_2) & \longrightarrow & T_1[x \leftarrow T_2] & \\ \mathcal{E}'(pc) & \longrightarrow & \delta(p,c) & \text{if } \delta(p,c) \text{ is defined} \\ (\lambda x.T) & \longrightarrow & \langle\lambda x.(\mathcal{E}T)\rangle & \\ \mathcal{E}\langle\lambda x.T\rangle & \longrightarrow & \langle\lambda x.T\rangle & \end{array} \tag{8.55}$$

[35] Operators $\mathcal{E}$ and $\mathcal{E}'$ correspond to meta-circular evaluator applicatives `eval` and `combine`. The name $\mathcal{C}$ was considered before $\mathcal{E}'$, but was rejected because it could also be mnemonic for *Cons*. The limited vocabulary of single-symbol operator names is one reason why $\mathfrak{f}$-calculi will introduce Lisp-like multi-character operator names.

As an illustration of the eager subterm-evaluation devices,

$$\begin{array}{lll}\mathcal{E}((\lambda\mathrm{x}.\mathrm{x})(\mathcal{Q}T)) & \longrightarrow_{\mathcal{E}} & \mathcal{E}(\langle\lambda\mathrm{x}.(\mathcal{E}\mathrm{x})\rangle(\mathcal{Q}T)) \\ & \longrightarrow_{\mathcal{E}} & \mathcal{E}'((\mathcal{E}\langle\lambda\mathrm{x}.(\mathcal{E}\mathrm{x})\rangle)(\mathcal{Q}(\mathcal{E}(\mathcal{Q}T)))) \\ & \longrightarrow_{\mathcal{E}} & \mathcal{E}'((\mathcal{E}\langle\lambda\mathrm{x}.(\mathcal{E}\mathrm{x})\rangle)(\mathcal{Q}T)) \\ & \longrightarrow_{\mathcal{E}} & \mathcal{E}'(\langle\lambda\mathrm{x}.(\mathcal{E}\mathrm{x})\rangle(\mathcal{Q}T)) \\ & \longrightarrow_{\mathcal{E}} & \mathcal{E}(\mathcal{Q}T) \\ & \longrightarrow_{\mathcal{E}} & T\,.\end{array} \tag{8.56}$$

Note that it is possible to construct a `cons` device from the facilities provided, e.g. $\mathcal{C}=\lambda\mathrm{x}.(\lambda\mathrm{y}.(\mathcal{Q}((\mathcal{E}\mathrm{x})(\mathcal{E}\mathrm{y}))))$ — which is, essentially, a *quasiquotation* construct (§7.3, §3.4.1). Then,

$$\mathcal{E}((\mathcal{C}T_1)T_2) \;\longrightarrow_{\mathcal{E}}^{+}\; (\mathcal{E}T_1)(\mathcal{E}T_2)\,. \tag{8.57}$$

In generating an equational theory $\boldsymbol{\lambda\mathcal{E}}$ from $\longrightarrow_{\mathcal{E}}$, the proof of inconsistency from $\lambda\mathcal{Q}$-calculus is foiled because there are no equations of the form $T_1T_2 =_{\mathcal{E}} T_3$ unless $T_3=T_1'T_2'$ with $T_1 =_{\mathcal{E}} T_1'$ and $T_2 =_{\mathcal{E}} T_2'$. Equation (8.44), which equated two combinations with arbitrary operands T_1 and T_2, has become

$$\mathcal{E}((\mathsf{KK})T_1) \;=_{\mathcal{E}}\; \mathcal{E}((\mathsf{KK})T_2)\,, \tag{8.58}$$

and its consequence Equation (8.45) has become

$$\mathcal{R}(\mathcal{E}((\mathsf{KK})T_1)) \;=_{\mathcal{E}}\; \mathcal{R}(\mathcal{E}((\mathsf{KK})T_2))\,; \tag{8.59}$$

and this doesn't lead in general to $T_1 =_{\mathcal{E}} T_2$ because operator $\mathcal{R}$ can't act until its operand is reduced to something of the form (T_1T_2). (In fact, its argument never achieves this form here, since $\mathcal{E}((\mathsf{KK})T) \longrightarrow_{\mathcal{E}}^{+} \langle\lambda\mathrm{x}.\langle\lambda\mathrm{y}.(\mathcal{E}\mathrm{x})\rangle\rangle$.)

8.4.2 Computation and logic

Trivialization of theory is not a computational problem. The λ-calculus augmented by quotation operator $\mathcal{Q}$, with its traditional semantics (preventing evaluation of its operand), is a valid programming system, and remains so with the addition of operators $\mathcal{E}$, $\mathcal{L}$, $\mathcal{R}$. The only *computational* difficulty in the previous subsection (§8.4.1) arose in a failed attempt to modify the programming system for the sake of its logical treatment — and even that difficulty, non-Church–Rosser-ness, isn't necessarily a computational problem either; though unwanted in this case, in some programming situations nondeterministic behavior might be *intended*.

This is an instance of a much broader principle. Programming power demands freedom to mix behaviors freely; this is the essence of the Smoothness Conjecture (§1.1.2), that removing restrictions on how language facilities can be used increases the potency of the language. Hence the early gravitation of Lisp interpretation technology toward using Lisp as its own meta-language, and hence Lisp's formidable abstractive

power. Yet, substantially the same blurring of descriptive levels in a *logical* system is cosmically fatal, invalidating the foundations of the system; it is the root cause of the classical antinomies — circular logic, a.k.a. *impredicativity* (§8.1.1), a.k.a. *tangled hierarchies* ([Hof79]). Logical power demands crisp separation between descriptor and described;[36] this is a recurring historical theme of §8.1.1 — presumed by Russell and Whitehead's *Principia*, proven by Gödel's Second Theorem, and witnessed by the logical failure of Church's λ-calculus (whose expressive flexibility fostered both computational power and logical antinomies).

Consider the classical Russell's Paradox: the set A of all sets that do not contain themselves, $A = \{X \mid X \notin X\}$, provably does not contain itself (by *reductio ad absurdum*), and also provably does contain itself (by *reductio ad absurdum* and the Law of the Excluded Middle). The Lisp analog would be an applicative predicate `A` of one argument, which assumes that its argument `X` is a predicate of one argument, and returns true iff `X` returns false on argument `X`:

```
($define! A
   ($lambda (X)
      (not? (X X)))).
```
(8.60)

When `A` is given itself as argument, it never terminates; but note the vast difference in scale of repercussions. Set A failing to resolve its own self-membership caused many of the greatest mathematical minds of the early twentieth century to reassess the whole infrastructure of their subject. Applicative `A` failing to return when given itself as argument just isn't a very big deal. Programmers are accustomed to working with algorithms that don't terminate on all inputs, and this one is already by nature at the mercy of the termination behavior of its argument; so, depending on circumstances, the programmer might not even consider nontermination of `(A A)` to be a bug, but just a practical restriction on how `A` should be used.

The key point to keep in mind, while struggling with the logical properties of calculi, is that the problem is how to reason about computation — not how to compute. For computation itself, the only problem that can arise is that we might have specified a different facility than we had meant to (on which we reassure ourselves, in part, by reasoning about its properties).

In contrasting computation with logic, brief mention may be made of the *Curry-Howard correspondence*. The original correspondence, foreshadowed by Curry and Feys in 1958 and sharpened by Howard in 1969 ([Bare84, §A.3]), identifies simple types τ in λ-calculus with propositions p in intuitionistic logic, such that there exists a term T with type τ iff there exists a proof $\mathcal{P}$ of proposition p — and, remarkably, the syntactic structure of T is isomorphic to that of $\mathcal{P}$. In 1990, Timothy Griffin showed

[36] One might frame this in quantum-mechanical terms, as crisp separation between observer and observed. Evidently quantum mechanics under the Copenhagen interpretation more resembles logic than computation.

an extended correspondence between simple types in Felleisen's then-recent $\lambda_v C^{\triangleright}$-calculus and proofs in *classical* logic, [Gri90],[37] sparking a small flurry of follow-up research in the years since (see [DoGhLe04]).

Note, however, that despite the temptation to intuit the Curry-Howard correspondence as relating computation with logic, in fact the correspondence is between two *logical* systems — one for constructing proofs of propositions in logic, and one for constructing simple types for terms in a calculus (thus reasoning *about* computation, rather than actually doing computation).

[37]Griffin was working at Rice University, to which Felleisen had relocated following his doctoral program.

Chapter 9

Pure vau calculi

9.0 Introduction

In the setting of computational semantics and calculi, *pure* means *without side-effects*. A function is pure if applying it never has side-effects. A calculus is pure if subterm-reduction never has side-effects.

Purity belongs to the same general class of well-behavedness properties as hygiene (Chapter 5). Like hygiene, purity is concerned with interactions between parts of a program, and its detailed definition must be parameterized in general by certain interactions that are permitted in the given language. The main specifically permitted interaction in Lisp/λ-calculus, for both properties, is function application.

However, hygiene —as presented in Chapter 5— starts from the position that *all* interactions are suspect, and then focuses in on plausibly eliminable interactions that may compromise static scope (such as operand or environment capture); side-effects are disregarded for Lisp hygiene, even though they are generally non-static, because they are considered too fundamental to the Lisp paradigm for their elimination to be plausible. Purity starts by looking only at actions within a subcomputation that affect the context in which the subcomputation occurs ("upward" influences, heedless of "downward" influences of the context on the subcomputation), and focuses in on influences by one subcomputation that must be resolved before some other, disjoint subcomputation can proceed. In the high-level view of a semantics/calculus, function application is the only upward influence excepted from the notion of side-effects.

Side-effects are a form of ill-behavedness, weakening the equational theory of the calculus in that (1) when a subterm reduction contains a side-effect, it cannot be equated compatibly with a resultant value, and (2) when a subterm reduction is predicated on side-effects of other, disjoint subterm reductions, it may not even *have* a unique resultant value (nor unique side-effects) with which to be equated.

In our understanding of the thesis, we take as given that Scheme is a reasonably well-behaved language, and use it as a standard against which to judge the change in well-behavedness when fexprs are introduced. Our interest in side-effects in the

presence of fexprs is, therefore, primarily focused on side-effects that were not already allowed in the absence of fexprs. Hygiene violations in the presence of fexprs create new opportunities for side-effects, by exposing a greater variety of contexts to possible influence; but since these new opportunities are contingent on bad hygiene, they are indirectly mitigated by the measures against bad hygiene discussed in Chapter 5. What we do want in order to support the well-behavedness claim of the thesis, and what the current chapter provides, is a demonstration that fexprs are not *themselves* inherently side-effect-ful, i.e., that there is such a thing as a pure $\mathfrak{f}$-calculus. Treating pure $\mathfrak{f}$-calculi first also allows us to work out the fundaments of the vau-based approach before introducing the complications of sequential control and state in Chapters 10–12.

9.0.1 Currying

Frege observed in 1893 that, for a minimalist theory of functions, it suffices to support functions of a single argument ([Ros82, §1]); an n-ary function can be modeled by a series of n unary functions, each of which builds the next knowing all its predecessors' arguments, until the last (nth) function in the series, knowing all n arguments, performs the operation of the modeled n-ary function. The principle was rediscovered in the 1920s by M. Schönfinkel, and the technique of modeling an n-ary function by a higher-order unary function is today commonly called *currying* (after Curry, whose combinatory logic (§8.1.2) extended Schönfinkel's work). Church also used this principle to simplify his treatment of functions, limiting λ-expressions to a single parameter.

However, the simplification to exclusively unary functions is only valid when studying computation alone. We expect $\mathfrak{f}$-calculi to address abstraction as well as computation, and this necessarily includes the treatment of arbitrarily structured operand lists. (Cf. the roles of generalized parameter trees in the derivations of `apply` and `list` in Chapter 4, and of binding constructs in Chapter 7.) Therefore, although currying will usually be apropos to some strictly computational subset of each $\mathfrak{f}$-calculus (a subset that will, in fact, be isomorphic or near-isomorphic to λ-calculus), devices in $\mathfrak{f}$-calculi that work directly with the structure of operand trees will not be subject to currying.

9.1 $\mathfrak{f}_e$-calculus

Fragments of a pure calculus were used in §4.3.1 to explain the semantics of Kernel. No deterministic order of reduction was assumed, so the system so described would not suffice as a primary definition of full Kernel, which is side-effect-ful; but for the pure subset of the language, we take it as a starting point. (Deterministically ordered semantics will be formulated in Chapter 10.) We call this system $\mathfrak{f}_e$-calculus (e being

the semantic-variable base letter for environments).

$\mathcal{k}_e$-calculus.

Syntax:

$$
\begin{array}{lll}
d \in \textit{PrimitiveData} & & \\
o \in \textit{PrimitiveOperatives} & & \\
s \in \textit{Symbols} \text{ (with total ordering } \leq) & & \\
B ::= s \leftarrow T & & \text{(Bindings)} \\
e ::= \langle\!\langle B^* \rangle\!\rangle & & \text{(Environments)} \\
S ::= d \mid o \mid e \mid \texttt{\#ignore} \mid \texttt{()} & & \\
\quad \mid \langle \text{operative } T\ T\ T\ T \rangle & & \text{(Self-evaluating terms)} \\
A ::= [\text{eval } T\ T] \mid [\text{combine } T\ T\ T] & & \text{(Active terms)} \\
T ::= S \mid s \mid \texttt{(}T\ \texttt{.}\ T\texttt{)} \mid \langle \text{applicative } T \rangle & & \\
\quad \mid A & & \text{(Terms)} \\
V \in \{T \mid \text{every active subterm of } T & & \\
\quad \text{is within an operative or} & & \\
\quad \text{environment}\} & & \text{(Values)}
\end{array}
\tag{9.1}
$$

where

bindings in an environment are in order by bound symbol; and
no two bindings in an environment bind the same symbol.

The separately stated constraints on environments enforce environment normalizations. Without these normalizations, the equational theory would be severely weakened (two environments differing only by permutation of visible bindings wouldn't be equal in the theory, and other equation failures would follow from that by compatibility); and without grammar constraints, the normalizations could only be enforced by reduction rule schemata. Schemata were used to enforce environment normalization in λS-calculus, (8.40) and (8.41); but in that calculus, environments didn't have their own dedicated syntax: they were represented by an admixture of λ- and σ-expressions, whose use for environments could not be constrained in the grammar because λ- and σ-expressions also had other uses in the semantics and calculus. Here the syntax of environments is dedicated; so, given the opportunity, we prefer to enforce the normalization grammatically, thereby simplifying the reduction rule schemata with which proofs will have to contend.

Technically, the separately stated constraints are context-sensitive, in the Chomsky sense; but here they cause no difficulty because they are *contextually local*. That is, although they limit how terms T can be constructed, and how contexts C can be constructed, they do not in any way limit which terms T can be used with which contexts C. (Contextually non-local syntax constraints reduce the advantage in equational strength afforded by compatibility, by reducing the set of equations $C[T_1] =_\bullet C[T_2]$ implied by any given $T_1 =_\bullet T_2$. We will introduce only one such in our $\mathcal{k}$-calculi, in Chapter 12, to support an innately non-local facility.)

Recall that the traditional treatment of λ-calculus had only one compound syntactic domain, T (e.g., (8.2)). Here in (9.1) there are six compound domains; but only one of them, B, exists to impose a structural constraint within the grammar. The other four auxiliary compound domains (e, S, A, V) are provided for convenient reference elsewhere, such as in specifying semantic functions and reduction rule schemata. Notably, S is the domain of terms that self-evaluate,

$$[\text{eval } S\ e] \longrightarrow S\,, \tag{9.2}$$

while V is the domain of terms that can be passed as operands to an operative, and A exists to facilitate the definition of V.

An auxiliary compound domain (i.e., other than the primary domain T and the simple domains, here d, o, s) acts as a structural constraint just when it occurs on the right-hand side of a nontrivial rule — the only such occurrence here being the B in "$e ::= \langle\!\langle B^* \rangle\!\rangle$". When a grammar is designed to avoid such structural constraints, it generally has a great many occurrences of T on the right-hand sides of nontrivial rules (in place of some other, structurally constraining nonterminal); and each occurrence of T on a right-hand side is another possible position for the meta-variable $\Box$ in contexts. Thus, the fewer structural constraints there are, the more is implied by the property of compatibility, fostering a strong reduction relation, and through it a strong equational theory.

Also, secondarily, lack of structural constraints simplifies the treatment of contexts. A naive mechanical generation of syntax rules for contexts C would introduce a distinct context-nonterminal for each nonterminal in the syntax of T that can be an ancestor of T; here, one would have nonterminals C, C_B, C_e, C_S, and C_A. (V isn't a nonterminal, so one wouldn't automatically have C_V.) But as long as the grammar is unambiguous, and each recursive occurrence of T only involves a single nontrivial syntax rule (e.g.,

$$T \ ::= \ A \ ::= \ [\text{eval } T\ T]\,), \tag{9.3}$$

we can abbreviate each corresponding occurrence of C to a single rule (in the example,

$$C \ ::= \ [\text{eval } C\ T] \mid [\text{eval } T\ C]\,), \tag{9.4}$$

generating the nontrivial syntax directly from C and eliding the intermediate nonterminals. The entire syntax for $\wr_e$-contexts only requires one auxiliary nonterminal, C_B, for the structural constraint implied by B on the nontrivial right-hand side $\langle\!\langle B^* \rangle\!\rangle$:

$$
\begin{aligned}
C \;::=\; & \Box \mid [\text{eval } C\ T] \mid [\text{eval } T\ C] \\
& \mid [\text{combine } C\ T\ T] \mid [\text{combine } T\ C\ T] \mid [\text{combine } T\ T\ C] \\
& \mid (C\ .\ T) \mid (T\ .\ C) \\
& \mid \langle \text{operative } C\ T\ T\ T\rangle \mid \langle \text{operative } T\ C\ T\ T\rangle \\
& \mid \langle \text{operative } T\ T\ C\ T\rangle \mid \langle \text{operative } T\ T\ T\ C\rangle \\
& \mid \langle \text{applicative } C\rangle \\
& \mid \langle\!\langle B^*\ C_B\ B^*\rangle\!\rangle \qquad \text{(Contexts)} \\
C_B \;::=\; & s \leftarrow C \qquad \text{(Binding contexts)}\,.
\end{aligned}
\tag{9.5}
$$

Constraints on the form of redexes, or of reducts, should be deferred to the reduction rule schemata, since one could imagine a different reduction relation on the same syntax without those constraints. (For example, in $\mathring{\jmath}_e$-calculus we don't syntactically constrain the first two subterms of a compound operative term, even though we may (and, in fact, do) require fully determined parameter trees when constructing an operative.) Constraints within the grammar should be limited to factors intrinsic to the function of the structure so constrained — as in the case of environments, whose whole reason for being would collapse if they were not lists of bindings.

The domain name *Values* is chosen for consistency with Plotkin's usage ([Plo75]). The immediate technical significance of the value domain is that it defines eager argument evaluation: values are what the argument-evaluation terms have to be reduced to before an applicative call, in order for the argument evaluation to be considered eager. The subjective underlying principle is that in any value V, a reducible subterm must represent *potential* subcomputation, to be realized only when V is invoked by a surrounding evaluation process. Applicatives may be unwrapped without invoking them, so the body of an applicative value must also be a value. The body of a compound operative, though, represents a computation to be performed just when the operative is called, so the operative is a value even if its body isn't; the analogous situation in Plotkin's λ_v-calculus (§8.3.2) is that a λ-expression is a value even though its body may be reducible.

The treatment of environments as values engages deep issues in the treatment of variables.

Explicit evaluation explicitly distinguishes between collecting operands into a local environment, and distributing them from there to the body of a compound operative; whereas the β-rules of λ-calculi collect and distribute in a single atomic step, via substitution. If operands are distributed via substitution, then variables can only be values (i.e., can only qualify as eagerly evaluated) if all operands are reduced to values before being distributed by substitution, so that all substitutions actually performed preserve the property of being a value; and since distribution is linked with collection, this means that substitutable variables can only be values if all argument evaluation is eager. However, variables distributed via explicit-evaluation environments —a.k.a. *symbols*— are not subject to this limitation. A symbol can be treated as a value even if it is bound to a non-value in the relevant environment, exactly because the symbol

can only be replaced by the non-value when its lookup is explicitly directed by a surrounding eval — so that the surrounding eval term isn't a value, but the symbol within it is.

So, if we define all environments to be values, and thus treat bound terms in an environment as potential computations awaiting invocation (i.e., lookup), we can support both eager and lazy argument evaluation under different circumstances *in the same calculus*, without compromising the status of symbols as values. This also has the merits that (1) by not requiring values in bindings, we avoid imposing a syntactic constraint for the preferably reduction-schematic choice of eager argument evaluation; and (2) by admitting more reducible subterms, we introduce greater flexibility in order of reduction, and thus strengthen the equational theory.

The semantic function for concatenating environments must account for the environment normalization constraints of the grammar (a small price to pay for simplifying proofs involving the schemata). For convenience of notation, we extend the ordering of symbols to bindings, with $(s \leftarrow T) \leq (s' \leftarrow T')$ iff $s \leq s'$; and universally quantify ω over B^*. The auxiliary environment functions are

$\mathcal{V}_e$-calculus.
Auxiliary functions (for managing environments):

$$
\begin{aligned}
\langle\!\langle\rangle\!\rangle \cdot e &= e \\
e \cdot \langle\!\langle\rangle\!\rangle &= e \\
\langle\!\langle B_1 \rangle\!\rangle \cdot \langle\!\langle B_2\, \omega \rangle\!\rangle &= \begin{cases} \langle\!\langle B_1 B_2\, \omega \rangle\!\rangle & \text{if } B_1 < B_2 \\ \langle\!\langle B_2 \rangle\!\rangle \cdot (\langle\!\langle B_1 \rangle\!\rangle \cdot \langle\!\langle \omega \rangle\!\rangle) & \text{if } B_2 < B_1 \\ \langle\!\langle B_1\, \omega \rangle\!\rangle & \text{otherwise} \end{cases} \\
\langle\!\langle \omega\, B \rangle\!\rangle \cdot e &= \langle\!\langle \omega \rangle\!\rangle \cdot (\langle\!\langle B \rangle\!\rangle \cdot e) \\
\mathit{lookup}(s, \langle\!\langle s' \leftarrow T \rangle\!\rangle \cdot e) &= \begin{cases} T & \text{if } s = s' \\ \mathit{lookup}(s, e) & \text{otherwise} \end{cases} \\
\mathit{match}(\texttt{\#ignore}, T) &= \langle\!\langle\rangle\!\rangle \\
\mathit{match}(\texttt{()}, \texttt{()}) &= \langle\!\langle\rangle\!\rangle \\
\mathit{match}(s, T) &= \langle\!\langle s \leftarrow T \rangle\!\rangle \\
\mathit{match}(\texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}, & \\
\quad \texttt{(}T_1'\ \texttt{.}\ T_2'\texttt{)}) &= \mathit{match}(T_1, T_1') \cdot \mathit{match}(T_2, T_2') \\
& \qquad \text{if } T_1, T_2 \text{ have no symbols in common.}
\end{aligned}
\tag{9.6}
$$

The reduction rule schemata fall into two groups: those concerning only structures of the general syntax from (9.1), and those concerning the behaviors of particular primitive operatives. We adopt the usual Lisp shorthand for eliding dots and internal parentheses in lists (§2.2.1).

There are six general schemata: three schemata for evaluating combinations, which were given (modulo semantic variable and reduction-relation names) in §4.3.1;

self-evaluation Schema (9.2) and a symbol-evaluation schema, which were implicit in §4.3.1; and an applicative-evaluation schema.

$\mathfrak{l}_e$-calculus.
Schemata (general):

$$
\begin{aligned}
&[\text{eval } S\ e] \longrightarrow S && \text{(9.7S)}\\
&[\text{eval } s\ e] \longrightarrow \mathit{lookup}(s,e) \quad \text{if } \mathit{lookup}(s,e) \text{ is defined} && \text{(9.7s)}\\
&[\text{eval } (T_1\ .\ T_2)\ e] \longrightarrow [\text{combine } [\text{eval } T_1\ e]\ T_2\ e] && \text{(9.7p)}\\
&[\text{eval } \langle \text{applicative } T\rangle] \longrightarrow \langle \text{applicative } [\text{eval } T]\rangle && \text{(9.7a)}\\
&[\text{combine } \langle \text{operative } V_1\ V_2\ T\ e_1\rangle\ V_3\ e_2]\\
&\longrightarrow [\text{eval } T\ \ \mathit{match}((V_1\ .\ V_2),(V_3\ .\ e_2))\cdot e_1] \quad \text{if } \mathit{match}((V_1\ .\ V_2),(V_3\ .\ e_2)) \text{ is defined} && \text{(9.7}\beta\text{)}\\
&[\text{combine } \langle \text{applicative } T\rangle\ (T_1\ \ldots T_m)\ e]\\
&\longrightarrow [\text{combine } T\ ([\text{eval } T_1\ e]\ \ldots [\text{eval } T_m\ e])\ e]\,. && \text{(9.7}\gamma\text{)}
\end{aligned}
\tag{9.7}
$$

As in §4.3.1, the body of a compound operative is limited to one expression since, without side-effects, sequentially evaluating a sequence of expressions would serve no purpose. Schema (9.7β) imposes eager argument evaluation on compound applicatives, by refusing to apply an underlying compound operative until the arguments have been reduced to values V_3. (We will want eager argument evaluation in the impure calculi of Chapters 10–12, so for convenience then we set things up that way now; but we could also construct a lazy-argument-evaluation calculus —$\mathfrak{l}_{en}$-calculus, versus $\mathfrak{l}_{ev}$-calculus— simply by using T_3 in the schema rather than V_3.)

As an alternative to Schema (9.7a), we could have depended applicatives from nonterminal S rather than T,

$$S ::= \langle \text{applicative } T\rangle\,; \tag{9.8}$$

evaluation of applicatives would then be covered by self-evaluation Schema (9.7S),

$$[\text{eval } \langle \text{applicative } T\rangle] \longrightarrow_{\mathfrak{l}e} \langle \text{applicative } T\rangle\,. \tag{9.9}$$

These two strategies (Syntax (9.8) and Schema (9.7a)) coincide when the underlying content of the applicative is self-evaluating, but may disagree in general. That is, both strategies have $[\text{eval } \langle \text{applicative } S\rangle] =_{\mathfrak{l}e} \langle \text{applicative } [\text{eval } S]\rangle$ for all S, but only Schema (9.7a) has $[\text{eval } \langle \text{applicative } T\rangle] =_{\mathfrak{l}e} \langle \text{applicative } [\text{eval } T]\rangle$ for all T. This difference should have no impact on correct programs, because the content of an applicative, if type correct, is always a combiner, hence all correctly typed applicatives eventually self-evaluate. However, in the absence of obvious semantic guidance on the choice, we prefer the strategy that maximizes equalities rather than inequalities.

By preference, most or all of the reduction rules for primitive operatives should be analogous to δ-rules of $\lambda\delta$-calculi (§8.2): because traditional δ-rules conform to a

simple, versatile fixed pattern proven not to compromise the Church–Rosser-ness of λ-calculus, they allow a programming language to be conveniently populated by any number of pedestrian primitive functions without laboriously extending the Church–Rosser proof for each individual function. The notion of δ-rules requires, however, some adaptation for use with $\mathfrak{f}$-calculi. The constraints on δ-rules for λ-calculi are that the arguments should be closed normal forms, and that the result should be closed:

- If the arguments weren't required to be normal, they might be reduced; and if the arguments weren't required to be closed, they might be altered by substituting for free variables. Either of these situations might destroy Church–Rosser-ness, because λ-calculus δ-rules don't have to be invariant under either kind of change to the arguments. (Lack of invariance under argument reduction was the root of non-Church–Rosser-ness in the $\lambda\mathcal{Q}$-calculus of §8.4.1, (8.47).)

- If the result weren't required to be closed, substitution for a variable that occurs free in the result could have a different effect if done before δ-reduction than if done after δ-reduction.[1]

$\mathfrak{f}_e$-calculus has nothing analogous to free variables: only an eval- or combine-operator can induce changes to a subterm,[2] and information cannot pass downward through an eval- or combine-operator ([eval ...] or [combine ...]). So a subterm can only change if it contains an active subterm, i.e., if it contains a redex, i.e., if it isn't a normal form. We don't want to require normal arguments, though, because we want some primitives to act on compound combiners that might not have normal forms (e.g., *$wrap*/*$unwrap*; re lack of normal forms, cf. λ_v-calculus, §8.3.2).

Definition 9.10 A $\mathfrak{f}_e$-calculus *δ-form* is a triple $\langle \pi_1, \pi_2, \pi_3 \rangle$ of semantic polynomials[3] over the term set $\mathfrak{T}_e$, such that

(1) in any term satisfying π_1, any active subterm must be entirely contained within a minimal nontrivial poly-context[4] that satisfies semantic variable V;

[1] Lack of invariance under substitution was the root of non-Church–Rosser-ness in the normal-argument λ-calculus variant discussed in Footnote 26 of §8.3.2.

[2] Strictly, this is true only because, in addition to omitting substitution from $\mathfrak{f}_e$-calculus, we are also handling environment normalization by means of (separately stated) grammar constraints rather than by reduction rule schemata.

[3] In case the notion of *semantic polynomial* falls short of intuitive evidence: a semantic polynomial has the structure of a term except that, at zero or more points in its syntax tree where subexpressions could occur, semantic variables occur instead, and for all permissible values of the semantic variables, the term will be syntactically correct. The differences from *poly-context* are that semantic variables are used instead of syntactic meta-variables, and the domains of the variables needn't be *Terms*, nor even subsets of *Terms*.

[4] A minimal nontrivial poly-context is one such that the only way to further weaken it, removing a constraint so as to be satisfied by a larger set of terms, would be to make it trivial (i.e., just a meta-variable). Formally, see Definition 13.1 in §13.1.1.

(2) in any term satisfying π_1, if a minimal nontrivial poly-context within the term satisfies semantic variable V, then every proper subterm of that poly-context in the term must be unconstrained by π_1;

(3) no semantic variable occurs more than once in π_1;

(4) π_2 is a semantic variable based on e (thus, quantified over *Environments*) that doesn't occur in π_1; and

(5) all semantic variables in π_3 also occur in π_1 or π_2.

A set Z of $\mathfrak{f}_e$-calculus δ-forms is *consistent* if for all $\langle \pi_1, \pi_2, \pi_3 \rangle, \langle \pi'_1, \pi'_2, \pi'_3 \rangle \in Z$, if $\langle \pi_1, \pi_2, \pi_3 \rangle$ is distinct from $\langle \pi'_1, \pi'_2, \pi'_3 \rangle$, then there is no term that satisfies both π_1 and π'_1. The class of all consistent sets of $\mathfrak{f}_e$-calculus δ-forms is called Δ_e.

A $\mathfrak{f}_e$-calculus *δ-rule schema* is a reduction rule schema of the form

$$[\text{combine } \pi_0 \ \pi_1 \ \pi_2] \ \longrightarrow \ \pi_3 \tag{9.11}$$

where $\pi_0 \in \mathit{PrimitiveOperatives}$ and $\langle \pi_1, \pi_2, \pi_3 \rangle$ is a $\mathfrak{f}_e$-calculus δ-form. ■

By Condition 9.10(1), if a redex under a δ-rule schema is transformed by reducing a subterm (under *any* schema of the calculus), the redex will still be a value; and by Condition 9.10(2), the redex will still be a redex under that same δ-rule schema. So the schema commutes with subterm reductions, and the addition of the δ-rule schema does not compromise Church–Rosser-ness. Condition 9.10(1) also enforces eager argument evaluation on δ-rules, analogously to Schema (9.7β) for compound operatives; that isn't necessary for a pure calculus, but will enforce determinism in the presence of side-effects in Chapter 10. Condition 9.10(2), which is stronger than strictly needed for the above (it says that π_1 never constrains proper subterms of an operative or environment — note that the left-hand side of Schema 9.7β *does* constrain proper subterms of an operative), will also be separately used in establishing well-behavedness properties in §14.2.2.

Here are the δ-rule schemata for the Kernel combiner-handling primitives:

$$\begin{aligned}
[\text{combine } \texttt{\$vau } (V_1\ V_2\ V_3)\ V_4] \ &\longrightarrow \ \langle \text{operative } V_1\ V_2\ V_3\ V_4 \rangle \\
[\text{combine } \texttt{\$wrap } (V_1)\ V_2] \ &\longrightarrow \ \langle \text{applicative } V_1 \rangle \\
[\text{combine } \texttt{\$unwrap } (\langle \text{applicative } V_1 \rangle)\ V_2] \ &\longrightarrow \ V_1 \,.
\end{aligned} \tag{9.12}$$

In general,

$\mathfrak{f}_e$-calculus.
Auxiliary functions (for δ-rules):

$$\delta\colon \mathit{PrimitiveOperatives} \to \Delta_e \tag{9.13}$$

Schemata (for δ-rules):

$$[\text{combine } o \ \pi_1 \ \pi_2] \ \longrightarrow \ \pi_3 \quad \text{if } \langle \pi_1, \pi_2, \pi_3 \rangle \in \delta(o) \,. \tag{9.13δ}$$

9.2 Equational weakness of $\mathfrak{l}_e$-calculus

In λ-calculus, two λ-expressions are equal whenever their bodies are equivalent when evaluated; that is, under the usual implicit-evaluation treatment of λ-calculus,

$$T_1 =_\beta T_2 \quad \text{implies} \quad (\lambda x.T_1) =_\beta (\lambda x.T_2)\,. \tag{9.14}$$

The analogous result for $\mathfrak{l}_e$-calculus *would* be that

$$\begin{aligned} &\forall e,\ [\text{eval } T_1\ e] =_{\mathfrak{l}e} [\text{eval } T_2\ e] \\ \text{implies} \\ &\forall V_1, V_2, e, \qquad \langle \text{operative } V_1\ V_2\ T_1\ e\rangle \\ &\qquad\qquad =_{\mathfrak{l}e} \langle \text{operative } V_1\ V_2\ T_2\ e\rangle\,; \end{aligned} \tag{9.15}$$

but this is not true, because $\mathfrak{l}_e$-calculus does not initiate evaluation of the body of a compound operative until the operative is called, and thereby given an operand tree. Something new must be added to the calculus, to express —and act on— intent to *partially* evaluate the body. Moreover, this addition will have to be structural: we can't make do with a minor adjustment in the schemata such as the quote-eval device in $\lambda\mathcal{E}$-calculus (§8.4.1), because we declare our intent-to-evaluate via an evaluation context, $[\text{eval } \Box\ T]$; this presupposes a term T for the intended environment; and we can't construct a term for the local environment before actually calling the operative, because we don't know what to bind to the parameters. Partial evaluation requires a way to express incomplete environments.

The quote-eval trick worked for $\lambda\mathcal{E}$-calculus because the evaluation frame of that calculus, $\mathcal{E}\Box$, was already designed to do partial evaluation, by pausing on subterms $\mathcal{E}x$ until some other term T was *substituted* for x. We will express incomplete environments here by introducing a substitution mechanism into the $\mathfrak{l}_e$-calculus, allowing construction of a local environment for a compound operative by binding the parameters to syntactic variables, for which values will be substituted from an operand tree when it becomes available. We call the resulting environment-with-substitution system $\mathfrak{l}_p$-calculus (p being mnemonic for *Partial evaluation*).

9.3 $\mathfrak{l}_x$-calculus

As a prelude to the full $\mathfrak{l}_p$-calculus, we first work out the infrastructure for substitution in $\mathfrak{l}$-calculi, by constructing a $\mathfrak{l}$-calculus with substitution but without environments, called $\mathfrak{l}_x$*-calculus* (x being the semantic-variable base letter for variables).

In discarding the syntax for environments themselves (nonterminal e), we must excise the environment operands from active terms.

$\mathfrak{l}_x$-calculus.
Syntax (active terms):
A ::= [eval T] | [combine T_1 T_2] (Active terms). (9.16)

The syntactic domain of symbols does not occur naturally in $\mathfrak{f}_x$-calculus, because without environments there is no way to give them an evaluation rule that will be consistent with the later extension to $\mathfrak{f}_p$-calculus. In its place, we restore the syntactic domain of substitutable variables as in λ-calculus,

$$\begin{array}{l}\mathfrak{f}_x\text{-calculus.}\\ \text{Syntax (substitution):}\\ \qquad x \in \textit{Variables}\\ \qquad T ::= x \qquad \text{(Terms)}\,.\end{array} \tag{9.17}$$

Variables are values, because we have attached them to the syntax by $T ::= x$ rather than $A ::= x$. As noted earlier (in discussion following (9.5)), for the integrity of the calculus, substitution must preserve the property of being a value; therefore, since variables are values, they can only be substituted for by values. (We will enforce this, below, by defining substitution only of values ((9.19)), even though we could as easily have defined substitution of arbitrary terms.)

Since hygienic substitution renames bound variables (§3.3.3), syntactic equivalence $\equiv_\alpha$ is again up to renaming of bound variables (which it wasn't in $\mathfrak{f}_e$-calculus because there was no such thing as a bound variable).

Because $\mathfrak{f}_e$-calculus used environments, which seem to be large monolithic structures,[5] it seemed natural for $\mathfrak{f}_e$-calculus to use compound definiends so that each local environment would be constructed all at once. Without environments, though, there is no psychological bias to treat the parsing and distribution of operand trees monolithically; and when we restore environments together with substitutable variables (in $\mathfrak{f}_p$-calculus, §9.4), disposition of operand trees will be separate from environment construction (the latter taking place when a compound operative is constructed, the former when it is called). We are therefore free to choose between operand disposition in a single complex act directed by a compound definiend; or operand disposition in a series of simple acts directed by elementary building blocks, analogous to currying of λ-calculus (§9.2). Given the choice, we prefer elementary building blocks, on the principle that both the dispositions themselves and their interactions should be more accessible to study when decomposed into simple acts.

We introduce a set of three elementary building blocks from which to construct operatives that parse and substitutively distribute their operand trees: one to parse a dotted pair, one to parse nil, and one to substitute for a single variable (analogous to $(\lambda x.T)$ of λ-calculus). Expecting that the fine-grained building blocks will lead to deep nestings of combiner constructors, we adopt a compact symbolic style of combiner construction operators, retaining the more verbose keyword style of $\mathfrak{f}_e$-calculus only for operators designating action (i.e., eval and combine).

[5] In the presence of environment mutation, it is difficult *not* to think of environments as large structures with object identity (cf. §§5.3.2, 7.1.1). At this point, we are biased toward that view of them by programming experience; formally, environment mutation will not be addressed until Chapter 12.

$\mathit{\imath}_x$-calculus.

Syntax (compound combiners):

$$\begin{array}{lll} S & ::= & \langle \mathit{\imath}_2.T\rangle \mid \langle \mathit{\imath}_0.T\rangle \mid \langle \mathit{\imath} x.T\rangle \qquad \text{(Self-evaluating terms)} \\ T & ::= & \langle T\rangle \qquad\qquad\qquad\qquad\quad \text{(Terms)}\,. \end{array} \tag{9.18}$$

As usual, $\mathrm{FV}(T)$ denotes the set of free variables of T, the difference from λ-calculus being that variables are bound by $\mathit{\imath}$ rather than λ.

$\mathit{\imath}_x$-calculus.

Auxiliary functions (substitution):

$$\begin{array}{rcl} x_1[x_2 \leftarrow V] & = & \begin{cases} V & \text{if } x_1 = x_2 \\ x_1 & \text{otherwise} \end{cases} \\ \langle \mathit{\imath} x_1.T\rangle[x_2 \leftarrow V] & = & \langle \mathit{\imath} x_3.((T[x_1 \leftarrow x_3])[x_2 \leftarrow V])\rangle \\ & & \quad \text{where } x_3 \notin \{x_2\} \cup \mathrm{FV}(T) \cup \mathrm{FV}(V) \\ P[\vec{T}][x \leftarrow V] & = & P[\sum_k \vec{T}(k)[x \leftarrow V]] \\ & & \quad \text{if } P \text{ doesn't involve any syntactic variable}\,. \end{array} \tag{9.19}$$

The rule for $P[\vec{T}][x \leftarrow V]$ covers, in effect, "everything else": all the syntactic structures that substitution passes over without affecting them — both primitive terms that aren't syntactic variables (primitive data, primitive operatives, ignore, and nil), and non-binding compound structures ($\mathit{\imath}_2$- and $\mathit{\imath}_0$-operatives, applicatives, evals, combines, and pairs).

$\mathit{\imath}_x$-calculus.

Schemata (compound combiner calls):

$$\begin{array}{lll} & \text{[combine } \langle \mathit{\imath}_2.T_0\rangle \text{ (}T_1 \text{ . } T_2\text{)]} & \\ \longrightarrow & \text{[combine [combine } T_0\ T_1\text{] } T_2\text{]} & (9.20\mathit{\imath}_2) \\ \text{[combine } \langle \mathit{\imath}_0.T\rangle \text{ ()]} & \longrightarrow \text{ T} & (9.20\mathit{\imath}_0) \\ \text{[combine } \langle \mathit{\imath} x.T\rangle\ V\text{]} & \longrightarrow\ T[x \leftarrow V] & (9.20\beta) \\ & \text{[combine } \langle T_0\rangle \text{ (}T_1 \ldots T_m\text{)]} & \\ \longrightarrow & \text{[combine } T_0 \text{ ([eval } T_1\text{]} \ldots \text{[eval } T_m\text{])]}\,. & (9.20\gamma) \end{array} \tag{9.20}$$

Schema $(9.20\mathit{\imath}_2)$ is essentially a currying operation (§9.0.1). It has no part in enforcing eager argument evaluation, since non-terminating T_1 or T_2 will still prevent termination of the larger term after the currying; and in later impure calculi, with side-effects, the currying transformation will preserve ordering of side-effects from T_0, T_1, T_2 as well.

The evaluation schemata differ from those of $\mathit{\imath}_e$-calculus in three ways: removal of the second operand to eval (since there are no environments), deletion of the symbol-evaluation schema (since there are no symbols), and adoption of the more compact notation for applicatives.

$\mathfrak{l}_x$-calculus.

Schemata (evaluation):

$$\begin{aligned} &[\text{eval } S] \longrightarrow S && (9.21\text{S}) \\ &[\text{eval } (T_1 \ . \ T_2)] \longrightarrow [\text{combine } [\text{eval } T_1] \ T_2] && (9.21\text{p}) \\ &[\text{eval } \langle T \rangle] \longrightarrow \langle [\text{eval } T] \rangle \, . && (9.21\text{a}) \end{aligned} \tag{9.21}$$

The definition of δ-rules for $\mathfrak{l}_e$-calculus was exceptional in its lack of provisions for substitutable variables (the only calculus we will consider with this lack), while δ-rules for $\mathfrak{l}_x$-calculus are exceptional in their lack of environments (our only $\mathfrak{l}$-calculus with this lack).

Definition 9.22 A $\mathfrak{l}_x$-calculus *δ-form* is a tuple $\langle \pi_1, \pi_3 \rangle$ of semantic polynomials over the term set $\mathfrak{T}_x$, satisfying Conditions 9.10(1)–9.10(3) and, additionally,

(5) all semantic variables in π_3 also occur in π_1;

(6) a syntactic variable is bound by π_3 at the position of a semantic variable in π_3 iff it is bound by π_1 at the position of that semantic variable;

(7) a syntactic variable can occur in a term satisfying π_3 only within a part of the term matching a semantic variable in π_3;

(8) the class of terms satisfying π_1 is closed under substitution; and

(9) π_1 does not require any syntactic variable to be bound more than once, nor to occur both bound and free.

A set Z of $\mathfrak{l}_x$-calculus δ-forms is *consistent* if for all $\langle \pi_1, \pi_3 \rangle, \langle \pi_1', \pi_3' \rangle \in Z$, if $\langle \pi_1, \pi_3 \rangle$ is distinct from $\langle \pi_1', \pi_3' \rangle$, then there is no term that satisfies both π_1 and π_1'. The class of all consistent sets of $\mathfrak{l}_x$-calculus δ-forms is called Δ_x.

A (particular) $\mathfrak{l}_x$-calculus *δ-rule schema* is a reduction rule schema of the form

$$[\text{combine } \pi_0 \ \pi_1] \longrightarrow \pi_3 \tag{9.23}$$

where $\pi_0 \in (\mathit{PrimitiveOperatives} - \{\texttt{\$vau}\})$ and $\langle \pi_1, \pi_3 \rangle \in \Delta_x$. ■

When a term is reduced via $\longrightarrow^{\mathfrak{l}x}$, any subterm T_1 satisfying the left side of a δ-rule $\pi \longrightarrow \pi'$ may be subjected to substitution for one of its syntactic variables. However, Condition 9.22(8) guarantees that the subterm T_1' resulting from this substitution will still satisfy π_1; and Conditions 9.22(6) and 9.22(7) guarantee that if T_1 were reduced via $\pi \longrightarrow \pi'$ to T_2, and T_2 were subjected to the same substitution as T_1, the result T_2' of the substitution would be the same as that of reducing T_1' via $\pi \longrightarrow \pi'$. Thus, as for $\mathfrak{l}_e$-calculus, addition of the δ-rule does not compromise Church–Rosser-ness. (Condition 9.22(9) prevents a class of pathological cases that will be dealt with in §13.1.2; cf. Definition 13.43.)

In general,

λ_x-calculus.
Auxiliary functions (for δ-rules):

$$\delta\colon (\mathit{PrimitiveOperatives} - \{\texttt{\$vau}\}) \to \Delta_x \tag{9.24}$$

Schemata (for δ-rules):

$$[\text{combine } o\ \pi_1] \longrightarrow \pi_3 \quad \text{if } \langle \pi_1, \pi_3 \rangle \in \delta(o)\,. \tag{9.24δ}$$

No (natural) λ_x-calculus schema can be constructed for `$vau`, because its natural behavior is heavily dependent on a source definiend — which requires symbols, which λ_x-calculus doesn't have.

All together, the syntax and schemata for λ_x-calculus are

λ_x-calculus.
Syntax:

$$\begin{array}{lll}
d \in \mathit{PrimitiveData} & & \\
o \in \mathit{PrimitiveOperatives} & & \\
x \in \mathit{Variables} & & \\
S ::= d \mid o \mid \texttt{\#ignore} \mid \texttt{()} & & \\
\qquad \mid \langle \lambda_2.T \rangle \mid \langle \lambda_0.T \rangle \mid \langle \lambda x.T \rangle & \text{(Self-evaluating terms)} & \\
A ::= [\text{eval } T] \mid [\text{combine } T\ T] & \text{(Active terms)} & \\
T ::= S \mid x \mid \texttt{(}T\ \texttt{.}\ T\texttt{)} \mid \langle T \rangle \mid A & \text{(Terms)} & \\
V \in \{T \mid \text{every active subterm of } T & & \\
\qquad \text{is within an operative}\} & \text{(Values)} &
\end{array}$$

Schemata:

$$\begin{array}{ll}
[\text{eval } S] \longrightarrow S & (9.25S) \\
[\text{eval } \texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}] \longrightarrow [\text{combine } [\text{eval } T_1]\ T_2] & (9.25p) \\
[\text{eval } \langle T \rangle] \longrightarrow \langle [\text{eval } T] \rangle & (9.25a) \\
[\text{combine } \langle \lambda x.T \rangle\ V] \longrightarrow T[x \leftarrow V] & (9.25\beta) \\
[\text{combine } \langle T_0 \rangle\ \texttt{(}T_1 \ldots T_m\texttt{)}] & \\
\quad \longrightarrow [\text{combine } T_0\ \texttt{(}[\text{eval } T_1] \ldots [\text{eval } T_m]\texttt{)}] & (9.25\gamma) \\
[\text{combine } o\ \pi_1] \longrightarrow \pi_3 \quad \text{if } \langle \pi_1, \pi_3 \rangle \in \delta(o) & (9.25\delta) \\
[\text{combine } \langle \lambda_2.T_0 \rangle\ \texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}] & \\
\quad \longrightarrow [\text{combine } [\text{combine } T_0\ T_1]\ T_2] & (9.25\lambda_2) \\
[\text{combine } \langle \lambda_0.T \rangle\ \texttt{()}] \longrightarrow \mathrm{T}\,. & (9.25\lambda_0)
\end{array} \tag{9.25}$$

9.4 λ_p-calculus

We're now ready to implement partial evaluation in the presence of environments (per §9.2), by merging λ_e-calculus with λ_x-calculus.

The syntax of $\mathfrak{l}_p$-calculus is almost entirely a merge of the syntax rules of $\mathfrak{l}_e$-calculus and $\mathfrak{l}_x$-calculus. It contains all the syntax rules of $\mathfrak{l}_e$-calculus except the compound-combiner forms; and it contains all the syntax rules of $\mathfrak{l}_x$-calculus except the active terms. The only syntax entirely new to $\mathfrak{l}_p$-calculus is a single new elementary building block for compound operatives, to effect dynamic-environment capture.

$\mathfrak{l}_x$-calculus.
Syntax (environment capture):
$S ::= \langle \epsilon . T \rangle$
Schemata (environment capture):
$[\text{combine } \langle \epsilon . T_0 \rangle \ T_1 \ e] \longrightarrow$
$[\text{combine } [\text{combine } T_0 \ e \ \langle\!\langle\rangle\!\rangle] \ T_1 \ e] .$ (9.26ϵ)

(9.26)

The complete syntax is

$\mathfrak{l}_p$-calculus.
Syntax:
$d \in \textit{PrimitiveData}$
$o \in \textit{PrimitiveOperatives}$
$s \in \textit{Symbols}$ (with total ordering $\leq$)
$x \in \textit{Variables}$

$B ::= s \leftarrow T$ (Bindings)
$e ::= \langle\!\langle B^* \rangle\!\rangle$ (Environments)

$S ::= d \mid o \mid e \mid \texttt{\#ignore} \mid \texttt{()}$
$\quad \mid \langle \mathfrak{l}_2 . T \rangle \mid \langle \mathfrak{l}_0 . T \rangle$
$\quad \mid \langle \mathfrak{l} x . T \rangle \mid \langle \epsilon . T \rangle$ (Self-evaluating terms)
$A ::= [\text{eval } T \ T] \mid [\text{combine } T \ T \ T]$ (Active terms)
$T ::= S \mid s \mid x \mid \texttt{(}T \texttt{ . } T\texttt{)} \mid \langle T \rangle \mid A$ (Terms)

$V \in \{T \mid$ every active subterm of T
is within an operative or
environment$\}$ (Values)

where
bindings in an environment are in order by bound symbol; and
no two bindings in an environment bind the same symbol.

(9.27)

The environment concatenation and *lookup* functions are retained from $\mathfrak{l}_e$-calculus,

$\mathfrak{k}_p$-calculus.
Auxiliary functions (retained from $\mathfrak{k}_e$-calculus):

$$\begin{aligned}
\langle\!\langle\rangle\!\rangle \cdot e &= e \\
e \cdot \langle\!\langle\rangle\!\rangle &= e \\
\langle\!\langle B_1 \rangle\!\rangle \cdot \langle\!\langle B_2\, \omega \rangle\!\rangle &= \begin{cases} \langle\!\langle B_1 B_2\, \omega \rangle\!\rangle & \text{if } B_1 < B_2 \\ \langle\!\langle B_2 \rangle\!\rangle \cdot (\langle\!\langle B_1 \rangle\!\rangle \cdot \langle\!\langle \omega \rangle\!\rangle) & \text{if } B_2 < B_1 \\ \langle\!\langle B_1\, \omega \rangle\!\rangle & \text{otherwise} \end{cases} \\
\langle\!\langle \omega\, B \rangle\!\rangle \cdot e &= \langle\!\langle \omega \rangle\!\rangle \cdot (\langle\!\langle B \rangle\!\rangle \cdot e) \\
\mathit{lookup}(s, \langle\!\langle s' \leftarrow T \rangle\!\rangle \cdot e) &= \begin{cases} T & \text{if } s = s' \\ \mathit{lookup}(s, e) & \text{otherwise}\,. \end{cases}
\end{aligned} \tag{9.28}$$

The *match* semantic function of $\mathfrak{k}_e$-calculus is no longer useful, since compound definiends will be processed at operative construction rather than operative call (details later in the section).

The general schemata of $\mathfrak{k}_x$-calculus are modified only by straightforwardly reintroducing an environment operand to eval and combine (illustrated momentarily). The conditions for a δ-form require another revision, to merge the environment-based conditions from $\mathfrak{k}_e$-calculus Definition 9.10 with the substitution provisions of $\mathfrak{k}_x$-calculus Definition 9.22.

Definition 9.29 A $\mathfrak{k}_p$-calculus *δ-form* is a tuple $\langle \pi_1, \pi_2, \pi_3 \rangle$ of semantic polynomials over the term set $\mathbb{T}_p$, satisfying Conditions 9.10(1)–9.10(5), Conditions 9.22(7)–9.22(9), and

(6) a syntactic variable is bound by π_3 at the position of a semantic variable in π_3 iff it is bound by π_1 or π_2 at the position of that semantic variable;

(8a) the class of terms satisfying π_2 is closed under substitution; and

(9a) π_2 does not require any syntactic variable to be bound more than once, nor to occur both bound and free.

A set Z of $\mathfrak{k}_p$-calculus δ-forms is *consistent* if for all $\langle \pi_1, \pi_2, \pi_3 \rangle, \langle \pi'_1, \pi'_2, \pi'_3 \rangle \in Z$, if $\langle \pi_1, \pi_2, \pi_3 \rangle$ is distinct from $\langle \pi'_1, \pi'_2, \pi'_3 \rangle$, then there is no term that satisfies both π_1 and π'_1. The class of all consistent sets of $\mathfrak{k}_p$-calculus δ-forms is called Δ_p.

A (particular) $\mathfrak{k}_p$-calculus *δ-rule schema* is a reduction rule schema of the form

$$[\text{combine } \pi_0\ \pi_1\ \pi_2] \longrightarrow \pi_3 \tag{9.30}$$

where $\pi_0 \in (\mathit{PrimitiveOperatives} - \{\texttt{\$vau}\})$ and $\langle \pi_1, \pi_2, \pi_3 \rangle \in \Delta_p$. ■

Semantic function δ now has type $(\mathit{PrimitiveOperatives} - \{\texttt{\$vau}\}) \to \Delta_p$. A schema can be constructed for `$vau`, because $\mathfrak{k}_p$-calculus has symbols (unlike $\mathfrak{k}_x$-calculus, which had no `$vau`-schema for this reason); but because operand acquisition in $\mathfrak{k}_p$-calculus uses substitution, the natural behavior of `$vau` must introduce new syntactic

variables bound over the body of the combiner, requiring some non-polynomial (i.e, non-δ-rule) auxiliary semantic device to select new syntactic variables that do not occur free in the body. Semantic function *definiend* takes as input a definiend tree, as detailed in §4.2, and a set of proscribed syntactic variables; and outputs a context (to form an operative around the body), an environment (providing local bindings of parameter symbols to syntactic variables), and an updated set of proscribed syntactic variables. A second, higher-level semantic function *vau* uses *definiend* to translate the operand tree of **`$vau`** into an operative. We write $\mathcal{P}_\omega(Z)$ for the set of finite subsets of a set Z.

$\mathring{\iota}_p$-calculus.
Auxiliary functions (definiend compilation):

$$\begin{aligned}
&\textit{definiend} : \textit{Terms} \times \mathcal{P}_\omega(\textit{Variables}) \\
&\quad \xrightarrow{p} \textit{Contexts} \times \textit{Environments} \times \mathcal{P}_\omega(\textit{Variables}) \\
&\qquad \textit{definiend}(\texttt{()}, \mathcal{X}) = \langle\langle \mathring{\iota}_0.\Box\rangle, \langle\!\langle\rangle\!\rangle, \mathcal{X}\rangle \\
&\qquad \textit{definiend}(\texttt{\#ignore}, \mathcal{X}) = \langle\langle \mathring{\iota}x.\Box\rangle, \langle\!\langle\rangle\!\rangle, (\mathcal{X} \cup \{x\})\rangle \\
&\qquad\qquad \text{where } x \notin \mathcal{X} \\
&\qquad \textit{definiend}(s, \mathcal{X}) = \langle\langle \mathring{\iota}x.\Box\rangle, \langle\!\langle s \leftarrow x\rangle\!\rangle, (\mathcal{X} \cup \{x\})\rangle \\
&\qquad\qquad \text{where } x \notin \mathcal{X} \\
&\qquad \textit{definiend}(\texttt{(}T_1\ \texttt{.}\ T_2\texttt{)}, \mathcal{X}) = \langle\langle \mathring{\iota}_2.C_1[C_2]\rangle, (e_1 \cdot e_2), \mathcal{X}_2\rangle \\
&\qquad\qquad \text{where } \textit{definend}(T_1, \mathcal{X}) = \langle C_1, e_1, \mathcal{X}_1\rangle \\
&\qquad\qquad \text{and } \textit{definend}(T_2, \mathcal{X}_1) = \langle C_2, e_2, \mathcal{X}_2\rangle
\end{aligned} \tag{9.31}$$

$$\begin{aligned}
&\textit{vau} : \textit{Terms} \times \textit{Environments} \xrightarrow{p} \textit{Terms} \\
&\textit{vau}(\texttt{(}T_1\ \texttt{\#ignore}\ T_2\texttt{)}, e) = C[[\text{eval } T_2\ (e' \cdot e)]] \\
&\quad \text{where } \textit{definiend}(T_1, (\mathrm{FV}(T_2) \cup \mathrm{FV}(e))) = \langle C, e', \mathcal{X}\rangle \\
&\textit{vau}(\texttt{(}T_1\ s\ T_2\texttt{)}, e) = \langle \epsilon.(\mathring{\iota}x.C[[\text{eval } T_2\ (e' \cdot \langle\!\langle s \leftarrow x\rangle\!\rangle \cdot e)]])\rangle \\
&\quad \text{where } x \notin (\mathrm{FV}(T_2) \cup \mathrm{FV}(e)) \\
&\quad \text{and } \textit{definiend}(T_1, (\mathrm{FV}(T_2) \cup \mathrm{FV}(e) \cup \{x\})) = \langle C, e', \mathcal{X}\rangle
\end{aligned}$$

Schemata (**`$vau`**):

$$[\text{combine } \texttt{\$vau}\ V\ e] \longrightarrow \textit{vau}(V, e) \quad \text{if } V \text{ is a valid operand tree for } \texttt{\$vau}\,. \tag{9.31υ}$$

All together, the schemata are

$\mathfrak{l}_p$-calculus.
Schemata:

$$[\text{eval } S\ e] \longrightarrow S \tag{9.32S}$$

$$[\text{eval } s\ e] \longrightarrow \mathit{lookup}(s, e) \quad \text{if } \mathit{lookup}(s, e) \text{ is defined} \tag{9.32s}$$

$$[\text{eval } (T_1\ .\ T_2)\ e] \longrightarrow [\text{combine } [\text{eval } T_1\ e]\ T_2\ e] \tag{9.32p}$$

$$[\text{eval } \langle T \rangle\ e] \longrightarrow \langle [\text{eval } T\ e] \rangle \tag{9.32a}$$

$$[\text{combine } \langle \mathfrak{l} x.T \rangle\ V\ e] \longrightarrow T[x \leftarrow V] \tag{9.32β}$$

$$\begin{aligned} &[\text{combine } \langle T_0 \rangle\ (T_1 \ldots T_m)\ e] \\ \longrightarrow\ &[\text{combine } T_0\ ([\text{eval } T_1\ e] \ldots [\text{eval } T_m\ e])\ e] \end{aligned} \tag{9.32γ}$$

(9.32)

$$[\text{combine } o\ \pi_1\ \pi_2] \longrightarrow \pi_3 \quad \text{if } \langle \pi_1, \pi_e, \pi_3 \rangle \in \delta(o) \tag{9.32δ}$$

$$\begin{aligned} &[\text{combine } \langle \epsilon.T_0 \rangle\ T_1\ e] \\ \longrightarrow\ &[\text{combine } [\text{combine } T_0\ e\ \langle\!\langle\rangle\!\rangle]\ T_1\ e] \end{aligned} \tag{9.32ϵ}$$

$$\begin{aligned} &[\text{combine } \langle \mathfrak{l}_2.T_0 \rangle\ (T_1\ .\ T_2)\ e] \\ \longrightarrow\ &[\text{combine } [\text{combine } T_0\ T_1\ e]\ T_2\ e] \end{aligned} \tag{9.32$\mathfrak{l}_2$}$$

$$[\text{combine } \langle \mathfrak{l}_0.T \rangle\ ()\ e] \longrightarrow \mathrm{T} \tag{9.32$\mathfrak{l}_0$}$$

$$[\text{combine } \texttt{\$vau}\ V\ e] \longrightarrow \mathit{vau}(V, e)$$
if V is a valid operand tree for $\texttt{\$vau}$. (9.32$\boldsymbol{v}$)

Chapter 10

Impure vau calculi — general considerations

10.0 Introduction

While the pure $\mathfrak{f}$-calculi of Chapter 9 demonstrate that our fexpr strategy is not itself side-effect-ful, that does not necessarily imply that our strategy can coexist peacefully with other, side-effect-ful features in a calculus. We therefore present, in this and the next two chapters, impure $\mathfrak{f}$-calculi incorporating the basic imperative features of sequential control (Scheme-style continuations) and sequential state (mutable environments), comparable to Felleisen's $\lambda_v Cd$- and $\lambda_v S\rho$-calculi (§8.3.3).

Our treatment of imperative facilities is broadly similar to Felleisen's in that "bubbling-up" schemata shift each side-effect-ful directive from its source to the upper syntactic limit of its influence,[1] and then substitution broadcasts the directive downward again to all points concerned. Since substitution is already involved, the only interesting feature of $\mathfrak{f}_e$-calculus (i.e., absence of substitution) is precluded; so we derive our imperative $\mathfrak{f}$-calculi from $\mathfrak{f}_p$-calculus.

However, our imperative facilities use *separate* substitution devices, for a total of four classes of substitution — one for partial evaluation (introduced in $\mathfrak{f}_x$-calculus), one for control (in $\mathfrak{f}C$-calculus), and two for state (in $\mathfrak{f}S$-calculus).

This is a striking departure from λ-calculus. Church's 1932 logic achieved an elegant economy by using only λ for binding variables (viewing existential and universal quantification as higher-order functions); and in the λ-calculus subset of his logic, λ plays such a central and ubiquitous role that, once immersed in λ-calculus, one tends

[1] We retain the term "bubbling up" from Felleisen's work, where it is especially appropriate since most of his imperative constructs cause a sort of churning transformation as they move upward through a term, as, e.g., $((\sigma x_\sigma.T_1)V)T_2 \longrightarrow (\sigma x_\sigma.(T_1T_2))V$. This characteristic churning occurs because his constructs imitate the functional structure of λ (thus his σ, $\mathcal{C}$, etc.); and our imperative constructs make no attempt to follow this paradigm, so they cause much less churning, some moving upward with scarcely a ripple; but we still appreciate the imagery of a construct that naturally tends upward because it is lighter than its surrounding context.

to imagine that variables and substitution can only take the forms given to them by λ. However, when Felleisen used these forms in his imperative calculi, they spawned complications. The delimiting syntactic frames for side-effects are essentially non-functional binding constructs; this became manifest in the later compatible revisions of his calculi ($\lambda_v Cd$- and $\lambda_v S\rho$-calculi), where the delimiting frames relied heavily on λ, and thereby became gratuitously entangled with issues of function application.[2] $\imath$-calculi, on the other hand, have neither historical nor structural investment in a single binding construct: the traditional operator name "λ" is not used and, while the binding construct $\langle \imath x.\Box \rangle$ in $\imath_p$-calculus acts via compound-operative calls, it is evidently a convenient add-on (to strengthen the equational theory) rather than a necessity, since $\imath_e$-calculus handles the calls with no substitution at all.

Embracing the notion of substitution as add-on, we therefore simplify our imperative calculi by adding on separate, customized substitution devices. Each uses a separate syntactic domain of variables, to prevent the formal occurrence of meaningless interactions between the different substitution devices. Each separate domain of variables has its own distinct binding construct(s) — operative-delimiting frames for $\imath_p$-calculus, control-delimiting frames for $\imath C$-calculus, state-delimiting and state-query-delimiting frames for $\imath S$-calculus. Each binding construct distributes information across its delimited syntactic region via a substitution function customized to the particular directive transaction.

Control- and state-delimiting constructs differ from the operative construct by being *persistent*, in that each delimiting frame serves as a catalyst for an unbounded number of bubble/substitute transactions, whereas an operative frame is always consumed in the act of substitution. All three imperative binding frames differ from the operative construct by being themselves capable of bubbling upward (necessary for control frames so that first-class continuations can be upward funargs; for state frames so that first-class environments can be upward funargs; and for state-query frames so that code fragments can be reasoned about before their time-dependent bindings are known). They differ from each other in when they can bubble upward (state frames and state-query frames interact with each other); and they use substitution functions with fundamentally different behaviors (control frames use a substitution that splices contexts into a term at selected points; state frames use one that splices bindings into environments, and another that excises environment identities).

[2] Actually, the use of λ as sole binding construct in Felleisen's $\lambda_v S$-calculi is illusory; each $\lambda_v S$-calculus has two distinct binding constructs, using two distinct substitution functions (at least; $\lambda_v S^{\triangleright}$-calculus could be viewed as having three constructs and three functions). The illusion of a single construct is created by overloading the symbol "λ" for both constructs, and requiring the assignable-variable binding construct to imitate the applicative-combination behavior of the declarative construct.

10.1 Multiple-expression operative bodies

In Lisp, the body of a `$lambda` expression is not required to contain just one expression; it may be a list of expressions, of arbitrary length. When the constructed applicative is called, the expressions in the body are evaluated from left to right, and the result of evaluating the last expression is returned (§2.2.3.2). Kernel's `$vau` works similarly (§4.2).

Hitherto, we have briefly deferred the complication of multi-expression bodies in our formal treatments as irrelevant, on the grounds that multi-expression bodies would serve no purpose in the absence of side-effects (§4.3.1, §9.1). Now that we are about to treat side-effects, we still omit the complication from our formal systems, but provide a technical justification: the Kernel report specifies, given a primitive operative `$vau` that requires a single-expression body, how to derive a library operative `$vau` supporting multi-expression bodies by exploiting eager argument evaluation ([Shu09, §5.1.1 (`$sequence`) and §5.3.1 (`$vau`)]). The simple-body primitive `$vau` therefore suffices both computationally and abstractively (the latter because the more advanced syntax can be supported without resorting to a meta-circular evaluator).

10.2 Order of argument evaluation

Historically, Scheme has differed from most Lisps by pointedly not requiring that the arguments to an applicative be evaluated in any particular order (such as left-to-right). Kernel also leaves the argument evaluation order unspecified; and, moreover, where in Scheme this was merely a deliberate omission, in Kernel the deliberate omission has other, inclusive consequences for the semantics.[3]

Nondeterministic order of argument evaluation is a thorny problem for formal calculi. Our whole strategy for treating well-behavior is based on Church–Rosserness, which fails in the presence of observably nondeterministic behavior. Nor can we require observably deterministic behavior with our nondeterministic argument-evaluation order: for arbitrary arguments, it is formally undecidable whether all evaluation orders will produce the same results; so if we want to prove observably deterministic behavior, we have to impose restrictive types on the arguments (and the less restrictive we want the types to be, the more complicated the type system will become, per the Smoothness Conjecture, §1.1.2).

Fortunately, having recognized the strategic difficulty in well-behaved nondeter-

[3]Kernel observes a uniform set of policies for handling cyclic lists, based on the side-effect-ful-ness and specified order of list processing (following Kernel design guideline *G3*: dangerous things should be difficult to do by accident). If the order of argument evaluation were specified as right-to-left, a cyclic operand list would be an error. If the order were specified left-to-right, argument evaluation would loop through the cycle forever (or until stopped by an interrupt or jump). Since the order is unspecified, each operand is evaluated exactly once, and an argument list is constructed with the same cyclic structure as the operand list. [Shu09, §3.9 (Self-referencing data structures)].

minism, we have no need to grapple with it in the current work, because *the Kernel design doesn't prohibit performing argument evaluation in a fixed order.* The design consequences of Kernel's argument evaluation order are contingent only on the order not being specified in the design; they don't require the design to take any position on the question of formal nondeterminism, and it doesn't.[4] Implementations are permitted to *use* some particular evaluation order, and our calculi are free to do likewise. All of our impure $\mathfrak{l}$-semantics impose right-to-left argument evaluation. (It is of some interest to consider what other choices would equally well support the formal results of following chapters, and we will remark on this as occasion warrants.)

10.3 $\mathfrak{l}_i$-semantics

The pure $\mathfrak{l}_p$-calculus was presented in §9.4 without an associated $\mathfrak{l}_p$-semantics; but each impure extension of $\mathfrak{l}_p$-calculus will have an attendant semantics. So, in using $\mathfrak{l}_p$-calculus as a common starting point for the impure calculi, we present first a suitable semantics, imposing deterministic order of evaluation on the pure calculus.

The term-syntax of $\mathfrak{l}_i$-semantics is just that of $\mathfrak{l}_p$-calculus. We do make a cosmetic change to its specification from (9.27): we rename domain *Variables* to *Partial-EvaluationVariables* (recalling, from §9.2, that partial evaluation was the purpose for which those variables were introduced), and add a subscript p to their semantic variable base name, in anticipation of adding other, distinct variable domains.

[4]Most of the Scheme reports, too, use theoretically noncommittal words to permit arbitrary argument evaluation orders (*unspecified order*, [KeClRe98, §4.1.3], [ClRe91b, §4.1.3]; *order is not specified*, [Cl85, §II.1]; *in any order*, [SteSu78a, §A]). Only the *R3RS* uses a theoretically loaded word, *indeterminate* ([ReCl86, §4.1.3]).

$\mathfrak{l}_i$-semantics.
Syntax (terms and values):

$$
\begin{array}{rcll}
d & \in & \textit{PrimitiveData} & \\
o & \in & \textit{PrimitiveOperatives} & \\
s & \in & \textit{Symbols} \text{ (with total ordering } \leq) & \\
x_p & \in & \textit{PartialEvaluationVariables} & \\
B & ::= & s \leftarrow T & \text{(Bindings)} \\
e & ::= & \langle\!\langle B^* \rangle\!\rangle & \text{(Environments)} \\
S & ::= & d \mid o \mid e \mid \texttt{\#ignore} \mid \texttt{()} & \\
 & & \quad \mid \langle \mathfrak{l}_2.T \rangle \mid \langle \mathfrak{l}_0.T \rangle & \\
 & & \quad \mid \langle \mathfrak{l} x_p.T \rangle \mid \langle \epsilon.T \rangle & \text{(Self-evaluating terms)} \\
A & ::= & [\text{eval } T\ T] \mid [\text{combine } T\ T\ T] & \text{(Active terms)} \\
T & ::= & S \mid s \mid x_p \mid \texttt{(}T\ \texttt{.}\ T\texttt{)} \mid \langle T \rangle \mid A & \text{(Terms)} \\
V & \in & \{T \mid \text{every active subterm of } T & \\
 & & \qquad \text{is within an operative or} & \\
 & & \qquad \text{environment}\} & \text{(Values)}
\end{array}
\tag{10.1}
$$

where
bindings in an environment are in order by bound symbol;
and no two bindings in an environment bind the same symbol.

Following Felleisen (§8.3.3.1), we specify the permissible redex positions by a restricted domain of evaluation contexts:[5]

$\mathfrak{l}_i$-semantics.
Syntax (contexts):

$$
\begin{array}{rcll}
E & ::= & \Box \mid \texttt{(}T\ \texttt{.}\ E\texttt{)} \mid \texttt{(}E\ \texttt{.}\ V\texttt{)} \mid \langle E \rangle & \\
 & & \mid [\text{eval } T\ E] \mid [\text{eval } E\ V] & \\
 & & \mid [\text{combine } T\ T\ E] & \\
 & & \mid [\text{combine } T\ E\ V] & \\
 & & \mid [\text{combine } E\ V\ V] & \text{(Evaluation contexts)}\,.
\end{array}
\tag{10.2}
$$

Evaluation contexts regulate *subterm evaluation order*; for example, productions $E ::= \texttt{(}T\ \texttt{.}\ E\texttt{)} \mid \texttt{(}E\ \texttt{.}\ V\texttt{)}$ enforce right-to-left pair-subterm reduction: when a pair structure, such as a list, contains non-values, they are reduced from right to left within the structure. In the presence of side-effects, subterm evaluation order becomes deeply entangled with differences between schemata in the semantics versus in its corresponding calculus; while order of *argument* evaluation is a separate though related issue, regulated by the schema that schedules argument evaluations (in the calculus, Schema (9.32γ)). We will return to these points below in §10.5, where we will have the complete semantics to refer back to.

[5]Technically, V isn't a nonterminal because its definition isn't a syntax production, so V oughtn't occur in a syntax production. We use it so as a shorthand for specifying a term T in the production and then verbally imposing an external constraint on the syntax.

All the auxiliary semantic functions are carried over unchanged from $\mathring{l}_p$-calculus, and we do not repeat their definitions here (noting that, if we were to repeat their definitions, we would add subscripts p to semantic variables x; these definitions occurred in (9.19), (9.28), and (9.31)).

The computation schemata are

$\mathring{l}_i$-semantics.
Schemata:

$$
\begin{aligned}
&E[[\text{eval } S\ e]] \longmapsto E[S] && \text{(10.3S)}\\
&E[[\text{eval } s\ e]] \longmapsto E[\mathit{lookup}(s,e)] \\
&\qquad\qquad \text{if } \mathit{lookup}(s,e) \text{ is defined} && \text{(10.3s)}\\
&\quad E[[\text{eval } (V_1\ .\ V_2)\ e]] \\
&\longmapsto E[[\text{combine } [\text{eval } V_1\ e]\ V_2\ e]] && \text{(10.3p)}\\
&E[[\text{eval } \langle V\rangle\ e]] \longrightarrow E[\langle[\text{eval } V\ e]\rangle] && \text{(10.3a)}\\
&E[[\text{combine } \langle \mathring{l} x_p.T\rangle\ V\ e]] \longmapsto E[T[x_p \leftarrow V]] && \text{(10.3}\beta\text{)}\\
&\quad E[[\text{combine } \langle V_0\rangle\ (V_1 \ldots V_m)\ e]] \\
&\longmapsto E[[\text{combine } V_0\ ([\text{eval } V_1\ e] \ldots [\text{eval } V_m\ e])\ e]] && \text{(10.3}\gamma\text{)}\\
&E[[\text{combine } o\ \pi_1\ \pi_2]] \longmapsto E[\pi_3] \\
&\qquad\qquad \text{if } \langle \pi_1, \pi_2, \pi_3\rangle \in \delta(o) && \text{(10.3}\delta\text{)}\\
&\quad E[[\text{combine } \langle \epsilon.T\rangle\ V\ e]] \\
&\longrightarrow E[[\text{combine } [\text{combine } T\ e\ \langle\rangle]\ V\ e]] && \text{(10.3}\epsilon\text{)}\\
&\quad E[[\text{combine } \langle \mathring{l}_2.T\rangle\ (V_1\ .\ V_2)\ e]] \\
&\longrightarrow E[[\text{combine } [\text{combine } T\ V_1\ e]\ V_2\ e]] && \text{(10.3}\mathring{l}_2\text{)}\\
&E[[\text{combine } \langle \mathring{l}_0.T\rangle\ ()\ e]] \longrightarrow E[T] && \text{(10.3}\mathring{l}_0\text{)}\\
&E[[\text{combine } \texttt{\$vau}\ V\ e]] \longmapsto E[vau(V,e)] \\
&\quad \text{if } V \text{ is a valid operand tree for } \texttt{\$vau}\,. && \text{(10.3}\upsilon\text{)}
\end{aligned}
\qquad (10.3)
$$

10.4 Alpha-renaming

Although the separate domains of variables in our impure calculi will have distinct binding constructs and customized substitution functions, the substitution functions on one domain of variables cannot entirely ignore the other domains of variables. Most 'substitution', whatever its precise form, involves intermixing fragments of terms; and intermixing of term fragments has the potential to capture variables of *any* kind (§3.3), regardless of which kind of variable originally motivated the intermixing. Therefore, the preventative measures to maintain hygiene —namely, selective α-renaming of bound variables— must account for all kinds of variables at once, rather than just the one kind whose substitution motivated its use.

Fortunately, a single α-renaming function can be devised that encompasses the hygiene needs of all substitution functions, and whose provision for each kind of variable is limited to a clause specifying its behavior on the binding construct for variables of that kind. As long as we define each substitution function by invoking the universal α-renaming function, each time we add a new domain of variables we have only to specify α-renaming on its binding construct, to preserve hygiene on all substitution functions (past, present, and future). The task of defining n separate domains of variables is therefore linear in n — rather than quadratic in n, as it would be if we had to provide separately for each variable domain in each substitution function.

To implement this strategy, we define each hygienic substitution function to be the composition of an unhygienic variant with function α. The unhygienic variant, noted by using floor-brackets "$\lfloor\,\rfloor$" rather than full square brackets "$[\,]$",[6] simply assumes that no variable renaming is required to avoid capturing. Function α takes as input a term and a set of proscribed variables, and returns a sanitized version of the term in which all bound variables have been renamed to avoid the proscribed set. In the case of ordinary partial-evaluation substitution,

$\natural_i$-semantics.
Auxiliary functions (substitution):

$$\begin{aligned}
T[x_p \leftarrow V] &= \alpha(T, \{x_p\} \cup \mathrm{FV}(V) \cup \mathrm{FV}(T))\lfloor x_p \leftarrow V \rfloor \\
x'_p\lfloor x_p \leftarrow V \rfloor &= \begin{cases} V & \text{if } x'_p = x_p \\ x'_p & \text{otherwise} \end{cases} \\
P[\vec{T}]\lfloor x_p \leftarrow V \rfloor &= P[\textstyle\sum_k \vec{T}(k)\lfloor x_p \leftarrow V \rfloor] \\
&\quad \text{if } P \text{ doesn't involve any subterm belonging to} \\
&\quad\quad \mathit{PartialEvaluationVariables} \\
\alpha(\langle \natural x_p.T\rangle, \mathcal{X}) &= \langle \natural x'_p.(\alpha(T, \mathcal{X} \cup \{x_p, x'_p\})\lfloor x_p \leftarrow x'_p \rfloor)\rangle \\
&\quad \text{where } x'_p \notin \mathcal{X} \\
\alpha(P[\vec{T}], \mathcal{X}) &= P[\textstyle\sum_k \alpha(\vec{T}(k), \mathcal{X})] \\
&\quad \text{if } P \text{ doesn't bind any syntactic variable}.
\end{aligned} \tag{10.4}$$

This is straightforwardly equivalent (for $\natural_p$-calculus and $\natural_i$-semantics) to the earlier definition from (9.19).

[6] The intended mnemonic is that hygienic substitution does at least as much as unhygienic substitution, and possible a little more, but there's a strict bound on how much more it might do.

10.5 Non-value subterms

Five of the $\mathfrak{k}$-semantic schemata (pair and applicative evaluation, and applicative, ϵ, and $\mathfrak{k}_2$ combination, (10.3p) (10.3a) (10.3γ) (10.3ϵ) and (10.3$\mathfrak{k}_2$)) do not merely wrap evaluation contexts around their $\mathfrak{k}_p$-calculus analogs ((9.32p) (9.32a) (9.32γ) (9.32ϵ) and (9.32$\mathfrak{k}_2$)), but also require values in positions where the $\mathfrak{k}_p$-calculus schemata allow arbitrary subterms— car and cdr of the pair in (10.3p), etc. The need for this restriction in the semantics is that, although the definition of *evaluation context* determines a unique path downward through the syntax tree of a term, it doesn't specify how far down to descend: a single large term T may be expressible in the form $E_k[T_k]$ for several different choices of E_k, T_k with T_k a $\mathfrak{k}_p$-calculus redex — necessarily, each such choice nested in its predecessor (e.g.,

$$[\text{combine } \langle[\text{eval } \texttt{not?}\ e]\rangle\ \texttt{(\#f)}\ e]\,, \tag{10.5}$$

which contains $\mathfrak{k}_p$-calculus redexes at evaluation contexts $\Box$ and [combine $\langle\Box\rangle$ `(#f)` e]). Placing value-subterm constraints on the pair/applicative schemata excludes all but the largest possible choice of E, ensuring deterministic order of computation.

The task for the corresponding calculus is then to admit multiple orders of reduction —to strengthen the equational theory— without altering the deterministic-order effect of any subterm on the rest of the computation. A subterm may have three kinds of 'effect on the rest of the computation': (1) any side-effect emitted by the subterm; (2) the value resulting from the subterm, if any; and (3) failure to reduce to a value, when that causes the surrounding computation to fail as well. We call (1) and (3) *non-local effects* (reserving the name *side-effect* for (1), an explicit upward-bubbling syntactic frame), and note that a value never has non-local effects: it cannot nonterminate (fail to reduce to a value); and, by design postulate (definition of *value*) in all our formal systems, it cannot ever emit a side-effect.

Requiring all active subterms to be contained within values, as in δ-rules (Condition (9.10(1))) and as in $\mathfrak{k}_i$-semantics (Schemata (10.3) — note that operatives are values by definition, so their bodies in Schemata (10.3β) etc. are unconstrained), limits non-local effects to those specified within the schema, where they are easy to regulate since their ordering is explicit. If a calculus schema is relaxed by allowing non-value subterms, there are three kinds of implicit ordering to be managed: ordering of non-local effects that were already latent in different subterms; ordering of latent effects of the subterms relative to side-effects that are introduced explicitly by the schema; and ordering of both of those —latent subterm effects and explicit schema effects— relative to effects that are *indirect consequences* of the schema.

Safety against the first two cases (latent subterm effects, and latent subterm effects with explicit schema effects) concerns only the schema itself; the definition of evaluation context, which determines the proper ordering of non-local effects; and some basic conventions about what a side-effect frame looks like, so that we actually know when we introduce one. Against the first case, it is sufficient that the schema does not alter the determined order of the effects of the subterms. Against the second

case, it is sufficient that the schema schedules any new explicit effects to occur after all the latent ones have completed. For example, pair evaluation Schema (9.32p) introduces no explicit side-effects, so it's safe against the second case; and against the first case, it preserves the left/right ordering of the subterms, which will suffice *if* evaluation contexts use the same subterm ordering for combines as they do for pairs (both right-to-left in (10.2)).

Whether Schema (9.32p) is safe against the third case —indirect consequences— is less obvious. Just as we assumed some basic conventions on what a side-effect frame looks like, we also assume that, besides side-effect frames, the only other possible redexes are eval frames and combine frames. Only redexes have the potential to have further, indirect consequences; so on the right-hand side of Schema (9.32p), the only expressions with this potential are the top-level combine frame, and the eval frame that is introduced around the operator. By treating the eval frame pessimistically as if it were already a side-effect, we can observe that, under the determined ordering of subterms, its consequences are positionally scheduled to occur *after* any effects of the operand subterm T_2 (an advantage for this calculus of right-to-left subterm ordering). What remains is to ensure that any indirect consequences of the eval frame will necessarily be scheduled after any effects that are already latent in its subterm T_1. This, though, is actually a recursive imposition —imposed on all the schemata of the calculus— of the constraint we are already trying to impose on each individual schema: that a top-level eval frame or combine frame will only induce side-effects that are scheduled to occur after all side-effects of the subterms. Any violation of this constraint has to start with some particular schema.

So as long as every schema ensures its own safety against cases one and two, and every schema avoids scheduling a non-top-level redex before the effects of some *other* subterm (as Schema (9.32p) avoids by scheduling the eval frame after any side-effects of T_2), the "recursive" subcase of case 3 will take care of itself.

10.6 $\mathfrak{l}_i$-calculus

Of the five $\mathfrak{l}_p$-calculus schemata subjected to value-subterm constraints by $\mathfrak{l}$-semantics, all but one of them satisfy the weaker effect-ordering safety requirements for an impure calculus. The outlier is (9.32γ), which —for nontrivial operand lists— interleaves argument evaluations with possibly-side-effect-ful subterm reductions. The only subterm whose side-effects are scheduled safely, to occur before those of any argument evaluations, is the rightmost subterm, a smallish detail that, for simplicity, we will not bother to exploit, although one certainly could do so. We provide safety in our impure $\mathfrak{l}$-calculi by basing them on a variant of $\mathfrak{l}_p$-calculus called $\mathfrak{l}_i$-calculus, which differs just by requiring the operator and operands of the γ-rule to be values.

The subscript i on the base letter is understood, and therefore omitted by convention, on the full names of impure $\mathfrak{l}$-calculi. (Thus, "$\mathfrak{l}C$-calculus" for "$\mathfrak{l}_iC$-calculus", etc.)

$\mathcal{l}_i$-calculus.
Schemata (amending $\mathcal{l}_p$-calculus):

$$\begin{array}{ll} & [\text{combine } \langle V_0 \rangle \ (V_1 \ \ldots \ V_m) \ e] \\ \longrightarrow & [\text{combine } V_0 \ ([\text{eval } V_1 \ e] \ \ldots \ [\text{eval } V_m \ e]) \ e] \qquad (10.6\gamma) \ . \end{array} \tag{10.6}$$

10.7 $\mathcal{l}_r$-calculi

When formally proving well-behavedness of impure $\mathcal{l}$-calculi in Chapter 14, it will be convenient to establish first proofs of well-behavedness for weaker variants of the calculi. For technical reasons (to be clarified in that chapter), these variants are called "regular"; they are named by adding a subscript r to the base letter — thus, $\mathcal{l}_r$-calculus (the generic regular calculus), $\mathcal{l}_r C$-calculus, etc.

Most of the constraints placed on the regular variants of the calculi are omissions of some subset of the schemata that involve impure frames; those omissions have no effect on the pure subset, $\mathcal{l}_i$-calculus, of the impure calculi, since $\mathcal{l}_i$-calculus evidently doesn't have any schemata involving impure frames. One constraint on the regular variants does affect the pure subset, though: the regular variants retain *all* the value-subterm constraints of $\mathcal{l}_i$-semantics (whereas $\mathcal{l}_i$-calculus does this only on the γ-rule). The generic $\mathcal{l}_r$-calculus is simply $\mathcal{l}_i$-semantics with the evaluation contexts removed from the schemata.

$\mathcal{l}_r$-calculus.
Schemata (amending $\mathcal{l}_i$-calculus):

$$\begin{array}{ll} [\text{eval } (V_1 \ . \ V_2) \ e] \longrightarrow [\text{combine } [\text{eval } V_1 \ e] \ V_2 \ e] & (10.7\text{p}) \\ [\text{eval } \langle V \rangle \ e] \longrightarrow \langle [\text{eval } V \ e] \rangle & (10.7\text{a}) \\ \quad [\text{combine } \langle \epsilon.T \rangle \ V \ e] & \\ \longrightarrow [\text{combine } [\text{combine } T \ e \ \langle\!\langle\rangle\!\rangle] \ V \ e] & (10.7\epsilon) \\ \quad [\text{combine } \langle \mathcal{l}_2.T \rangle \ (V_1 \ . \ V_2) \ e] & \\ \longrightarrow [\text{combine } [\text{combine } T \ V_1 \ e] \ V_2 \ e] & (10.7\mathcal{l}_2) \ . \end{array} \tag{10.7}$$

The equational strength of $\mathcal{l}_r$-calculus (compared to the purely reflexive equational theory of $\mathcal{l}_i$-semantics) comes entirely from compatibility — which should not be underestimated, noting that (10.6γ) systematically introduces parallel redexes, and that reduction of the body of an operative was the whole point of having substitution in the pure calculus.

The omitted schemata are always schemata that don't involve substitution (so that substitution can be dealt with by the generic theory, leaving only simpler non-substitutive cases to be added back in on a case-by-case basis), but whose omission leaves only schemata that cannot disable each other (as, for example, a $\mathcal{l}_p$-calculus $\mathcal{l}_2$-redex, per (9.32$\mathcal{l}_2$), could be temporarily disabled while a side-effect frame emitted

by T_2 bubbles up through it, as in

$$\begin{array}{ll} & [\text{combine } \langle \mathfrak{t}_2.T_0 \rangle \ (T_1 \ . \ [\text{side-effect } \langle \cdots \rangle \ T_2]) \ e] \\ \longrightarrow_\bullet & [\text{combine } \langle \mathfrak{t}_2.T_0 \rangle \ [\text{side-effect } \langle \cdots \rangle \ (T_1 \ . \ T_2)] \ e] \\ \longrightarrow_\bullet & [\text{side-effect } \langle \cdots \rangle \ [\text{combine } \langle \mathfrak{t}_2.T_0 \rangle \ (T_1 \ . \ T_2) \ e]] \end{array} \tag{10.8}$$

where the first and third terms are $\mathfrak{t}_2$-redexes but the second isn't).

The kinds of schemata that *may* be omitted by the regular calculi are

- garbage-collection schemata, most of which must be omitted because their use is contingent on some non-localized property of subterms that the generic theory doesn't provide for (such as a certain variable not occurring free in a subterm);
- bubbling-up schemata, which are often omitted if they don't involve substitution since they are major sources of interference with other schemata (as in (10.8)); and
- impure-frame simplification schemata that perform simple but potentially interfering transformations (often, *self*-interfering, as with a redex pattern $[a \ [a \ \Box]]$ in a term $[a \ [a \ [a \ T]]]$).

Chapter 11

Imperative control

11.0 Introduction

In this chapter, we introduce imperative control analogous to that of Felleisen's $\lambda_v C$-calculi. (Kernel's advanced continuation facilities ([Shu09, §7 (Continuations)]), which seem to require greater primitive support than offered here, have no bearing on the thesis of the dissertation.)

11.1 Common structures

Our control device uses two kinds of upward-bubbling syntactic frames:

[catch x_c $\Box$], the binding frame, which is initiated by a capture action (such as calling `call/cc`), and delimits the region within which x_c can be thrown to; and

[throw x_c $\Box$], which specifies the destination x_c, and delimits the term being sent to that destination.

The common syntax for control is

$\wr C$-calculus and $\wr C$-semantics.
Syntax (amending $\wr_i$-semantics):

$$\begin{aligned} x_c &\in \mathit{ControlVariables} \\ A_p &::= [\text{eval } T\ T] \mid [\text{combine } T\ T\ T] \quad \text{(Active partial-evaluation terms)} \\ A &::= A_p \mid [\text{catch } x_c\ T] \mid [\text{throw } x_c\ T] \quad \text{(Active terms)}\,. \end{aligned} \tag{11.1}$$

Because catch and throw are active, they can occur in a value only if contained within an operative or environment. Control variables are not terms.

The two activities of the control device are upward movement of a throw, and upward movement of a catch. When a throw bubbles upward, it deletes surrounding context until it arrives at a matching catch; as a calculus reduction,

$$\begin{array}{rl}[\text{catch } x_c \text{ E}[[\text{throw } x_c\ T]]] & \longrightarrow_{\wr C}^{*} [\text{catch } x_c\ [\text{throw } x_c\ T]] \\ & \longrightarrow_{\wr C} [\text{catch } x_c\ T]\,.\end{array} \tag{11.2}$$

This does not involve any substitution. When a catch bubbles upward, however, it must modify all matching throws. To see why, suppose a catch of variable x_c bubbles upward past a context E. (Assume x_c doesn't occur free in E.) Frame $E[[\text{catch } x_c\ \Box]]$ is transformed to $[\text{catch } x_c\ E[\Box]]$; but any matching subterm $[\text{throw } x_c\ T]$ that bubbles up to this frame would delete the E just within the catch; so, to preserve the meaning of the throw, $[\text{throw } x_c\ \Box]$ must be transformed to $[\text{throw } x_c\ E[\Box]]$. (To put it another way: $[\text{catch } x_c\ \Box]$ is the target of each matching $[\text{throw } x_c\ \Box]$, so as the target moves, the matching throws have to adjust their aim.) The substitution function to perform this transformation is

$\wr C$-calculus and $\wr C$-semantics.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[x_c \leftarrow C] & = & \alpha(T, \{x_c\} \cup \text{FV}(T) \cup \text{FV}(C))\lfloor x_c \leftarrow C \rfloor \\
[\text{throw } x_c\ T]\lfloor x_c' \leftarrow C \rfloor & = & \begin{cases} [\text{throw } x_c\ C[T\lfloor x_c' \leftarrow C \rfloor]] & \text{if } x_c = x_c' \\ [\text{throw } x_c\ \ (T\lfloor x_c' \leftarrow C \rfloor)] & \text{otherwise} \end{cases} \\
P[\vec{T}]\lfloor x_c' \leftarrow C \rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor x_c' \leftarrow C \rfloor] \\
 & & \text{if } P \text{ doesn't involve any throw} \\
T[x_c \leftarrow x_c'] & = & \alpha(T, \{x_c\} \cup \text{FV}(T) \cup \{x_c'\})\lfloor x_c \leftarrow x_c' \rfloor \\
[\text{throw } x_c\ T]\lfloor x_c' \leftarrow x_c'' \rfloor & = & \begin{cases} [\text{throw } x_c''\ (T\lfloor x_c' \leftarrow x_c'' \rfloor)] & \text{if } x_c = x_c' \\ [\text{throw } x_c\ (T\lfloor x_c' \leftarrow x_c'' \rfloor)] & \text{otherwise} \end{cases} \\
P[\vec{T}]\lfloor x_c' \leftarrow x_c'' \rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor x_c' \leftarrow x_c'' \rfloor] \\
 & & \text{if } P \text{ doesn't involve any throw} \\
\alpha([\text{catch } x_c\ T], \mathcal{X}) & = & [\text{catch } x_c'\ (\alpha(T, \mathcal{X} \cup \{x_c, x_c'\})\lfloor x_c \leftarrow x_c' \rfloor)] \\
 & & \text{where } x_c' \notin \mathcal{X}\,.
\end{array} \tag{11.3}$$

A control variable is free when it occurs in a throw not within a matching catch. α only has to be specified here for structures that bind control variables, as precisely all other cases are covered by (10.4).

Just as the syntactic domain of values was closed under substitutions $V_1[x_p \leftarrow V_2]$, it is also closed under substitutions $V[x_c \leftarrow C]$ since $\Box[x_c \leftarrow C]$ only modifies throws, which are already active.

In this type of substitution, function α alone is insufficient to guarantee hygiene. Hygiene also requires in general that C does not bind any variables — since if C did bind some variable x, $[\text{throw } x_c\ T]\lfloor x_c \leftarrow C \rfloor = [\text{throw } x_c\ C[T\lfloor x_c \leftarrow C \rfloor]]$ would

capture any free occurrences of x in T. As a matter of notational convenience, (11.3) isn't limited to the hygienic case; but in practice, all schemata that use this type of substitution will use non-binding C.

We make two changes to the definition of δ-rule from $\mathcal{l}_p$-calculus, Definition 9.29, by allowing polynomials over the extended term set $\mathfrak{T}_c$, rather than its subset $\mathfrak{T}_p$, and by excluding `$call/cc` from the set of δ-rule combiners (because, like `$vau`, its schema won't be strictly polynomial).

Definition 11.4 A $\mathcal{l}C$-calculus *δ-form* is a triple $\langle \pi_1, \pi_2, \pi_3 \rangle$ of semantic polynomials over the term set $\mathfrak{T}_c$ satisfying all the conditions of Definition 9.29 (9.10(1)–9.10(5), 9.22(7)–9.22(9), and 9.29(6)–9.29(9a)). The class of all consistent sets of $\mathcal{l}C$-calculus δ-forms is called Δ_c.

A $\mathcal{l}C$-calculus *δ-rule schema* is a reduction rule schema of the form

$$[\text{combine } \pi_0 \ \pi_1 \ \pi_2] \longrightarrow \pi_3 \tag{11.5}$$

where $\pi_0 \in (\mathit{PrimitiveOperatives} - \{\texttt{\$vau}, \texttt{\$call/cc}\})$ and $\langle \pi_1, \pi_2, \pi_3 \rangle$ is a $\mathcal{l}C$-calculus δ-form. ■

Semantic function δ now has type $(\mathit{PrimitiveOperatives} - \{\texttt{\$vau}, \texttt{\$call/cc}\}) \to \Delta_c$.

11.2 $\mathcal{l}C$-semantics

The additional computation rules for processing catch and throw are

$\mathcal{l}C$-semantics.
Schemata (catch and throw):

$$\begin{aligned}
E[[\text{catch } x_c \ T]] &\longmapsto [\text{catch } x'_c \ E[((T[x_c \leftarrow x'_c])[x'_c \leftarrow E])]] \\
&\quad \text{where } x'_c \notin \mathrm{FV}(E) \cup \mathrm{FV}([\text{catch } x_c \ T]) && \text{(11.6c)} \\
[\text{catch } x_c \ E[[\text{catch } x'_c \ T]]] &\longmapsto [\text{catch } x_c \ E[(((T[x'_c \leftarrow x''_c]) \\
&\qquad [x''_c \leftarrow E]) \\
&\qquad [x''_c \leftarrow x_c])]] \\
&\quad \text{where } x''_c \notin \mathrm{FV}(E) \cup \mathrm{FV}([\text{catch } x'_c \ T]) && \text{(11.6cc)} \\
[\text{catch } x_c \ V] &\longmapsto V \quad \text{if } x_c \notin \mathrm{FV}(V) && \text{(11.6g)} \\
[\text{catch } x_c \ E[[\text{throw } x_c \ T]]] &\longmapsto [\text{catch } x_c \ T] && \text{(11.6ct)}\,.
\end{aligned} \tag{11.6}$$

The third schema garbage-collects an unused catch; waiting until no other computation can be performed is a straightforward way to effect the garbage-collection without compromising determinism.

One can't just add these schemata onto $\mathcal{l}_i$-semantics, though, because all of the pre-existing computation schemata, (10.3), require a term of the form $E[A_p]$; introducing a top-level catch frame disables all of them. To correct this difficulty (which

only arises because the semantics isn't compatible), we introduce a schema that lifts each catch-less computation step to a catch-framed step:

$\not\ell C$-semantics.
Schemata (lifting unframed schemata):
$$[\text{catch } x_c\ E[A_p]] \longmapsto [\text{catch } x_c\ T] \quad \text{if } E[A_p] \longmapsto_c T\,. \tag{11.7}$$

(This is why we bothered to distinguish in the syntax, (11.1), between active terms A and active partial-evaluation terms A_p.)

11.3 $\not\ell C$-calculus

The computation schemata of our semantic systems are unbounded, in the sense that each schema requires its redex to match an evaluation context that spans the unbounded gap between the top level of the redex, and the targeted active subterm. The reduction rule schemata of our corresponding calculi, though, can afford to be strongly bounded, each matching its redex against a pattern of fixed, minimal size, focused narrowly on the neighborhood of an active subterm; the unbounded gap between redex and top-level term is bridged by compatibility. Fixed minimal patterns are simple, so we prefer them. Upward-bubbling frames move upward by just one level of syntax per reduction step; and frames interact with each other only when they actually make contact, with no intervening context at all.

We specify a single syntactic level of evaluation context, upward through which side-effects bubble, via a restricted syntactic domain of *singular* evaluation contexts (following [FeHi92, §3.1]). An evaluation context is singular iff it is nontrivial but is not the composition of any two nontrivial contexts:

$\not\ell$-calculus.
Syntax (contexts):
$$\begin{aligned} E^s ::= {} & (T\ .\ \square) \mid (\square\ .\ V) \mid \langle\square\rangle \\ & \mid [\text{eval } T\ \square] \mid [\text{eval } \square\ V] \\ & \mid [\text{combine } T\ T\ \square] \\ & \mid [\text{combine } T\ \square\ V] \\ & \mid [\text{combine } \square\ V\ V] \qquad \text{(Singular evaluation contexts)}\,. \end{aligned} \tag{11.8}$$

The syntactic domain of evaluation contexts is the closure of the singular evaluation contexts under n-ary composition. The general bubbling-up schema for throw is then

$$E^s[[\text{throw } x_c\ T]] \longrightarrow [\text{throw } x_c\ T]\,. \tag{11.9}$$

There is also a special bubbling-up schema for throw when its immediate context is another throw,

$$[\text{throw } x'_c\ [\text{throw } x_c\ T]] \longrightarrow [\text{throw } x_c\ T]\,. \tag{11.10}$$

This works because, algorithmically, the inner throw directs the result of T to destination x_c *rather than* to the outer throw; the destination of the outer throw is irrelevant, because no result is ever provided to it by the inner throw.

A catch can also bubble up through an evaluation context,

$$E^s[[\text{catch } x_c\ T]] \longrightarrow [\text{catch } x'_c\ E^s[((T[x_c \leftarrow x'_c])[x'_c \leftarrow E^s])] \quad \text{where } x'_c \notin \mathrm{FV}(E^s) \cup \mathrm{FV}([\text{catch } x_c\ T])\,. \tag{11.11}$$

There is no need for a throw to bubble up through a non-matching catch, because the catch can bubble up ahead of it until, assuming the throw does have a matching catch further up, the matching and non-matching catches merge.

In principle, a catch is actually a declarative construct —computationally it doesn't *do* anything, despite its potentially far-reaching substitutive consequences— so that one could allow a catch to bubble up through some non-evaluation contexts, such as $C = [\text{combine } \Box\ T_1\ e]$. This however would become fairly messy (under certain carefully circumscribed conditions, Church–Rosser-ness would require a catch to be able to *sink* back down into C, and undergo a sort of fission in doing so), so the current treatment omits it.

All together, the general schemata for $\lambda\!\!\!\!\!\!\lambda C$-calculus are

$\lambda\!\!\!\!\!\!\lambda C$-calculus.
Schemata (amending $\lambda\!\!\!\!\!\!\lambda_i$-calculus):

$$\begin{aligned}
&E^s[[\text{catch } x_c\ T]] \\
\longrightarrow\ &[\text{catch } x'_c\ E^s[((T[x_c \leftarrow x'_c])[x'_c \leftarrow E^s])] \\
&\quad \text{where } x'_c \notin \mathrm{FV}(E^s) \cup \mathrm{FV}([\text{catch } x_c\ T]) && \text{(11.12c)} \\
[\text{catch } x_c\ [\text{catch } x'_c\ T]] &\longrightarrow [\text{catch } x_c\ (T[x'_c \leftarrow x_c])] && \text{(11.12cc)} \\
[\text{catch } x_c\ T] &\longrightarrow T \quad \text{if } x_c \notin \mathrm{FV}(T) && \text{(11.12g)} \\
E^s[[\text{throw } x_c\ T]] &\longrightarrow [\text{throw } x_c\ T] && \text{(11.12t)} \\
[\text{throw } x'_c\ [\text{throw } x_c\ T]] &\longrightarrow [\text{throw } x_c\ T] && \text{(11.12tt)} \\
[\text{catch } x_c\ [\text{throw } x_c\ T]] &\longrightarrow [\text{catch } x_c\ T] && \text{(11.12ct)}\ .
\end{aligned} \tag{11.12}$$

The underlying operative of `call/cc` has schema

$$\begin{aligned}
&[\text{combine } \texttt{\$call/cc}\ (V_1)\ V_2] \\
\longmapsto\ &[\text{catch } x_c\ [\text{combine } V_1\ (\langle\langle\lambda\!\!\!\!\!\!\lambda_2.\langle\lambda\!\!\!\!\!\!\lambda x_p.\langle\lambda\!\!\!\!\!\!\lambda_0.[\text{throw } x_c\ x_p]\rangle\rangle\rangle\rangle)\ V_2]] \\
&\quad \text{where } x_c \notin \mathrm{FV}(V_1) \cup \mathrm{FV}(V_2)\,.
\end{aligned} \tag{11.13}$$

(This is the Scheme behavior of `call/cc`; in Kernel, the value passed to V_1 would be a first-class continuation, not a combiner.)

The regular variant calculus, $\lambda\!\!\!\!\!\!\lambda_r C$-calculus, retains the catch bubbling-up schema since it involves substitution, and must therefore omit the catch-throw simplification schema since the catch bubbling-up could disable it. The catch-catch and throw-throw simplifications are omitted since they would be self-interfering, and the garbage-collection since its use constraint is inherently non-regular. The only other general

schema, throw bubbling-up, is retained because there is simply nothing else left for it to interfere with.

$\not{t}_r C$-calculus.
Schemata (amending $\not{t}_r$-calculus):

$$\begin{array}{l} \quad E^s[[\text{catch } x_c\ T]] \\ \longrightarrow\ [\text{catch } x'_c\ E^s[((T[x_c \leftarrow x'_c])[x'_c \leftarrow E^s])] \\ \qquad \text{where } x'_c \notin \text{FV}(E^s) \cup \text{FV}([\text{catch } x_c\ T]) \\ E^s[[\text{throw } x_c\ T]]\ \longrightarrow\ [\text{throw } x_c\ T]\ . \end{array} \tag{11.14}$$

Schema (11.13) is also included in the regular variant $\not{t}_r C$-calculus.

Chapter 12

Imperative state

12.0 Introduction

In this chapter we introduce imperative state comparable to that of Felleisen's $\lambda_v S$-calculi, in the form of environment mutation. (Our mutable environments here are single-parented; Kernel supports multi-parent environments ([Shu09, §3.2, §4.2 (Environments)]), but multi-parented-ness of environments has no obvious analog in Felleisen's calculi.)

12.1 Common structures

As with imperative control, we describe first the syntax and auxiliary functions shared by both the semantics and the calculus ($\wr S$-semantics and $\wr S$-calculus); but, in contrast to the treatment of imperative control, here these common syntax and functions are sufficient only for the semantics. For the semantics goal of self-evident correctness, $\wr S$-semantics assumes that all bindings in all environments are mutable, and performs symbol lookup in a single atomic step. For the calculus goal of strong equational theory, $\wr S$-calculus provides additional syntax for degrees of immutability, with specialized mutable-to-immutable substitution functions so that stable bindings and environments can be exploited when they occur (as anticipated in Chapter 5); and provides additional syntax and substitution functions so that symbol lookup can be analyzed into multiple explicit steps.

12.1.1 State variables

A state variable x_s is the identity of a stateful environment. State variables are *compound* variables: they have the usual characteristics of variables (bound by a binding construct, subject to substitution functions), but also have operationally significant internal structure. Given a state variable x_s, the identity of its parent can

be uniquely reconstructed; and this parentage identification is invariant across both α-renaming of x_s and α-renaming of the parent of x_s.

Internally, an environment identity x_s is a [ ]-delimited nonempty string of primitive *state indices*,

$$\begin{array}{l} i \in \mathit{StateIndices} \text{ (with total ordering } \leq) \\ x_s ::= [i^+] \qquad \text{(State variables)}\,. \end{array} \tag{12.1}$$

The suffixes of x_s are the identities of its ancestors; thus, $[ii'i'']$ would have parent $[i'i'']$ and orphan grandparent $[i'']$. The leftmost index i in a state variable $x_s = [iw_i]$ distinguishes x_s from siblings with the same parent, $x'_s = [i'w_i]$; but i has no meaning independent of the suffix that follows it. That is, if $w_i \neq w'_i$, then the meaning of prefix i in $x_s = [iw_i]$ has nothing to do with its meaning in $x'_s = [iw'_i]$. In particular, α-renaming that replaces prefix i with i' in $[iw_i]$ would have no effect on the i prefix of $[iw'_i]$.

State renaming substitution uses notation $\square[x_s \leftarrow i]$, specifying that the prefixing index of x_s should be replaced with i. Quantifying semantic variables w_i over strings of state indices, the elementary operation is

$$\begin{array}{rcll} [w'_i i w_i][[iw_i] \leftarrow i'] & = & [w'_i i' w_i] & \\ [w'_i][[iw_i] \leftarrow i'] & = & [w'_i] & \text{if } iw_i \text{ isn't a suffix of } w'_i\,. \end{array} \tag{12.2}$$

Auxiliary function *path* maps a state variable x_s to the sequence of ancestors of x_s that are searched when looking up a symbol.

$$\begin{array}{l} \text{ł}S\text{-semantics.} \\ \text{Auxiliary functions (environment ancestry):} \\ \qquad \begin{array}{rcl} \mathit{path}([i]) & = & [i] \\ \mathit{path}([ii'w_i]) & = & [ii'w_i]\,\mathit{path}([i'w_i])\,. \end{array} \end{array} \tag{12.3}$$

State variables will usually be treated as atomic units. There will be a surgical intrusion of the internal structure of state variables into the schema for **`$vau`** (which must construct a child of its static environment); but the only widespread intrusions into the high-level treatment (i.e., above the level of defining certain auxiliary functions) will be (1) the existence of function *path*, (2) the possibility that substitutions for x_s may affect x'_s even if $x'_s \neq x_s$, and (3) the context-sensitive syntactic constraint that a free state variable cannot have a bound ancestor.

This syntactic constraint violates the principle of contextual locality (as discussed following (9.1)): not only does it restrict how terms can be constructed, and how contexts can be constructed, but it also restricts which terms are permissible in which contexts. Suppose x_s is, by its internal structure, a child of x'_s; context C binds x_s, but not x'_s; context C' binds x'_s, but not x_s; and term T contains a free occurrence of x_s. Then $C[T]$ is a term, and $C'[C[T]]$ is a term, but $C'[T]$ is not a term.

There is no purely syntactic way to eliminate the nonlocality, because it follows from the intrinsic purposes served by the syntactic elements involved (here, C', T, x_s, and x'_s), prior to the particular reduction rule schemata imposed on them.[1] A binding of x_s, as by C, uniquely determines the identity of the environment designated by free occurrences of x_s. In particular, α-renaming of x_s is possible iff the binding of x_s is available, because only then can we be sure that we have in hand all the instances of x_s that need to be renamed. Similarly, binding of x'_s, as by C', uniquely determines the identity of the environment designated by free occurrences of x'_s. However, because the ancestry of x_s is encoded within each occurrence of x_s, the identity of the parent x'_s of x_s is uniquely determined by a binding for x'_s (as in C'), *not* by a binding for x_s (as in C). Consider, then, what would happen if we α-renamed the x'_s bound by C' in $C'[T]$. The free occurrence of x_s in T signifies a child of the environment uniquely determined by the binding of x'_s in C'; therefore, the identity x_s of that environment must be updated to reflect the renaming of x'_s — but then, to guarantee consistency, *every* reference to the child environment x_s must be renamed at the same time; and we don't have all those references in hand, because the relevant binding of x_s isn't included in the given term, $C'[T]$. Thus, in order for α-renaming to function correctly, the binding of x'_s must encompass the bindings of its children.

As an alternative to the nonlocal constraint, we *could* change our understanding of binding, such that any substitution for x_s is blocked when it encounters a binding for any ancestor of x_s. In effect, free occurrences of x_s in T would not be free in $C'[T]$, but would not be explicitly bound, either. Special provisions would have to be made, throughout the machinery of the calculus, for these neither-this-nor-that variable occurrences: they would have to be viewed either as *permanently unbound*, or as *implicitly bound*. Terms with permanently unbound variable occurrences would be semantically useless, so admitting them to the syntax would add only manifestly useless formal equations to the theory. On the other hand, terms with implicitly bound variable occurrences could be trivially rewritten with explicit bindings, a pedestrian normalization task that would still require the rest of the calculus machinery to provide for the unnormalized terms. Therefore, if we are going to encode environment ancestry in the variables, we judge the nonlocal syntactic constraint superior to these alternatives.

If the ancestry of an environment were not encoded in the variables, there would be no need for the nonlocal syntax. However, for a usefully strong formal theory, we want to maximize local deductions about symbol lookups — and if a symbol binding isn't local to the environment at which its lookup starts, then the ancestry of that environment is prerequisite to any further reasoning about the lookup. Hence, locally encoding the ancestry strengthens the theory.

Any correct encoding of environment ancestry will be substantially equivalent to the one chosen here, inasmuch as it would logically require comparable supporting machinery in the same places in the calculus (as remarked earlier: auxiliary functions,

[1] Cf. the discussion of syntactic constraints following (9.5), §9.1.

schema for `$vau`, existence of function *path*, possibility that substitutions for one environment may affect other environments, and context-sensitive syntax constraint). The chosen encoding has particularly natural information granularity: When α-renaming an environment to avoid collisions, we are only allowed in general to modify that information that distinguishes the environment from its siblings, because in general we don't have access to the bindings of its ancestors and so cannot in any way alter the representations of their identities. A state index is *exactly* the unit of information we are able to modify during a single α-renaming, and dividing the encoded state variable into these units precisely minimizes the number of elements that need to be modified.

12.1.2 Environments and bindings

The binding frame for state variables has the form

$$[\text{state } X_s \ \Box]\,, \tag{12.4}$$

where X_s is a *state definiend*, specifying the identities of the environments defined by the frame, and $\Box$ is the delimited syntactic region within which the defined environments can be accessed.

Assignment is supported by a frame

$$[\text{set } [\![\omega_s]\!] \ \Box]\,, \tag{12.5}$$

where ω_s is a list of stateful bindings, each of the form $[x_s, s] \leftarrow V$. (The restriction of the right-hand sides of bindings to values V, rather than terms T as in the stateless calculi, will prevent a major sapping of equational strength in the stateful calculi.) Set frames are the sole repository of stateful bindings.

Operationally, all we need in the representation of a mutable environment is its identity,

$$e \ ::= \ \langle\!\langle x_s \rangle\!\rangle\,. \tag{12.6}$$

Lookup can then be handled with no additional syntax and no additional substitution, provided we don't care to partially evaluate incomplete subterms (which we never care to do in the semantics, but do routinely in the calculus), and are willing to embed a complete unspecified evaluation context E into the symbol-evaluation schema (which we do routinely in the semantics, but not in the calculus):

$$\begin{array}{ll} & [\text{state } X_s \ [\text{set } [\![\omega_s]\!] \ E[[\text{eval } s \ \langle\!\langle x_s \rangle\!\rangle]]]] \\ \longmapsto & [\text{state } X_s \ [\text{set } [\![\omega_s]\!] \ E[V]]] \\ & \quad \text{if } \mathit{lookup}([\mathit{path}(x_s), s], [\![\omega_s]\!]) = V \\ & \quad\quad \text{and } \mathit{path}(x_s) \text{ are all defined by } X_s\,. \end{array} \tag{12.7}$$

The state syntax common to both semantics and calculus is

$\wr S$-semantics.
Syntax (amending $\wr_i$-semantics):

$$
\begin{array}{rcll}
i & \in & \textit{StateIndices} \text{ (with total ordering } \leq) & \\
x_s & ::= & [i^+] & \text{(State variables)} \\
B_p & ::= & s \leftarrow V & \text{(Stateless bindings)} \\
B_s & ::= & [x_s, s] \leftarrow V & \text{(Stateful bindings)} \\
e & ::= & \langle\!\langle x_s \rangle\!\rangle & \text{(Environments)} \\
X_s & ::= & [x_s^*] & \text{(State definiends)} \\
A_p & ::= & [\text{eval } T\ T] \mid [\text{combine } T\ T\ T] & \text{(Active partial-evaluation terms)} \\
A & ::= & A_p \mid [\text{state } X_s\ T] \mid [\text{set } [\![B_s^*]\!]\ T] & \text{(Active terms)}
\end{array}
\tag{12.8}
$$

where

state variables in a state definiend are in order by first state index;
no two state variables in a state definiend have the same first state index;
bindings in a set list are in order primarily alphabetically by state variable, secondarily by symbol;
no two bindings in a set list have the same left-hand side; and
no free state variable in a state term [state X_s T] is descended from a state variable in X_s.

The definition of free variable set $\mathrm{FV}(T)$ has one subtle case, owing to the compound character of state variables. In a state term [state $[w_s]$ T], all proper ancestors of w_s (if any) are considered to "occur", and are therefore free unless also defined by the definiend $[w_s]$. That is,

$$
\begin{array}{rcl}
\textit{ancestors}(w_s) & = & \bigcup_{x_s \in w_s} \textit{path}(x_s) \\
\mathrm{FV}([\text{state } [w_s]\ T]) & = & (\mathrm{FV}(T) \cup \textit{ancestors}(w_s)) - w_s\,.
\end{array}
\tag{12.9}
$$

Under this definition, a state term [state $[[i''i'i][i]]$ T] would be syntactically illegal, under the last context-sensitive constraint of (12.8), because $[i]$ is defined by the definiend while its child $[i'i]$ occurs free.

$X_s \cdot X'_s$ denotes the sorted merge of definiends X_s, X'_s, provided no state variable in X_s has the same first state index as any state variable in X'_s. Semantic variables w_s are quantified over strings (vectors) of state variables. We extend the total ordering of state indices to partially order state variables by comparing first elements, $x_s \leq x'_s$ iff $x_s(1) \leq x'_s(1)$. (Also, besides treating strings as vectors, whenever convenient we use set operations to selectively delete elements from a string, e.g. "$w_s \cap \textit{path}(x_s)$" meaning the string of elements of w_s that are ancestors of x_s; or coerce strings to sets of their constituent elements, e.g. "$w_s \subseteq \mathrm{FV}(T)$" meaning every element of w_s belongs to $\mathrm{FV}(T)$.)

łS-semantics.
Auxiliary functions (state definiends):

$$\begin{array}{rcl}
[w_s] \cdot [\,] & = & [w_s] \\
[\,] \cdot [w_s] & = & [w_s] \\
[x_s] \cdot [x'_s w_s] & = & \begin{cases} [x_s x'_s w_s] & \text{if } x_s < x'_s \\ [x'_s] \cdot ([x_s] \cdot [w_s]) & \text{if } x'_s < x_s \end{cases} \\
[x_s x'_s w_s] \cdot X_s & = & [x_s] \cdot ([x'_s w_s] \cdot X_s)\,.
\end{array} \tag{12.10}$$

Delimited lists of stateful bindings, in set frames, are managed similarly to stateless environments, (9.28). We quantify semantic variables ω_s over sequences B_s^*, ω_p over B_p^*; and order stateful bindings (i.e., $B_s \leq B'_s$) primarily alphabetically by state variable, secondarily by symbol.

łS-semantics.
Auxiliary functions (stateful binding sets):

$$\begin{array}{rcl}
[\![\,]\!] \cdot [\![\omega_s]\!] & = & [\![\omega_s]\!] \\
[\![\omega_s]\!] \cdot [\![\,]\!] & = & [\![\omega_s]\!] \\
[\![B'_s]\!] \cdot [\![B_s\, \omega_s]\!] & = & \begin{cases} [\![B'_s B_s\, \omega_s]\!] & \text{if } B'_s < B_s \\ [\![B_s]\!] \cdot ([\![B'_s]\!] \cdot [\![\omega_s]\!]) & \text{if } B_s < B'_s \\ [\![B'_s\, \omega_s]\!] & \text{otherwise} \end{cases} \\
[\![\omega_s\, B_s]\!] \cdot [\![\omega'_s]\!] & = & [\![\omega_s]\!] \cdot ([\![B_s]\!] \cdot [\![\omega'_s]\!]) \\
\\
\mathrm{DV}_s([\![\,]\!]) & = & \{\} \\
\mathrm{DV}_s([\![[x_s, s] \leftarrow V\,]\!]) & = & \{x_s\} \\
\mathrm{DV}_s([\![\omega_s]\!] \cdot [\![\omega'_s]\!]) & = & \mathrm{DV}_s([\![\omega_s]\!]) \cup \mathrm{DV}_s([\![\omega'_s]\!]) \\
\\
\multicolumn{3}{l}{\mathit{lookup}([x_s, s], [\![\,[x'_s, s'] \leftarrow V\ \ \omega_s]\!])} \\
\qquad = & \multicolumn{2}{l}{\begin{cases} V & \text{if } [x_s, s] = [x'_s, s'] \\ \mathit{lookup}([x_s, s], [\![\omega_s]\!]) & \text{otherwise} \end{cases}} \\
\\
\multicolumn{3}{l}{\mathit{lookup}([x_s w_s, s], [\![\omega_s]\!])} \\
\qquad = & \multicolumn{2}{l}{\begin{cases} \mathit{lookup}([x_s, s], [\![\omega_s]\!]) & \text{if this is defined} \\ \mathit{lookup}([w_s, s], [\![\omega_s]\!]) & \text{otherwise} \end{cases}} \\
\\
x_s \times (s \leftarrow V) & = & [x_s, s] \leftarrow V \\
x_s \times \langle\!\langle\rangle\!\rangle & = & [\![\,]\!] \\
x_s \times \langle\!\langle B_p\, \omega_p \rangle\!\rangle & = & [\![x_s \times B_p]\!] \cdot (x_s \times \langle\!\langle \omega_p \rangle\!\rangle)\,.
\end{array} \tag{12.11}$$

Function $\mathrm{DV}_s([\![\omega_s]\!])$ extracts the set of state variables that should be defined by some enclosing state frame to support the left sides of bindings $[\![\omega_s]\!]$. $\mathrm{DV}_s([\![\omega_s]\!]) \subseteq \mathrm{FV}([\text{set } [\![\omega_s]\!]\ T])$.

Function $x_s \times \langle\!\langle \omega_p \rangle\!\rangle$ is an orthogonal alternative to redefining function *definiend*, which we assume unchanged from (9.31). Given partial-evaluation definiend X_p and

proscribed variables set $\mathcal{X}$, instead of modifying *definiend* to produce stateful bindings, we let it construct a stateless environment $\langle\!\langle \omega_p \rangle\!\rangle$ and retrofit the state variable to its contents, $x_s \times \langle\!\langle \omega_p \rangle\!\rangle$.

Because a state frame can bind multiple state variables in parallel, renaming state variables one at a time would be awkward. One would be forever picking apart compound state-definiends into their constituent parts, undermining the succinctness otherwise afforded by parallel binding, and with considerable intermediate-to-high-level exposure of the semi-encapsulated internal structure of the individual state variables — only to reassemble the disassembled compound definiends at earliest opportunity. Therefore, the state-variable renaming substitution function supports renaming multiple state variables in parallel. The general notation used is $\square[w_s \leftarrow w_i]$; each state variable $w_s(k)$ is renamed by changing its first state index to $w_i(k)$.

The empty string (for which we had no need before this) is denoted ϕ.

In a parallel renaming $x_s\lfloor w'_s \leftarrow w'_i\rfloor$, if x_s is descended from more than one of the w'_s, more than one of the indices in x_s must be changed — and this requires some care in the mechanical definition of the substitution, lest one index change $\square\lfloor w'_s(k) \leftarrow w'_i(k)\rfloor$ cause the target to no longer match the pattern $w'_s(k+j)$ for a later index change $\square\lfloor w'_s(k+j) \leftarrow w'_i(k+j)\rfloor$. For example, renaming $[i_2 i_1]\lfloor[i_1][i_2 i_1] \leftarrow i'_1 i'_2\rfloor$ ought to transform $[i_2 i_1]$ to $[i'_2 i'_1]$; but if the first index change $\square\lfloor[i_1] \leftarrow i'_1\rfloor$ is performed first, its result $[i_2 i'_1]$ won't match the pattern for the second index change, $\square\lfloor[i_2 i_1] \leftarrow i'_2\rfloor$. To elicit the correct behavior, when performing the first index change on the target $x_s = [i_2 i_1]$, we also perform that index change on the remaining patterns $w''_s = [i_2 i_1]$; the general relation is then

$$x_s\lfloor x'_s w'_s \leftarrow i' w'_i\rfloor \;=\; (x_s\lfloor x'_s \leftarrow i'\rfloor)\lfloor(w'_s\lfloor x'_s \leftarrow i'\rfloor) \leftarrow w'_i\rfloor\,, \tag{12.12}$$

and in this example,

$$\begin{aligned} [i_2 i_1]\lfloor[i_1][i_2 i_1] \leftarrow i'_1 i'_2\rfloor &= [i_2 i'_1]\lfloor[i_2 i'_1] \leftarrow i'_2\rfloor \\ &= [i'_2 i'_1]\lfloor\phi \leftarrow \phi\rfloor \\ &= [i'_2 i'_1]\,. \end{aligned} \tag{12.13}$$

$\natural S$-semantics.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[w_s \leftarrow w_i] & = & \alpha(T, w_s \cup \mathrm{FV}(T) \cup \\
 & & \qquad (w_s\lfloor w_s \leftarrow w_i\rfloor))\lfloor w_s \leftarrow w_i\rfloor \\
[w'_i i w_i]\lfloor [i w_i] \leftarrow i'\rfloor & = & [w'_i i' w_i] \\
x_s\lfloor x'_s \leftarrow i'\rfloor & = & x_s \qquad \text{if } x'_s \notin \mathit{path}(x_s) \\
x_s\lfloor \phi \leftarrow \phi\rfloor & = & x_s \\
x_s\lfloor x'_s w'_s \leftarrow i' w'_i\rfloor & = & (x_s\lfloor x'_s \leftarrow i'\rfloor)\lfloor (w'_s\lfloor x'_s \leftarrow i'\rfloor) \leftarrow w'_i\rfloor \\
\phi\lfloor w'_s \leftarrow w'_i\rfloor & = & \phi \\
(x_s w_s)\lfloor w'_s \leftarrow w'_i\rfloor & = & (x_s\lfloor w'_s \leftarrow w'_i\rfloor)(w_s\lfloor w'_s \leftarrow w'_i\rfloor) \\
\langle\!\langle x_s \rangle\!\rangle\lfloor w'_s \leftarrow w'_i\rfloor & = & \langle\!\langle x_s\lfloor w'_s \leftarrow w'_i\rfloor \rangle\!\rangle \\
[\,]\lfloor w'_s \leftarrow w'_i\rfloor & = & [\,] \\
[x_s w_s]\lfloor w'_s \leftarrow w'_i\rfloor & = & [x_s]\lfloor w'_s \leftarrow w'_i\rfloor \cdot [w_s]\lfloor w'_s \leftarrow w'_i\rfloor \\
[\text{state } [w_s]\ T]\lfloor w'_s \leftarrow w'_i\rfloor & = & [\text{state } [w_s]\lfloor w'_s \leftarrow w'_i\rfloor\ (T\lfloor w'_s \leftarrow w'_i\rfloor)] \\
([x_s, s] \leftarrow V)\lfloor w'_s \leftarrow w'_i\rfloor & = & [x_s\lfloor w'_s \leftarrow w'_i\rfloor, s] \leftarrow V\lfloor w'_s \leftarrow w'_i\rfloor \\
[\![\,]\!]\lfloor w'_s \leftarrow w'_i\rfloor & = & [\![\,]\!] \\
[\![B_s\, \omega_s]\!]\lfloor w'_s \leftarrow w'_i\rfloor & = & [\![B_s\lfloor w'_s \leftarrow w'_i\rfloor]\!] \cdot ([\![\omega_s]\!]\lfloor w'_s \leftarrow w'_i\rfloor) \\
[\text{set } [\![\omega_s]\!]\ T]\lfloor w'_s \leftarrow w'_i\rfloor & = & [\text{set } [\![\omega_s]\!]\lfloor w'_s \leftarrow w'_i\rfloor\ T\lfloor w'_s \leftarrow w'_i\rfloor] \\
P[\vec{T}]\lfloor w'_s \leftarrow w'_i\rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor x'_s \leftarrow i'\rfloor] \\
 & & \text{if } P \text{ doesn't involve any state variable}.
\end{array} \tag{12.14}$$

In addition to this explicit state-variable renaming, which is needed occasionally to maintain hygiene when rearranging terms (though, as in the control calculus of Chapter 11, most renaming will be handled quietly by function α), a second state-variable substitution function is provided to *delete* a state variable, for garbage collection (or, in principle —though not attempted here— because that environment provably won't be mutated). The notation for the deletion function is $\Box[x_s \not\leftarrow]$.

To maintain hygiene across state-variable deletion, renaming by function α avoids not only proscribed state variables, but proscribed state *indices*. The more stringent requirement arises because state-variable deletion causes structural rearrangement of its descendants:

$$[w'_i i w_i]\lfloor [i w_i] \not\leftarrow\rfloor \;=\; [w'_i w_i]\,. \tag{12.15}$$

If $[w'_i w_i]$ is already the name of another environment, the two environments are inadvertently merged. Function α chooses its renamings $\Box\lfloor [i w_i] \leftarrow i'\rfloor$ to guarantee not only that there is no proscribed variable $[i' w_i]$, and no proscribed variable with ancestor $[i' w_i]$ (lest we capture an ancestor of a free variable), but that no proscribed variable uses i' at all (which covers all the aforementioned cases, and deletion as well).

$\wr S$-semantics.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
\mathit{rename}(\phi, \mathcal{X}) & = & \langle \phi, \mathcal{X} \rangle \\
\mathit{rename}(x_s, \mathcal{X}) & = & \langle i, (\mathcal{X} \cup \{x_s \lfloor x_s \leftarrow i \rfloor\}) \rangle \\
 & & \quad \text{where } i > \mathit{max}(\bigcup_{[w_i] \in \mathcal{X}} w_i) \\
\mathit{rename}(w_s w'_s, \mathcal{X}) & = & \langle w_i w'_i, \mathcal{X}'' \rangle \\
 & & \quad \text{where } \mathit{rename}(w_s, \mathcal{X}) = \langle w_i, \mathcal{X}' \rangle \\
 & & \quad \text{and } \mathit{rename}(w'_s, \mathcal{X}') = \langle w'_i, \mathcal{X}'' \rangle \\
\alpha([\text{state } [w_s]\ T], \mathcal{X}) & = & [\text{state } [w_s \lfloor w_s \leftarrow w_i \rfloor] \\
 & & \quad \alpha(T, \mathcal{X}') \lfloor w_s \leftarrow w_i \rfloor] \\
 & & \quad \text{where } \mathit{rename}(w_s, \mathcal{X}) = \langle w_i, \mathcal{X}' \rangle\,.
\end{array} \tag{12.16}$$

A subtlety is that state-variable function *rename* always produces a vector of state indices in *increasing order*; consequently, the reduction relation of the semantics, which is meant to be deterministic, can expect the ordering of state variables within a state definiend to be unperturbed by required renaming. Further, because of the way $\alpha([\text{state } [w_s]\ T], \mathcal{X})$ uses *rename*, it will always assign larger indices to states bound in T than it does to states in w_s.

$\wr S$-semantics.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[x_s \not\leftarrow] & = & \alpha(T, \{x_s\} \cup \mathrm{FV}(T)) \lfloor x_s \not\leftarrow \rfloor \\
[w'_i i w_i] \lfloor [i w_i] \not\leftarrow \rfloor & = & [w'_i w_i] \qquad \text{if } w'_i \neq \phi \\
[i w_i] \lfloor [i w_i] \not\leftarrow \rfloor & = & \phi \\
x'_s \lfloor x_s \not\leftarrow \rfloor & = & x'_s \qquad \text{if } x_s \notin \mathit{path}(x'_s) \\
\langle\!\langle x'_s \rangle\!\rangle \lfloor x_s \not\leftarrow \rfloor & = & \langle\!\langle x'_s \lfloor x_s \not\leftarrow \rfloor \rangle\!\rangle \\
\phi \lfloor x_s \not\leftarrow \rfloor & = & \phi \\
(x'_s w_s) \lfloor x_s \not\leftarrow \rfloor & = & (x'_s \lfloor x_s \not\leftarrow \rfloor)(w_s \lfloor x_s \not\leftarrow \rfloor) \\
[\text{state } [w_s]\ T] \lfloor x_s \not\leftarrow \rfloor & = & [\text{state } ([w_s] \lfloor x_s \not\leftarrow \rfloor)\ (T \lfloor x_s \not\leftarrow \rfloor)] \\
([x'_s, s] \leftarrow V) \lfloor x_s \not\leftarrow \rfloor & = & [x'_s \lfloor x_s \not\leftarrow \rfloor, s] \leftarrow V \lfloor x_s \not\leftarrow \rfloor \qquad \text{if } x_s \neq x'_s \\
([x_s, s] \leftarrow V) \lfloor x_s \not\leftarrow \rfloor & = & \phi \\
[\![\,]\!] \lfloor x_s \not\leftarrow \rfloor & = & [\![\,]\!] \\
[\![B_s\, \omega_s]\!] \lfloor x_s \not\leftarrow \rfloor & = & [\![B_s \lfloor x_s \not\leftarrow \rfloor]\!] \cdot ([\![\omega_s]\!] \lfloor x_s \not\leftarrow \rfloor) \\
[\text{set } [\![\omega_s]\!]\ T] \lfloor x_s \not\leftarrow \rfloor & = & [\text{set } [\![\omega_s]\!] \lfloor x_s \not\leftarrow \rfloor\ T \lfloor x_s \not\leftarrow \rfloor] \\
P[\vec{T}] \lfloor x_s \not\leftarrow \rfloor & = & P[\sum_k \vec{T}(k) \lfloor x_s \not\leftarrow \rfloor] \\
 & & \quad \text{if } P \text{ doesn't involve any state variable}\,.
\end{array} \tag{12.17}$$

$T[x_s \not\leftarrow]$ is undefined if x_s occurs free in T as the identity in a first-class environment $\langle\!\langle x_s \rangle\!\rangle$, because in that case the substitution fails to produce a syntactically valid term.

(The free occurrence is replaced by the empty string; this produces invalid syntax only in the semantics, because the calculus will allow identity-less environments.)

We adjust the definition of δ-rule from $\mathfrak{f}_i$-calculus by allowing polynomials over the extended term set of $\mathfrak{f}S$-semantics, and by excluding `$define!` from the set of δ-rule combiners. We call the $\mathfrak{f}S$-semantics term set $\mathfrak{T}_{ss}$ (second "s" for "semantics") to distinguish it from the larger set $\mathfrak{T}_s$ of $\mathfrak{f}S$-calculus. (By limiting the polynomials to be 'over $\mathfrak{T}_{ss}$', we mean that the explicit syntax in the polynomials is restricted to that syntactic domain, *not* that the semantic variables within the polynomials are so constrained. As with all schemata and auxiliary functions, when shifting from $\mathfrak{f}S$-semantics to $\mathfrak{f}S$-calculus we broaden our interpretations of the semantic variables — as, most obviously, semantic variables based on "T" are quantified over $\mathfrak{T}_{ss}$ for the semantics, over $\mathfrak{T}_s$ for the calculus.)

A slight technical adjustment is also needed to δ-form Condition 9.10(2), because it specifies *unconstrained* subterms, whereas our term syntax now places constraints on arbitrary terms in arbitrary contexts.

Definition 12.18 A $\mathfrak{f}S$-calculus *δ-form* is a triple $\langle \pi_1, \pi_2, \pi_3 \rangle$ of semantic polynomials over the term set $\mathfrak{T}_{ss}$ satisfying all the conditions of Definition 9.29, except that Condition 9.10(2) is relaxed to allow subterm constraints inherent in their surrounding context. The class of all consistent sets of $\mathfrak{f}S$-calculus δ-forms is called Δ_s. ■

Semantic function δ now has type $(\mathit{PrimitiveOperatives} - \{\texttt{\$vau}, \texttt{\$define!}\}) \to \Delta_s$.

12.2 $\mathfrak{f}S$-semantics

The computation schemata for bubbling up of state and set frames are

$\wr S$-semantics.
Schemata (bubbling up):

$$\begin{aligned}
& E[[\text{state } [w_s]\ T]] \\
\longmapsto\ & [\text{state } [w_s]\lfloor w_s \leftarrow w_i \rfloor\ E[T[w_s \leftarrow w_i]]] \\
& \quad \text{where } \mathit{rename}(w_s, \mathrm{FV}(E[[\text{state } [w_s]\ T]])) = \langle w_i, \mathcal{X}\,\rangle \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.19\sigma)
\end{aligned}$$

$$\begin{aligned}
& [\text{state } [w_s]\ E[[\text{state } [w'_s]\ T]]] \\
\longmapsto\ & [\text{state } [w_s] \cdot ([w'_s]\lfloor w'_s \leftarrow w'_i \rfloor)\ E[T[w'_s \leftarrow w'_i]]] \\
& \quad \text{where } \mathit{rename}(w'_s, \mathrm{FV}(E[[\text{state } [w'_s]\ T]]) \cup w_s) = \langle w'_i, \mathcal{X}\,\rangle \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.19\sigma\sigma) \qquad (12.19)
\end{aligned}$$

$$\begin{aligned}
& [\text{state } [w_s]\ [\text{set } [\![\omega_s]\!]\ E[[\text{state } [w'_s]\ T]]]] \\
\longmapsto\ & [\text{state } [w_s] \cdot ([w'_s]\lfloor w'_s \leftarrow w'_i \rfloor)\ [\text{set } [\![\omega_s]\!]\ E[T[w'_s \leftarrow w'_i]]]] \\
& \quad \text{where } \mathit{rename}(w'_s, \mathrm{FV}([\text{set } [\![\omega_s]\!]\ E[[\text{state } [w'_s]\ T]]]) \cup w_s) \\
& \qquad\qquad = \langle w'_i, \mathcal{X}\,\rangle \qquad\qquad\qquad\qquad (12.19!\sigma)
\end{aligned}$$

$$\begin{aligned}
& [\text{state } X_s\ E[[\text{set } [\![\omega_s]\!]\ T]]] \\
\longmapsto\ & [\text{state } X_s\ [\text{set } [\![\omega_s]\!]\ E[T]]] \qquad\qquad\qquad\qquad (12.19!)
\end{aligned}$$

$$\begin{aligned}
& [\text{state } X_s\ [\text{set } [\![\omega_s]\!]\ E[[\text{set } [\![\omega'_s]\!]\ T]]]] \\
\longmapsto\ & [\text{state } X_s\ [\text{set } [\![\omega'_s]\!] \cdot [\![\omega_s]\!]\ E[T]]]\,. \qquad\qquad\qquad\qquad (12.19!!)
\end{aligned}$$

A set without a surrounding state is assumed to be an error, so we don't provide schemata for it.

The computation schema for symbol evaluation is

$\wr S$-semantics.
Schemata (symbol evaluation, amending $\wr_i$-semantics):

$$\begin{aligned}
& [\text{state } [w_s]\ [\text{set } [\![\omega_s]\!]\ E[[\text{eval } s\ \langle\!\langle x_s \rangle\!\rangle]]]] \qquad\qquad (12.20) \\
\longmapsto\ & [\text{state } [w_s]\ [\text{set } [\![\omega_s]\!]\ E[V]]] \\
& \quad \text{if } \mathit{lookup}([\mathit{path}(x_s), s], [\![\omega_s]\!]) = V \text{ and } \mathit{path}(x_s) \subseteq w_s\,.
\end{aligned}$$

The computation schema for `$vau`, (10.3`v`), doesn't itself need to be changed; but its auxiliary function *vau*, from (9.31), needs adaptation to use stateful environments (and will be adapted further in §12.3.6 to allow for degrees of immutability).

$\wr S$-semantics.
Auxiliary functions (definiend compilation):

$$\begin{array}{l} vau(\texttt{(}T_1\ \texttt{\#ignore}\ T_2\texttt{)}, \langle\!\langle [w_i] \rangle\!\rangle) \\ \quad = \ C[[\text{state } [[iw_i]]\ [\text{set } [iw_i] \times e_p\ [\text{eval } T_2\ \langle\!\langle [iw_i] \rangle\!\rangle]]]] \\ \qquad \text{if } \mathit{definiend}(T_1, \mathrm{FV}(T_2) \cup \{[w_i]\}) = \langle C, e_p, \mathcal{X} \rangle \\ \qquad\quad \text{and } [iw_i] \notin \mathrm{FV}(e_p) \cup \mathrm{FV}(T_2) \\ vau(\texttt{(}T_1\ s\ T_2\texttt{)}, \langle\!\langle [w_i] \rangle\!\rangle) \\ \quad = \ \langle \epsilon x_p . C[[\text{state } [[iw_i]]\ [\text{set } [iw_i] \times (e_p \cdot \langle\!\langle s \leftarrow x_p \rangle\!\rangle) \\ \qquad\qquad\qquad\qquad [\text{eval } T_2\ \langle\!\langle [iw_i] \rangle\!\rangle]]]] \rangle \\ \qquad \text{if } x_p \notin \mathrm{FV}(T_2), \\ \qquad\quad \mathit{definiend}(T_1, \mathrm{FV}(T_2) \cup \{[w_i]\}) = \langle C, e_p, \mathcal{X} \rangle, \\ \qquad\quad \text{and } [iw_i] \notin \mathrm{FV}(e_p) \cup \mathrm{FV}(T_2)\,. \end{array} \tag{12.21}$$

A fresh non-δ schema is needed for **`$define!`**. Otherwise, the stateless schemata (including the aforementioned for **`$vau`**) just need to be lifted.

$\wr S$-semantics.
Schemata (**`$define!`**; lifting):

$$\begin{array}{lll} & E[[\text{combine } \textbf{\texttt{\$define!}}\ \texttt{(}V_1\ \ V_2\texttt{)}\ \langle\!\langle x_s \rangle\!\rangle]] & \\ \longmapsto & E[[\text{combine } C[[\text{set } (x_s \times e_p)\ \texttt{\#inert}]]\ V_2\ \langle\!\langle \rangle\!\rangle]] & \\ & \quad \text{if } \mathit{definiend}(V_1, \{\}) = \langle C, e_p, \mathcal{X}\,\rangle & (12.22\mathtt{d}) \\ & [\text{state } X_s\ E[A_p]] & \\ \longmapsto & [\text{state } X_s\ T] \qquad \text{if } E[A_p] \longmapsto_s T & (12.22\bot\sigma) \\ & [\text{state } X_s\ [\text{set } [\![\omega_s]\!]\ E[A_p]]] & \\ \longmapsto & [\text{state } X_s\ [\text{set } [\![\omega_s]\!]\ T]] \qquad \text{if } E[A_p] \longmapsto_s T\,. & (12.22\bot!) \end{array} \tag{12.22}$$

There are four computation schemata for garbage collection, distinguished by whether the definiend of the state is or is not empty, and whether the body of the state is or is not a set. (Two schemata will suffice in the calculus, where we needn't impose a deterministic order of collection, and so don't care whether the body is a set.) To simplify the conditions on the schemata, we say x_s *is used in* T to mean that some free occurrence of x_s in T is the identity of a first-class environment $\langle\!\langle x_s \rangle\!\rangle$ occurring somewhere *other than* within the right-hand side of a stateful binding whose left-hand state variable is x_s (that is, the free occurrence of $\langle\!\langle x_s \rangle\!\rangle$ isn't within the V of a stateful binding $[x_s, s] \leftarrow V$).

$\wr S$-semantics.
Schemata (garbage collection):

$$\begin{array}{lll} & [\text{state } [\,]\ V] & \\ \longmapsto & \qquad V & (12.23\text{g0}) \end{array}$$

$$\begin{array}{lll} & [\text{state } [w_s]\ [\text{set } [\![]\!]\ V]] & \\ \longmapsto & [\text{state } [w_s] \qquad V\ ] & (12.23\text{g0!}) \end{array}$$

$$\begin{array}{lll} & [\text{state } [w_s x_s w'_s]\ V] & \\ \longmapsto & [\text{state } [w_s w'_s \cap \mathit{path}(x_s)] & \\ & \quad [\text{state } [w_s w'_s - \mathit{path}(x_s)] & \\ & \qquad V][x_s \not\leftarrow]] & \\ & \quad \text{if all of } w_s \text{ are used in } V, \text{ and } x_s \text{ is not used in } V & \\ & & (12.23\text{g}) \end{array} \tag{12.23}$$

$$\begin{array}{lll} & [\text{state } [w_s x_s w'_s]\ [\text{set } [\![\omega_s]\!]\ V]] & \\ \longmapsto & [\text{state } [w_s \quad w'_s]\ [\text{set } [\![\omega_s]\!]\ V]][x_s \not\leftarrow] & \\ & \quad \text{if all of } w_s \text{ are used in } [\text{set } [\![\omega_s]\!]\ V], & (12.23\text{g!}) \\ & \quad \text{and } x_s \text{ is not used in } [\text{set } [\![\omega_s]\!]\ V]\,. & \end{array}$$

(Note that in (12.23g!), some elements of ω_s might be deleted — and those are the same elements that make no contribution to determining whether x_s is used.)

12.3 $\wr S$-calculus

The state calculus, besides being compatible, has three features beyond its semantics: local schemata for lookup; immutable bindings; and immutable environments.

12.3.1 Syntax of lookup

In the strictly mutable case, symbol evaluation is a transaction between the point of evaluation, at the bottom of the syntax tree, and the point of binding (a set frame), somewhere higher in the syntax tree. As usual, we manage the transaction in the calculus by a directive frame that bubbles upward, and a substitution function that broadcasts a reply downward.

The directive frame has the form

$$[\text{get } [\![\omega_g]\!]\ \Box]\,, \tag{12.24}$$

where ω_g is a list of *stateful binding requests*, each of the form

$$x_g \leftarrow [x_s, s]\,, \tag{12.25}$$

where x_g is a *get variable*, bound by the get frame, and used within the scope of the frame, $\Box$, to designate the eventual resultant value of the requested lookup. When

the get frame encounters a set frame with binding $[x_s, s] \leftarrow V$, it broadcasts V to its body via substitution $\Box[x_g \leftarrow V]$; or, if the get frame encounters a state frame that binds x_s, it broadcasts lookup failure via $\Box[x_g \not\leftarrow]$.

The target for these substitutions is a special syntactic device, of either form

$$\begin{array}{ll} & [\text{receive } w_g] \\ \text{or} & [\text{receive } w_g/V]\,, \end{array} \tag{12.26}$$

where w_g is the sequence of get variables designating possible results along the environment-search path for the symbol evaluation, and optional $/V$ provides a value to use if all lookups w_g fail. For example, the following reductions lookup a symbol s in environment $\langle\!\langle x'_s \rangle\!\rangle$ with orphan parent $\langle\!\langle x_s \rangle\!\rangle$, where s is not locally bound (no binding with left-hand side $[x'_s, s]$), but is bound in the parent (binding $[x_s, s] \leftarrow V$). (We assume for the example that no α-renaming is needed.)

$$\begin{array}{ll} & [\text{state } [x_s x'_s]\ [\text{set } [\![[x_s, s] \leftarrow V]\!]\ E[[\text{eval } s\ \langle\!\langle x'_s \rangle\!\rangle]]]] \\ \longrightarrow_{ls} & [\text{state } [x_s x'_s]\ [\text{set } [\![[x_s, s] \leftarrow V]\!]\ E[[\text{get } [\![x'_g \leftarrow [x'_s, s]\ x_g \leftarrow [x_s, s]]\!] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\text{receive } x'_g x_g]]]]] \\ \longrightarrow^*_{ls} & [\text{state } [x_s x'_s]\ [\text{set } [\![[x_s, s] \leftarrow V]\!]\ [\text{get } [\![x'_g \leftarrow [x'_s, s]\ x_g \leftarrow [x_s, s]]\!] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E[[\text{receive } x'_g x_g]]]]] \\ \longrightarrow_{ls} & [\text{state } [x_s x'_s]\ [\text{set } [\![[x_s, s] \leftarrow V]\!]\ [\text{get } [\![x'_g \leftarrow [x'_s, s]]\!] \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad E[[\text{receive } x'_g/V]]]]] \\ \longrightarrow_{ls} & [\text{state } [x_s x'_s]\ [\text{get } [\![x'_g \leftarrow [x'_s, s]]\!] \\ & \qquad\qquad\qquad\qquad [\text{set } [\![[x_s, s] \leftarrow V]\!]\ E[[\text{receive } x'_g/V]]]]] \\ \longrightarrow_{ls} & [\text{state } [x_s x'_s]\ [\text{set } [\![[x_s, s] \leftarrow V]\!]\ E[V]]]\,. \end{array} \tag{12.27}$$

The syntax extension for mutable lookup is

$\wr S$-calculus.
Syntax (lookup, amending $\wr S$-semantics):

$$
\begin{array}{lcll}
x_g & \in & \textit{GetVariables} \text{ (with total ordering } \leq) & \\
B_g & ::= & x_g \leftarrow [x_s, s] & \text{(Stateful binding requests)} \\
D & ::= & \phi \mid /V & \text{(Optional default values)} \\
A & ::= & A_p \mid [\text{state } X_s\ T] \mid [\text{set } [\![B_s^*]\!]\ T] & \\
 & & \quad \mid [\text{get } [\![B_g^*]\!]\ T] & \\
 & & \quad \mid [\text{receive } x_g^*] & \text{(Active terms)} \\
T & ::= & S \mid s \mid x_p \mid [\text{receive } x_g^+ / V] & \\
 & & \quad \mid (T\ .\ T) \mid A & \text{(Terms)}
\end{array}
\tag{12.28}
$$

where
binding requests in a get list are in order by left-hand side;
no two binding requests in a get list have the same left-hand side; and
no two get variables in the get-variable list of a receive are identical.

A receive without a default value is an active term, because it might represent a lookup that will ultimately fail (reducing by substitution to [receive]), and so can only be used in situations where a nonterminating subcomputation is allowed. A receive *with* a default value cannot fail, because we already know that at least one of its binding requests was successful; and it must ultimately reduce to a value, because the right-hand sides of stateful bindings are required to be values; so we can safely use it in situations where a value is expected, provided the use doesn't depend on which value it reduces to. (Recall that δ-rules must be invariant under substitution, by Conditions 9.22(8) and 9.29(8a).)

12.3.2 Syntax of environments

Evaluation of a mutably bound symbol is a non-local event, i.e., it cannot be handled internally to the eval subterm that initiates it, because its binding, if any, is maintained by a set frame, higher up in the syntax tree. In contrast, evaluation of an immutably bound symbol can and should be a local event (as in $\wr_i$-calculus), promoting stronger equational theory (as anticipated in the treatment of hygiene, Chapter 5). To enable this local treatment, we distribute the representation of immutable bindings to the environment structures, $\langle\!\langle \ldots \rangle\!\rangle$:

$\natural S$-calculus.
Syntax (environments, amending $\natural S$-semantics):

$$\begin{array}{l} e \ ::= \ \langle\!\langle {x_s}^? \, B_p^* \rangle\!\rangle \quad \text{(Environments)} \\ \text{where} \\ \quad \text{bindings in an environment are in order by bound symbol; and} \\ \quad \text{no two bindings in an environment bind the same symbol.} \end{array} \tag{12.29}$$

When an environment is completely stable —no mutation can occur to it nor to any of its ancestors— we dispense with its state-variable identity altogether, leaving only a $\natural_i$-style environment $\langle\!\langle \omega_p \rangle\!\rangle$, thus recognizing equationally that stable environments with the same bindings are interchangeable.

12.3.3 Auxiliary functions

The state-variable substitution functions extend straightforwardly to cover the new syntax. Semantic variables ω_g are quantified over sequences of stateful binding requests, B_g^*.

$\natural S$-calculus.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
(s \leftarrow V)\lfloor w_s \leftarrow w_i \rfloor & = & s \leftarrow V\lfloor w_s \leftarrow w_i \rfloor \\
(B_p\, \omega_p)\lfloor w_s \leftarrow w_i \rfloor & = & B_p\lfloor w_s \leftarrow w_i \rfloor \ \omega_p\lfloor w_s \leftarrow w_i \rfloor \\
\langle\!\langle w_s\, \omega_p \rangle\!\rangle\lfloor w_s \leftarrow w_i \rfloor & = & \langle\!\langle w_s\lfloor w_s \leftarrow w_i \rfloor \ \omega_p\lfloor w_s \leftarrow w_i \rfloor \rangle\!\rangle \\
(x_g \leftarrow [x_s, s])\lfloor w_s \leftarrow w_i \rfloor & = & x_g \leftarrow [x_s\lfloor w_s \leftarrow w_i \rfloor, s] \\
[\![B_g\, \omega_g]\!]\lfloor w_s \leftarrow w_i \rfloor & = & [\![B_g\lfloor w_s \leftarrow w_i \rfloor]\!] \cdot [\![\omega_g]\!]\lfloor w_s \leftarrow w_i \rfloor \\
[\text{get } [\![\omega_g]\!]\ T]\lfloor w_s \leftarrow w_i \rfloor & = & [\text{get } [\![\omega_g]\!]\lfloor w_s \leftarrow w_i \rfloor \ T\lfloor w_s \leftarrow w_i \rfloor] \\
(s \leftarrow V)\lfloor x_s \not\leftarrow \rfloor & = & s \leftarrow V\lfloor x_s \not\leftarrow \rfloor \\
(B_p\, \omega_p)\lfloor x_s \not\leftarrow \rfloor & = & B_p\lfloor x_s \not\leftarrow \rfloor \ \omega_p\lfloor x_s \not\leftarrow \rfloor \\
\langle\!\langle w_s\, \omega_p \rangle\!\rangle\lfloor x_s \not\leftarrow \rfloor & = & \langle\!\langle w_s\lfloor x_s \not\leftarrow \rfloor \ \omega_p\lfloor x_s \not\leftarrow \rfloor \rangle\!\rangle \\
\langle\!\langle x_s\, \omega_p \rangle\!\rangle\lfloor x_s \not\leftarrow \rfloor & = & \langle\!\langle \omega_p\lfloor x_s \not\leftarrow \rfloor \rangle\!\rangle \\
(x_g \leftarrow [x'_s, s])\lfloor x_s \not\leftarrow \rfloor & = & x_g \leftarrow [x'_s\lfloor x_s \not\leftarrow \rfloor, s] \qquad \text{if } x'_s \neq x_s \\
[\![B_g\, \omega_g]\!]\lfloor x_s \not\leftarrow \rfloor & = & [\![B_g\lfloor x_s \not\leftarrow \rfloor]\!] \cdot ([\![\omega_g]\!]\lfloor x_s \not\leftarrow \rfloor) \\
[\text{get } [\![\omega_g]\!]\ T]\lfloor x_s \not\leftarrow \rfloor & = & [\text{get } [\![\omega_g]\!]\lfloor x_s \not\leftarrow \rfloor \ T\lfloor x_s \not\leftarrow \rfloor] \\
[\text{get } [\![\omega_g]\!] \cdot [\![x_g \leftarrow [x_s, s]]\!] & & \\
\qquad T]\lfloor x_s \not\leftarrow \rfloor & = & [\text{get } [\![\omega_g]\!]\ T[x_g \not\leftarrow]]\lfloor x_s \not\leftarrow \rfloor \,.
\end{array} \tag{12.30}$$

Recall that substitution $T\lfloor x_s \not\leftarrow \rfloor$ was undefined in the semantics, (12.17), when x_s occurred free in T either on the left-hand side of a stateful binding, or as the identity of a first-class environment $\langle\!\langle x_s \rangle\!\rangle$. Here, the extended syntax for environments admits $\langle\!\langle x_s \rangle\!\rangle\lfloor x_s \not\leftarrow \rfloor = \langle\!\langle \rangle\!\rangle$. When x_s occurs free on the left side of a stateful binding, we leave the substitution undefined, inhibiting deletion of x_s until and unless the stateful binding is eliminated (cf. §12.3.7). When x_s occurs free on the right side of a stateful

binding request, we define the state substitution by inducing a lookup-failure get substitution, $T[x_g \not\leftarrow]$.

In defining auxiliary functions to manage delimited lists of stateful binding requests, $[\![\omega_g]\!]$, we order requests by get variable (i.e., $(x_g \leftarrow [x_s, s]) \leq (x'_g \leftarrow [x'_s, s'])$ iff $x_g \leq x'_g$).

$\wr S$-calculus.
Auxiliary functions (stateful binding request sets):

$$\begin{aligned}
[\![\,]\!] \cdot [\![\omega_g]\!] &= [\![\omega_g]\!] \\
[\![\omega_g]\!] \cdot [\![\,]\!] &= [\![\omega_g]\!] \\
[\![B'_g]\!] \cdot [\![B_g\, \omega_g]\!] &= \begin{cases} [\![B'_g B_g\, \omega_g]\!] & \text{if } B'_g < B_g \\ [\![B_g]\!] \cdot ([\![B'_g]\!] \cdot [\![\omega_g]\!]) & \text{if } B_g < B'_g \end{cases} \\
[\![\omega_g\, B_g]\!] \cdot [\![\omega'_g]\!] &= [\![\omega_g]\!] \cdot ([\![B_g]\!] \cdot [\![\omega'_g]\!]) \qquad (12.31) \\
\mathrm{DV}_s([\![\,]\!]) &= \{\} \\
\mathrm{DV}_s([\![x_g \leftarrow [x_s, s]]\!]) &= x_s \\
\mathrm{DV}_s([\![\omega_g]\!] \cdot [\![\omega'_g]\!]) &= \mathrm{DV}_s([\![\omega_g]\!]) \cup \mathrm{DV}_s([\![\omega'_g]\!]) \\
\mathrm{DV}_g([\![\,]\!]) &= \phi \\
\mathrm{DV}_g([\![x_g \leftarrow [x_s, s]]\!]) &= x_g \\
\mathrm{DV}_g([\![\omega_g\, \omega'_g]\!]) &= \mathrm{DV}_g([\![\omega_g]\!])\, \mathrm{DV}_g([\![\omega'_g]\!])\,.
\end{aligned}$$

Get variables, like state variables, can be bound in parallel and so ought to be renameable in parallel. Semantic variables w_g are quantified over strings of state variables. As before ((12.16)), function *rename* always produces a renamed vector in increasing order, and $\alpha([\mathrm{get}\ \langle\!\langle \omega_g \rangle\!\rangle\ T], \mathcal{X})$ assigns larger revised names to variables bound in T that it does to variables bound by ω_g.

$\natural S$-calculus.
Auxiliary functions (substitution):

$$
\begin{array}{rcl}
T[w_g \leftarrow w'_g] & = & \alpha(T, w_g \cup w'_g \cup \mathrm{FV}(T))\lfloor w_g \leftarrow w'_g \rfloor \\
x_g\lfloor \phi \leftarrow \phi \rfloor & = & x_g \\
x_g\lfloor x'_g w'_g \leftarrow x''_g w''_g \rfloor & = & \left\{ \begin{array}{ll} x''_g & \text{if } x_g = x'_g \\ x_g\lfloor w'_g \leftarrow w''_g \rfloor & \text{otherwise} \end{array} \right. \\
(x_g \leftarrow [x_s, s])\lfloor w_g \leftarrow w'_g \rfloor & = & x_g\lfloor w_g \leftarrow w'_g \rfloor \leftarrow [x_s, s] \\
[\![\,]\!]\lfloor w_g \leftarrow w'_g \rfloor & = & [\![\,]\!] \\
[\![B_g\, \omega_g]\!]\lfloor w_g \leftarrow w'_g \rfloor & = & [\![B_g\lfloor w_g \leftarrow w'_g \rfloor]\!] \cdot [\![\omega_g]\!]\lfloor w_g \leftarrow w'_g \rfloor \\
[\mathrm{get}\ [\![\omega_g]\!]\ T]\lfloor w_g \leftarrow w'_g \rfloor & = & [\mathrm{get}\ [\![\omega_g]\!]\lfloor w_g \leftarrow w'_g \rfloor\ T\lfloor w_g \leftarrow w'_g \rfloor] \\
\phi\lfloor w_g \leftarrow w'_g \rfloor & = & \phi \\
(x_g w''_g)\lfloor w_g \leftarrow w'_g \rfloor & = & x_g\lfloor w_g \leftarrow w'_g \rfloor\, w''_g\lfloor w_g \leftarrow w'_g \rfloor \\
(/V)\lfloor w_g \leftarrow w'_g \rfloor & = & /(V\lfloor w_g \leftarrow w'_g \rfloor) \\
[\mathrm{receive}\ w''_g D]\lfloor w_g \leftarrow w'_g \rfloor & = & [\mathrm{receive}\ w''_g\lfloor w_g \leftarrow w'_g \rfloor \\
 & & \quad D\lfloor w_g \leftarrow w'_g \rfloor] \\
P[\vec{T}]\lfloor w_g \leftarrow w'_g \rfloor & = & P[\sum_k \vec{T}(k)\lfloor w_g \leftarrow w'_g \rfloor] \\
 & & \quad \text{if } P \text{ doesn't involve any get variable} \\
\mathit{rename}(x_g, \mathcal{X}) & = & \langle x'_g, (\mathcal{X} \cup \{x'_g\}) \rangle \\
 & & \quad \text{where } x'_g > \mathit{max}(\mathcal{X} \cap \mathit{GetVariables}) \\
\mathit{rename}(w_g w'_g, \mathcal{X}) & = & \langle w''_g w'''_g, \mathcal{X}'' \rangle \\
 & & \quad \text{where } \mathit{rename}(w_g, \mathcal{X}) = \langle w''_g, \mathcal{X}' \rangle \\
 & & \quad \text{and } \mathit{rename}(w'_g, \mathcal{X}') = \langle w'''_g, \mathcal{X}'' \rangle \\
\alpha([\mathrm{get}\ [\![\omega_g]\!]\ T], \mathcal{X}) & = & [\mathrm{get}\ [\![\omega_g]\!]\lfloor w_g \leftarrow w'_g \rfloor \\
 & & \quad \alpha(T, \mathcal{X}')\lfloor w_g \leftarrow w'_g \rfloor] \\
 & & \quad \text{where } w_g = \mathrm{DV}_g([\![\omega_g]\!]) \\
 & & \quad \text{and } \mathit{rename}(w_g, \mathcal{X} \cup w_g) = \langle w'_g, \mathcal{X}' \rangle\,.
\end{array}
\tag{12.32}
$$

Get-variable substitution on successful lookup is

$\wr S$-calculus.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[x_g \leftarrow V] & = & \alpha(T, \{x_g\} \cup \mathrm{FV}(T) \cup \mathrm{FV}(V))\lfloor x_g \leftarrow V \rfloor \\
[\text{get } [\![\omega_g]\!]\ T]\lfloor x_g \leftarrow V \rfloor & = & [\text{get } [\![\omega_g]\!]\ T\lfloor x_g \leftarrow V \rfloor)] \\
(/V)\lfloor x_g \leftarrow V \rfloor & = & /(V\lfloor x_g \leftarrow V \rfloor) \\
[\text{receive } w_g D]\lfloor x_g \leftarrow V \rfloor & = & [\text{receive } w_g(D\lfloor x_g \leftarrow V \rfloor)] \\
 & & \quad \text{if } x_g \notin w_g \\
[\text{receive } w_g x_g w'_g D]\lfloor x_g \leftarrow V \rfloor & = & [\text{receive } w_g/V] \\
 & & \quad \text{if } w_g \neq \phi \\
[\text{receive } x_g w_g D]\lfloor x_g \leftarrow V \rfloor & = & V \\
P[\vec{T}]\lfloor x_g \leftarrow V \rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor x_g \leftarrow V \rfloor] \\
 & & \quad \text{if } P \text{ doesn't involve any} \\
 & & \quad\quad \text{get variable}\,;
\end{array} \tag{12.33}$$

while on failed lookup,

$\wr S$-calculus.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[x_g \not\leftarrow] & = & \alpha(T, \{x_g\} \cup \mathrm{FV}(T))\lfloor x_g \not\leftarrow \rfloor \\
[\text{get } [\![\omega_g]\!]\ T]\lfloor x_g \not\leftarrow \rfloor & = & [\text{get } [\![\omega_g]\!]\ T\lfloor x_g \not\leftarrow \rfloor] \\
\phi\lfloor x_g \not\leftarrow \rfloor & = & \phi \\
(x'_g w_g)\lfloor x_g \not\leftarrow \rfloor & = & \begin{cases} w_g & \text{if } x_g = x'_g \\ x'_g(w_g\lfloor x_g \not\leftarrow \rfloor) & \text{otherwise} \end{cases} \\
(/V)\lfloor x_g \not\leftarrow \rfloor & = & /(V\lfloor x_g \not\leftarrow \rfloor) \\
[\text{receive } w_g D]\lfloor x_g \not\leftarrow \rfloor & = & [\text{receive } (w_g\lfloor x_g \not\leftarrow \rfloor)(D\lfloor x_g \not\leftarrow \rfloor)] \\
 & & \quad \text{if } w_g\lfloor x_g \not\leftarrow \rfloor \neq \phi \text{ or } D = \phi \\
[\text{receive } x_g/V]\lfloor x_g \not\leftarrow \rfloor & = & V\lfloor x_g \not\leftarrow \rfloor \\
P[\vec{T}]\lfloor x_g \not\leftarrow \rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor x_g \not\leftarrow \rfloor] \\
 & & \text{if } P \text{ doesn't involve any get variable}\,.
\end{array} \tag{12.34}$$

Mutable-to-immutable coercion of an environment x_s uses substitution $\Box[x_s \not\leftarrow]$, already defined; but mutable-to-immutable coercion of a binding $[x_s, s] \leftarrow V$ requires a new kind of substitution, for which we use, naturally, notation $\Box[[x_s, s] \leftarrow V]$.

$\natural S$-calculus.
Auxiliary functions (substitution):

$$\begin{array}{rcl}
T[[x_s, s] \leftarrow V] & = & \alpha(T, \{x_s\} \cup \mathrm{FV}(V) \cup \mathrm{FV}(T))\lfloor[x_s, s] \leftarrow V\rfloor \\
\mathit{children}(x_s, \phi) & = & \phi \\
\mathit{children}([w'_i], [i w_i] w_s) & = & \begin{cases} [i w_i]\mathit{children}([w'_i], w_s) & \text{if } w_i = w'_i \\ \mathit{children}([w'_i], w_s) & \text{otherwise} \end{cases} \\
\langle\!\langle x_s \rangle\!\rangle \cdot \langle\!\langle \omega_p \rangle\!\rangle & = & \langle\!\langle x_s\, \omega_p \rangle\!\rangle \\
\langle\!\langle x_s\, \omega_p \rangle\!\rangle\lfloor[x_s, s] \leftarrow V\rfloor & = & \langle\!\langle x_s \rangle\!\rangle \cdot (\langle\!\langle \omega_p \rangle\!\rangle\lfloor[x_s, s] \leftarrow V\rfloor \cdot \\
 & & \qquad \langle\!\langle s \leftarrow V \rangle\!\rangle) \\
[\text{state } [w_s]\ T]\lfloor[x_s, s] \leftarrow V\rfloor & = & [\text{state } [w_s] \\
 & & \qquad [\text{set } [\![\textstyle\sum_k [w'_s(k), s] \leftarrow V]\!] \\
 & & \qquad\quad T\lfloor[x_s, s] \leftarrow V\rfloor]] \\
 & & \qquad \text{where } w'_s = \mathit{children}(x_s, w_s) \\
P[\vec{T}]\lfloor[x_s, s] \leftarrow V\rfloor & = & P[\textstyle\sum_k \vec{T}(k)\lfloor[x_s, s] \leftarrow V\rfloor] \\
 & & \qquad \text{if } x_s \notin \mathrm{FV}(P)\,.
\end{array} \tag{12.35}$$

Correct use of substitution $\square[[x_s, s] \leftarrow V]$ is subject to some constraints not inherent in its definition. The substitution could misbehave severely if x_s occurs free as the identity of a first-class environment (that is, $\langle\!\langle x_s\, \omega_s \rangle\!\rangle$) in V; so any mutable-to-immutable coercion schema must forbid coercion in that case. The substitution will propagate correctly into children of x_s only if it is initiated from outside the state frames of all such children; so any coercion schema must require all possible children of x_s to be encompassed by the coerced binding.

12.3.4 Assignment

The computation schema for `$define!`, above in (12.22), is adapted for immutability by refusing to proceed if an attempt is made to mutate an immutable structure. Operationally, no computation that begins with a semantic term (as opposed to a calculus term outside the semantics) will ever attempt to mutate an immutable, because immutables would only arise from mutable-to-immutable coercion, and coercion would only be performed if there provably cannot be any attempt to mutate the coerced structure (§12.3.7). Hence, any positive behavior when attempting to mutate an immutable serves no useful purpose — and we would also have to worry about proving that the behavior doesn't violate well-behavedness.

$\wr S$-calculus.
Schemata (**`$define!`**):

$$\begin{aligned}
&[\text{combine } \texttt{\$define!}\ (V_1\ V_2)\ \langle\!\langle \omega_p \rangle\!\rangle] \\
\longrightarrow\ &\texttt{\#inert} \\
&\quad \text{if } \mathit{definiend}(V_1, \{\}) = \langle C, \langle\!\langle\rangle\!\rangle, \mathcal{X} \rangle && (12.36\texttt{dp}) \\
&[\text{combine } \texttt{\$define!}\ (V_1\ V_2)\ \langle\!\langle x_s\, \omega_p \rangle\!\rangle] \\
\longrightarrow\ &[\text{combine } C[[\text{set } (x_s \times e_p)\ \texttt{\#inert}]]\ V_2\ \langle\!\langle\rangle\!\rangle] \\
&\quad \text{if } \mathit{definiend}(V_1, \{\}) = \langle C, e_p, \mathcal{X} \rangle, \text{ and} && (12.36\texttt{ds}) \\
&\qquad e_p \text{ doesn't bind any symbol bound by } \omega_p\,.
\end{aligned} \tag{12.36}$$

Two schemata are provided here for simplifying set frames: one to merge consecutive sets, the other to garbage-collect a set with an empty list of bindings.

$\wr S$-calculus.
Schemata (set simplification):

$$\begin{aligned}
[\text{set } [\![\omega_s]\!]\ [\text{set } [\![\omega'_s]\!]\ T]] \ &\longrightarrow\ [\text{set } [\![\omega'_s]\!] \cdot [\![\omega_s]\!]\ T] && (12.37!!) \\
[\text{set } [\![\,]\!]\ T] \ &\longrightarrow\ T\,. && (12.37!0)
\end{aligned} \tag{12.37}$$

A set can bubble up through an evaluation context. As in the control calculus, frames bubble upward by just one level of syntax per reduction step; we borrow the definition of *singular evaluation context* from (11.8).

$\wr S$-calculus.
Schemata (set bubbling-up):

$$E^s[[\text{set } [\![\omega_s]\!]\ T]] \ \longrightarrow\ [\text{set } [\![\omega_s]\!]\ E^s[T]]\ . \tag{12.38}$$

12.3.5 Lookup

Symbol evaluation has two cases, depending on whether the binding is immutable (in which case evaluation is a local event), or mutable (in which case evaluation is a side-effect-ful event).

$\wr S$-calculus.
Schemata (symbol evaluation):

$$\begin{aligned}
[\text{eval } s\ \langle\!\langle w_s\, \omega_p \rangle\!\rangle] \ &\longrightarrow\ \mathit{lookup}(s, \langle\!\langle \omega_p \rangle\!\rangle) \\
&\qquad \text{if } \mathit{lookup}(s, \langle\!\langle \omega_p \rangle\!\rangle) \text{ is defined} && (12.39\text{s}p) \\
[\text{eval } s\ \langle\!\langle x_s\, \omega_p \rangle\!\rangle] \ &\longrightarrow\ [\text{get } [\![\textstyle\sum_k (w_g(k) \leftarrow [w_s(k), s])]\!]\ [\text{receive } w_g]] \\
&\qquad\qquad \text{where } w_s = \mathit{path}(x_s),\ \mathit{ar}(w_g) = \mathit{ar}(w_s), \\
&\qquad\qquad\quad \text{and } w_g \text{ is in strictly increasing order} \\
&\qquad \text{if } \mathit{lookup}(s, \langle\!\langle \omega_p \rangle\!\rangle) \text{ is undefined}\,. && (12.39\text{s}s)
\end{aligned} \tag{12.39}$$

Get resolution also has two cases, for success and failure.

$\wr S$-calculus.
Schemata (get resolution):

$$\begin{array}{ll} & [\text{set } [\![\omega_s]\!] \ [\text{get } [\![x_g \leftarrow [x_s, s] \ \ \omega_g]\!] \ T]] \\ \longrightarrow & [\text{set } [\![\omega_s]\!] \ [\text{get } [\![\omega_g]\!] \qquad\qquad T[x_g \leftarrow V]]] \\ & \quad \text{if } \mathit{lookup}([x_s, s], [\![\omega_s]\!]) = V \qquad (12.40?\top) \end{array} \qquad (12.40)$$

$$\begin{array}{ll} & [\text{state } [w_s] \ [\text{get } [\![\omega_g \ \ x_g \leftarrow [x_s, s] \ \ \omega'_g]\!] \ T]] \\ \longrightarrow & [\text{state } [w_s] \ [\text{get } [\![\omega_g \qquad\qquad \omega'_g]\!] \ T[x_g \not\leftarrow]]] \\ & \quad \text{if } x_s \in w_s, \text{ and no state variable in } \omega_g \text{ is} \\ & \quad\ \text{bound by } w_s\,. \qquad (12.40?\bot) \end{array}$$

These two schemata are written so that the binding requests are always processed in order from left to right — which has no effect on the equational theory (since any request that can't be resolved by a set will be able to bubble up through it, exposing the next request for processing), but will simplify proof of well-behavedness in Chapter 14.

Two consecutive gets can be merged; within a single get, two binding requests with the same right-hand side can be merged; and a get with no binding requests can be garbage-collected.

$\wr S$-calculus.
Schemata (get simplification):

$$\begin{array}{ll} & [\text{get } [\![\omega_g]\!] \ [\text{get } [\![\omega'_g]\!] \ T]] \\ \longrightarrow & [\text{get } [\![\omega_g]\!] \cdot ([\![\omega'_g]\!][w'_g \leftarrow w''_g]) \ T[w'_g \leftarrow w''_g]] \\ & \quad \text{where } w_g = \textstyle\sum_k (x_g \text{ such that } \omega_g(k) = (x_g \leftarrow [x_s, s])), \\ & \qquad\quad w'_g = \textstyle\sum_k (x_g \text{ such that } \omega'_g(k) = (x_g \leftarrow [x_s, s])), \\ & \qquad \text{and } \mathit{rename}(w'_g, w_g \cup \mathrm{FV}([\text{get } [\![\omega_g]\!] \ [\text{get } [\![\omega'_g]\!] T]])) \\ & \qquad\quad = \langle w''_g, \mathcal{X} \rangle \qquad (12.41??) \end{array} \qquad (12.41)$$

$$\begin{array}{ll} & [\text{get } [\![\omega_g]\!] \cdot [\![x_g \leftarrow [x_s, s]]\!] \cdot [\![x'_g \leftarrow [x_s, s]]\!] \ T] \\ \longrightarrow & [\text{get } [\![\omega_g]\!] \cdot [\![x_g \leftarrow [x_s, s]]\!] \qquad\qquad T[x'_g \leftarrow x_g]] \\ & \quad \text{if } x_g < x'_g \text{ are the leftmost such choices for this get} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.41?2) \end{array}$$

$$\begin{array}{ll} & [\text{get } [\![\,]\!] \ T] \\ \longrightarrow & \qquad T\,. \qquad (12.41?0) \end{array}$$

A get, like a set (§12.3.4), can bubble up through any singular evaluation context. A get can also bubble up through a set that doesn't resolve it — as illustrated above in Example (12.27); and as implicitly expected by the get-failure schema in (12.40), where the failing get must be immediately inside the state frame.

$\wr S$-calculus.
Schemata (get bubbling-up):

$$\begin{aligned}
& E^s[[\text{get } [\![\omega_g]\!]\ T]] \\
\longrightarrow\ & [\text{get } [\![\omega_g]\!][w_g \leftarrow w'_g]\ E^s[T[w_g \leftarrow w'_g]]] \\
& \quad \text{where } w_g = \textstyle\sum_k (x_g \text{ such that } \omega_g(k) = (x_g \leftarrow [x_s, s])) \\
& \quad\quad \text{and } \mathit{rename}(w_g, \mathrm{FV}(E^s[[\text{get } [\![\omega_g]\!]\ T]])) = \langle w'_g, \mathcal{X} \rangle \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.42{\uparrow}?)
\end{aligned} \tag{12.42}$$

$$\begin{aligned}
& [\text{set } [\![\omega_s]\!]\ [\text{get } [\![x_g \leftarrow [x_s, s]\ \ \omega_g]\!]\ T]] \\
\longrightarrow\ & [\text{get } [\![x'_g \leftarrow [x_s, s]]\!]\ [\text{set } [\![\omega_s]\!]\ [\text{get } [\![\omega_g]\!]\ T[x_g \leftarrow x'_g]]]] \\
& \quad \text{if } \mathit{lookup}([x_s, s], [\![\omega_s]\!]) \text{ is undefined, and} \\
& \quad\quad \mathit{rename}(x_g, \mathrm{FV}([\text{set } [\![\omega_s]\!]\ [\text{get } [\![\omega_g]\!] \cdot [\![x_g \leftarrow [x_s, s]]\!]\ T]])) \\
& \quad\quad = \langle x'_g, \mathcal{X} \rangle\,. \qquad\qquad (12.42{\uparrow}!?)
\end{aligned}$$

The second schema, which splits the definiend of a get in order to bubble up a single binding request, always chooses the first binding request, so as to preserve the overall ordering of binding requests in the term. A slightly weaker constraint, bubbling up the first binding request *that can't be resolved by the set* (regardless of whether it is actually the first binding request), would sometimes produce a single pattern (get nested in set) that is reducible in two different ways (bubbling up and get resolution), producing an unnecessary and inconvenient ambiguity. A still weaker constraint, bubbling up any unresolvable binding request at all, could permute the order of binding requests — which, besides further ambiguity, could potentially compromise Church–Rosser-ness; to preserve Church–Rosser-ness would then require an additional schema to permute a get definiend *in situ*, so that permutations caused by the second get bubbling-up schema could always be undone later.

12.3.6 Environments

Auxiliary function *vau*, (12.21), is adapted for immutability by treating immutable bindings of the static parent environment as initial mutable bindings for the local child environment. (Local immutable bindings could be introduced later by coercion; cf. §12.3.7.)

$\wr S$-semantics.
Auxiliary functions (definiend compilation):

$$\begin{aligned}
&\mathit{vau}(\texttt{(}T_1\ \texttt{\#ignore}\ T_2\texttt{)}, \langle\!\langle w_s\, \omega_p \rangle\!\rangle) \\
&\quad = \ C[[\text{state } [x_s]\ [\text{set } x_s \times (e_p \cdot \langle\!\langle \omega_p \rangle\!\rangle)\ [\text{eval } T_2\ \langle\!\langle x_s \rangle\!\rangle]]]] \\
&\qquad \text{if } \mathit{definiend}(T_1, \mathrm{FV}(T_2) \cup w_s) = \langle C, e_p, \mathcal{X} \rangle, \\
&\qquad\quad x_s \notin \mathrm{FV}(T_2), \text{ and either } w_s = \phi \text{ and } x_s = [i], \\
&\qquad\quad \text{or } w_s = [w_i] \text{ and } x_s = [i w_i] \in \mathrm{FV}(e_p) \\
&\mathit{vau}(\texttt{(}T_1\ s\ T_2\texttt{)}, \langle\!\langle w_s\, \omega_p \rangle\!\rangle) \\
&\quad = \ \langle \epsilon x_p . C[[\text{state } [x_s]\ [\text{set } x_s \times (e_p \cdot \langle\!\langle s \leftarrow x_p \rangle\!\rangle \cdot \langle\!\langle \omega_p \rangle\!\rangle) \\
&\qquad\qquad\qquad\qquad [\text{eval } T_2\ \langle\!\langle [x_s] \rangle\!\rangle]]]] \rangle \\
&\qquad \text{if } x_p \notin \mathrm{FV}(T_2), \\
&\qquad\quad \mathit{definiend}(T_1, \mathrm{FV}(T_2) \cup \{[w_i]\}) = \langle C, e_p, \mathcal{X} \rangle, \\
&\qquad\quad x_s \notin \mathrm{FV}(T_2), \text{ and either } w_s = \phi \text{ and } x_s = [i], \\
&\qquad\quad \text{or } w_s = [w_i] \text{ and } x_s = [i w_i] \in \mathrm{FV}(e_p)\,.
\end{aligned} \tag{12.43}$$

Three schemata here simplify state frames: one merges consecutive state frames, one deletes a state variable that is never used (as defined immediately before (12.23)), and one garbage-collects a state frame with an empty definiend.

$\wr S$-calculus.
Schemata (state simplification):

$$\begin{aligned}
&[\text{state } [w_s]\ [\text{state } [w'_s]\ T]] \\
\longrightarrow\ &[\text{state } [w_s] \cdot [w''_s]\ T[w'_s \leftarrow w''_s]] \\
&\quad \text{where } \mathit{rename}(w'_s, w_s \cup \mathrm{FV}([\text{state } [w'_s]\ T])) = \langle w''_s, \mathcal{X}\, \rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.44\sigma\sigma)
\end{aligned} \tag{12.44}$$

$$\begin{aligned}
&[\text{state } [w_s x_s w'_s]\ T] \\
\longrightarrow\ &[\text{state } [w_s w'_s]\ T][x_s \nleftarrow] \quad \text{if } x_s \text{ is not used in } T \qquad (12.44\sigma\mathrm{g})
\end{aligned}$$

$$\begin{aligned}
&[\text{state } [\,]\ T] \\
\longrightarrow\ &\qquad T\,. \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (12.44\sigma 0)
\end{aligned}$$

State frames, like the catch frames of $\wr C$-calculus (§11.3), are essentially declarative and therefore could in principle be allowed to bubble up through most contexts, even regardless of evaluation order. Arguably, state frames are even purer than catch frames, in that when a state frame bubbles up, the only substitution it performs is α-renaming, in contrast to the imperative context-substitution performed when a catch frame bubbles up. However, aggressive bubbling-up rules can become quite complicated (for example, some have to be matched with *sinking* schemata that reverse the bubbling-up in order to preserve Church–Rosser-ness); so, for simplicity, the only one we provide here is state frame bubbling up through a set frame, which is required for operational completeness since it is supported by $\wr S$-semantics (Schemata 12.19).

$\wr S$-calculus.
Syntax (contexts):

$$N^s_{\text{state}} ::= E^s \mid [\text{set } [\![\omega_s]\!]\ \Box] \qquad \text{(Singular non-state-blocking contexts)} \tag{12.45}$$

Schemata (state bubbling-up):

$$\begin{aligned} & N^s_{\text{state}}[[\text{state } [w_s]\ T]] \\ \longrightarrow\ & [\text{state } [w_s][w_s \leftarrow w_i]\ N^s_{\text{state}}[T[w_s \leftarrow w_i]]] \\ & \quad \text{where } \mathit{rename}(w_s, \text{FV}(N^s_{\text{state}}[[\text{state } [w_s]\ T]])) = \langle w_i, \mathcal{X} \rangle\,. \end{aligned}$$

12.3.7 Mutable-to-immutable coercion

The mitigations of bad hygiene discussed in Chapter 5 center on deducing that certain mutable bindings or environments cannot, in fact, be mutated. This would appear in $\wr S$-calculus as mutable-to-immutable coercion of structures in a term; and as a partial exploration of the issues involved, we have provided substitution functions suitable for the purpose ($\Box[x_s \not\leftarrow]$ and $\Box[[x_s, s] \leftarrow V]$).

However, in the current document we do not provide schemata for these coercions. The necessary requirements on such schemata are (1) restriction to cases for which we can prove there will never be an attempt to mutate the structure, and (2) inclusion of reductions sufficient to prove Church–Rosser-ness. These are both essentially proof-driven requirements, based on patterns of term evolution across extended reduction sequences (rather than on static patterns of term structure); and as such, they are unsurprisingly complex, and stylistically contrasting with the other schema specifications presented. It was judged that, in exchange for the significant added complexity, treatment of these schemata here would do little to further illuminate the thesis (beyond the *fact* that they are based on term evolution rather than static term structure, which we have now already noted), and would do substantially nothing to further illuminate the claim of well-behavedness of the $\wr$-calculi (which is the main focus of Part II of the dissertation).

12.3.8 $\wr_r S$-calculus

The schema alterations to $\wr_i$-calculus for $\wr S$-calculus are: `$define!`, (12.36); set simplification, (12.37); set bubbling-up, (12.38); symbol evaluation, (12.39); get resolution, (12.40); get simplification, (12.41); get bubbling-up, (12.42); state simplification, (12.44); and state bubbling-up, (12.45).

Whereas the regular control calculus, $\wr_r C$-calculus, retained bubbling-up schemata because they involved (nontrivial) substitution, here the bubbling-up schemata are not substitutive, so they can be omitted from the regular variant with impunity (which would, if fully implemented, eliminate (12.38), (12.42), and (12.45)). All of the impure-frame simplification schemata are omitted, for various reasons (eliminating schemata (12.37), (12.41), and (12.44)): the merge schemata are all omitted

due to self-interference; the variable-collection schemata are omitted because their constraints are inherently non-regular; and the empty-frame elimination schemata are omitted because they interfere with get resolution and get bubbling-up. The bubbling-up schemata for set and state do, in fact, interfere with get resolution, and are omitted; but get bubbling-up does not interfere with any of the other schemata being retained, so it stays in, lending a degree of nontriviality to the regular-variant state calculus.

The remaining $\mathring{l}_r S$-calculus schemata beyond $\mathring{l}_r$-calculus are ***$define!***, (12.36); symbol evaluation, (12.39); get resolution, (12.40); and get bubbling-up, (12.42).

Chapter 13

Substitutive reduction systems

13.0 Introduction

Ideally, we would prove for each λ-calculus (specifics for λ-calculi will be pursued in Chapter 14) the well-behavedness properties called for by Plotkin's paradigm (§8.3.2):

(1) Church–Rosser-ness of $\longrightarrow_\bullet$: if $T_1 \longrightarrow_\bullet^* T_2$ and $T_1 \longrightarrow_\bullet^* T_3$, then there exists T_4 such that $T_2 \longrightarrow_\bullet^* T_4$ and $T_3 \longrightarrow_\bullet^* T_4$.

This is generally the first well-behavedness property proven, because so many other results rely on it.

(2) Standardization of $\longrightarrow_\bullet^*$: there exists some standard order of reduction, such that if $T_1 \longrightarrow_\bullet^* T_2$, then there is a reduction from T_1 to T_2 that performs its reduction steps in standard order.

This property is intended to mediate the relation between $\bullet$-calculus and $\bullet$-semantics (specifically, Property (4), below); therefore, we want a standard order consistent with the deterministic order of reduction by the semantics. Imitating [Plo75], we will seek a standard order that first exercises redexes in evaluation contexts (skipping any that won't be exercised in this reduction), and then recursively applies the same principle to subterms.[1]

(3) Operational completeness of $\longrightarrow_\bullet^*$: $\longmapsto_\bullet^* \subseteq \longrightarrow_\bullet^*$.

Usually, and for all our λ-calculi, this result follows straightforwardly from the definitions.

[1]The reason Plotkin's, and our, notion of standardization doesn't *uniquely* determine order of reduction is that when a standard reduction sequence shifts from evaluation to recursion, the recursion is potentially on *all* subterms, rather than only on subterms in non-evaluation contexts — so that evaluation steps that aren't taken during the evaluation phase might still be taken during the recursion phase. Cf. Lemma 13.89.

(4) Operational soundness of $=_\bullet$: $=_\bullet \subseteq \simeq_\bullet$.[2]

In proving any of these properties, not least Church–Rosser-ness, substitution causes most of the problems. Two alternative substitutive reductions of a term may mangle each other's redexes — either transforming a redex, by substituting something else into it; or making multiple copies of a redex, by substituting it into something else, after which the different copies could then be differently mangled during further reduction. A particularly general solution to this problem was provided by Jan Willem Klop's 1980 dissertation, [Kl80]. Klop proved Church–Rosser-ness and standardization for a broad class of λ-like calculi that he called *regular combinatory reduction systems* (*regular CRSs*), which can have any number of reduction rule schemata, each of which can perform λ-calculus-style substitution (what we are calling here *partial-evaluation substitution*), provided the entire schemata set satisfies certain sufficient conditions to guarantee that exercising any one redex cannot disable any alternative redex.

The pure λ-calculi are regular CRSs, so we could get Church–Rosser-ness and standardization for those calculi directly from Klop (though the standardization might not be suitable to mediate operational soundness, as Klop's notion of standardization is descended from Curry and Feys' rather than Plotkin's). However, the impure λ-calculi are not CRSs, because they are not limited to partial-evaluation substitution. Moreover, Klop handles substitution, and hygiene (his sufficient conditions on substitution), by hardwiring them directly into the syntactic infrastructure of his meta-language — so that one literally *can't* formulate any other notion of substitution/hygiene without first recasting his work in a different meta-language.[3] Substitution functions in the impure λ-calculi are not plausibly hardwireable, because the impure λ-calculi treat substitution functions as a commodity; for the impure calculi, we have defined *nine* different substitution functions (including partial-evaluation substitution), each with its own distinct behavioral quirks.[4]

To treat the impure λ-calculi, we develop a general class of *regular substitutive reduction systems* (*regular SRSs*), which can have multiple schemata using multiple substitution functions, provided certain sufficient conditions are met by the schemata *and* the substitution functions. The substitution functions are explicit, hence subject to explicit analysis, rather than built into the meta-language. Regular SRSs are similar to Klop's regular CRSs, reformulated to allow more general forms of substi-

[2]Formal equality $=_\bullet$ is sound because it implies operational equivalence $\simeq_\bullet$, but would only be complete if operational equivalence implied it, which we do not claim. Similarly, calculus reduction $\longrightarrow_\bullet^*$ is complete because semantics reduction $\longmapsto_\bullet^*$ implies it, but would only be sound if it implied the semantics reduction, which would defeat the purpose of the calculus since the calculus would then be identical to the semantics.

[3]The situation is reminiscent of quantum mechanics, which protects its basic metaphysical principles from assault by requiring the physicist to express all questions in a form that presupposes those principles.

[4]$\Box[x_p \leftarrow T]$, $\Box[x_c \leftarrow C]$, $\Box[x_c \leftarrow x'_c]$, $\Box[w_s \leftarrow w_i]$, $\Box[x_s \not\leftarrow]$, $\Box[w_g \leftarrow w'_g]$, $\Box[x_g \leftarrow V]$, $\Box[x_g \not\leftarrow]$, $\Box[[x_s, s] \leftarrow V]$. One of these, $\Box[[x_s, s] \leftarrow V]$, isn't actually used here (§12.3.7).

tution. Church–Rosser-ness, standardization, and operational soundness theorems are proven over the entire class of regular SRSs, and the pure λ-calculi and "regular variant" impure λ-calculi (based on λ_r-calculus) are shown to belong to this class. The remaining schemata of each impure λ-calculus are separately shown to preserve well-behavedness from its regular subset.

To formulate the criteria for regular SRSs, we will decompose terms into compositions of poly-contexts, and identify various roles that poly-contexts can play in schemata and substitutions. Some significant roles amongst these are

- *selective* poly-contexts, which are always either entirely inside a redex, or entirely outside it.
- *decisively reducible* poly-contexts, which guarantee that the matching term is a redex. In a regular SRS, every minimal decisively reducible poly-context must be selective.
- *suspending* poly-contexts, which can be involved in a minimal reducible context *above* them in the syntax tree, but cannot be involved in a minimal reducible context *below* them in the syntax tree. (In λ-calculus, the suspending contexts are the ones of the form $(\lambda x.\Box)$.) Suspending poly-contexts don't contribute to determining regularity, and they aren't involved in the proof of Church–Rosser-ness, but they are central to defining standard order(s) of reduction.

13.1 Substitution

13.1.1 Poly-contexts

We'll need some additional terminology and notation for working with poly-contexts.

Definition 13.1

Term T *satisfies* poly-context P if there exist $\vec{T}$ such that $T = P[\vec{T}]$.

Poly-context P_2 *satisfies* poly-context P_1 if every T satisfying P_2 satisfies P_1.

Poly-context P is *trivial* if P is a meta-variable.

Poly-context P is *minimal nontrivial* if P is nontrivial, and for every poly-context P' satisfied by P, either P' is trivial or P' satisfies P.

Poly-context P is *singular* if P is nontrivial, $ar(P) = 1$, $P[\Box]$ is a context, and for every context C satisfied by P, either C is trivial, or C satisfies P.

Context C *minimally satisfies* poly-context P (or, is a minimal context satisfying P) if C satisfies P, and for every context C' satisfying P and satisfied by C, $C' = C$. ■

A trivial poly-context is satisfied by every term. If some nontrivial poly-context P were satisfied by every term, one could prove that there is only one term; but we will

assume that there is more than one term, therefore for every nontrivial poly-context there exists a term that doesn't satisfy it.

For every nontrivial poly-context P, there exist minimal nontrivial poly-contexts satisfied by P; and all minimal nontrivial poly-contexts satisfied by P satisfy each other. In λ-calculus, the minimal nontrivial poly-contexts are those of the forms x, c, $(\Box_j\ \Box_k)$ where $j \neq k$, and $(\lambda x.\Box_k)$.

For every nontrivial context C, there exists one and only one singular poly-context satisfied by C. In λ-calculus, the singular poly-contexts are those of arity one with the forms $(T\ \Box_1)$, $(\Box_1\ T)$, and $(\lambda x.\Box_1)$.

For every poly-context P, the contexts C minimally satisfying P can be formed from P by replacing one of the meta-variable occurrences of P with $\Box$, and the rest with terms. In λ-calculus, the contexts minimally satisfying $P = (\Box_1\ \Box_1)$ are those of the forms $(\Box\ T)$, $(T\ \Box)$; contexts of the form $((\lambda x.\Box)\ T)$ satisfy P, but non-minimally.

Definition 13.2 Poly-context P is *monic* if each meta-variable of P occurs at most once in P. P is *epic* if each meta-variable of P occurs at least once in P. P is *iso* if it is both monic and epic. ■

These definitions of *monic*, *epic*, and *iso* view P as a way to decompose any term T that satisfies P into a vector of subterms, $T = P[\vec{T}]$. Subterms of T appear in P as meta-variable occurrences; and the position of each subterm of T in the resulting vector $\vec{T}$ is determined by the meta-variable index on the corresponding meta-variable occurrence in P. If the mapping from meta-variable occurrences to indices is a monomorphism (a.k.a. one-to-one, injective, meaning that no meta-variable index occurs more than once), the poly-context is monic; if an epimorphism (onto, surjective, meaning every allowed meta-variable index occurs at least once), the poly-context is epic; if an isomorphism, the poly-context is iso.

For example, poly-context $P = [\text{combine } \Box_1\ (\Box_2\ \Box_2)\ e]$ isn't monic, because index 2 occurs more than once; but P may be epic, depending on its arity: P has every index up to 2, so if $ar(P) = 2$ then P is epic, while if $ar(P) \geq 3$ then P is not epic. On the other hand, $P = [\text{combine } \Box_1\ (\Box_3\ \Box_4)\ e]$ is monic, but cannot be epic because its arity is at least 4 and index 2 doesn't occur. Finally, $P = [\text{combine } \Box_1\ (\Box_3\ \Box_2)\ e]$ is monic, and iff $ar(P) = 3$ then P is also epic, hence iso.

All minimal nontrivial poly-contexts are monic (because if some $\Box_k$ occurs more than once in P, then one can construct a strictly less constraining nontrivial P' by re-indexing the meta-variable occurrences of P). For every poly-context P, there exists an iso minimal nontrivial P' satisfied by P; and for given P, all such P' differ from each other only by permutation of their meta-variable indices.

Contexts may be coerced to poly-contexts of arity 1, by implicitly replacing the context meta-variable $\Box$ with $\Box_1$; as an explicit conversion, $P = C[\Box_1]$. This provides a simple way to specify that a poly-context is unary and iso. (Cf. *singular poly-context*, Definition 13.1.)

Definition 13.3 A *branch* of a poly-context P_1 is a poly-context P_2 such that for some poly-contexts $P, \vec{P},\ P_1 = P[\vec{P}],\ P$ is epic, and $P_2 \in \vec{P}$. If P is nontrivial, P_2 is a *proper* branch of P_1.

A *normal prime factorization* of a poly-context P_1 is an expression of P_1 as a composition of iso minimal nontrivial poly-contexts and trivial poly-contexts, such that no such composition involves fewer instances of trivial poly-contexts. The iso minimal nontrivial poly-contexts are the *prime factors* in that factorization. ■

In λ-calculus, poly-context $(\lambda \mathrm{x}.\Box_1)\Box_2$ has exactly four branches: $(\lambda \mathrm{x}.\Box_1)\Box_2,\ \lambda \mathrm{x}.\Box_1$, $\Box_1$, and $\Box_2$. It has exactly two normal prime factorizations: $(\Box_1\Box_2)[(\lambda \mathrm{x}.\Box_1), \Box_2]$ and $(\Box_2\Box_1)[\Box_2, (\lambda \mathrm{x}.\Box_1)]$. The stipulation "no composition involves fewer trivial poly-contexts" guarantees that a normal prime factorization will never apply a trivial poly-context (as in $(\lambda \mathrm{x}.\Box_1) = \Box_1[(\lambda \mathrm{x}.\Box_1)]$), will never apply a nontrivial poly-context to a vector of the form $\sum_{k=1}^{n} \Box_k$ (as in $(\lambda \mathrm{x}.\Box_1) = (\lambda \mathrm{x}.\Box_1)[\Box_1]$), and will never unnecessarily use a vector of trivial poly-contexts to rearrange the meta-variable indices of a prime factor (as in $(\Box_2\Box_1) = (\Box_1\Box_2)[\Box_2, \Box_1]$).

Every poly-context P has one or more normal prime factorizations; and the number of instances of trivial factors is fixed (by definition, it has to be the minimum possible), while the instances of prime factors can only vary by permutation of the meta-variable indices, so the number of normal prime factorizations of P is always finite. Every branch of P satisfies some prime factor in each normal prime factorization of P; and each prime factor in each normal prime factorization of P is satisfied by some branch of P.

For any objects $\vec{O}, O$, and integer k with $1 \leq k \leq ar(\vec{O})$, notation "$\vec{O}\backslash\!\!\backslash^{k}_{O}$" signifies the vector of objects formed from $\vec{O}$ by replacing the kth element of the vector with O. That is,

$$\vec{O}\backslash\!\!\backslash^{k}_{O} \;=\; \sum_j \begin{cases} O & \text{if } j = k \\ \vec{O}\,(j) & \text{otherwise}\,. \end{cases} \tag{13.4}$$

Often, $\vec{O}, O$ are terms. The notation can also be used to convert an m-ary poly-context P into a unary poly-context by replacing all but one of its meta-variables with terms; then, the pre-existing vector elements are terms, but the kth element spliced in is $\Box_1$ (the expected name for the meta-variable of a unary poly-context). For example,

$$\begin{aligned} (\Box_1\ \Box_2\ \Box_3)[\langle T_1, T_2, T_3\rangle\backslash\!\!\backslash^{2}_{\Box_1}] &= (T_1\ \Box_1\ T_3) \\ (\Box_1\ \Box_2\ \Box_3)[\langle T_1, T_2, T_3\rangle\backslash\!\!\backslash^{2}_{\Box_1}][T_4] &= (T_1\ T_4\ T_3) \\ &= (\Box_1\ \Box_2\ \Box_3)[\langle T_1, T_2, T_3\rangle\backslash\!\!\backslash^{2}_{T_4}]\,. \end{aligned} \tag{13.5}$$

For any integer $m \geq 0$, $\vec{\Box}_m$ is the vector of length m whose elements are meta-variables with successive indices starting from 1; thus, $\vec{\Box}_m$ is the m-ary prefix of infinite vector $\sum_k \Box_k$. (E.g., $\vec{\Box}_3 = \langle \Box_1, \Box_2, \Box_3\rangle$.)

Definition 13.6 Suppose poly-context P, and semantic polynomial π.

P *satisfies* π if every term satisfying P satisfies π.

P *minimally satisfies* π if P satisfies π and, for all poly-contexts P', if P satisfies P', and P' satisfies π, then P' satisfies P. ■

For later proofs, it will be desirable that for each T satisfying π, if P_1 and P_2 are satisfied by T and minimally satisfy π, then P_1 and P_2 must satisfy each other. There are some pathological reasons why this might not be so:

- if, when constructing a term, separate subterms are not independent of each other.
- if, when constructing a term, a context uniquely determines the top-level structure of the subterms that can occur in it.
- if, when constructing an expression in the domain of quantification of a semantic variable, separate subterms are not independent of each other.

Assumptions 13.7

(a) For every iso poly-context P and terms $\vec{T}$ of like arity, if for each $\vec{T}(k)$ there exists some term satisfying P in which the occurrence of $\Box_k$ is replaced by $\vec{T}(k)$, then $P[\vec{T}] \in \mathit{Terms}$.

(b) For every semantic polynomial π and contexts $\vec{C}$, if π is not a semantic variable, then there exists T such that every $(\vec{C}(k))[T] \in \mathit{Terms}$, and T does not satisfy π.

(c) For every semantic polynomial π, term T, and contexts $\vec{C}$, if no semantic variable occurs more than once in π, T satisfies π and all the $\vec{C}$, and all the $\vec{C}$ satisfy π, then there exists an iso poly-context P satisfied by T and by all the $\vec{C}$ and satisfying π. ■

The second assumption, 13.7(b), is (as stated in the corresponding bulleted item above) about contexts, not about semantic polynomials despite its use of one. It says that no set of contexts $\vec{C}$ can restrict the term that replaces the meta-variable in a way that would require any particular fragment of concrete syntax — not even a fragment that is less than a minimal nontrivial poly-context. The notion of a fragment of concrete syntax is expressed by the semantic polynomial in the assumption, which is required to be *not a semantic variable*, so that it must specify some fragment of concrete syntax. (For example, semantic polynomial "$(\lambda x.T)$" specifies the parentheses and the λ and the dot, but can be satisfied by infinitely many minimal nontrivial poly-contexts that do not satisfy each other: $(\lambda \mathrm{x}.\Box_1)$, $(\lambda \mathrm{y}.\Box_1)$, $(\lambda \mathrm{z}.\Box_1)$, etc.) By the assumption, no matter how small the fragment of concrete syntax included in semantic polynomial π, as long as *some* concrete syntax is included in it, there will

always exist a term that can be used in all the contexts of interest ($\vec{C}$) but that does not satisfy π.

The third assumption, 13.7(c), is meant to constrain the domains over which semantic variables are quantified. Suppose that a semantic variable is quantified over some class of syntactic structures that may occur in terms — say, *Structs*. (The structures don't have to *be* terms in order to *contain* subterms; for example, in any λ-calculus except λ_x-calculus, a syntactic structure $s \leftarrow V$ can occur in terms, and contains a term, but is not itself a term.) The assumption says that whenever a structure belongs to *Structs*, and each of several separate subterms of the structure is individually unconstrained by conformance of the whole to *Structs*, then conformance of the whole to *Structs* can't require those subterms to somehow correlate with each other. Violating this would be a a bizarre phenomenon, but is imaginable; one could, for example, have three subterms and require that two out of three be identical, but it doesn't matter which two; so the subterms aren't independent even though any *one* of them can vary arbitrarily.

That assumption is expressed in terms of semantic polynomials rather than semantic variables because, being built up from Definition 13.6, it is only relevant to a semantic expression that signifies a term (i.e., only when there exists T satisfied by π) — so that if it were restricted to semantic variables, there would be no constraint on subterms of non-term semantic variables (e.g., ω_s in §12.1.2). Since the assumption is applied to polynomials, it constrains the syntactic domain of every semantic variable that occurs *within* a polynomial signifying a term, even though the semantic variable itself might never signify a term.

Lemma 13.8 If some term satisfies poly-context P, and P satisfies semantic polynomial π, then each meta-variable occurrence in P occurs in some part of P that matches a semantic variable occurrence in π. ■

For example, P satisfying $\pi = (\lambda x.T)$ can only have meta-variable occurrences within those parts of P that match semantic variables x and T. (That's assuming that x and T are the semantic variables, while λ is not a semantic variable. In the particular case of the syntax and notational conventions of λ-calculus, it happens that an expression matching semantic variable x cannot contain a general term, and therefore has no place for a meta-variable occurrence within it; so for λ-calculus, meta-variable occurrences in P would be restricted to the part of P that matches semantic variable T. Evidently, poly-contexts minimally satisfying π would then be exactly those of the form $(\lambda x.\Box_k)$, such as $(\lambda \mathrm{x}.\Box_1)$ or $(\lambda \mathrm{y}.\Box_{27})$.)

Proof. Suppose $P[\vec{T}] \in \mathit{Terms}$, P satisfies π, and P contains an occurrence of $\Box_k$ that is not contained within the part of P matching any semantic variable occurrence of π. Let π' be the part of π that matches that occurrence of $\Box_k$. π' is not a semantic variable, because if it were then the occurrence of $\Box_k$ would in fact be occurring within the part of P matching an occurrence of that semantic variable. Let $\vec{C}$ consist of just those contexts formed from P by replacing one occurrence of $\Box_k$ with $\Box$,

and the rest of the meta-variable occurrences $\Box_j$ with $\vec{T}(j)$; thus, $(\vec{C}(j))[\vec{T}(k)] = T$. By Assumption 13.7(b), let T' be a term that does not satisfy π', such that each $(\vec{C}(j))[T'] \in \mathit{Terms}$. By Assumption 13.7(a), $P[\vec{T}\backslash^k_{T'}] \in \mathit{Terms}$; but since π' matches one of the $\Box_k$ occurrences in P, and T' does not satisfy π, term $P[\vec{T}\backslash^k_{T'}]$ does not satisfy π. Therefore P does not satisfy π, a contradiction. ■

Lemma 13.9 If T satisfies semantic polynomial π, then there exists an epic poly-context P satisfied by T and minimally satisfying π. ■

Proof. There are only finitely many epic poly-contexts satisfied by T, at least one of which satisfies π (because T itself can be viewed as an epic poly-context with arity zero). The satisfaction relation between poly-contexts is a preorder (i.e., reflexive and transitive), since it is derived from the subset relation between sets of terms. Therefore, there exists at least one epic poly-context satisfied by T and satisfying π such that, if any *epic* poly-context P' is satisfied by P and satisfies π, then P' satisfies P. If there exists a non-epic P' satisfied by P and satisfying π, then by re-indexing there exists an epic P'' satisfying and satisfied by P', and by transitivity of satisfaction P'' satisfies P; ergo, P minimally satisfies π. ■

Lemma 13.10 Suppose semantic polynomial π.

If T satisfies poly-contexts P_1 and P_2, and P_1 and P_2 satisfy π, then there exists poly-context P satisfied by P_1 and P_2 and satisfying π. ■

Proof. Suppose T satisfies P_1 and P_2, and P_1 and P_2 satisfy π.

Case 1: No one semantic variable occurs more than once in π.

Let $\vec{C}$ consist of every context that can be formed from P_1 or P_2 by replacing one meta-variable occurrence with $\Box$ and the rest with the subterms that replace them in T. Then T satisfies all the $\vec{C}$, and since each $\vec{C}(j)$ satisfies either P_1 or P_2, all the $\vec{C}$ satisfy π. By Assumption 13.7(c), let P be an iso poly-context satisfied by T and by all the $\vec{C}$ and satisfying π.

Suppose $k \in \{1, 2\}$, and T' satisfies P_k. Then the subterm replacing $\Box_j$ of P_k in T' can also replace $\Box$ in some $\vec{C}(i)$ (by Assumption 13.7(a)). Since $\vec{C}(i)$ satisfies P, any changes to the term caused by that replacement into $\vec{C}(i)$ must occur entirely within a meta-variable of P; therefore, making simultaneous replacements for all the meta-variables of P_k to produce T' only changes things within meta-variables of P, and since T' is a term, it satisfies P. So P_k satisfies P.

Case 2: At least one semantic variable occurs more than once in π.

Let π' be formed from π by replacing duplicate semantic variables with distinct semantic variables quantified over the same syntactic domains. By Lemma 13.9, let poly-context P' be satisfied by T and minimally satisfy π'; and by Case 1, assume

without loss of generality that P' is iso. By Lemma 13.8, each meta-variable in P' occurs within a part of P' that matches a semantic variable in π'.

Let P be constructed from P' as follows: For each semantic variable in π, take the part of P' that matches the first instance of that semantic variable, and copy that part to all the other places in P' that match instances of the same semantic variable. (If any of the copied parts of P' contains a meta-variable occurrence, the modified poly-context will not be monic; and if any of the parts that were overwritten contained a meta-variable occurrence, the modified poly-context will not be epic.)

Each time a part of P' was copied, since it was copied to a place that matched the same semantic variable in π —and T satisfies π— both places match the same subexpression in T, which is known to satisfy the copied part of P'; therefore, T satisfies P. The constraints introduced by copying exclude terms satisfying P' that do not satisfy π; therefore, P satisfies π. Suppose $k \in \{1, 2\}$. By Lemma 13.8, P_k and P can only differ from each other within the parts that match semantic variables of π. If T' satisfies P_k, then T' satisfies P', and therefore the part of T' matching any given meta-variable of π must satisfy whichever part of P' matches that meta-variable throughout P; therefore, T' satisfies P. So P_k satisfies P. ■

Theorem 13.11 Suppose semantic polynomial π.

If T satisfies π, then there exists poly-context P minimally satisfying π such that, for all poly-contexts P', if P' is satisfied by T and satisfies π, then P' satisfies P. ■

Proof. Follows immediately from Lemmata 13.9 and 13.10. ■

13.1.2 α-equivalence

What we have been calling "substitution functions" can be separated into two distinct classes, depending on which binary relation they are used to support. Recall that each of our calculi is founded on two distinct binary relations: an equivalence $\equiv_\alpha$ between terms, and a reduction relation $\longrightarrow^\bullet$ between $\equiv_\alpha$-classes of terms.[5] This subsection addresses the class of functions that support $\equiv_\alpha$, which we call *renaming functions*.[6]

The class of functions that support $\longrightarrow^\bullet$, which we call *substitutive functions*, will be treated in §13.1.3.

[5]This two-relation strategy was (as related in §8.1.2) established by Church and Rosser's proof of the Church–Rosser property, where the two relations were induced by Postulates I and II of Church's 1932 logic.

[6]In retrospect, it appears (to the author) that this treatment of renaming functions is much complicated by its rather extreme aversion to concrete syntax. The subject is comparatively straightforward for λ-calculus exactly because variables are concrete syntactic atoms, and have consequently

As usual, we assume a syntactic domain *Terms*, equipped with a congruence (i.e., compatible equivalence) $\equiv_\alpha$. *Terms* is not required to be freely generated over its CFG; that is, for context C and term T, in general $C[T]$ might not be a term. When explicitly postulating the existence of a context+term or poly-context+term-vector structure, postulation of its membership in *Terms* may be elided; thus, for example, we may say simply "suppose $P[\vec{T}]$" rather than "suppose term $P[\vec{T}]$" or "suppose $P[\vec{T}] \in \mathit{Terms}$".

Definition 13.12 Suppose binary relation R on terms.
R is *compatible* if for all C, T_k,
if $C[T_1], C[T_2] \in \mathit{Terms}$ and $\langle T_1, T_2\rangle \in R$ then $\langle C[T_1], C[T_2]\rangle \in R$.
R is *constructive* if for all C, T_k,
if $C[T_1] \in \mathit{Terms}$ and $\langle T_1, T_2\rangle \in R$ then $C[T_2] \in \mathit{Terms}$. ■

From further assumptions, it will follow that $\equiv_\alpha$ is constructive (Corollary 13.27).

We assume a countably infinite syntactic domain *Vars* of variables, over which we quantify semantic variables x; and we assume that *Vars* is equipped with a partial ordering $\sqsubseteq$, read "is descended from". In any concrete calculus, *Vars* will be the union of all particular classes of variables. For example, in $\natural S$-calculus, $\mathit{Vars} = \mathit{PartialEvaluationVariables} \cup \mathit{StateVariables} \cup \mathit{GetVariables}$; $\sqsubseteq$ extends trivially (i.e., reflexively) to the naturally flat classes of variables, $\mathit{PartialEvaluationVariables} \cup \mathit{GetVariables}$. Each occurrence of a variable x is assumed to also be an occurrence of every ancestor of x; but we make no assumption about how this is accomplished (such as the concrete implementation of state variables in §12.1.1). Variables are not required to be terms.

Definition 13.13 Suppose variable sets $\mathcal{X}_k \subseteq \mathit{Vars}$.
$\mathcal{X}_1$ is *orthogonal to* $\mathcal{X}_2$, denoted $\mathcal{X}_1 \perp \mathcal{X}_2$, if for all $x_1 \in \mathcal{X}_1$ and $x_2 \in \mathcal{X}_2$, $x_1 \not\sqsubseteq x_2$ and $x_2 \not\sqsubseteq x_1$. ■

The family of renaming functions is denoted $\mathcal{F}_\alpha$. Formally, renaming functions act on the union of *Terms* and *Vars*, mapping each variable to a variable, and each term to a term. $\mathcal{F}_\alpha$ is assumed to be closed under finite composition, and to include the null composition, i.e., the identity function, which we denote $f_{\mathbf{id}}$. Renaming functions in our $\natural$-calculi have the forms $\Box[x_p \leftarrow x'_p]$, $\Box[x_c \leftarrow x'_c]$, $\Box[w_s \leftarrow w_i]$, $\Box[w_g \leftarrow w'_g]$, and all compositions of finitely many thereof.

Each renaming function f has a unique *complement*, which is another renaming function that attempts to undo what f does. Two renaming functions f, g may map

self-evident properties that, in the absence of this concrete grounding, must be tediously enumerated. Extreme abstraction here was chosen as a precaution against unnecessary concrete assumptions when entering unfamiliar territory; but, now that completion of this treatment has scouted the territory, it seems one might find a more felicitous intermediate point between the extreme abstraction here, and the mechanical intricacy of compound state variables in §12.1.1.

each input to the same output (i.e., for all T, $f(T) = g(T)$, and for all x, $f(x) = g(x)$), yet still have different complements — that is, $\mathcal{F}_\alpha$ are *intensional* functions, not necessarily uniquely defined by their input–output pairs. The complement of a composition is always the reverse-order composition of the complements: if the complement of f is f', and of g is g', then the complement of $f \circ g$ is $g' \circ f'$. The complement of the complement of any f is f. $f_{\mathbf{id}}$ is self-complementary.

For example, in λ-calculus let

$$\begin{array}{lcll} f_1 & = & \Box[\mathrm{y} \leftarrow \mathrm{x}] & \\ f_2 & = & \Box[\mathrm{z} \leftarrow \mathrm{x}] & \\ f_3 & = & f_2 \circ f_1 \;=\; (\Box[\mathrm{y} \leftarrow \mathrm{x}])[\mathrm{z} \leftarrow \mathrm{x}] & \\ f_4 & = & f_1 \circ f_2 \;=\; (\Box[\mathrm{z} \leftarrow \mathrm{x}])[\mathrm{y} \leftarrow \mathrm{x}]\,. & \end{array} \tag{13.14}$$

For all x, $f_3(x) = f_4(x)$. However, f_3 and f_4 are distinguished by their complements; the complements are

$$\begin{array}{lcll} f'_1 & = & \Box[\mathrm{x} \leftarrow \mathrm{y}] & \\ f'_2 & = & \Box[\mathrm{x} \leftarrow \mathrm{z}] & \\ f'_3 & = & f'_1 \circ f'_2 \;=\; (\Box[\mathrm{x} \leftarrow \mathrm{z}])[\mathrm{x} \leftarrow \mathrm{y}] & \\ f'_4 & = & f'_2 \circ f'_1 \;=\; (\Box[\mathrm{x} \leftarrow \mathrm{y}])[\mathrm{x} \leftarrow \mathrm{z}]\,, & \end{array} \tag{13.15}$$

and $f'_3(\mathrm{x}) = \mathrm{z}$, while $f'_4(\mathrm{x}) = \mathrm{y}$. Also, for all x, $f'_1(x) = f'_4(x)$ and $f'_2(x) = f'_3(x)$, but again these are distinguished by their complements: $f_1(\mathrm{z}) = \mathrm{z}$ while $f_4(\mathrm{z}) = \mathrm{x}$, and $f_2(\mathrm{y}) = \mathrm{y}$ while $f_3(\mathrm{y}) = \mathrm{x}$.

As usual, each variable occurrence in any term is either free or bound, and each poly-context binds a finite set of variables. Bindings are built up by induction from iso minimal nontrivial poly-contexts: in general, P binds x iff some non-term branch of P satisfies some iso minimal nontrivial poly-context that binds x. (From this it follows, for example, that a term T does not bind any x.) For this chapter, the finite set of variables that occur free in T is denoted $\mathrm{Free}(T)$, and the finite set of variables bound by P is denoted $\mathrm{Bind}(P)$. The set of all ancestors of variables in a set $\mathcal{X} \subseteq \mathit{Vars}$ is denoted $\mathrm{Above}(\mathcal{X}) \supseteq \mathcal{X}$, of descendants, $\mathrm{Below}(\mathcal{X}) \supseteq \mathcal{X}$; any variable x may be coerced to singleton set $\{x\}$. When an occurrence of x in T is free, all of the variable occurrences it contains are free (these being occurrences of $\mathrm{Above}(x)$, as just mentioned). Minimal nontrivial P contains at least one bound occurrence, and no free occurrences, of each variable that it binds. Free occurrences in T of $x \in \mathrm{Bind}(C)$ are bound in $C[T]$. For all T, the free set of T is closed under ancestry, i.e., $\mathrm{Free}(T) = \mathrm{Above}(\mathrm{Free}(T))$; hence, for all T satisfying minimal nontrivial P, $\mathrm{Free}(T) \cap \mathrm{Below}(\mathrm{Bind}(P)) = \{\}$.

For compound poly-contexts P, it will sometimes be necessary to distinguish between variables bound at different meta-variables of P. The set of variables bound *by P at $\Box_k$*, denoted $\mathrm{Bind}(P \text{ at } \Box_k)$, is the set of variables whose free occurrences in a subterm would be captured by P if the subterm replaced $\Box_k$. For any term $P[\vec{T}]$, $\mathrm{Bind}(P \text{ at } \Box_k) = \mathrm{Bind}(P[\vec{T} \backslash \overset{k}{\Box}])$.

Lemma 13.16
For all $C[T] \in \mathit{Terms}$, $\text{Free}(T) \cap (\text{Below}(\text{Bind}(C)) - \text{Bind}(C)) = \{\}$. ■

Proof. Suppose $C[T] \in \mathit{Terms}$, $x_2 \sqsubseteq x_1$, $x_2 \in \text{Free}(T)$, and $x_1 \in \text{Bind}(C)$. $x_2 \in \text{Free}(T)$ means there is a free occurrence of x_2 in T, which by assumption is also a free occurrence of every ancestor of x_2, including x_1; but occurrences of x_1 in T are not free in $C[T]$ because $x_1 \in \text{Bind}(C)$, and by assumption, when a variable occurrence is free, all the variable occurrences it contains are also free; therefore, x_2 must not be free in $C[T]$, which is to say, $x_2 \in \text{Bind}(C)$. ■

This lemma is the only permitted restriction on construction of terms $C[T]$ from arbitrary C and T.

Assumptions 13.17
(a) For every finite $\mathcal{X} \subset \mathit{Vars}$, there exists T such that $\text{Free}(T) = \text{Above}(\mathcal{X})$.
(b) For every x, there exists minimal nontrivial P such that $\text{Bind}(P) = x$ and $\text{Free}(P) = \text{Above}(x) - x$.
(c) For all C and T, if $\text{Free}(T) \cap (\text{Below}(\text{Bind}(C)) - \text{Bind}(C)) = \{\}$, then $C[T] \in \mathit{Terms}$. ■

Lemma 13.18 If $\mathcal{X} \subset \mathit{Vars}$ is finite, then $\text{Above}(\mathcal{X})$ is finite. ■

Proof. Suppose finite $\mathcal{X} \subset \mathit{Vars}$. By Assumption 13.17(a), there exists T such that $\text{Free}(T) = \text{Above}(\mathcal{X})$. By assumption (in the prose, above), the free set of every term is finite. ■

A renaming $f \in \mathcal{F}_\alpha$ affects a term T only by changing the names of variable occurrences within T (i.e., by causing them to become occurrences of different variables instead). This changing of variable occurrence names in T is determined *up to* $\equiv_\alpha$ by the behavior of f on variables; and $\equiv_\alpha$ is determined, in turn, up to internal use of renaming functions.

Assumptions 13.19 Suppose $f \in \mathcal{F}_\alpha$, variables x, x_k, and terms T, T_k.
(a) $f(T)$ differs from T only by the names of variable occurrences.
(b) Free occurrences of x in T become free occurrences of $f(x)$ in $f(T)$.
(c) Bound variable occurrences in T become bound variable occurrences in $f(T)$.
(d) If $T_1 \equiv_\alpha T_2$ then $f(T_1) \equiv_\alpha f(T_2)$.
(e) If $x_2 \sqsubseteq x_1$ then $f(x_2) \sqsubseteq f(x_1)$.
(f) If $x_2 \sqsubseteq x_1$ and $f(x_1) \neq x_1$ then $f(x_2) \neq x_2$. ■

Definition 13.20 Suppose $f \in \mathcal{F}_\alpha$ with complement f', terms T, T', and variables x, x'.

T' is an *α-image through f* of T, denoted $T \Rrightarrow_f T'$, if $f(T) \equiv_\alpha T'$ and $f'(T') \equiv_\alpha T$.

x' is an *α-image through f* of x, denoted $x \Rrightarrow_f x'$, if $f(x) = x'$ and $f'(x') = x$. ■

Lemma 13.21 Suppose $f, f_k \in \mathcal{F}_k$, and terms T_k.

(a) If $T_1 \Rrightarrow_{f_1} T_2$ and $T_2 \Rrightarrow_{f_2} T_3$, then $T_1 \Rrightarrow_{f_2 \circ f_1} T_3$.

(b) If $T_1 \Rrightarrow_f T_2$ and f' is the complement of f, then $T_2 \Rrightarrow_{f'} T_1$.

(c) If $T_1 \Rrightarrow_f T_2$ and $T_3 \Rrightarrow_f T_4$, then $T_1 \equiv_\alpha T_3$ iff $T_2 \equiv_\alpha T_4$. ■

Proof. (a) follows from Assumption 13.19(d).

(b) and (c) follow from Assumption 13.19(d) and the fact that the complement of the complement of f is f. ■

Intuitively, $O \Rrightarrow_f O'$ means that renaming f on object O is *reversible*. Complementarity provides a unique determination of how f must be reversed, so that $T_1 \Rrightarrow_f T$ and $T_2 \Rrightarrow_f T$ always imply $T_1 \equiv_\alpha T_2$. Recalling Example (13.14), $\mathrm{z} \Rrightarrow_{f_3} \mathrm{x}$ and $\mathrm{y} \Rrightarrow_{f_4} \mathrm{x}$, but $\mathrm{y} \not\Rrightarrow_{f_3} \mathrm{x}$ and $\mathrm{z} \not\Rrightarrow_{f_4} \mathrm{x}$; without complements, there would be no distinguishing f_3 from f_4, and we would have $\mathrm{y} \Rrightarrow_f \mathrm{x}$ and $\mathrm{z} \Rrightarrow_f \mathrm{x}$ despite $\mathrm{y} \not\equiv_\alpha \mathrm{x}$.

Now that we've established Lemma 13.21(c), we mostly won't need to mention complementarity hereafter.

Definition 13.22 Suppose $f \in \mathcal{F}_\alpha$, and $O \in \mathit{Terms} \cup \mathit{Vars}$.

O is *hygienic to f*, denoted $O \parallel f$, if $O \Rrightarrow_f f(O)$.

O is *orthogonal to f*, denoted $O \perp f$, if $O \Rrightarrow_f O$. ■

We will routinely use notations $\parallel f$ and $\perp f$ with sets of objects, meaning that the relation holds for all elements of the set. (For example, $\mathit{Vars} \perp f_{\mathbf{id}}$.)

Lemma 13.23 Suppose $f \in \mathcal{F}_\alpha$.

If $x_2 \sqsubseteq x_1$ and $x_1 \not\perp f$, then $x_2 \not\perp f$.

If $x_2 \sqsubseteq x_1$ and $x_1 \nparallel f$, then $x_2 \nparallel f$. ■

Proof. Suppose $x_2 \sqsubseteq x_1$, and let f' be the complement of f. By Assumption 13.19(f), if $f(x_1) \neq x_1$ then $f(x_2) \neq x_2$, and if $f'(x_1) \neq x_1$ then $f'(x_2) \neq x_2$, which is to say, by definition, that if $x_1 \not\perp f$ then $x_2 \not\perp f'$. Since $\mathcal{F}_\alpha$ is closed under composition, $f' \circ f \in \mathcal{F}_\alpha$; and by definition, $x_k \parallel f$ iff $x_k \perp f' \circ f$; so the second result follows from the first.[7] ■

[7]Assumption 13.19(f) is closely related to multi-parent environments, which (as noted at the top of Chapter 12) are not supported by the $\wr S$-calculus presented there, but which we would like the abstract treatment here to be capable of handling. If, instead of Assumption 13.19(f), we

Assumptions 13.24 Suppose $f \in \mathcal{F}_\alpha$, and term T.
(a) If $\text{Free}(T) \parallel f$, then $T \parallel f$.
(b) $\text{Free}(T) \perp f$ iff $T \perp f$. ■

The converse of Assumption 13.24(a) doesn't have to be assumed:

Lemma 13.25 Suppose $f \in \mathcal{F}_\alpha$, and term T.
If $T \parallel f$, then $\text{Free}(T) \parallel f$. ■

Proof. Suppose $T \parallel f$. Let f' be the complement of f. By Definition 13.22, $f'(f(T)) \equiv_\alpha T$. Since the complement of a composition is the reversed composition of the complements, $f' \circ f$ is self-complementary; therefore, since $f'(f(T)) \equiv_\alpha T$, by Definition 13.22, $T \perp (f' \circ f)$. By Assumption 13.24(b), $\text{Free}(T) \perp (f' \circ f)$. By Definition 13.22, for all $x \in \text{Free}(T)$, $f'(f(x)) = x$, so by Definition 13.22, $x \parallel f$. ■

Theorem 13.26 Suppose terms T_k.
$T_1 \equiv_\alpha T_2$ iff $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_2$. ■

Proof. By Definitions 13.20 and 13.22, $\text{Free}(T_1) \perp f_{\mathbf{id}}$. Therefore, by Assumption 13.24(b), $T_1 \perp f_{\mathbf{id}}$; and by Definition 13.22, $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_1$.

For implication left-to-right, suppose $T_1 \equiv_\alpha T_2$. By Definition 13.20 and Assumption 13.19(d), since $\equiv_\alpha$ is transitive, $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_2$.

For implication right-to-left, suppose $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_2$. By Definition 13.20, $f_{\mathbf{id}}(T_1) \equiv_\alpha T_2$; but $f_{\mathbf{id}}(T_1) \equiv_\alpha T_1$, so by transitivity of $\equiv_\alpha$, $T_1 \equiv_\alpha T_2$. ■

Corollary 13.27 $\equiv_\alpha$ is constructive. ■

Proof. Suppose $C[T_1] \in \mathit{Terms}$ and $T_1 \equiv_\alpha T_2$; we wish to show $C[T_2] \in \mathit{Terms}$. By the preceding theorem, $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_2$; by Definition 13.20, $T_1 \parallel f_{\mathbf{id}}$; by Assumption 13.24(a), $\text{Free}(T_1) \parallel f_{\mathbf{id}}$; and by Assumptions 13.19(b) and 13.19(c), $\text{Free}(T_2) = \text{Free}(T_1)$. Therefore, by Assumption 13.17(c), $C[T_2] \in \mathit{Terms}$. ■

While renaming $f(T)$ affects directly only the free variables of T, for structural-inductive treatment of α-renaming we want to transform an iso minimal nontrivial

assumed that for all x, $\text{Above}(x)$ is linearly ordered —i.e., all environments are single-parented— then Assumption 13.19(e) and the fact that $\sqsubseteq$ is a partial order would suffice for Lemma 13.23. However, in the presence of multi-parented variables, we might have the anomaly of a variable x whose parents x_k are *permuted* by $(f' \circ f)$, yet the permutation does not affect the identity of x, and thus $x \parallel f$ despite $x_k \not\parallel f$. Recognizing that these variables would be the identities of Kernel environments, the reason this anomaly can't occur in Kernel is that the ordering of parents is significant: if environment e has parents e_1 and e_2, then symbols not bound locally are looked up in e_1 first, while if the parents were permuted, they would be looked up in e_2 first.

poly-context P by applying a renaming to its bound, as well as free, variable occurrences. We write this type of renaming $f(P)$, and avoid notational confusion by defining it only for iso minimal nontrivial P. (If we tried to define $f(P)$ for arbitrary P, it would collide with notation $f(T)$, since terms are a special case of poly-contexts (with no meta-variable occurrences).)

Definition 13.28 Suppose $f \in \mathcal{F}_\alpha$, and iso minimal nontrivial poly-context P.

$f(P)$ denotes the poly-context produced from P by applying f to every variable occurrence; that is, for all x, all occurrences of x become occurrences of $f(x)$. ■

Definition 13.29 Suppose iso minimal nontrivial poly-contexts P_k.

P_2 is an *α-form* of P_1, denoted $P_1 \sim_\alpha P_2$, if there exists $g \in \mathcal{F}_\alpha$ such that all of the following conditions hold.

- For all $P_1[\vec{T}]$, if $\vec{T} \parallel g$ then $P_1[\vec{T}] \equiv_\alpha P_2[\sum_k g(\vec{T}(k))]$.
- $P_2 = g(P_1)$.
- $\mathrm{Free}(P_1) \cup \mathrm{Bind}(P_1) \parallel g$.

The relation $P_1 \sim_\alpha P_2$ is then *mediated by* g. ■

In the first condition, $\vec{T} \parallel g$ must imply $P_1[\vec{T}] \equiv_\alpha P_2[\sum_k g(\vec{T}(k))]$; but the converse is not required, not even given the other two conditions. This occurs because renaming functions can be extensionally identical but have different complements, and a mediating function of $P_1 \sim_\alpha P_2$ may be $\not\parallel$ to some variables that don't occur in P_1. For example, in λ-calculus, let $P_1 = \lambda\mathrm{x}.\Box_1$, $P_2 = \lambda\mathrm{y}.\Box_1$, and $g = (\Box[\mathrm{x} \leftarrow \mathrm{y}])[\mathrm{x} \leftarrow \mathrm{z}]$. g has complement $g' = (\Box[\mathrm{z} \leftarrow \mathrm{x}])[\mathrm{y} \leftarrow \mathrm{x}]$, and $\mathrm{y}, \mathrm{z} \not\parallel g$. For all T, $P_1[T] \equiv_\alpha P_2[g(T)]$ iff $\mathrm{y} \notin \mathrm{Free}(T)$; so $T \parallel g$ does imply $P_1[T] \equiv_\alpha P_2[g(T)]$; and since $P_2 = g(P_1)$ and $\mathrm{x} \parallel g$, g mediates $P_1 \sim_\alpha P_2$. Furthermore, $g(\mathrm{z}) = \mathrm{z}$, so free occurrences of z in T don't preclude $P_1[T] \equiv_\alpha P_2[g(T)]$; but they *do* preclude $T \parallel g$. Most simply, $P_1[\mathrm{z}] \equiv_\alpha P_2[g(\mathrm{z})]$ despite $\mathrm{x} \not\parallel g$.

In our concrete calculi, it will always be possible to find a mediating g for which the converse does hold; but we don't need that for the abstract treatment, and even the hygienic behavior of that g won't be unique in general. For example, in $\imath S$-calculus, consider $P = [\mathrm{state}\ [[\mathrm{i}_1][\mathrm{i}_2]]\ \Box_1]$; then $P \sim_\alpha P$ mediated by $f_{\mathbf{id}}$, but also $P \sim_\alpha P$ mediated by any g that hygienically swaps $[\mathrm{i}_1]$ with $[\mathrm{i}_2]$, such as $g = ((\Box[[\mathrm{i}_3] \leftarrow \mathrm{i}_2])[[\mathrm{i}_2] \leftarrow \mathrm{i}_1])[[\mathrm{i}_1] \leftarrow \mathrm{i}_3]$.

Lemma 13.30 Suppose iso minimal nontrivial poly-contexts P_k.

If $P_1 \sim_\alpha P_2$ and $\mathrm{Bind}(P_1) \cap \mathrm{Bind}(P_2) = \{\}$, then $\mathrm{Bind}(P_1) \perp \mathrm{Bind}(P_2)$. ■

Proof. Suppose $P_1 \sim_\alpha P_2$ and $\mathrm{Bind}(P_1) \cap \mathrm{Bind}(P_2) = \{\}$. Then $\mathrm{Free}(P_1) = \mathrm{Free}(P_2)$ (by Assumptions 13.19(b) and 13.19(c) and Theorem 13.26).

Suppose $x_1 \in \text{Bind}(P_1)$, $x_2 \in \text{Bind}(P_2)$, and $x_2 \sqsubset x_1$. Then x_1 must occur in P_2 (since any occurrence of x_2 is an occurrence of all its ancestors); and since $\text{Bind}(P_1) \cap \text{Bind}(P_2) = \{\}$, $x_1 \in \text{Free}(P_2)$. But since $\text{Free}(P_1) = \text{Free}(P_2)$, x_1 would have to be both free and bound in P_1, which is prohibited (by the paragraph preceding Lemma 13.16).

The symmetric case of $x_1 \sqsubset x_2$ follows by similar reasoning. ■

Just as $\equiv_\alpha$ generalizes to $\Rrightarrow_f$, $\sim_\alpha$ generalizes to $\rightsquigarrow_f$. For $P_1 \rightsquigarrow_f P_2$, P_1 is α-renamed to P_1' (i.e., $P_1 \sim_\alpha P_1'$), then f is applied naively as a homomorphism, and the resulting P_2' is α-renamed to P_2. In building the infrastructure for this, we start with the naive homomorphism.

Definition 13.31 Suppose $f \in \mathcal{F}_\alpha$, and iso minimal nontrivial poly-context P.

P is *weakly hygienic to* f, denoted $P \mid f$, if for all terms $P[\vec{T}]$, $f(P[\vec{T}]) \equiv_\alpha (f(P))[\sum_k f(\vec{T}(k))]$.

P is *strongly hygienic to* f, denoted $P \parallel f$, if $P \mid f$ and $\text{Free}(P) \cup \text{Bind}(P) \parallel f$. ■

Notation $P \parallel f$ is strictly disjoint from $T \parallel f$, since bound variables of P must be $\parallel f$, while bound variables of T may be $\not\parallel f$. The only ambiguous case would be a syntactic expression that is both a term and an iso minimal nontrivial poly-context; and in that case, the expression would have no bound variables, so that the two notations would mean the same thing.

Lemma 13.32 Suppose $f \in \mathcal{F}_\alpha$, iso minimal nontrivial P, and term $P[\vec{T}]$.

If $P \parallel f$, then $P[\vec{T}] \parallel f$ iff $\vec{T} \parallel f$. ■

Proof. Suppose $P \parallel f$. By the definition, $\text{Free}(P) \cap \text{Bind}(P) \parallel f$. Therefore, by Assumption 13.17(c), $\text{Free}(P[\vec{T}]) \parallel f$ iff $\text{Free}(\vec{T}) \parallel f$. Therefore, by Assumption 13.24(a) and Lemma 13.25, $P[\vec{T}] \parallel f$ iff $\vec{T} \parallel f$. ■

Definition 13.33 Suppose $f \in \mathcal{F}_\alpha$, and iso minimal nontrivial poly-contexts P_k.

P_2 is an *α-image through f* of P_1, denoted $P_1 \rightsquigarrow_f P_2$, if there exists P_1' such that $P_1 \sim_\alpha P_1'$, $P_1' \parallel f$, and $f(P_1') \sim_\alpha P_2$. Further, if $P_1 \sim_\alpha P_1'$ is mediated by g_1 and $f(P_1) \sim_\alpha P_2$ is mediated by g_2', then $P_1 \rightsquigarrow_f P_2$ is *mediated by* $g_2' \circ f \circ g_1$. ■

Definition 13.34 Suppose terms T_k and poly-contexts P_k.

$T_1 \Rrightarrow_f T_2$ *satisfies* $P_1 \rightsquigarrow_f P_2$ mediated by g if the following conditions all hold.

- $T_1 \Rrightarrow_f T_2$.
- $P_1 \rightsquigarrow_f P_2$ mediated by g.

- There exist $\vec{T_1}, \vec{T_2}$ such that $T_1 = P_1[\vec{T_1}]$, $T_2 = P_2[\vec{T_2}]$, and for all $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \Rrightarrow_g \vec{T_2}(k)$.

$T_1 \Rrightarrow_f T_2$ satisfies $P_1 \rightsquigarrow_f P_2$ if there exists g such that $T_1 \Rrightarrow_f T_2$ satisfies $P_1 \rightsquigarrow_f P_2$ mediated by g.

$T_1 \equiv_\alpha T_2$ satisfies $P_1 \sim_\alpha P_2$ mediated by g if the following conditions all hold.

- $T_1 \equiv_\alpha T_2$.
- $P_1 \sim_\alpha P_2$ mediated by g.
- There exist $\vec{T_1}, \vec{T_2}$ such that $T_1 = P_1[\vec{T_1}]$, $T_2 = P_2[\vec{T_2}]$, and for all $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \Rrightarrow_g \vec{T_2}(k)$.

$T_1 \equiv_\alpha T_2$ satisfies $P_1 \sim_\alpha P_2$ if there exists g such that $T_1 \equiv_\alpha T_2$ satisfies $P_1 \sim_\alpha P_2$ mediated by g. ■

Each $f \in \mathcal{F}_\alpha$ has an associated set of variables called its *active set*, denoted $\mathrm{Act}(f)$. The active variables are, conceptually, those directly involved by f, and therefore potentially relevant to hygienic use of f. For example, in λ-calculus, $f = \Box[\mathrm{x} \leftarrow \mathrm{y}]$ has $\mathrm{Act}(f) = \{\mathrm{x}, \mathrm{y}\}$. Variables descended from $\mathrm{Act}(f)$, but not themselves active, are modified indirectly by f; for example, $\wr S$-calculus $f = \Box[[\mathrm{i}_1] \leftarrow \mathrm{i}_2]$ has $\mathrm{Act}(f) = \{[\mathrm{i}_1], [\mathrm{i}_2]\}$, so $x = [\mathrm{i}_3\mathrm{i}_2]$ is not active under f, but $x \not\perp f$ (in fact, $x \nparallel f$).

The active set of a renaming f is always the same as the active set of its complement. In simple renamings, such as $f = \Box[\mathrm{x} \leftarrow \mathrm{y}]$, the active variables are either modified by f or modified by its complement; but $\mathrm{Act}(f)$ must also be *closed* under f and its complement (so that if C doesn't bind any active variable of f, and doesn't capture x, then it doesn't capture $f(x)$) — and it is therefore possible that $x \in \mathrm{Act}(f)$ even though $x \perp f$. For example, $f = (\Box[\mathrm{x} \leftarrow \mathrm{y}])[\mathrm{y} \leftarrow \mathrm{x}]$ has $\mathrm{Act}(f) = \{\mathrm{x}, \mathrm{y}\}$, but $\mathrm{x} \perp f$.

Variables active under f but not hygienic to f are called *skew*; the skew set of f is denoted $\mathrm{Skew}(f) = \{x \in \mathrm{Act}(f) \mid x \nparallel f\}$. The skew set of f is typically different from the skew set of its complement, as, in λ-calculus, $f = \Box[\mathrm{x} \leftarrow \mathrm{y}]$ has $\mathrm{Skew}(f) = \{\mathrm{y}\}$, while its complement $f' = \Box[\mathrm{y} \leftarrow \mathrm{x}]$ has $\mathrm{Skew}(f') = \{\mathrm{x}\}$. (Note, however, that symmetry is possible, as with $f = ((\Box[\mathrm{x} \leftarrow \mathrm{z}])[\mathrm{y} \leftarrow \mathrm{x}])[\mathrm{z} \leftarrow \mathrm{y}]$, for which $\mathrm{Act}(f) = \mathrm{Act}(f') = \{\mathrm{x}, \mathrm{y}, \mathrm{z}\}$ and $\mathrm{Skew}(f) = \mathrm{Skew}(f') = \{\mathrm{z}\}$.)

Assumptions 13.35 Suppose $f \in \mathcal{F}_\alpha$, and iso minimal nontrivial poly-context P.

(a) $T \perp f$ iff $\mathrm{Free}(T) \cap \mathrm{Act}(f) \perp f$.

(b) $T \parallel f$ iff $\mathrm{Free}(T) \cap \mathrm{Skew}(f) \parallel f$.

(c) If $\mathrm{Bind}(P) \cap \mathrm{Act}(f) = \{\}$, then $P \mid f$.

(d) If $\mathrm{Free}(P[\vec{T}]) \perp f$ and $\mathrm{Bind}(P) \parallel f$, then $P[\vec{T}] \equiv_\alpha (f(P))[\sum_k f(\vec{T}(k))]$. ■

Since $x \in \mathrm{Skew}(f)$ implies by definition that $x \nparallel f$, Assumption 13.35(b) could have been stated equivalently as

$T \parallel f$ iff $\mathrm{Free}(T) \cap \mathrm{Skew}(f) = \{\}$.
Assumption 13.35(a) cannot be restated this way since, as noted above, $x \in \mathrm{Act}(f)$ does not necessarily imply $x \not\perp f$.

Assumptions 13.36

(a) If $T_1 \equiv_\alpha T_2$, then there exist iso minimal nontrivial P_1, P_2 such that $T_1 \equiv_\alpha T_2$ satisfies $P_1 \sim_\alpha P_2$.

(b) If P_1 is iso minimal nontrivial, $\mathcal{X} \subset \mathit{Vars}$ is finite, and $\mathcal{X} \perp \mathrm{Bind}(P_1)$, then there exist P_2 such that $P_1 \sim_\alpha P_2$ and $\mathrm{Bind}(P_2) \perp \mathcal{X} \cup \mathrm{Bind}(P_1)$.

(c) If $P_1 \sim_\alpha P_2$ are iso minimal nontrivial, and $\mathrm{Bind}(P_1) \perp \mathrm{Bind}(P_2)$, then there exists g mediating $P_1 \sim_\alpha P_2$ such that $\mathrm{Skew}(g) = \mathrm{Bind}(P_2)$ and $\mathrm{Act}(g) = \mathrm{Bind}(P_1) \cup \mathrm{Bind}(P_2)$. ■

Assumption 13.36(a) guarantees that given $T_1 \equiv_\alpha T_2$ can be decomposed into $P_1 \sim_\alpha P_2$ and $\vec{T_1}(k) \Rrightarrow_g \vec{T_2}(k)$. It wouldn't be enough to provide $P_1 \sim_\alpha P_2$ with each P_k satisfied by T_k; we need to know there is a mediating g that actually covers the particular case of $P_1[\vec{T_1}] \equiv_\alpha P_2[\vec{T_2}]$, which is why we provided Definition 13.34.

Assumption 13.36(b) guarantees that it's always possible to α-rename P_1 to avoid capturing any given variables. Assumption 13.36(c) then guarantees that this α-renaming can be performed hygienically on any term $P_1[\vec{T_1}]$ (provided we include in the proscribed $\mathcal{X}$ any stray free variables of $\vec{T_1}$). Together with Assumption 13.35(c), this means that for every $T_1 \parallel f$ there exists a $T_2 \equiv_\alpha T_1$ such that f can be applied to T_2 as a naive homomorphism.

Theorem 13.37

If $T_1 \Rrightarrow_f T_2$, then there exist iso minimal nontrivial P_1, P_2 such that $T_1 \Rrightarrow_f T_2$ satisfies $P_1 \rightsquigarrow_f P_2$. ■

Proof. Suppose $T_1 \Rrightarrow_f T_2$. Let $T_1 = P_1[\vec{T_1}]$, where P_1 is iso minimal nontrivial. By Assumption 13.36(b), let $P_1 \sim_\alpha P_1'$ such that $\mathrm{Bind}(P_1') \perp \mathrm{Bind}(P_1)$ and $\mathrm{Bind}(P_1') \cap (\mathrm{Free}(\vec{T_1}) \cup \mathrm{Act}(f)) = \{\}$ (using $\mathcal{X} = \{x \in (\mathrm{Free}(\vec{T_1}) \cup \mathrm{Act}(f)) \mid x \perp \mathrm{Bind}(P_1)\}$). By Assumption 13.36(c), let g_1 mediate $P_1 \sim_\alpha P_1'$ such that $\mathrm{Skew}(g_1) = \mathrm{Bind}(P_1')$. By Assumption 13.35(b), $\vec{T_1} \parallel g_1$. Let $\vec{T_1'} = \sum_k g_1(\vec{T_1}(k))$; then by Definition 13.29, $P_1[\vec{T_1}] \Rrightarrow_f P_1'[\vec{T_1'}]$. By Assumption 13.35(c), $P_1' \parallel f$. By Assumption 13.24(a), $\mathrm{Free}(\vec{T_1}) - \mathrm{Bind}(P_1) \parallel f$; and by Assumptions 13.19(b) and 13.19(c), $\mathrm{Free}(P_1'[\vec{T_1'}]) \parallel f$; so, $f(P_1'[\vec{T_1'}]) \Rrightarrow_f (f(P_1'))[\sum_k f(\vec{T_1'}(k))]$. By Assumption 13.19(d), $(f(P_1'))[\sum_k f(\vec{T_1'}(k))] \equiv_\alpha T_2$. By Assumption 13.36(a), let P_2 be iso minimal nontrivial such that $(f(P_1'))[\sum_k f(\vec{T_1'}(k))] \equiv_\alpha T_2$ satisfies $f(P_1') \sim_\alpha P_2$. By Definitions 13.33 and 13.34, $T_1 \Rrightarrow_f T_2$ satisfies $P_1 \rightsquigarrow_f P_2$. ■

Theorem 13.38 $\sim_\alpha$ is an equivalence relation. ■

Proof. Trivially, for any iso minimal nontrivial P, $P \sim_\alpha P$ mediated by $f_{\mathbf{id}}$; hence, reflexivity.

Suppose $P_1 \sim_\alpha P_2 \sim_\alpha P_3$. By Assumptions 13.17(a) and 13.17(c), let $T_1 \equiv_\alpha T_3$ such that T_1 satisfies P_1 and T_3 satisfies P_3. By Assumption 13.36(a), let iso minimal nontrivial P_3' be satisfied by T_3 such that $P_1 \sim_\alpha P_3'$. Then the only possible difference between P_3 and P_3' is permutation of the meta-variable indices; but by varying any one of the subterms of T_1 without varying the others (via Assumptions 13.17), and tracing the results through to P_3 and P_3' via their respective $\sim_\alpha$ connections, corresponding meta-variable occurrences must in fact have the same indices. So $P_1 \sim_\alpha P_3$.

Symmetry follows from Lemma 13.21(b) (reversing the direction of the mediating function). ■

Theorem 13.39 Suppose poly-contexts P_k.
$P_1 \sim_\alpha P_2$ iff $P_1 \rightsquigarrow_{f_{\mathbf{id}}} P_2$. ■

Proof. Suppose $P_1 \sim_\alpha P_2$. To satisfy the definition of $\rightsquigarrow_{f_{\mathbf{id}}}$ (Definition 13.33), we want P_1' such that $P_1 \sim_\alpha P_1'$, $P_1' \parallel f_{\mathbf{id}}$, and $f_{\mathbf{id}}(P_1') \sim_\alpha P_2$. Let $P_1' = P_2$. $P_2 \parallel f_{\mathbf{id}}$ (by definition of that relation, Definition 13.31), and $P_2 \sim_\alpha P_2$ (by Theorem 13.38); therefore, $P_1 \rightsquigarrow_{f_{\mathbf{id}}} P_2$.

Suppose $P_1 \rightsquigarrow_{f_{\mathbf{id}}} P_2$. By definition of $\rightsquigarrow_{f_{\mathbf{id}}}$, there exists P_1' such that $P_1 \sim_\alpha P_1'$, $P_1' \parallel f_{\mathbf{id}}$, and $f_{\mathbf{id}}(P_1') \sim_\alpha P_2$. $f_{\mathbf{id}}(P_1') = P_1'$; so $P_1' \sim_\alpha P_2'$, and by transitivity of $\sim_\alpha$ (Theorem 13.38), $P_1 \sim_\alpha P_2$. ■

Lemma 13.40 Suppose iso minimal nontrivial poly-context P, and $f \in \mathcal{F}_\alpha$.
If $\mathrm{Free}(P[\vec{T}]) \cup \mathrm{Bind}(P) \parallel f$, then $P[\vec{T}] \Rrightarrow_f (f(P))[\sum_k f(\vec{T}(k))]$. ■

Proof. Suppose $\mathrm{Free}(P[\vec{T}]) \cup \mathrm{Bind}(P) \parallel f$. Proceed by induction on the number of variables $x \in \mathrm{Free}(P[\vec{T}])$ such that $x \not\perp f$.

Base case: $\mathrm{Free}(P[\vec{T}]) \perp f$. The result is just Assumption 13.35(d).

Inductive step: there are n free variables in $P[\vec{T}]$ that are $\not\perp f$, and the result holds for all terms and renaming functions such that strictly fewer than n free variables of the term are $\not\perp$ to the renaming function.

Let $x_1 \in \mathrm{Free}(P[\vec{T}])$ and $x_1 \not\perp f$, such that $(\mathrm{Above}(x_1) - x_1) \perp f$ (by Lemma 13.18). (Incidentally, $x_1 \in \mathrm{Act}(f)$, by Assumptions 13.35(a) and 13.17(a).)

We will show $x_1 \Rrightarrow_{f_1} x_2 \Rrightarrow_{f_2} f(x_1)$ and $f = f_2 \circ g \circ f_1$, such that g is subject to the inductive hypothesis, and enough hygiene is maintained to complete the proof.

By Assumption 13.17(b), let P_1 be a minimal nontrivial poly-context such that $\mathrm{Bind}(P_1) = x_1$ and $\mathrm{Free}(P_1) = \mathrm{Above}(x_1) - x_1$; and by Assumption 13.17(a), let

$\vec{T_1}$ have the same arity as P_1 and $\text{Free}(\vec{T_1}) = \text{Above}(x_1)$. By Assumption 13.17(c), $P_1[\vec{T_1}] \in \mathit{Terms}$. By Assumption 13.35(d), $P_1[\vec{T_1}] \equiv_\alpha (f(P_1))[\sum_k f(\vec{T_1}(k))]$. By Assumption 13.36(a) (and varying subterms via Assumptions 13.17), $P_1 \sim_\alpha f(P_1)$. Let $P_3 = f(P_1)$ and $x_3 = \text{Bind}(P_3) = f(x_1)$.

We want $P_2 \sim_\alpha P_1$ such that $\text{Bind}(P_2)$ doesn't occur at all in $P[\vec{T}]$, nor in $f(P)$, nor in $\sum_k f(\vec{T}(k))$. Let $\mathcal{X}$ be the set of all of these variables that are $\perp x_1$. By Assumption 13.36(a), let $P_1 \sim_\alpha P_2$ such that $\text{Bind}(P_2) \perp \mathcal{X} \cup x_1$. Let $x_2 = \text{Bind}(P_2)$; then x_2 doesn't occur at all in $P[\vec{T}]$, nor in $f(P)$, nor in $\sum_k f(\vec{T}(k))$. By Theorem 13.38, $P_2 \sim_\alpha P_3$. By Lemma 13.30, $x_3 \perp x_2$. By Assumption 13.36(c), let f_1 mediate $P_1 \sim_\alpha P_2$, and f_2 mediate $P_2 \sim_\alpha P_3$, such that $\text{Act}(f_1) = \{x_1, x_2\}$, $\text{Act}(f_2) = \{x_2, x_3\}$, $\text{Skew}(f_1) = x_2$, and $\text{Skew}(f_2) = x_3$. Let f_1' be the complement of f_1, and f_2' of f_2. $\vec{T} \parallel f_1$, and $(\sum_k f(\vec{T}(k))) \parallel f_2'$.

Let $g = f_2' \circ f \circ f_1'$. By Assumption 13.35(c) and Definition 13.31, $P \parallel f_1$ and $f(P) \parallel f_2'$; therefore, also by Definition 13.31, $f_1(P[\vec{T}]) \equiv_\alpha (f_1(P))[\sum_k f_1(\vec{T}(k))]$ and $f_2'((f(P))[\sum_k f(\vec{T}(k))]) \equiv_\alpha (f_2'(f(P)))[\sum_k f_2'(f(\vec{T}(k)))]$. By definition, $x_2 \perp g$. For x free in $f_1(P[\vec{T}])$ but not descended from x_2, $x \perp g$ iff $x \perp f$. Therefore, the number of free variables in $f_1(P[\vec{T}])$ that are $\not\perp g$ is strictly less than n; so by inductive hypothesis, $f_1(P[\vec{T}]) \Rrightarrow_g (g(f_1(P)))[\sum_k g(f_1(\vec{T}(k)))] \equiv_\alpha (f_2'(f(P)))[\sum_k f_2'(f(\vec{T}(k)))]$. Therefore, Q.E.D. ■

Because α-renaming can't change the structure of a term (Assumption 13.19(a)), induction using $\rightsquigarrow$ is always straightforward. For convenience we will now generalize the terminology of *α-form* and *α-image* from minimal nontrivial poly-contexts to arbitrary poly-contexts. However, because the details of the situation described by this terminology can actually be quite complicated, we prefer not to hide these details in proofs (whose purpose is, after all, to convince the reader that the conclusion really follows from the premises). Therefore, we do not extend this generalization to the symbolic notations, $\sim_\alpha$ and $\rightsquigarrow$; and when treating these compound relationships in proofs, we view each relationship as a projection between prime factorizations via $\rightsquigarrow$. If we were to attempt a generalization of $\sim_\alpha$ and $\rightsquigarrow_f$, as such, to compound poly-contexts, it would have to abandon the idea of a single mediating function. In a compound P, each prime factor of P is transformed uniformly, but different prime factors may be transformed differently from each other. For example, in λ-calculus, $P_1 = ((\lambda \text{x}.\Box_1)\,((\lambda \text{x}.\Box_2)\,\Box_3))$ could be α-renamed to $P_2 = ((\lambda \text{y}.\Box_1)\,((\lambda \text{z}.\Box_2)\,\Box_3))$; the two occurrences of x in P_1 are differently transformed, and each subterm $\Box_k$ is differently transformed. So one would have to view $P_1 \rightsquigarrow_{f_{\mathbf{id}}} P_2$ as being mediated by a *vector* of renaming functions, $\vec{f} = \langle \Box[\text{x} \leftarrow \text{y}], \Box[\text{x} \leftarrow \text{z}], f_{\mathbf{id}} \rangle$.

Definition 13.41 Suppose $f \in \mathcal{F}_\alpha$, and poly-contexts P_k.

P_2 is an *α-image of P_1 through f* if for all $P_1[\vec{T}] \in \mathit{Terms}$, if $\mathrm{Free}(\vec{T}) = \{\}$ then $P_1[\vec{T}] \Rrightarrow_f P_2[\vec{T}]$.

P_2 is a *α-form of P_1* if P_2 is an α-image of P_1 through $f_{\mathbf{id}}$. ■

If P_2 is an α-image of P_1, then relation $P_1[\vec{T_1}] \Rrightarrow_f P_2[\vec{T_2}]$ can always be decomposed (by recursive application of Theorem 13.37) into a structure-preserving isomorphic projection of a prime factorization of P_1, via $\rightsquigarrow$ relations between prime factors, onto a prime factorization of P_2. Then the P_k differ only by the names of variable occurrences (this follows for the syntax of the P_k by Assumption 13.19(a), and for the meta-variable indices by varying the $\vec{T}$ independently).

Theorem 13.42 Suppose $f \in \mathcal{F}_\alpha$.

If $C[T] \in \mathit{Terms}$ and $T \Rrightarrow_f T'$, then there exists an α-image $C'[T']$ of $C[T]$. ■

Proof. Suppose $C[T] \in \mathit{Terms}$ and $T \Rrightarrow_f T'$. If the result holds for singular C, the general result follows by induction; so suppose C is singular. Without loss of generality, let $C = P[\Box, \vec{T}]$ where P is iso minimal nontrivial.

Let n be the number of variables $x \in \mathrm{Free}(C[T]) \cup \mathrm{Bind}(P)$ such that $x \not\parallel f$. If $n = 0$, the result follows from Lemma 13.40. Suppose $n \geq 1$, and the result holds for all strictly smaller n. By Lemma 13.23, let $x \in \mathrm{Free}(C[T]) \cup \mathrm{Bind}(P)$ such that $x \not\parallel f$ and $(\mathrm{Above}(x) - x) \parallel f$. Then $x \notin \mathrm{Free}(T)$ (by Assumption 13.35(b)). By Assumptions 13.17 and 13.36, let $g \in \mathcal{F}_\alpha$ such that $\mathrm{Free}(C[T]) \cup \mathrm{Bind}(C) \parallel g$, $g(x) \parallel f$, and $T \perp g$. By Lemma 13.40, $C[T] \Rrightarrow_g (g(P))[T, \sum_k g(\vec{T}(k))]$; and by inductive hypothesis, $(g(P))[T, \sum_k g(\vec{T}(k))]$ has an α-image $C'[T']$. ■

Definition 13.43 Suppose poly-context P, set of poly-contexts $\mathcal{P}$, and set of variables $\mathcal{X}$.

P is *in general position*[8] if for all poly-contexts $P', \vec{P'}$,
if $P'[\vec{P'}]$ is satisfied by P, and x is bound by any $\vec{P'}(k)$,
then x does not occur in P'.

P is *in $\mathcal{X}$-general position* if P is in general position and for all poly-contexts P' satisfied by P, $\mathrm{Bind}(P') \cap \mathcal{X} = \{\}$.

P is *in $(\mathcal{X} \cup \mathcal{P})$-general position* if P is in $\mathcal{X} \cup \mathrm{Free}(\mathcal{P})$-general position. ■

As usual, P may be a term, while $\mathcal{P}$ may include terms. Every minimal nontrivial poly-context is in general position. Note that general position is not compositional,

[8]The terminology *general position* is borrowed from algebraic geometry, where it refers to an arrangement of particular things that is an instance of the general case, free from pathologies such as three points that happen to be collinear, four points that happen to be coplanar (when considering more than 2 dimensions), two lines that happen to be parallel, and so on.

i.e., a composition of poly-contexts might not be in general position even though all the parts are in general position; e.g., $C_1[C_2]$ might not be in general position even though C_1 is in general position and C_2 is in general position.

Theorem 13.44 Suppose poly-context P, and set of variables $\mathcal{X}$.

If $\mathcal{X}$ is finite, then there exists an α-form P' of P such that P' is in $\mathcal{X}$-general position. ■

Proof. Descend recursively through the structure of P, α-renaming each prime factor to avoid binding any proscribed variables, as allowed by Assumptions 13.36(b) and 13.36(a) and Theorem 13.37. ■

Theorem 13.45

If $C[T] \in \mathit{Terms}$, and T is in general position, then there exists an α-image $C'[T]$ of $C[T]$ such that $C'[T]$ is in general position. ■

Proof. Suppose $C[T] \in \mathit{Terms}$. If the result holds for singular C, the general result follows by induction; so suppose C is singular. Without loss of generality, let $C = P[\Box, \vec{T}]$ where P is iso minimal nontrivial. If any of the $\vec{T}$ are not in P-general position, they can be replaced by $\equiv_\alpha$ terms that are (by Theorem 13.44); suppose they are already in P-general position. The only possible remaining violations of general position are variables occurring in P that occur bound in T. These variables do not occur free in T, since T is in general position; so they can be diverted, one by one, as in the proof of Theorem 13.42 (where the variables to be diverted were those $\not\parallel f$, which also could not occur free in T). ■

Theorem 13.46 Suppose poly-context P.

If P is in general position, then P is in P-general position. ■

That is, if P is in general position, and x occurs free in P, then x does not occur bound in P.

Proof. Suppose P is in general position, P satisfies P', and x is bound by P'. Then there must be some prime factor of P that binds x, call it P''. Since P is in general position, x cannot occur anywhere in P *outside* the scope of P''. Since P'' binds x, any occurrence of x in a subterm within the scope of P'' is bound in P, not free in P. Finally, x cannot occur free within P'' itself, because a minimal nontrivial poly-context doesn't have free occurrences of variables that is binds (assumption stated in prose before Lemma 13.16). Therefore, x does not occur free in P. ■

Lemma 13.47 Suppose $f \in \mathcal{F}_\alpha$, and iso minimal nontrivial poly-contexts P_k.

If $P_1 \rightsquigarrow_f P_2$ and $P_1[\vec{T_1}] \in \mathit{Terms}$, then there exists $g \in \mathcal{F}_\alpha$ mediating $P_1 \rightsquigarrow_f P_2$ such that for each $x \in \mathrm{Free}(\vec{T_1})$, either $x \parallel g,\ x \not\parallel f$, or $f(x) \in \mathrm{Bind}(P_2)$. ■

Proof. Suppose $P_1 \rightsquigarrow_f P_2$ and $P_1[\vec{T_1}] \in \mathit{Terms}$.

Let $\vec{T_1'}$ have the same arity but $\mathrm{Free}(\vec{T_1'}) = \mathrm{Free}(\vec{T_1}) - \mathrm{Below}(\mathrm{Skew}(f))$ (by Assumption 13.17(a)); then $P_1[\vec{T_1'}] \in \mathit{Terms}$ (by Assumption 13.17(c)). For all $x \in \mathrm{Free}(\vec{T_1}) - \mathrm{Free}(\vec{T_1'}),\ x \not\parallel f$ (by Lemma 13.23). For all $x \in \vec{T_1'},\ x \parallel f$ (by Assumptions 13.17(a) and 13.35(b) and Lemma 13.25). Let $\mathcal{X}'$ be the co-image of $\mathrm{Bind}(P_2)$ under $\Rrightarrow_f$; that is, $\mathcal{X}' = \{x \mid (x \parallel f) \wedge (f(x) \in \mathrm{Bind}(P_2))\}$. $\mathcal{X}'$ is finite (since f must be one-to-one on variables $x \parallel f$). Let $\mathcal{X}$ be the set of all $x \in \mathrm{Free}(\vec{T_1'}) \cup \mathcal{X}' \cup \mathrm{Act}(f)$ such that $x \perp \mathrm{Bind}(P_1)$. By Assumption 13.36(b), let $P_1 \sim_\alpha P_1'$ such that $\mathrm{Bind}(P_1') \perp \mathcal{X} \cup \mathrm{Bind}(P_1')$; hence, $\mathrm{Bind}(P_1') \cap (\mathrm{Free}(\vec{T_1'}) \cup \mathcal{X}' \cup \mathrm{Act}(f)) = \{\}$.

By Assumption 13.35(c), $P_1' \mid f$. Since $P_1 \rightsquigarrow_f P_2$, $\mathrm{Free}(P_1) \parallel f$; and therefore, since $\mathrm{Bind}(P_1') \cap \mathrm{Act}(f) = \{\}$, $\mathrm{Bind}(P_1') \parallel f$ (by Assumptions 13.17(a) and 13.35(b)). So $P_1' \parallel f$. By Lemma 13.21(c) (and, primarily, Assumption 13.36(a)), $f(P_1') \sim_\alpha P_2$. Because $\mathrm{Bind}(P_1') \cap \mathcal{X}' = \{\}$, $\mathrm{Bind}(f(P_1')) \cap \mathrm{Bind}(P_2) = \{\}$; hence, by Lemma 13.30, $\mathrm{Bind}(f(P_1')) \perp \mathrm{Bind}(P_2)$.

By Assumption 13.36(c), let g_1 mediate $P_1 \sim_\alpha P_1'$ such that $\mathrm{Skew}(g_1) = \mathrm{Bind}(P_1')$ and $\mathrm{Act}(g_1) = \mathrm{Bind}(P_1) \cup \mathrm{Bind}(P_1')$, and let g_2 mediate $f(P_1') \sim_\alpha P_2$ such that $\mathrm{Skew}(g_2) = \mathrm{Bind}(P_2)$ and $\mathrm{Act}(g_2) = \mathrm{Bind}(f(P_1')) \cup \mathrm{Bind}(P_2)$. Let $g = g_2 \circ f \circ g_1$; then g mediates $P_1 \rightsquigarrow_f P_2$.

Suppose $x \in \mathrm{Free}(\vec{T_1}),\ x \parallel f$, and $f(x) \notin \mathrm{Bind}(P_2)$. x either is or is not bound by P_1. If x is bound by P_1, then $x \parallel g_1,\ g_1(x) \parallel f$, and $f(g_1(x)) \parallel g_2$, so $x \parallel g$. If x is not bound by P_1, then $x \perp g_1$, we're already supposing $x \parallel f$, and since $f(g_1(x)) = f(x)$ isn't bound by P_2, it is $\parallel g_2$, so again $x \parallel g$. ■

Theorem 13.48 Suppose poly-contexts P_k.

Suppose P_2 is an α-image of P_1, P_1 is iso, and P_2 is in general position. If T_1 satisfies P_1, then there exists an α-image of T_1 that satisfies P_2. ■

Note that the theorem doesn't mention renaming functions. If P_2 is an α-image of P_1 *through* f, etc., and T_1 satisfies P_1, then there must exist some g such that T_1 has an α-image through g satisfying P_2; but in general g might not $= f$.

The iso precondition guarantees that each meta-variable occurrence can be considered in its own context, without any required correlations with other meta-variable occurrences.

Proof. Suppose P_2 is an α-image of P_1, P_1 is iso, P_2 is in general position, and $P_1[\vec{T_1}] \in \mathit{Terms}$. If P_1 and P_2 are trivial, the result follows immediately; so suppose

they are nontrivial. Let $f \in \mathcal{F}_\alpha$ such that $P_1[\vec{T_1'}] \Rrightarrow_f P_2[\vec{T_2'}]$. Then $P_1[\vec{T_1'}] \Rrightarrow_f P_2[\vec{T_2'}]$ projects a prime factorization of P_1 via $\rightsquigarrow$ onto a prime factorization of P_2. This projection ultimately transforms each $\vec{T_1'}(k)$ through some renaming $\vec{f}(k)$ that mediates the projection via $\rightsquigarrow$ of the closest prime factor within which $\Box_k$ occurs in P_1 (this prime factor being a leaf in the prime factorization of P_1). We would like to map $P_1[\vec{T_1}]$ through the same projection; and this will be possible iff for each k, $\text{Free}(\vec{T_1}(k)) \parallel \vec{f}(k)$, which is to say, $\text{Free}(\vec{T_1}(k)) \cap \text{Skew}(\vec{f}(k)) = \{\}$.

Consider any $x \in \text{Free}(\vec{T_1}(k))$. The occurrence of $\Box_k$ in P_1 falls within the scope of a column of prime factors. Consider the prime factors in this column, in ascending order (that is, starting from the factor at the bottom of the column, which is within the scopes of all the others, and proceeding upward to the factor at the top of column, which is not within the scopes of any of the rest of the column, and which is satisfied by P_1). At each step, call the current factor P_1'. P_1' is projected to corresponding prime factor P_2' of P_2 by $\rightsquigarrow_{f''}$ mediated by f'. When P_1' is at the bottom of the column, $f' = \vec{f}(k)$, the renaming applied to $\vec{T_1'}(k)$; when P_1' is at the top of the column, $f'' = f$, the renaming applied to $P_1[\vec{T_1'}]$ as a whole. Assume without loss of generality, by Lemma 13.47, that either $x \parallel f'$, or $x \nparallel f''$, or $f''(x) \in \text{Bind}(P_2')$.

If $x \parallel f'$, the projection treats free occurrences of x hygienically; it doesn't matter then how x is handled by factors further up the column, since that handling is known to result in hygiene at f'. Suppose $x \nparallel f'$. Then x does not occur in P_1', because if it did that would imply $x \parallel f'$ (by definition of mediating function of $\rightsquigarrow$; cf. Definitions 13.29, 13.33, and 13.31).

Suppose $x \parallel f''$; then, by our assumption without loss of generality, $f''(x) \in \text{Bind}(P_2')$. But then, by Lemma 13.46, $f''(x) \notin \text{Free}(P_2)$. Therefore, $x \parallel f''$ is not the result of a binding further up the column, and the behavior of f'' on x is inherited from f, i.e., $x \parallel f$ and $f(x) = f''(x)$. So $x \notin \text{Free}(P_1)$. That being the case, we can devise a renaming function (via Assumptions 13.17(b), 13.36(b), and 13.36(c)) that will transform x to some other variable that doesn't occur at all in $P_1[\vec{T_1}]$ and isn't the co-image of any variable that occurs in P_2; applying this devised renaming function to $P_1[\vec{T_1}]$ eliminates x from $\vec{T_k}(k)$, and any internal α-renamings involved remain reversible — so this case can always be eliminated, and we assume $x \nparallel f''$. Proceeding up the column, at each level we have $x \nparallel f'$ and either $x \nparallel f''$ or $f''(x) \in \text{Bind}(P_2')$, and we can eliminate the latter in the same way as above, until the only case unaccounted for is that $x \nparallel f$, at the top of the column. But since $x \nparallel f$ and $P_1 \rightsquigarrow_f P_2$, it must be that $x \notin \text{Free}(P_1)$, and we can again eliminate x by deriving an applying a renaming function to the term as a whole. ■

13.1.3 Substitutive functions

The class of term-transformations that facilitate the reduction relation $\longrightarrow^\bullet$, we call

substitutive functions. These are the substitutions invoked for substantive transformations by reduction rule schemata (i.e., they are used to make things happen, in contrast to renaming functions that are used to *prevent* unintended things from happening).

Both classes of functions, renaming and substitutive, behave as naive homomorphisms on a term *if* the term is first suitably prepared by means of α-renaming (to avoid bad hygiene). Klop assumed that this α-renaming would always have been performed before substitution, which was credible because he was using a simple and already well-understood class of variables, so that α-renaming itself could be taken as given. For our treatment of renaming functions in §13.1.2, we used an explicit, fine-grained approach, steering warily well clear of circularity (as the functions being treated there were themselves responsible for the α-renaming) by orchestrating hygienic renaming behavior one syntactic induction step at a time, and by formally deriving our primary notion of hygiene from reversibility (Definitions 13.22 and 13.20); but now that we have the machinery of renaming to draw on, we will invoke it wholesale (though still explicitly, in difference from Klop) to provide suitable hygienic α-renaming for substitutive functions — and, notwithstanding hygiene provisions, our substitutive functions will not be reversible in general.[9]

A substitutive function takes two or more inputs: in order, a variable (conceptually, to be substituted for), a term (to be substituted into), and a monic poly-context and some fixed number of additional terms (to be substituted). The output is a term. The third and later inputs are (up to a point) treated as a unit, $P[\Box, \vec{T}]$, which is either a context or a term depending on whether or not $\Box_1$ occurs in P. The third input P might be constrained idiosyncratically by the function, and might not be copied verbatim during substitution; it serves as a constraint on the subterms and on free/bound variables. Later inputs $\vec{T}$ are copied verbatim when substituted, and are constrained only by what subterms are syntactically admissible in P (cf. Lemma 13.16). In the traditional λ-calculus β-rule, $P = (\Box_1\ \Box_2)$ supports traditional call-by-name substitution $\Box[x_p \leftarrow T]$, where P itself entirely disappears during substitution (and its disappearance creates no danger of upward variable capture since P binds no variables). More complex P, possibly including binding branches, can limit the substituted structure to a proper subset of terms (as in $\Box[x_g \leftarrow V]$, where binding branches of P are part of the copied structure V, and copying them prevents upward capture of variables in non-P subterms of V).

Given some explicit preconditions which, in the event, will have been guaranteed by preparatory α-renaming, substitutive $f(x, T, P, \vec{T})$ naively descends through the structure of T, determining where copies of the $\vec{T}$ are to be placed. That is, it maps T homomorphically to a poly-context $P' = (f(x, P))T$, and then replaces the meta-variables of P' by $\vec{T}$: $f(x, T, P, \vec{T}) = ((f(x, P))T)[\vec{T}]$.

Associated with each substitutive f is also a variable transformation, called f_{v},

[9]This is a natural consequence of the fact that $\equiv_\alpha$ is symmetric, while $\longrightarrow^\bullet$ generally is not.

which acts on variables in the second input (the term substituted into) to produce their analogs in the output. This transformation is used to describe hygiene conditions, notably including hygiene when the variable substituted for (first input to f) is eliminated from the term substituted into (second input to f) — as happens in λ-calculus substitutions, get resolutions ($\Box[x_g \leftarrow V]$, $\Box[x_g \not\leftarrow]$) and state deletions ($\Box[x_s \not\leftarrow]$).

Definition 13.49 Suppose $n \geq 0$; f is a partial function that maps a variable, a term, a monic poly-context of arity $n+1$, and a term vector of arity n to a term, f: $\mathit{Vars} \times \mathit{Terms} \times \mathit{PolyContexts}_{n+1} \times \mathit{Terms}^n \xrightarrow{p} \mathit{Terms}$; and f_{v} is a partial function that maps two variables to a variable, f_{v}: $\mathit{Vars} \times \mathit{Vars} \xrightarrow{p} \mathit{Vars}$.

f is *substitutive with variable transformation* f_{v} if all of the following conditions hold.

(a) If any of the following conditions holds, then $f(x, T, P, \vec{T})$ is undefined.

- $P[\Box, \vec{T}]$ is not a context.
- x occurs in $P[\Box, \vec{T}]$.
- Any of $\mathrm{Bind}(P[\Box, \vec{T}])$ occur in T.
- T is not in $(\mathrm{Above}(x) \cup P[\Box, \vec{T}])$-general position.
- x_1 and x_2 occur in T, $x_1 \neq x_2$, and $f_{\mathrm{v}}(x, x_1) = f_{\mathrm{v}}(x, x_2)$.

If $x' \neq x$, then $f_{\mathrm{v}}(x, x')$ is defined.

(b) If $f(x, T, P, \vec{T})$ is defined, $x \notin \mathrm{Free}(T')$, $ar(\vec{T'}) = n$, and $f(x, T', P, \vec{T'})$ is not prohibited by Condition (a), then $f(x, T', P, \vec{T'}) \equiv_\alpha T'$.

If $x' \not\sqsubseteq x$, then $f_{\mathrm{v}}(x, x') = x'$.

(c) If $f(x, T_1, P_1, \vec{T_1})$ and $f(x, T_2, P_2, \vec{T_2})$ are defined and $P_1[T_1, \vec{T_1}] \equiv_\alpha P_2[T_2, \vec{T_2}]$, then $f(x, T_1, P_1, \vec{T_1}) \equiv_\alpha f(x, T_2, P_2, \vec{T_2})$.

If $x_1 \sqsubseteq x_2$, and $f_{\mathrm{v}}(x, x_1)$ and $f_{\mathrm{v}}(x, x_2)$ are defined, then $f_{\mathrm{v}}(x, x_1) \sqsubseteq f_{\mathrm{v}}(x, x_2)$.
If $f_{\mathrm{v}}(x, x_1) \sqsubseteq f_{\mathrm{v}}(x, x_2)$, and $x \notin f_{\mathrm{v}}(x, x_1) \cup f_{\mathrm{v}}(x, x_2)$, then $x_1 \sqsubseteq x_2$.

(d) For all $g \in \mathcal{F}_\alpha$, if $f(x_1, T_1, P_1, \vec{T_1})$ is defined, $x_1, T_1, \vec{T_1} \parallel g$, and every prime factor of P_1 is $\parallel g$, then there exist $x_2, T_2, P_2, \vec{T_2}$ such that $x_1 \Rrightarrow_g x_2$; $T_1 \Rrightarrow_g T_2$; for all $k \leq n$, $\vec{T_1}(k) \Rrightarrow_g \vec{T_2}(k)$; P_2 results from applying g to all the prime factors of P; and $f(x_1, T_1, P_1, \vec{T_1}) \Rrightarrow_g f(x_2, T_2, P_2, \vec{T_2})$.

For all $g \in \mathcal{F}_\alpha$, if $f_{\mathrm{v}}(x, x')$ is defined and $x, x' \parallel g$, then $f_{\mathrm{v}}(x, x') \Rrightarrow_g f_{\mathrm{v}}(g(x), g(x'))$.

(e) If $f(x, T, P, \vec{T})$ is defined, then there exists poly-context P_1 such that for all $\vec{T'}$, if $f(x, T, P, \vec{T'})$ is defined then $f(x, T, P, \vec{T'}) = P_1[\vec{T'}]$. For given x T and P, this P_1 if it exists may be denoted $f(x, T, P)$; $f(x, T, P)$ is undefined iff no $f(x, T, P, \vec{T})$ is defined.

(f) If $f(x, P_1[\vec{T_1}], P)$ is defined, then there exists poly-context P_1' such that for all $\vec{T_2}$, if $f(x, P_1[\vec{T_2}], P)$ is defined then

$f(x, P_1[\vec{T_2}], P) = P_1'[\vec{\Box_n}, \sum_{k=1}^{ar(P_1)} f(x, \vec{T_2}(k), P)]$.

For given x P_1 and P, this P_1' if it exists may be denoted $f(x, P_1, P)$; $f(x, P_1, P)$ is undefined iff no $f(x, P_1[\vec{T_1}], P)$ is defined. For given x, the partial function from poly-contexts P_1 to poly-contexts $f(x, P_1, P)$ may be denoted $f(x, P)$.

(g) If $f(x, P_1, P)$ is defined, and $\Box_k$ occurs j times in P_1, then $\Box_{k+n}$ occurs $\leq j$ times in $f(x, P_1, P)$.

(h) If $f(x, T, P)$ is defined, $(f(x, T, P))[\vec{T}] \in \mathit{Terms}$, and $f(x, T, P, \vec{T})$ is not prohibited by Condition (a), then $f(x, T, P, \vec{T})$ is defined.

(i) If $x_1' \in \text{Free}((f(x, P))P_1)$, then
either $x \in \text{Free}(P_1)$ and $x_1' \in \text{Free}(P)$,
or there exists $x_1 \in \text{Free}(P_1)$ with $x_1' = f_{\text{v}}(x, x_1)$.

If $x_1' \in \text{Bind}((f(x, P))P_1 \text{ at } \Box_k)$ for $1 \leq k \leq n$, then
either $x_1' \in \text{Bind}(P \text{ at } \Box_{k+1})$
or there exists $x_1 \in \text{Bind}(P_1)$ with $x_1' = f_{\text{v}}(x, x_1)$.

If $x_1' \in \text{Bind}((f(x, P))P_1 \text{ at } \Box_{k+n})$ for $1 \leq k \leq ar(P_1)$, then
there exists $x_1 \in \text{Bind}(P_1 \text{ at } \Box_k)$ with $x_1' = f_{\text{v}}(x, x_1)$.

For $1 \leq k \leq n$, and C satisfying $((f(x, P))P_1)[\vec{T}\backslash^{k}_{\Box_1}]$,
$\text{Bind}(P \text{ at } \Box_{k+1}) \subseteq \text{Bind}(C)$.

For $1 \leq k \leq ar(P_1)$, and C satisfying $((f(x, P))P_1)[\vec{T}\backslash^{k+n}_{\Box_1}]$,
$\{f_{\text{v}}(x, x_1) \mid x_1 \in \text{Bind}(P_1 \text{ at } \Box_k)\} \subseteq \text{Bind}(C)$.

(j) For every x, there exists a P such that $f(x, P)$ is defined.

Then f is substitutive. The family of substitutive functions is denoted $\mathcal{F}_\beta$.

Substitutive f is *trivial* if for all $f(x, P)$ and T_1 such that Condition 13.49(a) does not prohibit $(f(x, P))T_1$, $(f(x, P))T_1 \equiv_\alpha T_1$. ■

Condition 13.49(a) forbids f from being defined in a broad set of cases where naive substitution might cause variable capture; such cases will be reabsorbed into the treatment later, mainly by positing invariance under unspecified preliminary α-renaming. Conditions 13.49(c) and 13.49(d) guarantee that the behavior of f will be, for its part, invariant under different choices of preliminary α-renaming. Conditions 13.49(e) and 13.49(f) break down f into its two naive phases — naive homomorphism $(f(x, P))T$, and naive polynomial substitution $((f(x, P))T)[\vec{T}]$. Condition 13.49(i) prohibits $(f(x, P))T$ from introducing new free variables into T, or exposing previously bound variables of T or $\vec{T}$ to possible capture by surrounding contexts.

The definition does not require the variable transformation to be unique; a given f might be substitutive with variable transformation h_{v} for multiple distinct variable

transformations h_{v}. This is harmless, because we only need to know that such a transformation exists. However, the behavior of every substitutive function considered here will, in fact, uniquely determine its variable transformation (via Condition 13.49(i)).

It may happen, and in the impure λ-calculi it does happen, that a substitutive function f is meant to substitute for one of several kinds of variables — substitution $\Box[x \leftarrow T]$ is only performed with a partial-evaluation variable x, $\Box[x \leftarrow C]$ only with a control variable x, etc. Restricting the domain of the first input of f would needlessly further complicate the definition. Such specialized substitutions f are reconcilable with the definition by giving f trivial behavior when its first input does not belong to the intended class — thus, $T_1[x_c \leftarrow T_2] \equiv_\alpha T_1$, $T[x_p \leftarrow C] \equiv_\alpha T$, etc. The well-behavedness requirements of the definition allow this tactic, because renaming a variable of one class cannot produce a variable of a different class, so that f can behave in entirely different ways on different classes of variables while remaining invariant under renaming.

Definition 13.50 Suppose substitutive f.

Poly-context P_1 is *hygienic at* (x, P) *to* f, denoted $P_1 \mid f(x, P)$, if $(f(x, P))P_1$ is defined.

Poly-context P_1 is *hygienic at* x *to* f, denoted $P_1 \mid f(x)$, if there exists P such that $P_1 \mid f(x, P)$. ■

In characterizing the behavior of an arbitrary poly-context P_1 under a substitutive function f, we will want the notion of an $f(x)$-*hygienic image* of P_1, which is any poly-context that P_1 may be transformed into when a term containing P_1 is α-renamed for naive treatment under $f(x)$. Precisely,

Definition 13.51 Suppose substitutive f, and poly-contexts P_k.

P_2 is an $f(x)$-*hygienic form of* P_1 if $P_2 \mid f(x)$ and $P_1 \sim_\alpha P_2$.

P_2 is an $f(x)$-hygienic *image* of P_1 if there exist C_1, C_2 such that C_2 is an $f(x)$-hygienic form of C_1, and $C_2[P_2]$ is an $f(x)$-hygienic form of $C_1[P_1]$.

P_2 is an $f(x)$-hygienic image of P_1 *under* P_3 if there exist $C_1, C_2, C_3, C_3', \vec{T}_3$, and k such that $C_1 = C_3[P_3[\vec{T}_3 \backslash^k_{C_3'}]]$, C_2 is an $f(x)$-hygienic form of C_1, and $C_2[P_2]$ is an $f(x)$-hygienic form of $C_1[P_1]$. Further, P_2 is then an $f(x)$-hygienic image of P_1 under P_3 *at* k.

P_2 is an $f(x, P)$-*hygienic form of* P_1 if $P_2 \mid f(x, P)$ and $P_1 \sim_\alpha P_2$; and so on. ■

Every $f(\cdots)$-hygienic form of a poly-context P is an $f(\cdots)$-hygienic image of P (using $C = C' = \Box$ in the definition).

Lemma 13.52 Suppose f is substitutive, and $k \in \mathbb{N}$.

For given $f(x, P)$, there exists $\vec{T}$ with arity k such that $\mathrm{Free}(\vec{T}) = \{\}$, each $(f(x, P))\vec{T}(j)$ exists, and no two terms in $\vec{T}$ have the same syntactic depth.

There exists $\vec{T}$ with arity k such that $\mathrm{Free}(\vec{T}) = \{\}$, $\vec{T} \mid f(x)$, and no two terms in $\vec{T}$ have the same syntactic depth. ■

The significance of different syntactic depths is that when any two terms have different shapes, no combination of renamings can possibly make them $\equiv_\alpha$ to each other.

Proof. It suffices to show the first half of the lemma, as the second half follows form it by Condition 13.49(j).

Suppose $f(x, P)$ exists. Let $x' \perp \text{Above}(x)$, and $\perp$ all variables occurring in P (by Lemma 13.18 and Assumptions 13.17(b) and 13.36(b)). Further, since $\sqsubseteq$ is a partial order and has (again by Lemma 13.18) no infinite ascending chains, assume without loss of generality that x' is an orphan, i.e., $\text{Above}(x') = x'$. By Assumption 13.17(b) and Condition 13.49(b), let P' be minimal nontrivial such that $\text{Bind}(P') = x'$, $\text{Free}(P') = \{\}$, and there exist $T', \vec{T'}$ such that T' satisfies P' and $f(x, T', P, \vec{T'})$ is defined. Then $\text{Bind}(P') \perp x$, and $\text{Free}(P') = \{\}$.

By Assumption 13.17(a) and Condition 13.49(b), let $\text{Free}(T) = \{\}$ such that for some $\vec{T'}$, $f(x, T, P, \vec{T'})$ exists. By Assumption 13.36(b), let poly-contexts $\vec{P'}$ be n α-forms of P' that all involve variables $\perp$ to each other and to x and all variables in P; by Condition 13.49(d), each $f(x, \vec{P'}(k), P)$ is defined. By Assumption 13.17(c), let

$$\vec{T} \;=\; \sum_{j=1}^{k} \begin{cases} T & \text{if } j = 1 \\ (\vec{P'}(j))[\sum_{i=1}^{ar(P')} \vec{T}(j-1)] & \text{otherwise}\,. \end{cases} \blacksquare$$

Lemma 13.53 Suppose substitutive f, and poly-contexts P_k.

If $P_1 \sim_\alpha P_2$ are both $\mid f(x, P)$, then $f(x, P_1, P) \sim_\alpha f(x, P_2, P)$.

If $P_1 \sim_\alpha P_2$ are both $\mid f(x)$, then there exists P such that both are $\mid f(x, P)$.

If $P_1 \sim_\alpha P_2$ are both $\mid f(x)$, then there exists P such that $f(x, P_1, P) \sim_\alpha f(x, P_2, P)$. $\blacksquare$

Proof. The third part of the lemma follows immediately from the first two.

For the second part of the lemma, suppose $P_1 \sim_\alpha P_2$ are both $\mid f(x)$. Let P be such that $(f(x, P))P_1$ is defined (per the definition of $\mid f(x)$). By taking a suitable α-image of P (via Assumption 13.36(b) and Condition 13.49(c)), let P be such that $(f(x, P))P_1$ and $(f(x, P))P_2$ are both defined.

For the first part of the lemma, suppose $P_1 \sim_\alpha P_2$ are $\mid f(x, P)$. Let $\text{Free}(\vec{T_0}) = \{\}$ with $ar(\vec{T_0}) = ar(f) - 3$. By Lemma 13.52 and Condition 13.49(b), there exist terms $\vec{T_1} \mid f(x, P)$ such that $P_1[\vec{T_1}] \equiv_\alpha P_2[\vec{T_1}]$ and for each $\vec{T_1}(k)$, $f(x, \vec{T_1}(k), P) \equiv_\alpha \vec{T_1}(k)$. By Conditions 13.49(f) and 13.49(h),

$$\begin{array}{lll} f(x, P_1[\vec{T_1}], P, \vec{T_0}) & \equiv_\alpha & f(x, P_1, P)[\vec{T_0}, \vec{T_1}] \\ f(x, P_2[\vec{T_1}], P, \vec{T_0}) & \equiv_\alpha & f(x, P_2, P)[\vec{T_0}, \vec{T_1}]\,, \end{array} \tag{13.54}$$

and by Condition 13.49(c),

$$f(x, P_1, P)[\vec{T_0}, \vec{T_1}] \;\equiv_\alpha\; f(x, P_2, P)[\vec{T_0}, \vec{T_1}]\,. \tag{13.55}$$

Again by Lemma 13.52, we can choose structurally distinct alternatives to $\vec{T_1}$ that also satisfy the above. Likewise, we can choose structurally distinct alternatives to $\vec{T_0}$ that satisfy the above. Since $f(x, P_1, P)$ and $f(x, P_2, P)$ continue to correspond as all these parameters change, their meta-variable occurrences must match up one-to-one, and by Assumption 13.36(a), $f(x, P_1, P) \sim_\alpha f(x, P_2, P)$. ■

Definition 13.56 Suppose substitutive functions f_k with variable transformations $f_{k\mathrm{v}}$.

f_1 *distributes over* f_2 if, for all $x_1 \not\sqsubseteq x_2$ and $T, P_1, P_2, \vec{T_1}, \vec{T_2}$ such that $f_1(x_1, T, P_1, \vec{T_1})$, $f_2(x_2, T, P_2, \vec{T_2})$, and $f_1(x_1, P_2[T', \vec{T_2}], P_1, \vec{T_1})$ (for some T') are defined, there exist $P_3, \vec{T_3}$ such that

$$P_3[\Box_1, \vec{T_3}] \;=\; (f_1(x_1, P_2[\Box_1, \vec{T_2}], P_1))[\vec{T_1}, \Box_1] \tag{13.57}$$

and

$$\begin{array}{l} f_1(x_1, f_2(x_2, T, P_2, \vec{T_2}), P_1, \vec{T_1}) \;= \\ f_2(f_{1\mathrm{v}}(x_1, x_2), f_1(x_1, T, P_1, \vec{T_1}), P_3, \vec{T_3})\,. \end{array} \tag{13.58}$$ ■

13.2 Reduction

Previous chapters have blithely followed the convention, from §8.2, that binary reduction relations are understood to be relations between equivalence classes under α-renaming. Since we now need to work explicitly with the internal mechanics of α-renaming, we refine that convention to a precise property of binary relations.

Definition 13.59 Suppose R, Q are binary relations on terms.

Infix operator $\longrightarrow^R$ denotes R.

The *concatenation of* R *with* Q, denoted $R \cdot Q$, is
$R \cdot Q = \{\langle T_1, T_3\rangle \mid \exists T_2 \text{ such that } T_1 \longrightarrow^R T_2 \longrightarrow^Q T_3\}$.

R is a *reduction relation* if for all T_1, T_2, T_1', T_2', if $T_1 \longrightarrow^R T_2$, $T_1 \equiv_\alpha T_1'$, and $T_2 \equiv_\alpha T_2'$, then $T_1' \longrightarrow^R T_2'$.

Infix operator $\longrightarrow_R$ denotes the least reduction relation containing the compatible closure of R.

Infix operator $\longrightarrow_R^+$ denotes the transitive closure of $\longrightarrow_R$.

Infix operator $\longrightarrow_R^?$ denotes the union of $\equiv_\alpha$ with $\longrightarrow_R$.

Infix operator $\longrightarrow_R^*$ denotes the transitive closure of $\longrightarrow_R^?$. ■

So $\longrightarrow_R^* = (\equiv_\alpha \cup \longrightarrow_R^+)$.

Lemma 13.60 Suppose $f \in \mathcal{F}_\alpha$.

$\Rrightarrow_f$ is a reduction relation. ■

Proof. Suppose $T_1 \Rrightarrow_f T_2$, $T_1 \equiv_\alpha T_1'$, and $T_2 \equiv_\alpha T_2'$. By Theorem 13.26, $T_1 \Rrightarrow_{f_{\mathbf{id}}} T_1'$; therefore, by Assumptions 13.19(b) and 13.19(c), $\text{Free}(T_1') = \text{Free}(T_1)$, and by Assumption 13.24(a), $T_1' \parallel f$. By Assumption 13.19(d), $f(T_1') \equiv_\alpha f(T_1)$; so since $\equiv_\alpha$ is transitive, by Definition 13.20, $T_1' \Rrightarrow_f T_2'$. ■

Theorem 13.61 Suppose R is a binary relation on terms.
$\longrightarrow_R$ is compatible. ■

Proof. Suppose $C[T_1], C[T_2] \in \mathit{Terms}$, and $T_1 \longrightarrow_R T_2$. By definition of $\longrightarrow_R$, let $C'[T_1'], C'[T_2'] \in \mathit{Terms}$ such that $T_1 \equiv_\alpha C'[T_1']$, $T_2 \equiv_\alpha C'[T_2']$, and $T_1' \longrightarrow^R T_2'$. Since $\equiv_\alpha$ is constructive (Corollary 13.27), $C[C'[T_1']], C[C'[T_2']] \in \mathit{Terms}$. Since $C[C'[T_1']], C[C'[T_2']] \in \mathit{Terms}$ and $T_1' \longrightarrow^R T_2'$, by definition of $\longrightarrow_R$, $C[C'[T_1']] \longrightarrow_R C[C'[T_2']]$. Since $\equiv_\alpha$ is compatible, $C[T_1] \equiv_\alpha C[C'[T_1']]$ and $C[T_2] \equiv_\alpha C[C'[T_2']]$; therefore, by definition of $\longrightarrow_R$, $C[T_1] \longrightarrow_R C[T_2]$. ■

Theorem 13.62

(a) If $\mathcal{R}$ is a set of reduction relations, then $\bigcup \mathcal{R}$ is a reduction relation.

(b) If R, Q are reduction relations, then $R \cdot Q$ is a reduction relation.

(c) If R is a binary relation on terms, then $\longrightarrow_R^+$, $\longrightarrow_R^?$, and $\longrightarrow_R^*$ are reduction relations. ■

Proof. For (a), suppose $\mathcal{R}$ is a set of reduction relations, $\langle T_1, T_2\rangle \in \bigcup \mathcal{R}$, $T_1 \equiv_\alpha T_1'$, and $T_2 \equiv_\alpha T_2'$. Since $\langle T_1, T_2\rangle \in \bigcup \mathcal{R}$, let $R \in \mathcal{R}$ such that $T_1 \longrightarrow^R T_2$. Since $R \in \mathcal{R}$, R is a reduction relation, so $T_1' \longrightarrow^R T_2'$, and $\langle T_1', T_2'\rangle \in \bigcup \mathcal{R}$.

For (b), suppose R, Q are reduction relations, $T_1 \longrightarrow^{R\cdot Q} T_2$, $T_1 \equiv_\alpha T_1'$, and $T_2 \equiv_\alpha T_2'$. Since $T_1 \longrightarrow^{R\cdot Q} T_2$, let $T \in \mathit{Terms}$ such that $T_1 \longrightarrow^R T \longrightarrow^Q T_2$. Since R, Q are reduction relations, $T_1' \longrightarrow^R T \longrightarrow^Q T_2'$. By definition of $R\cdot Q$, $T_1' \longrightarrow^{R\cdot Q} T_2'$.

By Lemma 13.60 and Theorem 13.26, $\equiv_\alpha$ is a reduction relation; and by the definition of $\longrightarrow_R$, $\longrightarrow_R$ is a reduction relation. (c) then follows immediately from (a) and (b). ■

Definition 13.63 Suppose R, Q are binary relations on terms.

R and Q *cooperate* if for all T_1, T_2, T_3, if $T_1 \longrightarrow^R T_2$ and $T_1 \longrightarrow^Q T_3$ then there exists T_4 such that $T_2 \longrightarrow^Q T_4$ and $T_3 \longrightarrow^R T_4$.

R has the *diamond property* if R cooperates with itself.

R is *Church–Rosser* if $\longrightarrow_R^*$ has the diamond property. ■

Lemma 13.64 Suppose R is a binary relation on terms.

If $\longrightarrow_R$ has the diamond property, then $\longrightarrow_R^+$ and $\longrightarrow_R^*$ have the diamond property. ■

Proof. The result for $\longrightarrow_R^+$ is simply by induction: if $T_1 \longrightarrow_R^+ T_2$ in k_1 steps, and $T_1 \longrightarrow_R^+ T_3$ in k_2 steps, then the requisite T_4 has $T_3 \longrightarrow_R^+ T_4$ in k_1 steps and $T_2 \longrightarrow_R^+ T_4$ in k_2 steps. For $\longrightarrow_R^*$, the zero-step case using $\equiv_\alpha$ is provided by the fact that $\longrightarrow_R$ is, by definition, a reduction relation. ■

Definition 13.65 Suppose R is a binary relation on terms, and f is substitutive.

f strictly distributes over R if, for all $x, T_1, P, \vec{T}$, if $T_1 \longrightarrow^R T_2$ and $f(x, T_1, P, \vec{T})$ is defined, then for some $T_3 \equiv_\alpha T_2$, $f(x, T_1, P, \vec{T}) \longrightarrow^R f(x, T_3, P, \vec{T})$.

f distributes over R if f strictly distributes over $R \cup \equiv_\alpha$. ■

Non-strict distributivity allows for the possibility that $f(x, P)$ washes out the difference between T_1 and T_2, i.e., $f(x, T_1, P, \vec{T}) \equiv_\alpha f(x, T_3, P, \vec{T})$.

Definition 13.66 Suppose R is a binary relation on terms.

R is *α-closed* if R is a reduction relation and, for all $f \in \mathcal{F}_\alpha$ and terms T_1, T_2, if $T_1 \longrightarrow^R T_2$ and $T_1 \Rrightarrow_f f(T_1)$, then $f(T_1) \longrightarrow^R f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$.

The set of all α-closed binary relations is denoted $\mathcal{R}_\alpha$. ■

Free variables in a term T are, in general, both obstacles to term construction (limiting what contexts T can occur in) and opportunities for nontrivial term transformation (subjecting T to alteration by substitutive functions that target that variable). Consequently, while constructive R might eliminate some free variables, it presumably will not introduce new ones as this would tend to sabotage constructivity; and while Church–Rosser R might eliminate some free variables, it presumably will not introduce new ones as this would tend to sabotage Church–Rosser-ness.[10] Therefore, for typical well-behaved step $T_1 \longrightarrow^R T_2$, $T_1 \Rrightarrow_f f(T_1)$ will imply $T_2 \Rrightarrow_f f(T_2)$, but $T_2 \Rrightarrow_f f(T_2)$ will not necessarily imply $T_1 \Rrightarrow_f f(T_1)$.

Theorem 13.67 Suppose R is a binary relation on terms.

R is α-closed iff R is a reduction relation and for all $f \in \mathcal{F}_\alpha$, R cooperates with $\Rrightarrow_f$. ■

[10] Suppose R is compatible and includes λ-calculus reduction, and $T_1 \longrightarrow^R T_2$ introduces a new free variable x. Consider term $(\lambda x.T_1)T_3$.

$$\begin{array}{llll} (\lambda x.T_1)T_3 & \longrightarrow^R & T_1 & \longrightarrow^R \quad T_2 \\ (\lambda x.T_1)T_3 & \longrightarrow^R & (\lambda x.T_2)T_3 & \longrightarrow^R \quad T_2[x \leftarrow T_3]\,, \end{array}$$

but since T_2 contains a free x, in general $T_2[x \leftarrow T_3] \neq_R T_2$ (i.e., they won't have a common reduct).

Proof. Suppose R is α-closed, $T_1 \longrightarrow^R T_2$, and $T_1 \Rrightarrow_f T_3$. By the definition of $\Rrightarrow_f$ (Definition 13.20), $T_3 \equiv_\alpha f(T_1)$ and $T_1 \Rrightarrow_f f(T_1)$. By the definition of α-closed (Definition 13.66), $f(T_1) \longrightarrow^R f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$. By the definition of reduction relation (Definition 13.59), $T_3 \longrightarrow^R f(T_2)$. So by the definition of cooperation (Definition 13.63), R cooperates with $\Rrightarrow_f$.

Suppose R is a reduction relation, $T_1 \longrightarrow^R T_2$, $T_1 \Rrightarrow_f f(T_1)$, and for all $g \in \mathcal{F}_\alpha$, R cooperates with $\Rrightarrow_g$. By the definition of cooperation, let $T_2 \Rrightarrow_f T_4$ such that $f(T_1) \longrightarrow^R T_4$. By the definition of $\Rrightarrow_f$, $T_4 \equiv_\alpha f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$. By the definition of reduction relation, $f(T_1) \longrightarrow^R f(T_2)$. So by definition, R is α-closed. ■

Theorem 13.68 Suppose R is a binary relation on terms.
If R is α-closed, then $\longrightarrow_R$ is α-closed. ■

Proof. Suppose $R \in \mathcal{R}_\alpha$.

Suppose $T_1 \longrightarrow_R T_2$ and $T_1 \Rrightarrow_f f(T_1)$; we will show that $f(T_1) \longrightarrow_R f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$. Since $T_1 \longrightarrow_R T_2$, by Definition 13.59, let $C'[T_1'], C'[T_2'] \in \mathit{Terms}$ such that $T_1 \equiv_\alpha C'[T_1']$, $T_2 \equiv_\alpha C'[T_2']$, and $T_1' \longrightarrow^R T_2'$. $C'[T_1'] \Rrightarrow_f f(T_1)$; so, by Assumption 13.36(a), let $f(T_1) = C''[T_1'']$ and $g \in \mathcal{F}_\alpha$ such that $C'[T_1'] \Rrightarrow_f f(T_1)$ satisfies $C' \rightsquigarrow_f C''$ mediated by g. So $T_1' \Rrightarrow_g T_1''$. Let $T_2'' = g(T_2')$. By definition of α-closed, $T_1'' \longrightarrow^R T_2''$ and $T_2' \Rrightarrow_g T_2''$. Since g mediates $C' \rightsquigarrow_f C''$, $C'[T_2'] \Rrightarrow_f C''[T_2'']$. Because $\longrightarrow_R$ contains the compatible closure of R, $C''[T_1''] \longrightarrow_R C''[T_2'']$. ■

Theorem 13.69

(a) If $\mathcal{R}$ is a set of α-closed relations, then $\bigcup \mathcal{R}$ is α-closed.

(b) If R, Q are α-closed relations, then $R \cdot Q$ is α-closed.

(c) If R is an α-closed relation, then $\longrightarrow_R^+$, $\longrightarrow_R^?$, and $\longrightarrow_R^*$ are α-closed. ■

Proof. For (a), suppose $\mathcal{R}$ is a set of α-closed relations. By Theorem 13.62(a), $\bigcup \mathcal{R}$ is a reduction relation. Suppose $\langle T_1, T_2 \rangle \in \bigcup \mathcal{R}$ and $T_1 \Rrightarrow_f f(T_1)$. Since $\langle T_1, T_2 \rangle \in \bigcup \mathcal{R}$, let $R \in \mathcal{R}$ such that $T_1 \longrightarrow^R T_2$. Since $R \in \mathcal{R}$, R is α-closed, so $T_2 \Rrightarrow_f f(T_2)$ and $f(T_1) \longrightarrow^R f(T_2)$; so $\langle f(T_1), f(T_2) \rangle \in \bigcup \mathcal{R}$.

For (b), suppose R, Q are α-closed relations. By Theorem 13.62(b), $R \cdot Q$ is a reduction relation. Suppose $T_1 \longrightarrow^{R \cdot Q} T_2$ and $T_1 \Rrightarrow_f f(T_1)$. Since $T_1 \longrightarrow^{R \cdot Q} T_2$, let $T \in \mathit{Terms}$ such that $T_1 \longrightarrow^R T \longrightarrow^Q T_2$. Since R, Q are α-closed, $T_2 \Rrightarrow_f f(T_2)$ and $f(T_1) \longrightarrow^R f(T) \longrightarrow^Q f(T_2)$. By definition of $R \cdot Q$, $f(T_1) \longrightarrow^{R \cdot Q} f(T_2)$.

By Theorem 13.62(c) and Assumption 13.19(d), $\equiv_\alpha$ is α-closed; and by Theorem 13.68, for any α-closed relation R, $\longrightarrow_R$ is α-closed. (c) then follows immediately from (a) and (b). ■

Definition 13.70 Suppose R is a binary relation on terms.

Poly-context P is *selective in* R if, for all terms $\vec{T_1}, T_2$,

if $\quad P[\vec{T_1}] \longrightarrow_R T_2$ but $P[\vec{T_1}] \not\longrightarrow^R T_2$,

then for some term T_3 and integer k, $\vec{T_1}(k) \longrightarrow_R T_3$ and $T_2 \equiv_\alpha P[\vec{T_1}\backslash^k_{T_3}]$. ■

Selectivity of P in R simplifies structural-inductive reasoning, by allowing P to be treated as an indivisible unit: if $P[\vec{T_1}] \longrightarrow_R T_2$, then the redex is either $P[\vec{T_1}]$ itself, or a subterm of some $\vec{T_1}(k)$ (possibly the trivial subterm, $\vec{T_1}(k)$ itself). As a counterexample, take the λ-calculus η- and β-rules,

$$\begin{array}{rcll} \lambda x.(Tx) & \longrightarrow & T \quad \text{if } x \notin \mathrm{FV}(T) & (\eta) \\ (\lambda x.T_1)T_2 & \longrightarrow & T_1[x \leftarrow T_2]\,, & (\beta) \end{array} \tag{13.71}$$

treating the left-hand sides of the rules as poly-contexts. The left-hand side of the β-rule, $(\lambda x.\Box_1)\Box_2$, isn't selective in $\boldsymbol{\eta}$, because $\lambda x.\Box_1$ might be an $\boldsymbol{\eta}$-redex. Thus, $\boldsymbol{\eta}$-reducing a proper subterm of a $\boldsymbol{\beta}$-redex could result in a term that isn't a $\boldsymbol{\beta}$-redex (such as

$$(\lambda \mathrm{x.(yx)})T \quad \longrightarrow_\eta \quad \mathrm{y}T\,). \tag{13.72}$$

Similarly, the left-hand side of the η-rule, $\lambda x.(\Box_1 x)$, isn't selective in $\boldsymbol{\beta}$, because $(\Box_1 x)$ might be a $\boldsymbol{\beta}$-redex, so that $\boldsymbol{\beta}$-reducing a proper subterm of an $\boldsymbol{\eta}$-redex could result in a term that isn't an $\boldsymbol{\eta}$-redex (such as

$$\lambda \mathrm{x.}((\lambda \mathrm{y.y})\mathrm{x}) \quad \longrightarrow_\beta \quad \lambda \mathrm{x.x}\,). \tag{13.73}$$

For every binary term relation R, every minimal nontrivial P is selective in R, and every trivial P is selective in R (because in either of these cases, *every* proper subterm of $P[\vec{T_1}]$ is a subterm of one of the $\vec{T_1}$).

Definition 13.74 Suppose R is a binary relation on terms, and P is a poly-context.

T is *reducible in* R if T is an R-redex (that is, if $\exists\, T'$ such that $T \longrightarrow^R T'$).

P is *decisively reducible in* R if every term satisfying P is reducible in R.

P is *decisively irreducible in* R if no term satisfying P is reducible in R.

P is *decisive in* R if it is decisively either reducible in R or irreducible in R. ■

In λ-calculus, a term is reducible in $\longrightarrow^\beta$ iff it satisfies $((\lambda x.\Box_1)\Box_2)$ for some x. So $P = (\lambda x.\Box_1)$ is decisively irreducible in $\longrightarrow^\beta$. On the other hand, suppose $P = (\Box_1\, \Box_2)$; then $P[T_1, T_2]$ might or might not be a redex, depending on T_1, so P is not decisive in $\longrightarrow^\beta$. Finally, suppose $P = ((\lambda x.\Box_1)\Box_2)$. Then P is decisively reducible in $\longrightarrow^\beta$, and all $C[P]$ are decisively reducible in $\longrightarrow_\beta$.

In λ_v-calculus, $P = ((\lambda x.\Box_1)\Box_2)$ is not decisive, because whether or not $P[T_1, T_2]$ is a β_v-redex depends on whether or not T_2 is a *value*. $((\lambda x_1.\Box_1)(\lambda x_2.\Box_2))$ is decisive in $\longrightarrow^v$, though, since all matching terms are redexes.

Definition 13.75 Suppose R is a binary relation on terms.

$\longrightarrow_R^{\|?}$ is the smallest binary relation on terms such that for all nontrivial polycontexts P and terms $\vec{T_1}, \vec{T_2}, T_3,$ if

- $P[\vec{T_1}] \in \mathit{Terms}$,
- for all $1 \leq k \leq ar(P),\ \vec{T_1}(k) \longrightarrow_R^{\|?} \vec{T_2}(k),$ and
- either $P[\vec{T_2}] \equiv_\alpha T_3$, or P is decisively reducible in R and there exists T_4 such that $P[\vec{T_2}] \longrightarrow^R T_4 \equiv_\alpha T_3$,

then $P[\vec{T_1}] \longrightarrow_R^{\|?} T_3$.

$\longrightarrow_R^{\|*}$ denotes the transitive closure of $\longrightarrow_R^{\|?}$. ■

Relation $\longrightarrow_R^{\|?}$ is a straightforward generalization of the "parallel reduction" relation that was central to Plotkin's ([Plo75]) proofs of Church–Rosser-ness and standardization for λ_v-calculus. Its generalized form will be used here for the analogous results on SRSs, in proofs approximately generalizing Plotkin's. Its use for Church–Rosser-ness is commonly attributed to Martin-Löf using some ideas of W. Tait.[11]

Lemma 13.76 Suppose R is a binary relation on terms.

If R is α-closed, then $\longrightarrow_R \subseteq \longrightarrow_R^{\|?} \subseteq \longrightarrow_R^* = \longrightarrow_R^{\|*}$. ■

Proof. Suppose R is α-closed, and $T_1 \longrightarrow_R T_2$. By definition of $\longrightarrow_R$, let $T_1 \equiv_\alpha C'[T_1']$ and $T_2 \equiv_\alpha C'[T_2']$ such that $T_1' \longrightarrow^R T_2'$. Let $C'[T_1'] \equiv_\alpha T_1$ satisfy $C' \sim_\alpha C$ mediated by f; then $T_1' \parallel f$. Since R is α-closed, $T_2' \parallel f$ and $f(T_1') \longrightarrow^R f(T_2')$. Since f mediates $C' \sim_\alpha C,\ C[f(T_2')] \equiv_\alpha T_2$. So $T_1 \longrightarrow_R^{\|?} T_2$.

On the other hand, suppose $T_1 \longrightarrow_R^{\|?} T_2$. Assume inductively that the result holds for proper subterms of T_1; then by definition of $\longrightarrow_R^{\|?}$, any subterm reductions can be performed first, by compatibility, and then the top-level reduction, if any, can be performed last. So $T_1 \longrightarrow_R^* T_2$.

Since $\longrightarrow_R \subseteq \longrightarrow_R^{\|?}$, the transitive closures are similarly related, $\longrightarrow_R^+ \subseteq \longrightarrow_R^{\|*}$. By the definition of $\longrightarrow_R^{\|?},\ \equiv_\alpha \subseteq \longrightarrow_R^{\|?}$; and $\longrightarrow_R^* = (\longrightarrow_R^+ \cup \equiv_\alpha)$, so $\longrightarrow_R^* \subseteq \longrightarrow_R^{\|*}$. ■

Lemma 13.77 Suppose R is a binary relation on terms.

If R is α-closed, and $\longrightarrow_R^{\|?}$ has the diamond property, then $\longrightarrow^R$ is Church–Rosser. ■

[11] Before this technique emerged in the early 1970s, proofs of the Church–Rosser theorem for λ-calculus were much messier. It takes advantage of the fact that λ-calculus terms have tree structure, a property on which earlier techniques had failed to capitalize. See [Ros82, §4], [Bare84, §3.2].

Proof. Suppose R is α-closed, and $\longrightarrow_R^{\|?}$ has the diamond property. Since $\longrightarrow_R^{\|?}$ has the diamond property, so does its transitive closure $\longrightarrow_R^{\|*}$; and by Lemma 13.76, $\longrightarrow_R^{\|*} = \longrightarrow_R^*$. So $\longrightarrow_R^*$ has the diamond property, which is to say that $\longrightarrow^R$ is Church–Rosser. ■

For a standardization theorem, showing the existence of *some* standard order of reduction would not suit our purpose: we mean the theorem to mediate a proof of operational soundness of $=_\bullet$, for which (as mentioned at the top of the chapter) the standard order of reduction should first exercise redexes in evaluation contexts, and then recursively apply the same principles to subterms. Moreover, evaluation contexts are simply a way of specifying the deterministic order of $\longmapsto_\bullet$, which is specific to some $\bullet$-semantics and which, therefore, we do not wish to fix in our abstract treatment. Consequently, even if we meant to stop our abstract treatment at standardization, leaving all the rest of operational soundness to the treatments of individual calculi, we would still need to generalize the notion of evaluation context for use in our abstract treatment. (In fact, we are going to push the abstract treatment beyond standardization into operational-soundness territory.) Our generalization of evaluation context will build on two notions, *evaluation order* and *suspending poly-context*.

Evaluation order is just a consistent way of ordering the subterms of any term; it will be used to decide which subterms get reduced first, regardless of what reduction relation will actually be used on them. Recall that if two iso minimal nontrivial poly-contexts satisfy each other, then they differ only by permutation of their meta-variable indices. We simply choose one of these permutations to be the order of subterm evaluation, and require that this choice be invariant under α-renaming.

Definition 13.78 An *evaluation order* is a set $\mathcal{E}$ of iso minimal nontrivial poly-contexts such that

- for every T, there exists $P \in \mathcal{E}$ that satisfies T, and
- if $T_1 \Rrightarrow_f T_2$, and each T_k satisfies $P_k \in \mathcal{E}$, then $T_1 \Rrightarrow_f T_2$ satisfies $P_1 \rightsquigarrow_f P_2$. ■

For example, in ordinary λ-calculus, $(\Box_1\, \Box_2) \in \mathcal{E}$ would cause left-to-right evaluation, while $(\Box_2\, \Box_1) \in \mathcal{E}$ would cause right-to-left. The definition would not allow both of these to occur in $\mathcal{E}$:

Lemma 13.79 Suppose evaluation order $\mathcal{E}$.
If T satisfies $P_1 \in \mathcal{E}$, and T satisfies $P_2 \in \mathcal{E}$, then $P_1 = P_2$. ■

Proof. Suppose T satisfies $P_1, P_2 \in \mathcal{E}$. Since P_1, P_2 are iso minimal nontrivial, they can only differ by permutation of their meta-variable indices. Since $T \Rrightarrow_{f_{\mathbf{id}}} T$ (Theorem 13.26), $P_1 \rightsquigarrow_{f_{\mathbf{id}}} P_2$. By varying individual subterms one can show that each meta-variable occurrence in P_2 must have the same index as the occurrence in the same position in P_1; therefore, P_1 and P_2 are identical. ■

A suspending poly-context is one that prevents $\longmapsto_\bullet$ from applying to subterms, effectively suspending computation. That is, if C is suspending, then any composition of contexts involving C (in general, $C_1[C[C_2]]$) *cannot* be an evaluation context. In λ_v-calculus, the minimal suspending contexts are those of the form $(\lambda x.\Box)$. In $\mathfrak{f}$-calculi, the minimal suspending contexts are those that construct operatives or environments (as evident in the definition of *value*, which is a term T such that every active subterm of T occurs within an operative or environment (cf. (10.1))). For the abstract treatment, suspending contexts will be derived from the reduction relation independent of evaluation order.

Definition 13.80 Suppose R is a binary relation on terms, and P is a poly-context. P is *suspending in* R if all of the following conditions hold.

- P is minimal nontrivial;
- all poly-contexts $C[P]$ are decisive in R; and
- there exists context C such that $C[P]$ is decisively reducible in R, but C is not decisive in R. ■

In λ-calculus, suppose $P = (\lambda x.\Box_1)$. P is minimal nontrivial, and decisively irreducible in $\longrightarrow^\beta$; for nontrivial C, whether or not any $C[P[T]]$ is a β-redex is independent of T; and $C = (\Box\, x')$ is not decisively reducible in $\longrightarrow^\beta$, but $C[P] = ((\lambda x.\Box_1)\, x')$ is; therefore, P is suspending in $\longrightarrow^\beta$. On the other hand, suppose $P = (\Box_1\, \Box_2)$; then $P[T_1, T_2]$ might or might not be a β-redex, depending on T_1, so P is not decisive in $\longrightarrow^\beta$, therefore not suspending in $\longrightarrow^\beta$. Finally, suppose $P = ((\lambda x.\Box_1)\Box_2)$. Then P is decisively reducible in $\longrightarrow^\beta$, and all $C[P]$ are decisive in $\longrightarrow^\beta$; but P isn't minimal nontrivial, so it can't be suspending.

In λ_v-calculus, the suspending poly-contexts are, again, those of the form $P = (\lambda x.\Box_k)$ — although the choices of C for which $C[P]$ is decisively reducible in $\longrightarrow^v$ are a proper subset of those for which it is decisively reducible in $\longrightarrow^\beta$.

Definition 13.81 Suppose R is a binary relation on terms.

An *R-evaluation normal form* is a term N such that every R-reducible subterm of N occurs within a poly-context that is suspending in R. ■

That is, for all C and T, if $N = C[T]$ and T is reducible in R, then there exist contexts C_k such that $C = C_1[C_2[C_3]]$ and C_2 satisfies some poly-context that is suspending in R. Note that an R-evaluation normal form is not necessarily a *value*, in the usual sense, because not every "active term" is a redex; e.g., $\mathfrak{f}_p$-calculus term [eval () x] is an evaluation normal form.

Lemma 13.82 Suppose binary term relation R.

If $T_1 \longrightarrow_R T_2$, and T_1 is an R-evaluation normal form, then so is T_2. ■

Proof. Suppose $T_1 \longrightarrow_R T_2$, and T_1 is an R-evaluation normal form. By definition of R-evaluation normal form (Definition 13.81), every R-redex in T_1 is within an R-suspending context; and $T_1 \longrightarrow^R T_2$ can only modify one of these redexes, so that by definition of R-suspending poly-context (Definition 13.80), any new redex introduced into T_2 is also within an R-suspending poly-context. ■

Definition 13.83 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

An $R, \mathcal{E}$-*evaluation context* is a context E, not decisively reducible in R, such that either $E = \Box$, or $E = P[\vec{P}]$ such that $P \in \mathcal{E}$ is not suspending in R and, for some $1 \leq k \leq ar(P)$ and all $1 \leq j \leq ar(P)$,

if $j < k$ then $\vec{P}(j)$ is an R-evaluation normal form;

if $j = k$ then $\vec{P}(j)$ is an $R, \mathcal{E}$-evaluation context; and

if $j > k$ then $\vec{P}(j)$ is a term. ■

Lemma 13.84 Suppose binary term relation R, and poly-context P_1.

If R is α-closed and constructive, and P_1 is iso and in general position, then

(a) if P_1 is selective in R, then so are all its α-images.

(b) if P_1 is decisively reducible in R, then so are all its α-images.

(c) if P_1 is decisively irreducible in R, then so are all its α-images.

(d) if P_1 is decisive in R, then so are all its α-images. ■

Proof. Suppose R is α-closed and constructive, P_1 is iso and in general position, and P_2 is an α-image of P_1. Since P_1 is iso, its α-image P_2 is iso; so by Theorem 13.48, every term satisfying P_2 has an α-image satisfying P_1.

For (a), suppose P_1 is selective in R, $P_2[\vec{T_2}] \longrightarrow_R T_2$, and $P_2[\vec{T_2}] \not\longrightarrow^R T_2$. By the above, let $P_2[\vec{T_2}] \Rrightarrow_f P_1[\vec{T_1}]$. Since R is α-closed, $T_2 \Rrightarrow_f T_1$ such that $P_1[\vec{T_1}] \longrightarrow_R T_1$ and $P_1[\vec{T_1}] \not\longrightarrow^R T_1$. Since P_1 is selective, let $\vec{T_1}(k) \longrightarrow_R T_3$ and $T_1 \equiv_\alpha P_1[\vec{T_1}\backslash^k_{T_3}]$. In a projection of prime factors of $P_2[\vec{T_2}] \Rrightarrow_f P_1[\vec{T_1}]$, let g mediate the projection at $\Box_k$ in P_2, so that $\vec{T_2}(k) \Rrightarrow_g \vec{T_1}(k)$. Since R is α-closed, let $\vec{T_2}(k) \longrightarrow_R T_4 \Rrightarrow_g T_3$. Since R is constructive, $P_2[\vec{T_2}] \longrightarrow_R P_2[\vec{T_2}\backslash^k_{T_4}] \Rrightarrow_f P_1[\vec{T_1}\backslash^k_{T_3}] \equiv_\alpha T_1$. By Lemma 13.21(c), $P_2[\vec{T_2}\backslash^k_{T_4}] \equiv_\alpha T_2$. Therefore, P_2 is selective.

For (b), (c), and (d): since every term satisfying P_2 has an α-image satisfying P_1, and R is α-closed, if every term satisfying P_1 is (ir)reducible in R, then so is every term satisfying P_2. ■

Lemma 13.85 Suppose binary term relation R, evaluation order $\mathcal{E}$, and poly-context P_1.

If R is α-closed and constructive, then

(a) if P_1 is iso and suspending in R, then so are all its α-images.

(b) if T_1 is an R-evaluation normal form, then so are all its α-images.

(c) if C_1 is an $R, \mathcal{E}$-evaluation context, then so are all its general-position α-images. ■

Proof. Suppose R is α-closed and constructive.

For (a), suppose P_1 is iso minimal nontrivial; all poly-contexts $C[P_1]$ are decisive in R; and $C_1'[P_1]$ is a poly-context decisively reducible in R, but C_1' is not decisive in R. Suppose P_2 is an α-image of P_1; then P_2 is iso minimal nontrivial. Since P_1 and P_2 are minimal nontrivial, they are in general position. We need to show that P_2 also has the other two properties.

Suppose poly-context $C_2[P_2]$. By Theorems 13.42 and 13.45, there exists an α-image $C_1[P_1]$ of $C_2[P_2]$ such that $C_1[P_1]$ is in general position. Since P_1 is suspending, $C_1[P_1]$ is decisive in R; therefore, by Lemma 13.84(d), $C_2[P_2]$ is decisive in R.

Let $C_1''[P_1]$ be an α-image of $C_1'[P_1]$ in general position (by Theorem 13.45). Since P_1 is suspending, $C_1''[P_1]$ is decisive in R. Since $C_1''[P_1]$ is a general-position α-image of a poly-context decisively reducible in R, and R is α-closed, $C_1''[P_1]$ cannot be decisively irreducible in R (by Lemma 13.84(c)); so $C_1''[P_1]$ is decisively reducible in R. Since C_1'' is a general-position α-image of C_1', and C_1' is not decisive in R, C_1'' cannot be decisive in R (by Lemma 13.84(b)).

By Theorem 13.42, there exists a general-position α-image $C_2''[P_2]$ of $C_1''[P_1]$. Since $C_1''[P_1]$ is in general position, $C_2''[P_2]$ must be decisively reducible in R (by Lemma 13.84(b)); and since C_2'' is in general position, it must not be decisive in R (by Lemma 13.84(d)).

(b) follows immediately from (a) and α-closure of R.

For (c), suppose C_2 is an α-image of C_1 in general position, and proceed by induction on the depth of $\Box$ within C_1, i.e., the number of singular contexts whose composition is C_1. When $C_1 = \Box$, $C_2 = C_1$. Suppose $C_1 = P[\vec{P}]$ such that $P \in \mathcal{E}$, and let k have the properties enumerated in the definition (Definition 13.83). C_2 is not decisively reducible in R, by Lemma 13.84(b). The image of P in C_2 is not suspending, by (a). Suppose $1 \leq j \leq ar(P)$. The case of $j < k$ carries over to C_2 by (b); $j = k$ carries over by the inductive hypothesis; and $j > k$ carries over trivially. ■

(c) covers only α-images in general position because, for some R, a context C_1 that is not decisively reducible might have an α-image that *is* decisively reducible (just not one in general position, by Lemma 13.84(b)). This would happen if, for example, terms of a certain form are redexes only if some variable, bound at the top level of the term, *does not occur free* in some subterm — say, $\lambda x.(T_1 T_2)$ is a redex iff $x \notin \text{Free}(T_2)$; then $\lambda\text{x}.(\Box_1(\lambda\text{y}.\Box_2))$, is not decisively reducible, but its α-image $\lambda\text{x}.(\Box_1(\lambda\text{x}.\Box_2))$ is.

This general pattern of behavior, in which a term is reducible only if some variable is bound but not used, is not pathological; rather, it is typical of garbage-collection schemata.

Definition 13.86 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

The $R, \mathcal{E}$-*evaluation relation*, denoted $\longmapsto_R^{\mathcal{E}}$, is the least reduction relation such that if E is an $R, \mathcal{E}$-evaluation context, $T_1 \longrightarrow^R T_2$, and $E[T_1], E[T_2] \in \mathit{Terms}$, then $E[T_1] \longmapsto_R^{\mathcal{E}} E[T_2]$. ■

Theorem 13.87 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

If R is α-closed, then $\longmapsto_R^{\mathcal{E}}$ is α-closed. ■

Proof. Suppose R is α-closed, $T_1 \longmapsto_R^{\mathcal{E}} T_2$, and $T_1 \Rrightarrow_f f(T_1)$; we will show that $f(T_1) \longmapsto_R^{\mathcal{E}} f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$. Since $\longmapsto_R^{\mathcal{E}} \subseteq \longrightarrow_R$ (by definition), $f(T_1) \longrightarrow_R f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$ (by Theorem 13.68).

Let E be an $R, \mathcal{E}$-evaluation context such that $T_1 \equiv_\alpha E[T_1']$, $T_2 \equiv_\alpha E[T_2']$, and $T_1' \longrightarrow^R T_2'$ (by Definition 13.86). Let C be an α-image of E, T_1'' of T_1', and T_2'' of T_2', such that $f(T_1) \equiv_\alpha C[T_1'']$, $f(T_2) \equiv_\alpha C[T_2'']$, and $T_1'' \longrightarrow^R T_2''$ (following the proof of Theorem 13.68). Assume $C[T_k'']$ are in general position (by Theorem 13.44); then C is an $R, \mathcal{E}$-evaluation context (by Lemma 13.85(c)), therefore $f(T_1) \longmapsto_R^{\mathcal{E}} f(T_2)$ (by definition). ■

Definition 13.88 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

$\vec{T}$ with arity $n \geq 1$ is an *R-reduction sequence* if for all $1 \leq k < n$, $\vec{T}(k) \longrightarrow^R \vec{T}(k+1)$.

Two R-reduction sequences $\vec{T_1}$ and $\vec{T_2}$ are *concatenable* if $\vec{T_1}(ar(\vec{T_1})) = \vec{T_2}(1)$. Their *concatenation*, denoted $\vec{T_1} \cdot \vec{T_2}$, is then $\sum_k \begin{cases} \vec{T_1}(k) & \text{if } k \leq ar(\vec{T_1}) \\ \vec{T_2}(k + 1 - ar(\vec{T_1})) & \text{otherwise.} \end{cases}$

A vector of terms $\vec{T}$ is an $R, \mathcal{E}$-*standard reduction sequence* if any of the following conditions holds.

(a) $P \in \mathcal{E}$ with arity zero, and
$\vec{T} = \langle P \rangle$.

(b) $\vec{T_1}$ with arity ≥ 2 is an $\longmapsto_R^{\mathcal{E}}$-reduction sequence,
$\vec{T_2}$ is an $R, \mathcal{E}$-standard reduction sequence, and
$\vec{T} = \vec{T_1} \cdot \vec{T_2}$.

(c) $P \in \mathcal{E}$ with arity $n \geq 1$;
for all $1 \leq k \leq n$, $\vec{T_k}$ is an $R, \mathcal{E}$-standard reduction sequence with arity m_k;

for all $1 \leq k \leq n,\ C_k = P\left[\sum_j \left\{ \begin{array}{ll} \vec{T_j}(m_j) & \text{if } j < k \\ \Box & \text{if } j = k \\ \vec{T_j}(1) & \text{if } j > k \end{array} \right.\right]$ is a context;
for all $1 \leq k \leq n,\ \vec{T'_k} = \sum_j C_k[\vec{T_k}(j)]$ is a vector of terms; and
$\vec{T}$ is the concatenation of all the $\vec{T'_k},\ \vec{T} = \vec{T'_1} \cdot \ldots \cdot \vec{T'_n}$.

(d) $\vec{T'}$ is an $R, \mathcal{E}$-standard reduction sequence with arity m;
$ar(\vec{T}) = m$; and
for all $1 \leq j \leq m,\ \vec{T}(j) \equiv_\alpha \vec{T'}(j)$.

An $R, \mathcal{E}$-standard reduction sequence (or R-reduction sequence) $\vec{T}$ is *from* T_1 *to* T_2 if $\vec{T}(1) = T_1$ and $\vec{T}(ar(\vec{T})) \equiv_\alpha T_2$. ■

An $R, \mathcal{E}$-standard reduction sequence is *not* necessarily an R-reduction sequence (because in general R might not be compatible), but *is* necessarily an $\longrightarrow_R$-reduction sequence. Every $\longmapsto_R^{\mathcal{E}}$-reduction sequence is an $R, \mathcal{E}$-standard reduction sequence. Note that the set of $R, \mathcal{E}$-standard reduction sequences is not closed under concatenation; sequences $\vec{T'_k}$ in Criterion 13.88(c) have to be concatenated in order of increasing k.

The shorthand terminology of a reduction sequence "from T_1 to T_2" means that the first element is $=$ to T_1, but only that the last element is $\equiv_\alpha$ to T_2, because the weaker constraint on the last element avoids having to make special provisions for a degenerate case. If $R, \mathcal{E}$-standard reduction sequence $\vec{T}$ is from T_1 to T_2, and $ar(\vec{T}) \geq 2$, then we could always replace the last element of $\vec{T}$ with T_2 (by Criterion 13.88(d)); but when $ar(\vec{T}) = 1$, we can't simultaneously have the first element $= T_1$ and the last element $= T_2$ unless $T_1 = T_2$ — whereas usually this will be the reflexive case of $T_1 \longrightarrow_R^* T_2$, which is $T_1 \equiv_\alpha T_2$ (not $T_1 = T_2$).

Even when $R, \mathcal{E}$-evaluation is deterministic (which is not necessary for arbitrary R and $\mathcal{E}$, so that it will have to proven for regular SRSs in Theorem 13.105), the order in which redexes are exercised in an $R, \mathcal{E}$-standard reduction sequence is not always unique. The non-uniqueness stems from the fact that while Criterion 13.88(b) requires its prefixed subsequence to contain only evaluation steps, Criterion 13.88(c) does not *forbid* its subsequences to contain evaluation steps. For example, consider a right-to-left $\mathring{\imath}_p$-calculus standard reduction sequence starting with term [eval T e]. A prefix of the sequence, via Criterion 13.88(b), contains only evaluation steps; suppose none of these are top-level. Then they must all be reductions of subterm T; and subsequently, Criterion 13.88(c) allows subsequences to standardly reduce first e, and then T. Since e is already a value, hence an evaluation normal form, any reductions of e are certainly not evaluation steps. However, if the subsequent standard reduction of subterm T involves any evaluation steps (which it might, if T is not already an evaluation normal form by then), these steps will also be evaluation steps for the

whole term [eval T e] — so that the whole standard reduction sequence may contain evaluation steps (reducing subterm T) that follow non-evaluation steps (reducing subterm e). This mild ambiguity of order makes the definition both easier to state, and easier to satisfy; and it entails no fundamental disadvantage, because given a standard reduction sequence, it will always be possible to find one in which all the evaluation steps are done first.

Lemma 13.89 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

If $\vec{T}$ is an $R, \mathcal{E}$-standard reduction sequence from T_1 to T_2, then there exists an $R, \mathcal{E}$-standard reduction sequence $\vec{T'}$ from T_1 to T_2, with $ar(\vec{T'}) = ar(\vec{T})$, such that all the $R, \mathcal{E}$-evaluation steps in $\vec{T'}$ occur consecutively at the start of the sequence. ■

Proof. Suppose $\vec{T}$ is an $R, \mathcal{E}$-standard reduction sequence from T_1 to T_2, and $ar(\vec{T}) = m$. Proceed by induction on m, and within consideration of given m, by induction on the size of T.

If $m = 1$ or $m = 2$, the result is trivial; so suppose $m \geq 3$ and the result holds for all smaller m. If the first step of $\vec{T}$ is an $R, \mathcal{E}$-evaluation step, then the result follows immediately by applying the inductive hypothesis to the rest of $\vec{T}$ (from $\vec{T}(2)$ to $\vec{T}(m)$ with arity $m-1$); so suppose that the first step of $\vec{T}$ is not an $R, \mathcal{E}$-evaluation step. If no later step of $\vec{T}$ is an $R, \mathcal{E}$-evaluation step, then $\vec{T}$ already has the desired property; so suppose some later step of $\vec{T}$ is an $R, \mathcal{E}$-evaluation step.

Since the first step of $\vec{T}$ is not an $R, \mathcal{E}$-evaluation step, $\vec{T}$ must be an $R, \mathcal{E}$-standard reduction sequence via Criterion 13.88(c). Let P, n, $\vec{T_k}$, and m_k be as in the criterion. The inductive hypothesis applies to each $\vec{T_k}$ (since it is no longer than $\vec{T}$ and its first term is a subterm of T), so assume without loss of generality that in each $\vec{T_k}$, all $R, \mathcal{E}$-evaluation steps occur consecutively at the start. Let k be the smallest integer such that $\vec{T_k}(1)$ is not an R-evaluation normal form; then the first $R, \mathcal{E}$-evaluation step in $\vec{T}$ must be due to the first step in $\vec{T_k}$ (because without some evaluation step in $\vec{T_k}$, no reduction of any other subterm could be an evaluation step; by supposition there *is* an evaluation step somewhere; and if there is an evaluation step anywhere, the first step of $\vec{T_k}$ must be one). Adjust sequence $\vec{T}$ by moving that one step to the front of the sequence (while still performing the rest of the steps of $\vec{T_k}$ in their former place in the sequence); then the resulting sequence $\vec{T}$ is $R, \mathcal{E}$-standard, has the same length as $\vec{T}$, and is from T_1 to T_2, but starts with an $R, \mathcal{E}$-evaluation step, reducing it to a previously solved case (which was handled by induction on m). ■

Definition 13.90 Suppose binary term relation R, and evaluation order $\mathcal{E}$.

T is *R-observable* if T is a minimal nontrivial R-evaluation normal form.

T is $R, \mathcal{E}$-*normalizable* if there exists an R-evaluation normal form N such that $T \longmapsto_R^{\mathcal{E}*} N$.

T_1 and T_2 are $R, \mathcal{E}$-*operationally equivalent*, denoted $T_1 \simeq_R^{\mathcal{E}} T_2$, if, for every C such that $C[T_1], C[T_2] \in \mathit{Terms}$ and $\mathrm{Free}(C[T_1], C[T_2]) = \{\}$, both of the following conditions hold.

(a) $C[T_1]$ is $R, \mathcal{E}$-normalizable iff $C[T_2]$ is $R, \mathcal{E}$-normalizable.

(b) For all R-observable T_3, $C[T_1] \longmapsto_R^{\mathcal{E}*} T_3$ iff $C[T_2] \longmapsto_R^{\mathcal{E}*} T_3$. ■

In general, the precondition that $\mathrm{Free}(C[T_1], C[T_2]) = \{\}$ makes operational equivalence easier to prove, by allowing the proof to ignore terms with non-eliminable free variables; but none of the proofs in this chapter will need it.

13.3 Substitutive reduction systems

Definition 13.91 An *SRS concrete schema*, κ, is a structure of the form

$$\begin{array}{ll} P_0[P_1[\sum_{k=1}^{n_1} \Box_{k+n_0-1}], & \sum_{k=2}^{n_0} \Box_{k-1}] \\ P'[\sum_{k=1}^{n_1} f(x, \Box_{k+n_0-1}, P_0), & \sum_{k=2}^{n_0} \Box_{k-1}], \end{array} \longrightarrow \tag{13.92}$$

where

(a) P_0 and P_1 are iso.

(b) $n_0 = ar(P_0)$ and $n_1 = ar(P_1)$.

(c) f is substitutive with variable transformation f_{v}.

(d) κ is satisfiable. A pair of terms $\langle T_1, T_2 \rangle$ *satisfies* κ if there exists a vector of terms $\vec{T}$ with arity $n_0 + n_1 - 1$ such that replacing each $\Box_k$ of κ with $\vec{T}(k)$ — *except* within the occurrences of P_0 on the right-hand side, as the third input to f— defines T_1 on the left-hand side and T_2 on the right-hand side. κ is satisfiable if there exists a pair of terms that satisfies it.

(e) If f is nontrivial, then $\mathrm{Bind}(P_0 \text{ at } \Box_1) = \{\}$.

$\mathrm{Below}(x) \cap \mathrm{Bind}(P_0) = \{\}$.

If $1 \leq k \leq n_1$, then $x \in \mathrm{Bind}(P_1 \text{ at } \Box_k)$ iff f is not trivial.

If $\Box_k$ occurs in P', and $k \leq n_1$, then
$\mathrm{Bind}(P' \text{ at } \Box_k) = \{f_{\mathrm{v}}(x, x') \mid x' \in \mathrm{Bind}(P_1 \text{ at } \Box_k)\} \cup \mathrm{Bind}(P_0 \text{ at } \Box_1)$.

If $\Box_k$ occurs in P', and $k > n_1$, then
$\mathrm{Bind}(P' \text{ at } \Box_k) - \mathrm{Bind}(P_1) = \mathrm{Bind}(P_0 \text{ at } \Box_{k+1-n_1})$.

(f) $\mathrm{Free}(P') \subseteq \mathrm{Free}(P_0[\vec{\Box}_{n_0} \backslash^1_{P_1}])$. ■

For example, in λ-calculus, let f be the substitutive function with $f(x, T_1, (\Box_1\, \Box_2), T_2) = T_1\lfloor x \leftarrow T_2 \rfloor$, suitably restricted per Condition 13.49(a); $f_{\mathrm{v}}(x, x') = x'$, restricted to $x' \neq x$; and $P_0 = (\Box_1\, \Box_2)$. Then

$$(\lambda \mathrm{x}.\Box_2)\Box_1 \quad \longrightarrow \quad f(\mathrm{x}, \Box_2, P_0, \Box_1) \tag{13.93}$$

is an SRS concrete schema. $P_1 = \lambda \mathrm{x}.\Box_1$; $P' = \Box_1$; no variables are bound by P_0; $\mathrm{x} \in \mathrm{Bind}(P_1 \text{ at } \Box_1)$ and f is nontrivial; $\mathrm{Bind}(P' \text{ at } \Box_1) = \{f_{\mathrm{v}}(\mathrm{x}, x') \mid x' \in \mathrm{Bind}(P_1 \text{ at } \Box_1)\} = \{\}$ since $f_{\mathrm{v}}(\mathrm{x}, \mathrm{x})$ is undefined; and $\Box_2$ doesn't occur in P'. Note that the definition of SRS concrete schema uses behavior of f to dictate whether P_1 binds x, and behavior of f_{v} to dictate whether P' binds x.

By itself, this one concrete schema apparently doesn't cover the entire β-rule,

$$(\lambda x.T_2)T_1 \quad \longrightarrow \quad T_2[x \leftarrow T_1]\,, \tag{13.94}$$

not only because the β-rule allows any variable x to be bound, but also because, even for $x = \mathrm{x}$, Condition 13.49(a) requires f to be undefined whenever T_2 is not in $(\mathrm{x} \cup T_1)$-general position. However, if one starts with the binary relation naively induced by the concrete schema (pairs of terms that *satisfy* the concrete schema), and then takes the α-closure of that induced relation —the smallest α-closed relation containing it— one gets exactly the enumerated reduction relation of the β-rule.

Bad hygiene in a concrete schema involves a variable and (usually) a binding — either a binding that ceases to bind the variable when the schema is applied, freeing it so that it can be captured by surrounding scope (as an "upward funarg"; a variant of this being that the variable is simply introduced where it didn't exist before, with no matching binding); or a binding that starts to bind the variable when the schema is applied, locally capturing it (as a "downward funarg"). There are three cases, depending on where the variable and the binding are: the variable is within one of the subterms (i.e., not in the top-level poly-contexts, P_0 P_1 and P'), and the binding is also within a subterm; or the variable is within one of the subterms, and the binding is in the top-level schema poly-contexts; or the variable itself is in the top-level schema poly-contexts.

Bad hygiene due to bindings within subterms is prevented by provisions of the definition of substitutive function, especially Conditions 13.49(a) and 13.49(i), while gaps in coverage caused by Condition 13.49(a) are smoothed over when the α-closure is taken. Bad hygiene due to variables within the top-level schema poly-contexts is prevented, or at least bounded, by Condition 13.91(f) in the definition of concrete schema. Bad hygiene due to variables within the second input to f (that is substituted into) interacting with bindings in the top-level schema poly-contexts, is prevented by Conditions 13.49(i), 13.49(b), and 13.91(e) (noting that variables descended from x, the first input to f, must be bound either by P_1 or internally by the second input to f). There only remains bad hygiene due to variables within the later inputs to f, or within subterms copied without passing through f, interacting with bindings in the top-level schema poly-contexts; the above example isn't subject to this, because

$\text{Bind}(P') = \{\}$, but in general this last form of bad hygiene must be provided for by some other means. The means used is to define an *induced hygienic relation* that excludes that type of hygiene violation (similarly to Condition 13.49(a)).

Definition 13.95 Suppose SRS concrete schema κ satisfying (13.92).

The *induced hygienic relation of* κ is the set of all pairs of terms $\langle P_0[P_1[\vec{T_1}], \vec{T_0}],\ P'[\sum_k f(x, \vec{T_1}(k), P_0, \vec{T_0}),\ \vec{T_0}]\rangle$ such that for all $j < n_0$ and $k \leq n_1$, $\text{Bind}(P' \text{ at } \Box_k\ ) \cap \text{Free}(P_0[\vec{\Box}_{n_0} \backslash \vec{T_0}(j)^{j+1}]) = \{\}$ and $\text{Bind}(P' \text{ at } \Box_{n_1+j}) \cap \text{Free}(P_0[\vec{\Box}_{n_0} \backslash \vec{T_0}(j)^{j+1}]) = \{\}$. ■

In other words, when some $\vec{T_0}(j)$ is copied from outside the scope of P_1 to inside the scope of P', the copying must not capture any free variables of $\vec{T_0}(j)$.

Definition 13.96 Suppose SRS concrete schema κ satisfying (13.92).

The *α-closure of* κ, denoted $\longrightarrow^\kappa$, is the smallest α-closed relation containing the induced hygienic relation of κ.

A *concrete SRS*, $\mathcal{K}$, is a set of SRS concrete schemata. The *α-closure of* $\mathcal{K}$, denoted $\longrightarrow^\mathcal{K}$, is the union of the α-closures of the elements of $\mathcal{K}$. $\mathcal{K}$ is *α-normal* if no term is reducible in the α-closures of more than one element of $\mathcal{K}$; that is, there do not exist T, T_1, T_2 and $\kappa_1 \neq \kappa_2 \in \mathcal{K}$ such that $T \longrightarrow^{\kappa_1} T_1$ and $T \longrightarrow^{\kappa_2} T_2$. An *$\alpha$-normal form* of $\mathcal{K}$ is an α-normal concrete SRS $\mathcal{K}'$ such that $\longrightarrow^\mathcal{K} = \longrightarrow^{\mathcal{K}'}$. $\mathcal{K}$ is *α-normalizable* if it has an α-normal form.

An *SRS schema*, σ, is a reduction rule schema such that, for some α-normalizable concrete SRS $\mathcal{K}$, $\longrightarrow^\mathcal{K} = \longrightarrow^\sigma$ and the left-hand side of each element of $\mathcal{K}$ minimally satisfies the left-hand side of σ. $\mathcal{K}$ is then a *concrete form* of σ.

An *SRS*, $\mathcal{S}$, is a set of SRS schemata. Its enumerated reduction relation, $\longrightarrow^\mathcal{S}$, is the union of the enumerated relations of its schemata. A *concrete form of* $\mathcal{S}$ is the union of any set of concrete forms of each of its schemata. ■

Evidently, if $\mathcal{K}$ is a concrete form of SRS $\mathcal{S}$, then $\longrightarrow^\mathcal{S} = \longrightarrow^\mathcal{K}$.

Recalling the previous example, the α-closure of Concrete Schema (13.93) is exactly $\longrightarrow^\beta$, the enumerated relation of the λ-calculus β-rule.

Lemma 13.97 Suppose SRS $\mathcal{S}$.

If there does not exist any term T that satisfies the left-hand sides of two different schemata in $\mathcal{S}$, then $\mathcal{S}$ has an α-normal concrete form. ■

Proof. Suppose $\mathcal{S}$ does not have an α-normal concrete form. By the definition of SRS schema, for every $\sigma \in \mathcal{S}$, let $\mathcal{K}_\sigma$ be an α-normal concrete form of σ. Let $\mathcal{K} = \bigcup_{\sigma\in\mathcal{S}} \mathcal{K}_\sigma$. $\mathcal{K}$ is a concrete form of $\mathcal{S}$; therefore, by supposition, $\mathcal{K}$ is not α-normal. By definition of α-normal, let $\kappa_1 \neq \kappa_2 \in \mathcal{K}$ and $T \in \mathit{Terms}$ such that T is reducible in both $\longrightarrow^{\kappa_1}$ and $\longrightarrow^{\kappa_2}$. Let $\sigma_1, \sigma_2 \in \mathcal{S}$ be the schemata in whose concrete

forms κ_1 and κ_2 occur; thus, $\kappa_1 \in \mathcal{K}_{\sigma_1}$ and $\kappa_2 \in \mathcal{K}_{\sigma_2}$. Because T is reducible in both $\longrightarrow^{\kappa_1}$ and $\longrightarrow^{\kappa_2}$, any concrete SRS containing both κ_1 and κ_2 is not α-normal; and we chose each of the $\mathcal{K}_\sigma$ to be α-normal; therefore, $\sigma_1 \neq \sigma_2$. For $k \in \{1, 2\}$, since $\longrightarrow^{\kappa_k} \subseteq \longrightarrow^{\sigma_k}$, T is reducible in $\longrightarrow^{\sigma_k}$, hence T must satisfy the left-hand side of σ_k. ■

The converse (if $\mathcal{S}$ has an α-normal concrete form, then no term satisfies the left-hand sides of two different schemata in $\mathcal{S}$) isn't theoretically necessary. The trouble is that there really isn't any formal definition of what an SRS schema σ can look like: it can use arbitrary *semantic* notations, i.e., anything we expect a human audience to understand, as long as its left-hand side is a semantic polynomial, and its enumerated relation $\longrightarrow^\sigma = \longrightarrow^\mathcal{K}$ for some α-normal $\mathcal{K}$ matching the polynomial. This is exactly why we defined concrete schemata in the first place, to give us something precisely defined that we could prove theorems about. In particular, notations elsewhere in an SRS schema (other than on the left-hand side) could place rather arbitrary restrictions on reducible terms, so that two different SRS schemata in a single α-normalizable SRS might even have *identical* left-hand sides, as long as no term is reducible in the enumerated relations of both.

Lemma 13.98 If $\mathcal{S}$ is an SRS, and $T \longrightarrow^\mathcal{S} T'$, then $\text{Free}(T') \subseteq \text{Free}(T)$. ■

Proof. Suppose $\mathcal{S}$ is an SRS, and $T \longrightarrow^\mathcal{S} T'$. Let κ belong to a concrete form of $\mathcal{S}$ such that $T \longrightarrow^\kappa T'$. Then for some $f \in \mathcal{F}_\alpha$, $T, T' \parallel f$ and $\langle f(T), f(T')\rangle$ is in the induced hygienic relation of κ; and by Assumptions 13.19(b) and 13.19(c), $\text{Free}(f(T')) \subseteq \text{Free}(f(T))$ iff $\text{Free}(T') \subseteq \text{Free}(T)$. So it suffices to show $\text{Free}(T') \subseteq \text{Free}(T)$ when $\langle T, T'\rangle$ is in the induced hygienic relation of κ; suppose $\langle T, T'\rangle$ is in the induced hygienic relation of κ.

Let $\kappa = (P_0[P_1[\sum_{k=1}^{n_1} \Box_{k+n_0-1}],\ \sum_{k=2}^{n_0} \Box_{k-1}] \longrightarrow$
$P'[\sum_{k=1}^{n_1} f(x, \Box_{k+n_0-1}, P_0),\ \sum_{k=2}^{n_0} \Box_{k-1}])$,
$T = P_0[P_1[\vec{T_1}], \vec{T_0}]$, and $T' = P'[\vec{T_1'}, \vec{T_0}]$.

Consider a free occurrence of x' in T'; we will show that x' occurs free in T.

Case 1: the free occurrence of x' is in P'. By Condition 13.91(f), $\text{Free}(P') \subseteq \text{Free}(P_0[\vec{\Box}_{n_0} \backslash^1_{P_1}])$; so $x' \in \text{Free}(T)$.

Case 2: the free occurrence of x' is in $\vec{T_0}(k)$ (as it appears directly under P'). Since x' is not bound by P' at $\Box_{k+n_1}$ (else the occurrence would not be free in T'), it cannot be bound by P_0 at $\Box_{k+1}$ either (by Condition 13.91(e)). Therefore $x' \in \text{Free}(T)$.

Case 3: the free occurrence of x' is in $\vec{T_1'}(k)$. x' is not bound by P' at $\Box_k$, since its occurrence at that position is free. By Condition 13.49(e), either the occurrence is in $f(x, \vec{T_1}(k), P_0)$, or the occurrence is in a copy of some $\vec{T_0}(j)$.

Case 3a: the free occurrence of x' is in $f(x, \vec{T_1}(k), P_0)$. Either $x' \in \text{Free}(P_0)$, or there exists $x'' \in \text{Free}(\vec{T}(k))$ such that $x' = f_\text{v}(x, x'')$ (by Condition 13.49(i)). If

$x' \in \text{Free}(P_0)$ then $x' \in \text{Free}(T)$; so assume the latter. Since x' is not bound by P' at $\Box_k$, x'' is not bound by P_1 at $\Box_k$ or P_0 at $\Box_1$ (by Condition 13.91(e)). If $x'' \sqsubseteq x$, then x is not bound by P_1 (by Lemma 13.16), so f is trivial (by Condition 13.91(e)); and since f is trivial, f_{v} must be the identity function on *Vars* (by Condition 13.49(i)); so $\vec{T_1}(k) \equiv_\alpha \vec{T_1'}(k)$, and since x' isn't bound by P' at $\Box_k$, it isn't bound by P_1 at $\Box_k$ or P_0 at $\Box_1$ (by Condition 13.91(e)), and $x' \in \text{Free}(T)$. Suppose $x'' \not\sqsubseteq x$. Then $x'' = x$ (by Condition 13.49(b)), and again, since x' isn't bound by P' at $\Box_k$, it isn't bound by P_1 at $\Box_k$ or P_0 at $\Box_1$, and $x' \in \text{Free}(T)$.

Case 3b: the free occurrence of x', within $\vec{T_1'}(k)$, is in a copy of $\vec{T_0}(j)$. Since x' isn't bound by $f(x, \vec{T_1}(k), P_0)$ at $\Box_j$, it isn't bound by P_0 at $\Box_{j+1}$ (by Condition 13.49(i)), so $x' \in \text{Free}(T)$. ■

Theorem 13.99 If $\mathcal{S}$ is an SRS, then $\longrightarrow_{\mathcal{S}}^*$ is constructive. ■

Proof. Follows from Assumption 13.17(c) and Lemma 13.98. ■

Theorem 13.100 If $\mathcal{S}$ is an SRS, then $\longrightarrow^{\mathcal{S}}$ is α-closed. ■

Proof. Suppose $\mathcal{S}$ is an SRS, $f \in \mathcal{F}_\alpha$, $T_1 \longrightarrow^{\mathcal{S}} T_2$, and $T_1 \Rrightarrow_f f(T_1)$. By the definition of SRS, let $\sigma \in \mathcal{S}$ such that $T_1 \longrightarrow^\sigma T_2$, let $\mathcal{K}$ be a concrete form of σ, and let $\kappa \in \mathcal{K}$ such that $T_1 \longrightarrow^\kappa T_2$. Since $\longrightarrow^\kappa$ is, by definition, the smallest α-closed relation containing the induced hygienic relation of κ, $f(T_1) \longrightarrow^\kappa f(T_2)$ and $T_2 \Rrightarrow_f f(T_2)$. Since $\kappa \in \mathcal{K}$ and $\sigma \in \mathcal{S}$, $f(T_1) \longrightarrow^{\mathcal{S}} f(T_2)$. So $\longrightarrow^{\mathcal{S}}$ is α-closed. ■

13.4 Regularity

Definition 13.101 Suppose SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.
$\mathcal{K}$ is a *regular* concrete form of $\mathcal{S}$ if all of the following conditions hold.

(a) $\mathcal{K}$ is a concrete form of $\mathcal{S}$.

(b) $\mathcal{K}$ is α-normal.

(c) The left-hand side of every $\kappa \in \mathcal{K}$ is in general position and selective in $\longrightarrow^{\mathcal{S}}$.

(d) Every $f \in \mathcal{F}_\beta$ used in $\mathcal{K}$ distributes over $\longrightarrow_{\mathcal{S}}$, and strictly distributes over $\longrightarrow^{\mathcal{S}}$.

(e) For every $f \in \mathcal{F}_\beta$ used in $\mathcal{K}$, and every x, P, and P_0, if $f(x, P, P_0)$ is defined and P satisfies the left-hand side of some $\sigma \in \mathcal{S}$, then $f(x, P, P_0)$ satisfies the left-hand side of some $\sigma' \in \mathcal{S}$.

$\mathcal{S}$ is *regular* if both of the following conditions hold.

(f) $\mathcal{S}$ has a regular concrete form.

(g) For every $\sigma \in \mathcal{S}$ and $T \in \mathit{Terms}$, if T satisfies the left-hand side of σ, then T is reducible in $\longrightarrow^{\sigma}$. ■

Given strict distributivity over $\longrightarrow^{\mathcal{S}}$, distributivity over $\longrightarrow_{\mathcal{S}}$ simply means that $f(x, P[\vec{T}])$ never makes more than once copy of $f(x, \vec{T}(k))$.

Condition 13.101(e) supports distributivity over $\longrightarrow_{\mathcal{S}}^{\|?}$: while selectivity of left-hand sides (Condition 13.101(c)) guarantees that a top-level reduction $P_1[\vec{T_1}] \longrightarrow^{\mathcal{S}} T_2$ will not interfere with subterm reductions $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$, Condition 13.101(e) preserves this property through f so that $f(x, P_1[\vec{T_1}], \vec{T}) \longrightarrow^{\mathcal{S}} f(x, T_2, \vec{T})$ will not interfere with $f(x, \vec{T_1}(k), \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, \vec{T_2}(k), \vec{T})$.

Condition 13.101(g) allows suspending contexts (Definition 13.80) to be determined by examining just the left-hand sides of the schemata. (Garbage collection schemata are apt to violate this condition.) This also implies that the set of poly-contexts satisfying the left-hand side of each schema is closed under α-images (by Definition 13.96).

Lemma 13.102 Suppose regular SRS $\mathcal{S}$.

If κ is an element of a regular concrete form of $\mathcal{S}$, then the left-hand side of κ is decisively reducible in $\longrightarrow^{\kappa}$. ■

Proof. Suppose $\mathcal{K}$ is a regular concrete form of $\mathcal{S}$, $\kappa \in \mathcal{K}$, P is the left-hand side of κ, and $P[\vec{T}] \in \mathit{Terms}$. By definition of concrete form of an SRS, let $\sigma \in \mathcal{S}$ such that κ is in a concrete form of σ. Let π be the left-hand side of σ. Since $\mathcal{S}$ is regular, every term satisfying π is reducible in $\longrightarrow^{\sigma}$; and since κ is in a concrete form of σ, P minimally satisfies π; so $P[\vec{T}]$ is reducible in $\longrightarrow^{\sigma}$. Therefore, let $\kappa' \in \mathcal{K}$ such that $P[\vec{T}]$ is reducible in $\longrightarrow^{\kappa'}$. Let P' be the left-hand side of κ'; then P' minimally satisfies π. By definition of $\longrightarrow^{\kappa'}$, let $P'[\vec{T'}]$ be an α-image of $P[\vec{T'}]$ such that $P'[\vec{T'}]$ is reducible in the induced hygienic relation of κ'.

Since $P'[\vec{T'}]$ is an α-image of $P[\vec{T}]$, let P'_0 be an α-image of P satisfied by $P'[\vec{T'}]$ (by Theorem 13.37). P is iso (by Condition 13.91(a)), so P'_0 is iso (because it's an α-image of P); and P is in general position (because $\mathcal{K}$ is a regular concrete form of $\mathcal{S}$); therefore, every term satisfying P'_0 is an α-image of a term satisfying P (by Theorem 13.48). Further, every term satisfying P satisfies π, and every term satisfying π is reducible in $\longrightarrow^{\sigma}$ (because $\mathcal{S}$ is regular); and since $\longrightarrow^{\sigma}$ is α-closed, every term satisfying P'_0 is reducible in $\longrightarrow^{\sigma}$. But $\longrightarrow^{\sigma}$ is, by definition, the enumerated reduction relation of σ, so every term reducible in $\longrightarrow^{\sigma}$ must satisfy π. Therefore, every term satisfying P'_0 must satisfy π; and since $P'[\vec{T'}]$ satisfies P'_0 and P' minimally satisfies π, P'_0 satisfies P'.

Let $P[\vec{T_0}] \in \mathit{Terms}$ with $\mathrm{Free}(\vec{T_0}) = \{\}$ (by Condition 13.49(b) and Assumption 13.17(a)). Then $P[\vec{T_0}]$ is reducible in both $\longrightarrow^{\kappa}$ and $\longrightarrow^{\kappa'}$ (by Conditions

13.91(d) and 13.49(b)); therefore, since $\mathcal{K}$ is α-normal (part of regularity), $\kappa = \kappa'$. So $P[\vec{T}]$ is reducible in $\longrightarrow^{\kappa}$. ■

Lemma 13.103 Suppose regular SRS $\mathcal{S}$, and substitutive f.
If f is used in a regular concrete form of $\mathcal{S}$, then f distributes over $\longrightarrow_{\mathcal{S}}^{\|?}$. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$, and proceed by induction on the size of T_1. Suppose $f(x, T_1, P_0, \vec{T})$ is defined. By the definition of $\longrightarrow_{\mathcal{S}}^{\|?}$ (Definition 13.75), let $T_1 = P_1[\vec{T_1}]$ for nontrivial P_1, and $P_1[\vec{T_2}] \in \mathit{Terms}$, such that each $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ and either $P_1[\vec{T_2}] \equiv_\alpha T_2$ or $P_1[\vec{T_2}] \longrightarrow^{\mathcal{S}} T_2$.

Case 1: $P_1[\vec{T_2}] \equiv_\alpha T_2$. The conclusion follows immediately from the inductive hypothesis, the homomorphic behavior of substitutive functions (Condition 13.49(f)), and α-closure (Theorem 13.100).

Case 2: $P_1[\vec{T_2}] \longrightarrow^{\mathcal{S}} T_2$. Let $\sigma \in \mathcal{S}$ such that $P_1[\vec{T_2}] \longrightarrow^{\sigma} T_2$. Let $\mathcal{K}$ be a regular concrete form of $\mathcal{S}$ (Condition 13.101(f)), and $\kappa \in \mathcal{K}$ such that $P_1[\vec{T_2}] \longrightarrow^{\kappa} T_2$. $\longrightarrow^{\kappa} \subseteq \longrightarrow^{\sigma}$ (by Condition 13.101(b)). Let P be the left-hand side of κ, and P_1' the α-image of P satisfied by $P_1[\vec{T_2}]$. Since P is iso, in general position, and selective in $\longrightarrow^{\mathcal{S}}$ (Conditions 13.91(a) and 13.101(c)), P_1' is selective in $\longrightarrow^{\mathcal{S}}$ (by Lemma 13.84(a)); therefore, assume without loss of generality that P_1 satisfies P_1'. P is decisively reducible in $\longrightarrow^{\kappa}$ (by Lemma 13.102), therefore P_1 is decisively reducible in $\longrightarrow^{\kappa}$ (by Lemma 13.84(b)).

From the previous case, $f(x, P_1[\vec{T_1}], P_0, \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, P_1[\vec{T_2}], P_0, \vec{T})$. By the homomorphic behavior of substitutive functions, $f(x, P_1[\vec{T_1}], P_0)$ is composed from $f(x, P_1, P_0)$ and the $f(x, \vec{T_1}(k), P_0)$, while $f(x, P_1[\vec{T_1}], P_0)$ is composed from $f(x, P_1, P_0)$ and the $f(x, \vec{T_2}(k), P_0)$. By regularity Condition 13.101(e), $f(x, P_1, P_0)$ satisfies the left-hand side of some $\sigma' \in \mathcal{S}$. Because f distributes over $\longrightarrow^{\mathcal{S}}$, $f(x, P_1[\vec{T_2}], P_0, \vec{T}) \longrightarrow^{\mathcal{S}?} f(x, T_2, P_0, \vec{T})$. Therefore, by definition of $\longrightarrow_{\mathcal{S}}^{\|?}$ (Definition 13.75), $f(x, P_1[\vec{T_1}], P_0, \vec{T}) \longrightarrow^{\mathcal{S}} f(x, T_2, P_0, \vec{T})$. ■

Theorem 13.104 (Church–Rosser-ness)
Suppose regular SRS $\mathcal{S}$.
$\longrightarrow^{\mathcal{S}}$ is Church–Rosser. ■

Proof. We will show that $\longrightarrow_{\mathcal{S}}^{\|?}$ has the diamond property; therefore, by Theorem 13.100 and Lemma 13.77, $\longrightarrow^{\mathcal{S}}$ is Church–Rosser.

Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$. Assume, inductively, that the result holds for all proper subterms of T_1. (Eventually this induction comes down to T_1 with no proper subterms, in which case the result does hold for all proper subterms of T_1.)

Case 1: neither $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ nor $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ involves a top-level $\mathcal{S}$-reduction. Let P be iso minimal nontrivial such that $T_1 = P[\vec{T_1}]$, $T_2 \equiv_\alpha P[\vec{T_2}]$, and $T_3 \equiv_\alpha P[\vec{T_3}]$. Then for all $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ and $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_3}(k)$. By the inductive assumption, for each k let $\vec{T_4}(k) \in \mathit{Terms}$ such that $\vec{T_2}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_4}(k)$ and $\vec{T_3}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_4}(k)$. Let $T_4 = P[\vec{T_4}]$ (by Theorem 13.99). By the definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, $P[\vec{T_2}] \longrightarrow_{\mathcal{S}}^{\|?} T_4$ and $P[\vec{T_3}] \longrightarrow_{\mathcal{S}}^{\|?} T_4$; therefore, since $\longrightarrow^{\mathcal{S}}$ is α-closed (by Theorem 13.100), $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_4$ and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_4$, neither involving a top-level $\longrightarrow^{\mathcal{S}}$-reduction.

Case 2: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ does not involve a top-level $\mathcal{S}$-reduction, and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ involves *only* a top-level $\mathcal{S}$-reduction (i.e., $T_1 \longrightarrow^{\mathcal{S}} T_3$). Let $\sigma \in \mathcal{S}$ such that $T_1 \longrightarrow^{\sigma} T_3$; let $\mathcal{K}$ be an α-normal concrete form of σ; let $\kappa \in \mathcal{K}$ such that $T_1 \longrightarrow^{\kappa} T_3$; and let κ be as given in Schema 13.92. Let P be the left-hand side of κ. Since T_1 is reducible in $\longrightarrow^{\kappa}$, let $T_1 \equiv_\alpha P[\vec{T_1}]$ such that $P[\vec{T_1}]$ is reducible in the induced hygienic relation of κ to $P'[\vec{T_3}]$. Since $\mathcal{K}$ is α-normal, $P'[\vec{T_3}] \equiv_\alpha T_3$. Since $\longrightarrow^{\mathcal{S}}$ is α-closed (by Theorem 13.100), $\longrightarrow_{\mathcal{S}}^{\|?}$ is α-closed; and since $\mathcal{S}$ is regular, P is selective (Condition 13.101(c)); so let $T_2 \equiv_\alpha P[\vec{T_2}]$ such that for all $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$.

By the internal structure of κ, for $2 \leq k \leq n_0$, $\vec{T_3}(k-1) = \vec{T_1}(k-1)$; and for $1 \leq k \leq n_1$, $\vec{T_3}(k+n_0-1) = f(x, \vec{T_1}(k+n_0-1), P_0, \sum_{j=2}^{n_0} \vec{T_1}(j-1))$. Since $\mathcal{S}$ is regular, f distributes over $\longrightarrow_{\mathcal{S}}^{\|?}$ (Lemma 13.103); so $f(x, \vec{T_1}(k+n_0-1), P_0, \sum_{j=2}^{n_0} \vec{T_1}(j-1)) \longrightarrow_{\mathcal{S}}^{\|?} f(x, \vec{T_2}(k+n_0-1), P_0, \sum_{j=2}^{n_0} \vec{T_1}(j-1))$ (for suitable choice of $\vec{T_2}$). By Condition 13.49(e), $(f(x, \vec{T_1}(k+n_0-1), P_0))[\sum_{j=2}^{n_0} \vec{T_1}(j-1)] \longrightarrow_{\mathcal{S}}^{\|?} (f(x, \vec{T_2}(k+n_0-1), P_0))[\sum_{j=2}^{n_0} \vec{T_1}(j-1)]$. By Condition 13.101(e), the schema left-hand sides exercised by this relation are all independent of the $\sum_{j=2}^{n_0} \vec{T_1}(j-1)$; therefore, by definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, $f(x, \vec{T_1}(k+n_0-1), P_0, \sum_{j=2}^{n_0} \vec{T_1}(j-1)) \longrightarrow_{\mathcal{S}}^{\|?} f(x, \vec{T_2}(k+n_0-1), P_0, \sum_{j=2}^{n_0} \vec{T_2}(j-1))$.

Let $\vec{T_4} = \sum_k \begin{cases} \vec{T_2}(k) & \text{if } k < n_0 \\ f(x, \vec{T_2}(k), P_0, \sum_{j=2}^{n_0} \vec{T_2}(j-1)) & \text{otherwise} \end{cases}$. Let $T_4 = P'[\vec{T_4}] \in \mathit{Terms}$ (by Theorem 13.99). Since each $\vec{T_3}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_4}(k)$, by definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, $P'[\vec{T_3}] \longrightarrow_{\mathcal{S}}^{\|?} P'[\vec{T_4}]$; and by the structure of κ, $P[\vec{T_2}] \longrightarrow^{\mathcal{S}} P'[\vec{T_4}]$. Therefore, since $\longrightarrow^{\mathcal{S}}$ is α-closed, $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_4$ and $T_2 \longrightarrow^{\mathcal{S}} T_4$.

Case 3: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ does not involve a top-level $\mathcal{S}$-reduction, and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ does involve a top-level $\mathcal{S}$-reduction. By definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, let $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3' \longrightarrow^{\mathcal{S}} T_3$ such that $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3'$ does not involve a top-level $\mathcal{S}$-reduction. By Case 1, let $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_4'$ and $T_3' \longrightarrow_{\mathcal{S}}^{\|?} T_4'$ such that neither of these relations involves a top-level

$\longrightarrow^{\mathcal{S}}$-reduction. By Case 2, let $T_4' \longrightarrow^{\mathcal{S}} T_4$ and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_4$. By definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, since $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_4' \longrightarrow^{\mathcal{S}} T_4$, the former doesn't involve a top-level $\longrightarrow^{\mathcal{S}}$-reduction, and the left-hand sides of all concrete schemata are selective in $\longrightarrow^{\mathcal{S}}$, $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_4$.

Case 4: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ both involve top-level $\mathcal{S}$-reductions. By definition of $\longrightarrow_{\mathcal{S}}^{\|?}$, let $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2' \longrightarrow^{\mathcal{S}} T_2$ and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3' \longrightarrow^{\mathcal{S}} T_3$ such that each $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_k'$ does not involve a top-level $\longrightarrow^{\mathcal{S}}$-reduction. By Case 1, let $T_2' \longrightarrow_{\mathcal{S}}^{\|?} T_4'$ and $T_3' \longrightarrow_{\mathcal{S}}^{\|?} T_4'$ such that neither of these relations involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction. By Case 2, for $k \in \{2,3\}$ let $T_k \longrightarrow_{\mathcal{S}}^{\|?} T_{k,4}$ and $T_4' \longrightarrow^{\mathcal{S}} T_{k,4}$. Since $\mathcal{S}$ is regular, let $\mathcal{K}$ be an α-normal concrete form of $\mathcal{S}$; then there is at most one $\kappa \in \mathcal{K}$ such that T_4' is $\longrightarrow^{\kappa}$-reducible, so $T_{2,4} \equiv_\alpha T_{3,4}$. So $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_{k,4}$ and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_{k,4}$. ■

Theorem 13.105 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.
$\longmapsto_{\mathcal{S}}^{\mathcal{E}}$ is deterministic up to $\equiv_\alpha$. ■

Proof. Suppose $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_2$ and $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3$; we will show $T_2 \equiv_\alpha T_3$. Let E_2, E_3 be evaluation contexts, $T_1 \equiv_\alpha E_2[T_1'] \equiv_\alpha E_3[T_1'']$, $T_2 \equiv_\alpha E_2[T_2']$, and $T_3 \equiv_\alpha E_3[T_3'']$, such that $T_1' \longrightarrow^{\mathcal{S}} T_2'$ and $T_1'' \longrightarrow^{\mathcal{S}} T_3''$, and $E_2[T_1']$, $E_2[T_2']$, $E_3[T_1'']$, and $E_3[T_3'']$ are in general position (by Theorems 13.100, 13.87, and 13.44, and Lemma 13.85(c)).

Suppose $E_2 = \Box$. Then $E_3[T_1''] \equiv_\alpha T_1'$, so $E_3[T_1'']$ is reducible in $\longrightarrow^{\mathcal{S}}$. Since $\mathcal{S}$ is an SRS, let P satisfied by $E_3[T_1'']$ be the left-hand side of some κ in a concrete form of $\mathcal{S}$; by the definition of regular SRS, P is selective in $\longrightarrow^{\mathcal{S}}$, and by Lemma 13.102, P is decisively reducible in $\longrightarrow^{\mathcal{S}}$. Therefore, $E_3 = \Box$, and $T_1' \equiv_\alpha T_1''$. Since $\mathcal{S}$ is regular, it has an α-normal concrete form, in which $T_1' \longrightarrow^{\mathcal{S}} T_2'$ and $T_1' \longrightarrow^{\mathcal{S}} T_3''$ must use the same concrete schema. So $T_2' \equiv_\alpha T_3''$.

On the other hand, suppose $E_2 = (P_2[\vec{P_2}])[\Box]$ such that $P_2 \in \mathcal{E}$ (possible by definition of evaluation order, Definition 13.78), and let k have the properties enumerated in the definition of $R, \mathcal{E}$-evaluation context (Definition 13.83). If $E_3 = \Box$ then, reasoning symmetrically to the above, $E_2 = \Box$; so $E_3 \neq \Box$. Let $E_3 = (P_3[\vec{P_3}])[\Box]$ such that $P_3 \in \mathcal{E}$. For all $j < k$, $\vec{P_2}(j)$ is an $\mathcal{S}$-evaluation normal form; therefore, in term $E_3[T_1'']$, the α-image of $\vec{P_2}(j)$ is also an $\mathcal{S}$-evaluation normal form (by Lemma 13.85(b)); and an $\mathcal{S}$-evaluation normal form can't be reducible in $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$, so the α-image of $\vec{P_2}(j)$ doesn't contain the meta-variable occurrence, and the α-image of $\vec{P_2}(j)$ is $\vec{P_3}(j)$. So $\vec{P_3}(j)$ is an $\mathcal{S}$-evaluation normal form. Similarly, since the subterm of $E_3[T_1'']$ at $\Box_k$ in P_3 is reducible in $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$, it isn't an $\mathcal{S}$-evaluation normal form; so since E_3 is an $\mathcal{S}, \mathcal{E}$-evaluation context, so is $(\vec{P_3}(k))[\Box]$. Since $(\vec{P_2}(k))[\Box]$ is an $R, \mathcal{E}$-evaluation context in general position in which the meta-variable has strictly lesser depth than in E_2, and $(\vec{P_3}(k))[\Box]$ is its α-image in general position, by induction on the depth of the meta-variable, Q.E.D. ■

Plotkin's development of a standardization theorem for λ_v-calculus ([Plo75]) depended on starting with an arbitrary sequence of "parallel reduction" steps —generalized into our treatment as $\longrightarrow_{\mathcal{S}}^{\|?}$ steps— and then unpacking these parallel steps, one at a time from right to left, into a standard reduction sequence.

The parallel reduction, in essence, chooses a subset of the possible β-redexes in the term, and exercises all the selected redexes at once, *from the bottom up*. Exercising the lower redexes doesn't sabotage those above them because of the selectivity condition in the definition of regular SRS (Condition 13.101(c)). Standard reduction order contrasts with this bottom-up order by consistently exercising high-level redexes before those below them, a rearrangement of ordering that is possible because of the homomorphic and copying behaviors of substitutive functions, and the distributivity condition in the definition of regular SRS. During the rearrangement, though, the number of individual redexes to be exercised may increase dramatically, because exercising a high-level redex may make many copies of some of its subterms, so that if the parallel reduction step exercised a redex in any of those copied subterms, the corresponding redexes in all the copies must be exercised separately by an equivalent standard reduction sequence. Plotkin's technique uses induction on the number of these redexes, for which purpose he defines a "reduction size" for each parallel reduction step that provides an upper bound on the number of redexes that will be exercised after the rearrangement; and central to this size metric is the number of times a subterm is copied during the exercise of a redex above it. In λ_v-calculus reduction $(\lambda x.T)V \longrightarrow^v T[x \leftarrow V]$, redexes in T are copied exactly once, while redexes in V are copied once for each free occurrence of x in T. For our abstract treatment of SRSs, the number of copies is more involved, depending on the structure of the schema and the homomorphic part of the behavior of its substitutive function.

Definition 13.106 Suppose poly-context P.
$\#(\Box_k, P)$ is the number of occurrences of $\Box_k$ in P. ■

In substitutive function application $f(x, T, P_0, \vec{T})$, the number of copies of each $\vec{T}(k)$ is just $\#(\Box_k, f(x, T, P_0))$.

Definition 13.107 Suppose SRS concrete schema κ is as given in Schema 13.92, P is the left-hand side of κ, and term $P[\vec{T}]$ is reducible in the induced hygienic relation of κ.

For $n_0 \leq h \leq ar(P)$, $\#(h, \kappa, P, \vec{T}) = \#(\Box_h, P')$.
For $1 \leq h < n_0$,

$$\#(h, \kappa, P, \vec{T}) = \#(\Box_h, P'\left[\sum_k \left\{ \begin{array}{ll} \Box_k & \text{if } k < n_0 \\ (f(x, \vec{T}(k), P_0))[\sum_{j=2}^{n_0} \Box_{j-1}] & \text{otherwise} \end{array} \right]\right).$$

Suppose further poly-context P_2 and term $P_2[\vec{T_2}]$ are α-images of, respectively, P and $P[\vec{T}]$ (so that $P_2[\vec{T_2}]$ is reducible in the α-closure of κ). For $1 \leq i \leq ar(P_2)$, $\#(h, \kappa, P_2, \vec{T_2}) = \#(h, \kappa, P, \vec{T})$. ■

The generalization to P_2 is unambiguous since the behavior of f is continuous under α-renaming (Conditions 13.49(c) and 13.49(d)).

If $P_1[\vec{T_1}] \longrightarrow^{\mathcal{S}} T_2$ and T is a subterm of $\vec{T_1}(k)$, the number of images of T in T_2 —that is, the number of distinct subterms of T_2 that could be altered if the occurrence of T in $\vec{T_1}(k)$ were replaced by some T'— is $\#(k, \kappa, P_1, \vec{T_1})$ (by the behavior of substitutive functions, specifically Conditions 13.49(e) and 13.49(g)).

Definition 13.108 Suppose SRS $\mathcal{S}$.

$T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ *with reduction size* n if $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ and any of the following conditions holds.

(a) $T_1 = P[\vec{T_1}]$ for nontrivial P;
for all $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k;
$P[\vec{T_2}] \equiv_\alpha T_2$; and
$n = \sum_{k=1}^{ar(P)}(n_k \times \#(\Box_k, P))$.

(b) $T_1 = P[\vec{T_1}]$ for nontrivial P;
for all $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k;
$P[\vec{T_2}] \longrightarrow^{\mathcal{S}} T_2$ via SRS concrete schema κ in a regular concrete form of $\mathcal{S}$;
for all $1 \leq k \leq ar(P)$, $\#(k, \kappa, P, \vec{T_2})$ is defined; and
$n = 1 + \sum_{k=1}^{ar(P)}(n_k \times \#(k, \kappa, P, \vec{T_2}))$. ■

Reduction size of a given step is not required to be unique: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n and with reduction size m doesn't necessarily imply $n = m$.

Lemma 13.109 Suppose regular SRS $\mathcal{S}$.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$, then there exists n such that $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n. ■

Proof. Suppose the proposition holds for all proper subterms of T_1. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$. Following the definition of $\longrightarrow_{\mathcal{S}}^{\|?}$ (Definition 13.75), there are two cases, depending on whether $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction.

Case 1: no top-level $\longrightarrow^{\mathcal{S}}$-reduction. Follows from the definition of reduction size without a top-level reduction (Criterion 13.108(a)), and the inductive hypothesis.

Case 2: top-level $\longrightarrow^{\mathcal{S}}$-reduction. Follows from the definition of reduction size with a top-level reduction (Criterion 13.108(b)), and the inductive hypothesis. ■

When $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, one can always choose the nontrivial P in the definition to be *iso* (because when there isn't a top-level reduction, the subterm occurrences are just added together, and when there *is* a top-level reduction, the definition of $\#(k, \kappa, P, \vec{T_2})$ requires P to be an α-image of the left-hand side from Scheme 13.92, which is iso by Condition 13.91(a)).

Lemma 13.110 Suppose regular SRS $\mathcal{S}$, and substitutive f.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, f is used in a regular concrete form of $\mathcal{S}$, and $f(x, T_1, P_0, \vec{T})$ and $f(x, T_2, P_0, \vec{T})$ are defined, then $f(x, T_1, P_0, \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, T_2, P_0, \vec{T})$ with reduction size $\leq n$. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, f is used in a regular concrete form of $\mathcal{S}$, and $f(x, T_1, P_0, \vec{T})$ and $f(x, T_2, P_0, \vec{T})$ are defined. Proceed by induction on n. There are two cases, depending on whether or not $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction (Criterion 13.108(a) or 13.108(b)).

Case 1: no top-level $\longrightarrow^{\mathcal{S}}$-reduction. Let $T_1 = P_1[\vec{T_1}]$ for $P_1 \in \mathcal{E}$; for $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k; $P_1[\vec{T_2}] \equiv_\alpha T_2$; and $n = \sum_{k=1}^{ar(P_1)} n_k$. There are three sub-cases, depending on how many of the n_k are non-zero.

Case 1a: all of the n_k are zero. Then $n = 0$, $T_1 \equiv_\alpha T_2$, and the proposition is just invariance of f over $\equiv_\alpha$ (Condition 13.49(c)).

Case 1b: at least two of the n_k are non-zero. Then the proposition follows immediately from the inductive hypothesis.

Case 1c: exactly one of the n_k is non-zero. Then $n = n_k$, and we can consider just $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n, which involves a smaller term on the left-hand side. By doing this repeatedly, we can reduce the problem either to Case 1b above, or to Case 2 below.

Case 2: top-level $\longrightarrow^{\mathcal{S}}$-reduction. Let $T_1 = P_1[\vec{T_1}]$ for nontrivial P_1; for $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k; $P_1[\vec{T_2}] \longrightarrow^{\mathcal{S}} T_2$ via SRS concrete schema κ in a regular concrete form of $\mathcal{S}$; for $1 \leq k \leq ar(P_1)$, $\#(k, \kappa, P_1, \vec{T_2})$ be defined; and $n = 1 + \sum_{k=1}^{ar(P_1)}(n_k \times \#(k, \kappa, P_1, \vec{T_2}))$. For the $\#(k, \kappa, P_1, \vec{T_2})$ to be defined, P_1 must be an α-image of the left-hand side of κ; so P_1 is iso.

Consider the structure of the reduction $f(x, T_1, P_0, \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, T_2, P_0, \vec{T})$, which parallels that of $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$. For suitable $\vec{T_3}$ element-wise $\equiv_\alpha$ to the $\vec{T_2}$, we have $f(x, P_1[\vec{T_1}], P_0, \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, P_1[\vec{T_3}], P_0, \vec{T}) \longrightarrow^{\mathcal{S}} f(x, T_2, P_0, \vec{T})$, via subterm reductions $f(x, \vec{T_1}(k), P_0, \vec{T}) \longrightarrow_{\mathcal{S}}^{\|?} f(x, \vec{T_2}(k), P_0, \vec{T})$ with reduction size $\leq n_k$ (by inductive hypothesis). Let κ_f be the SRS concrete schema used, in the same regular concrete form of $\mathcal{S}$ as κ, for the top-level step $f(x, P_1[\vec{T_3}], P_0, \vec{T}) \longrightarrow^{\mathcal{S}} f(x, T_2, P_0, \vec{T})$. The left-hand side of κ_f is selective (by Condition 13.101(c)), therefore so are its α-images (by Lemma 13.84(a)), and one such α-image must occur in $f(x, T_1, P_0, \vec{T})$. The actual redexes within the $\vec{T_1}$ must map to actual redexes within $f(x, T_1, P_0, \vec{T})$, which are entirely independent of this selective part of T_1. The reduction size from $f(x, T_1, P_0, \vec{T})$ to $f(x, T_2, P_0, \vec{T})$ is then, by definition, one plus the reduction size for each actual redex times the number of images of that redex projected by κ_f. But the number of images of any one of these actual redexes in $f(x, T_2, P_0, \vec{T})$ cannot be larger than the

number of its images in T_2, because $f(x, \Box, P_0, \vec{T})$ is guaranteed not to make multiple copies of any subterm of $\Box$ (by Condition 13.49(g)). Therefore, Q.E.D. ■

Lemma 13.111 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n via Criterion 13.108(b), then there exists T_3 such that $T_1 \longrightarrow^{\mathcal{S}} T_3$ and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size $\leq n-1$. ■

Proof. Suppose the proposition holds for all smaller n, and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n via Criterion 13.108(b). Let $T_1 = P[\vec{T_1}]$ for nontrivial P; for $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k; $P[\vec{T_2}] \longrightarrow^{S} T_2$ via SRS concrete schema κ in a regular concrete form of $\mathcal{S}$; and $n = 1 + \sum_{k=1}^{ar(P)}(n_k \times \#(k, \kappa, P, \vec{T_2}))$.

Let f be the substitutive function used by κ; and let P_0, P_1, and P' be the poly-contexts on the left- and right-hand sides of κ as in Schema 13.92. For the $\#(k, \kappa, P, \vec{T_2})$ to be defined, P must be an α-image of the left-hand side of κ. Since all the reduction relations involved are α-closed, and the definition of $\#(k, \kappa, P, \vec{T_2})$ is invariant across α-images, assume without loss of generality that P is the left-hand side of κ.

If any of the $\#(k, \kappa, P, \vec{T_2})$ is zero, then that subterm reduction makes no contribution to n, and it suffices to prove the proposition when no reduction is performed on that subterm, i.e., when $n_k = 0$; so assume without loss of generality that for all k, if $\#(k, \kappa, P, \vec{T_2}) = 0$ then $n_k = 0$.

Proceed by considering what would happen if parallel step $P[\vec{T_1}] \longrightarrow_{\mathcal{S}}^{\|?} T_2$ were modified by *not reducing* some one of the subterms, $\vec{T_1}(k)$. This modification of the parallel step is possible for all k, because every α-image of the left-hand side of κ is selective in $\longrightarrow^{\mathcal{S}}$ (by Condition 13.101(c) and Lemma 13.84(a)) and decisively $\longrightarrow^{\mathcal{S}}$-reducible (by Lemma 13.102); and if $n_k > 0$, then the resulting parallel step with have strictly lesser reduction size (because then also $\#(k, \kappa, P, \vec{T_2}) \neq 0$), so the inductive hypothesis will apply to that resulting parallel step. Moreover, by this selectivity, the top-level reduction in the parallel step will be via κ.

Case 1: all of the n_k are zero. Then $P[\vec{T_1}] \equiv_\alpha P[\vec{T_2}]$; $T_1 \longrightarrow^{\mathcal{S}} T_2$; and $T_2 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size 0.

Case 2: for some $k < ar(P_0)$, $n_k \geq 1$. Let $\vec{T_2'} = \vec{T_2}\backslash^{k}_{\vec{T_1}(k)}$. Let $P[\vec{T_2'}] \longrightarrow^{\mathcal{S}} T_2'$; then $P[\vec{T_1}] \longrightarrow_{\mathcal{S}}^{\|?} T_2'$ with reduction size $n - (n_k \times \#(k, \kappa, P, \vec{T_2}))$. By the inductive hypothesis, let $P[\vec{T_1}] \longrightarrow^{\mathcal{S}} T_3$ via κ, and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2'$ with reduction size $\leq n - 1 - (n_k \times \#(k, \kappa, P, \vec{T_2'}))$. Since the terms being substituted into by κ are the same in $\vec{T_2'}$ as in $\vec{T_2}$ (namely, $\vec{T_2}(j)$ for $j \geq ar(P_0)$), by definition $\#(k, \kappa, P, \vec{T_2'}) = \#(k, \kappa, P, \vec{T_2})$ (Definition 13.107). So the only difference between T_2' and T_2 is that $\#(k, \kappa, P, \vec{T_2})$

verbatim copies of $\vec{T_1}(k)$ in T_2' are replaced by $\vec{T_2}(k)$ in T_2; and each of these verbatim copies is an image through $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2'$ of a verbatim copy in T_3. Let $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_3'$ by reducing all these verbatim copies of $\vec{T_1}(k)$ to $\vec{T_2}(k)$. As these verbatim copies are projected through $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2'$, they are invariant (by the hygiene constraint imposed by the definition of induced hygienic relation, Definition 13.95); and the same is true of the copies of $\vec{T_2}(k)$ in T_3' (by Lemma 13.98), so $T_3' \longrightarrow_{\mathcal{S}}^{\|?} T_2$, with the same reduction size as $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2'$. As they do not interfere with each other, the two parallel steps $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_3' \longrightarrow_{\mathcal{S}}^{\|?} T_2$ can be merged into one, increasing the reduction size of the second step by exactly n_k for each image of $\vec{T_2}$ that reaches T_2; so $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ with reduction size $\leq n-1$.

Case 3: for all $i < ar(P_0)$, $n_i = 0$, but for some $k \geq ar(P_0)$, $n_k \geq 1$. Let $T_1 = P[\vec{T_1}] \longrightarrow^{\mathcal{S}} P'[\vec{T_1'}]$ and $P[\vec{T_2}] \longrightarrow^{\mathcal{S}} P'[\vec{T_2'}] \equiv_\alpha T_2$. From the structure of κ, for $i < ar(P_0)$, $\vec{T_1'}(i) \equiv_\alpha \vec{T_1}(i)$ and $\vec{T_2'}(i) \equiv_\alpha \vec{T_2}(i)$, while for $i \geq ar(P_0)$, $\vec{T_1'}(i) \equiv_\alpha f(x, \vec{T_1}(i), P_0, \sum_{j<ar(P_0)} \vec{T_1}(j))$ and $\vec{T_2'}(i) \equiv_\alpha f(x, \vec{T_2}(i), P_0, \sum_{j<ar(P_0)} \vec{T_2}(j))$ (assuming, without loss of generality, that $\vec{T_1}(i)$ and $\vec{T_2}(i)$ have the correct form for f to be defined on them per Condition 13.49(a)). For $i < ar(P_0)$, $\vec{T_1'}(i) \equiv_\alpha \vec{T_2}(i) = \vec{T_2'}(i)$ (because, by supposition, $n_i = 0$). For $i \geq ar(P_0)$, $f(x, \vec{T_1}(i), P_0, \sum_{j<ar(P_0)} \vec{T_1}(j)) \longrightarrow_{\mathcal{S}}^{\|?} f(x, \vec{T_2}(i), P_0 \sum_{j<ar(P_0)} \vec{T_1}(j))$ with reduction size $\leq n_k$ (by Lemma 13.110). Therefore, $P'[\vec{T_1'}] \longrightarrow_{\mathcal{S}}^{\|?} P'[\vec{T_2'}]$ with reduction size $n-1$. ■

Lemma 13.112 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, then there exists an $\mathcal{S}, \mathcal{E}$-standard reduction sequence $\vec{T}$ from T_1 to T_2 with arity $\leq n+1$. ■

That is, the reduction size is an upper bound on the number of reduction steps needed for a standard reduction sequence (the number of terms in the sequence being one more than the number of reduction steps).

Proof. Proceed by induction on n.

Suppose the proposition holds for all smaller n, and $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n. Following the definition of reduction size, there are two cases, depending on whether $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction.

Case 1: no top-level $\longrightarrow^{\mathcal{S}}$-reduction. Let $T_1 = P_1[\vec{T_1}]$ for $P_1 \in \mathcal{E}$; for $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k; $P_1[\vec{T_2}] \equiv_\alpha T_2$; and $n = \sum_{k=1}^{ar(P_1)} n_k$. There are three sub-cases, depending on how many of the n_k are non-zero.

Case 1a: all of the n_k are zero. Then $n = 0$, $T_1 \equiv_\alpha T_2$, and a reduction sequence with arity 1 suffices.

Case 1b: at least two of the n_k are non-zero. Then all of the n_k are less than n, so by the inductive hypothesis, for each $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ there exists an $\mathcal{S},\mathcal{E}$-standard reduction sequence from $\vec{T_1}(k)$ to $\vec{T_2}(k)$ with arity $\leq n_k + 1$. Since $\longrightarrow_{\mathcal{S}}^{\|?}$ is constructive (by Theorem 13.99), we can perform these sequences of reductions on the subterms of $T_1 = P_1[\vec{T_1}]$, reducing each subterm $\vec{T_1}(k)$ to $\vec{T_2}(k)$ before beginning to reduce $\vec{T_1}(k+1)$; call this sequence $\vec{T}$. $\vec{T}$ starts with T_1, ends with $P_1[\vec{T_2}] \equiv_\alpha T_2$, has arity $\leq n+1$, and is an $\mathcal{S},\mathcal{E}$-standard reduction sequence (by Criterion 13.88(c)).

Case 1c: exactly one of the n_k is non-zero. Then $n = n_k$, and we can consider just $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n, which involves a smaller term on the left-hand side. By doing this repeatedly, we can reduce the problem either to Case 1b above, or to Case 2 below.

Case 2: top-level $\longrightarrow^{\mathcal{S}}$-reduction. Follows immediately from Lemma 13.111, the inductive step, and Criterion 13.88(b). ■

Lemma 13.113 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, and T_2 is an $\mathcal{S}$-evaluation normal form, then there exists T_3 such that $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}*} T_3$, T_3 is an $\mathcal{S}$-evaluation normal form, and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size $\leq n$. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, and T_2 is an $\mathcal{S}$-evaluation normal form. If $n = 0$, the result is trivial; so suppose $n \geq 1$, and the proposition holds for all smaller n.

If T_1 is an $\mathcal{S}$-evaluation normal form, then the result follows immediately from reflexivity of $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$. Suppose T_1 is not an $\mathcal{S}$-evaluation normal form.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction, the result follows immediately from Lemma 13.111 and the inductive step. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n does not involve a top-level $\longrightarrow^{\mathcal{S}}$-reduction.

Following Criterion 13.108(a), let $T_1 = P_1[\vec{T_1}]$ for $P_1 \in \mathcal{E}$; for all $1 \leq k \leq ar(P_1)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size n_k; $P_1[\vec{T_2}] \equiv_\alpha T_2$; and $n = \sum_{k=1}^{ar(P_1)} n_k$.

If P_1 were suspending in $\longrightarrow^{\mathcal{S}}$, then T_1 would be an $\mathcal{S}$-evaluation normal form (per Definition 13.81); so P_1 is not suspending. Therefore, all of the $\vec{T_2}$ are $\mathcal{S}$-evaluation normal forms, and at least one of the $\vec{T_1}$ is not an $\mathcal{S}$-evaluation normal form. Let k be the smallest integer such that $\vec{T_1}(k)$ is not an $\mathcal{S}$-evaluation normal form.

It suffices to show that there exists an $\mathcal{S}$-evaluation normal form T_3 such that $\vec{T_1}(k) \longmapsto_{\mathcal{S}}^{\mathcal{E}*} T_3$, and $T_3 \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size $\leq n_k$. For then, because all the $\vec{T_1}$ with indices less than k are $\mathcal{S}$-evaluation normal forms, $P_1[\vec{T_1}] \longmapsto_{\mathcal{S}}^{\mathcal{E}*} P_1[\vec{T_1}\backslash_{T_3}^{k}]$; and $P_1[\vec{T_1}\backslash_{T_3}^{k}] \longrightarrow_{\mathcal{S}}^{\|?} P_1[\vec{T_2}]$ with reduction size $\leq n$. So we have the same n and P_1, but fewer non-normal subterms, and by induction on the number of non-normal subterms, we're done.

If $n_k < n$, the sufficient condition follows from the inductive hypothesis. Otherwise, we have the same reduction size with a smaller left-hand term T_1, and by induction on the size of T_1 we can reduce the problem to some other case already solved. ■

Definition 13.114 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.

A poly-context P is *locally $\mathcal{S},\mathcal{E}$-irregular* if there exist
poly-contexts $P_1, \vec{P_1}$ with $P_1 \in \mathcal{E}$ and $P = P_1[\vec{P_1}]$,
term T that is not an $\mathcal{S}$-evaluation normal form,
and integers j, k
such that T satisfies $\vec{P_1}(j),\ j < k$, and $\vec{P_1}(k)$ is nontrivial.

A poly-context P is *$\mathcal{S},\mathcal{E}$-regular* if none of its branches are locally $\mathcal{S},\mathcal{E}$-irregular.

$\mathcal{E}$ is *$\mathcal{S}$-regular* if for every $\sigma \in \mathcal{S}$, every poly-context P minimally satisfying the left-hand side of σ is $\mathcal{S},\mathcal{E}$-regular. ■

Without $\mathcal{S}$-regularity (or something similar to it), there would be no standardization theorem for regular SRSs. To see why, consider a combination of $\mathcal{S}$ and $\mathcal{E}$ for which the standardization theorem is *false*: ordinary call-by-name λ-calculus with right-to-left evaluation. Here, $\mathcal{S}$ is the singleton set $\{((\lambda x.T_1)T_2) \longrightarrow T_1[x \leftarrow T_2]\}$, and $(\Box_2\, \Box_1) \in \mathcal{E}$. The difficulty is that a standard reduction sequence, as we have defined it (Definition 13.88), is always a concatenation of an evaluation sequence (via $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$) followed by standard reductions of subterms. Once the sequence has left its evaluation phase, it cannot decide later to do a top-level reduction after all. For this $\mathcal{S}$ and $\mathcal{E}$, when reducing a term of the form $(T_2\, T_1)$, if we want to do a top-level reduction, and the operator T_2 is not yet of the form $(\lambda x.\Box)$, we have to first get it into that form by using $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$ on the whole term $(T_2\, T_1)$. Unfortunately, because we are using right-to-left evaluation order $\mathcal{E}$, $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$ can't reduce the left-hand subterm until and unless it first succeeds in reducing the right-hand subterm to an $\mathcal{S}$-evaluation normal form (such as a value). So a term such as

$$(((\lambda \mathrm{z}.(\lambda \mathrm{y}.\mathrm{z}))\,\mathrm{x})\,((\lambda \mathrm{x}.(\mathrm{xx}))(\lambda \mathrm{x}.(\mathrm{xx})))) \tag{13.115}$$

is reducible to x by means of $\longrightarrow_{\mathcal{S}}^{*}$, but not by means of any standard reduction — because the right-hand subterm is non-normalizable (it reduces to itself via $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$), while the top-level reduction requires a preliminary reduction of the left-hand subterm (which reduces via $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$ to $(\lambda \mathrm{y}.\mathrm{x})$).[12] To couch this in general terms, standardization fails because the poly-context on the left-hand side of a concrete schema, $((\lambda x.\Box_2)\Box_1)$, is ordered so that a potentially non-normalizable subterm (here, $\Box_1$) is followed by a subterm that is nontrivial and therefore might only be obtainable via reduction

[12] As a corollary, since Term (13.115) can't be normalized (reduced to x) via a standard reduction, it can't be normalized via $\mathcal{S},\mathcal{E}$-evaluation $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$, either, so that when call-by-name λ-calculus with right-to-left evaluation fails standardization, it also fails operational soundness.

(here, $\lambda x.\Box_2$). This is the pathological extreme of the situation defined above as *local* $\mathcal{S},\mathcal{E}$*-irregularity*: in the definition, a non-normal subterm T precedes a non-trivial $\vec{P_1}(k)$. (The definition is more inclusive in that it doesn't require T to be non-normalizable, merely non-normal — a simplification made because non-normality is usually decidable while non-normalizability is not.)

In contrast, when $\mathcal{S}$ is the call-by-value λ_v-calculus (with a suitably straightforward choice of SRS schemata), all possible evaluation orders are $\mathcal{S}$-regular.

Lemma 13.116 Suppose regular SRS $\mathcal{S}$, and $\mathcal{S}$-regular evaluation order $\mathcal{E}$.
If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3$, then there exists T_4 such that $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}+} T_4 \longrightarrow_{\mathcal{S}}^{\|?} T_3$. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n, and $T_2 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3$. Assume without loss of generality that $T_2 = E[T_2']$, $T_3 = E[T_3']$, E is an $\mathcal{S},\mathcal{E}$-evaluation context, and $T_2' \longrightarrow^{\mathcal{S}} T_3'$. Proceed by induction on n; and within consideration of given n, proceed by induction on the size of T_1.

If $n = 0$, the result is immediate with $T_4 = T_3$; suppose $n \geq 1$, and the proposition holds for all smaller n.

Case 1: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ involves a top-level reduction. Then it can be factored out to the left (via Lemma 13.111), giving $T_1 \longrightarrow^{\mathcal{S}} T_1' \longrightarrow_{\mathcal{S}}^{\|?} T_2 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3$ where the parallel step has reduction size $< n$. The result follows from the inductive hypothesis.

Case 2: $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ does not involve a top-level reduction.

Case 2a: $E = \Box$. Then $T_2 \longmapsto^{\mathcal{S}} T_3$.

If T_1 is $\longrightarrow^{\mathcal{S}}$-reducible then, since $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ doesn't involve a top-level reduction, $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_3$ (by Definition 13.75 and Conditions 13.101(c) and 13.101(g)), and we're done. Suppose T_1 is not $\longrightarrow^{\mathcal{S}}$-reducible.

Let poly-context P, satisfied by T_2, minimally satisfy the left-hand side of some $\sigma \in \mathcal{S}$. Assume without loss of generality that T_2 is chosen (via α-renaming) to maximize how much of P is shared by T_1. One or more branches of P are not complete in T_1, since T_1 is not $\longrightarrow^{\mathcal{S}}$-reducible; of these branches, one of them is earliest according to ordering $\mathcal{E}$; and this earliest branch must be exercised by the parallel step, $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$. Since the parallel step doesn't involve a top-level reduction, the $\longrightarrow_{\mathcal{S}}$-reduction of this earliest branch can be factored out to the left, producing $T_1 \longrightarrow_{\mathcal{S}} T_1' \longrightarrow_{\mathcal{S}}^{\|?} T_2 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3$ where the parallel step has reduction size $< n$. Because $\mathcal{E}$ is $\mathcal{S}$-regular, $T_1 \longrightarrow_{\mathcal{S}} T_1'$ is an $\mathcal{S},\mathcal{E}$-evaluation step, $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_1'$; and the result follows from the inductive hypothesis.

Case 2b: $E = P[\vec{P}]$ such that $P \in \mathcal{E}$ and, for some $1 \leq k \leq ar(P)$ and all $1 \leq j \leq ar(P)$,

if $j < k$ then $\vec{P}(j)$ is an $\mathcal{S}$-evaluation normal form,

if $j = k$ then $\vec{P}(j)$ is an $\mathcal{S},\mathcal{E}$-evaluation context, and

if $j > k$ then $\vec{P}(j)$ is a term.

Since $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ doesn't involve a top-level reduction, assume without loss of generality that $T_1 = P[\vec{T_1}],\ T_2 = P[\vec{T_2}]$, for each $1 \leq j \leq ar(P),\ \vec{T_1}(j) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(j)$ with reduction size n_j, and $n = \sum_{j=1}^{ar(P)} n_j$. Suppose inductively that the proposition holds for all subterms of T_1, particularly the $\vec{T_1}$.

For each $j < k,\ \vec{T_2}(j) = \vec{P}(j)$ is an $\mathcal{S}$-evaluation normal form; so, by Lemma 13.113, $\vec{T_1}(j) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(j)$ with reduction size n_j can be factored into $\vec{T_1}(j) \longmapsto_{\mathcal{S}}^{\mathcal{E}*} \vec{T_1'}(j) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(j)$ with reduction size $\leq n_j$. Let $\vec{T_1'}$ be these intermediate terms for $j < k$, $\vec{T_1'}(j) = \vec{T_1}(j)$ for $j \geq k$; then $T_1 = P[\vec{T_1}] \longmapsto_{\mathcal{S}}^{\mathcal{E}*} P[\vec{T_1'}] \longrightarrow_{\mathcal{S}}^{\|?} P[\vec{T_2}] = T_2$ with reduction size $\leq n$. We have $\vec{T_1'}(k) = \vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k) = T_2' \longmapsto_{\mathcal{S}}^{\mathcal{E}} T_3'$ where the parallel step has reduction size $n_k \leq n$; so, by either inductive hypothesis (the one on n, or within that the one on subterms of T_1), $\vec{T_1}(k) \longmapsto_{\mathcal{S}}^{\mathcal{E}+} T_2'' \longrightarrow_{\mathcal{S}}^{\|?} T_3'$, and the result follows immediately. ■

Lemma 13.117 Suppose regular SRS $\mathcal{S}$, and $\mathcal{S}$-regular evaluation order $\mathcal{E}$.

If $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$, and there exists an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_2 to T_3, then there exists an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_1 to T_3. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n; $\vec{T}$ is an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_2 to T_3; and $m = ar(\vec{T})$. Proceed by induction on m; within consideration of given m, proceed by induction on n; and within consideration of given m and n, proceed by induction on the size of T_1.

If $m = 1$, then $\vec{T} = \langle T_2 \rangle$, and the result follows immediately (by Lemma 13.112). Suppose $m \geq 2$, and the result holds for all smaller m.

If $n = 0$, then $\vec{T}\backslash_{T_1}^{1}$ is an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_1 to T_3 (by Criterion 13.88(d)). Suppose $n \geq 1$, and the result holds for this m for all smaller n.

Suppose the result holds for this m and n for all proper subterms of T_1 (which is trivially true if T_1 has no proper subterms). There are two cases, depending on whether $T_1 \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size n involves a top-level $\longrightarrow^{\mathcal{S}}$-reduction.

Case 1: no top-level $\longrightarrow^{\mathcal{S}}$-reduction. There are two subcases, depending on whether the first step of $\vec{T}$ is an evaluation step (i.e., whether $\vec{T}(1) \longmapsto_{\mathcal{S}}^{\mathcal{E}} \vec{T}(2)$).

Case 1a: $\vec{T}(1) \longmapsto_{\mathcal{S}}^{\mathcal{E}} \vec{T}(2)$. The result follows from Lemma 13.116, and the inductive hypothesis on m.

Case 1b: $\vec{T}(1) \not\longmapsto_{\mathcal{S}}^{\mathcal{E}} \vec{T}(2)$. Then the status of $\vec{T}$ as a standard reduction sequence is not deducible from Criterion 13.88(b) (neither directly, nor indirectly through Criterion 13.88(d) since $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$ is α-closed by Theorem 13.87); therefore, it must be deducible, directly or indirectly, from Criterion 13.88(c). Assume without loss generality that $P \in \mathcal{E},\ T_1 = P[\vec{T_1}],\ T_2 = P[\vec{T_2}],\ T_3 = P[\vec{T_3}]$, and for each $1 \leq k \leq ar(P)$, $\vec{T_1}(k) \longrightarrow_{\mathcal{S}}^{\|?} \vec{T_2}(k)$ with reduction size $\leq n$ and there is an $\mathcal{S},\mathcal{E}$-standard reduction

sequence from $\vec{T_2}(k)$ to $\vec{T_3}(k)$ with arity $\leq m$. The inductive hypothesis (on m, n, and the size of T_1) applies to each of these subterms, and an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_1 to T_3 can be spliced together from the subsequences via Criterion 13.88(c).

Case 2: top-level $\longrightarrow^{\mathcal{S}}$-reduction. The top-level $\longrightarrow^{\mathcal{S}}$-reduction can be factored out to the left (via Lemma 13.111), giving $T_1 \longrightarrow^{\mathcal{S}} T_1' \longrightarrow_{\mathcal{S}}^{\|?} T_2$ with reduction size $< n$. The result then follows from the inductive hypothesis. ■

Theorem 13.118 (Standardization)
Suppose regular SRS $\mathcal{S}$, and $\mathcal{S}$-regular evaluation order $\mathcal{E}$.
$T_1 \longrightarrow_{\mathcal{S}}^{*} T_2$ iff there exists an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_1 to T_2. ■

Proof. For right-to-left implication, suppose $\vec{T}$ is an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_1 to T_2. By definition of this, $\vec{T}(1) = T_1$ and $\vec{T}(ar(\vec{T})) \equiv_\alpha T_2$. Since every $R,\mathcal{E}$-standard reduction sequence is necessarily an R-reduction sequence, $T_1 \longrightarrow_{\mathcal{S}}^{*} \vec{T}(ar(\vec{T}))$; therefore, since $\longrightarrow_{\mathcal{S}}$ is a reduction relation and $\equiv_\alpha \subseteq \longrightarrow_{\mathcal{S}}^{*}$, $T_1 \longrightarrow_{\mathcal{S}}^{*} T_2$.

For left-to-right implication, suppose $T_1 \longrightarrow_{\mathcal{S}}^{*} T_2$. Proceed by induction on the number of $\longrightarrow_{\mathcal{S}}$-steps in $T_1 \longrightarrow_{\mathcal{S}}^{*} T_2$. If $T_1 \equiv_\alpha T_2$, the result follows immediately. Suppose $T_1 \longrightarrow_{\mathcal{S}} T_3 \longrightarrow_{\mathcal{S}}^{*} T_2$, and the proposition holds for $T_3 \longrightarrow_{\mathcal{S}}^{*} T_2$. Then since $\longrightarrow_{\mathcal{S}} \subseteq \longrightarrow_{\mathcal{S}}^{\|?}$, the result follows from Lemma 13.117. ■

Lemma 13.119 Suppose regular SRS $\mathcal{S}$, and evaluation order $\mathcal{E}$.
If $T_1 \longrightarrow_{\mathcal{S}} T_2$, $T_1 \not\mapsto_{\mathcal{S}}^{\mathcal{E}} T_2$, and T_1 is not an $\mathcal{S}$-evaluation normal form, then T_2 isn't either. ■

Proof. Suppose $T_1 \longrightarrow_{\mathcal{S}} T_2$, $T_1 \not\mapsto_{\mathcal{S}}^{\mathcal{E}} T_2$, and T_1 is not an $\mathcal{S}$-evaluation normal form. Because T_1 is not an $\mathcal{S}$-evaluation normal form, let $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T$. Whatever $\longrightarrow^{\mathcal{S}}$-redex in T_1 is exercised by $T_1 \longmapsto_{\mathcal{S}}^{\mathcal{E}} T$, that redex is not the one exercised by $T_1 \longrightarrow_{\mathcal{S}} T_2$. Each of these two redexes satisfies the left-hand side of some schema in $\mathcal{S}$; let the poly-contexts minimally satisfying these schema left-hand sides be P and P_2, respectively. (That is, P is satisfied by the redex leading to T, and P_2 by the redex leading to T_2.) Both P and P_2 are selective in $\longrightarrow^{\mathcal{S}}$ and decisively $\longrightarrow^{\mathcal{S}}$-reducible (by Conditions 13.101(c) and 13.101(g)). By the definition of $\longmapsto_{\mathcal{S}}^{\mathcal{E}}$ (Definition 13.86), the redex satisfying P does not occur within the redex satisfying P_2 (because P_2 is decisively $\longrightarrow^{\mathcal{S}}$-reducible); and even if P_2 occurs within the redex satisfying P, exercising it cannot disrupt that redex's satisfaction of P (because P is selective in $\longrightarrow^{\mathcal{S}}$). So T_2 necessarily contains a subterm satisfying P that is not within any poly-context suspending in $\longrightarrow^{\mathcal{S}}$, and T_2 is not an $\mathcal{S}$-evaluation normal form. ■

Theorem 13.120 (Operational soundness)
Suppose regular SRS $\mathcal{S}$, and $\mathcal{S}$-regular evaluation order $\mathcal{E}$.
If $T_1 =_{\mathcal{S}} T_2$ then $T_1 \simeq^{\mathcal{E}}_{\mathcal{S}} T_2$. ■

Proof. Suppose $T_1 =_{\mathcal{S}} T_2$, $C[T_1], C[T_2] \in \mathit{Terms}$, and $\mathrm{Free}(C[T_1], C[T_2]) = \{\}$. Since $\longrightarrow_{\mathcal{S}}$ is compatible, $C[T_1] =_{\mathcal{S}} C[T_2]$.

For Condition 13.90(a), it suffices to show that if $C[T_1]$ is $\mathcal{S},\mathcal{E}$-normalizable, then so is $C[T_2]$; implication in the other direction follows by symmetry. Suppose $C[T_1] \longmapsto^{\mathcal{E}*}_{\mathcal{S}} T_1'$, and T_1' is an $\mathcal{S}$-evaluation normal form.

By the Church–Rosser theorem (Theorem 13.104), since $T_1 =_{\mathcal{S}} T_2$, let $T_1' \longrightarrow^*_{\mathcal{S}} T$ and $T_2 \longrightarrow^*_{\mathcal{S}} T$. Since T_1' is an $\mathcal{S}$-evaluation normal form, T must be also (by Lemma 13.82). By the standardization theorem (Theorem 13.118), since $T_2 \longrightarrow^*_{\mathcal{S}} T$, let $\vec{T_2}$ be an $\mathcal{S},\mathcal{E}$-standard reduction sequence from T_2 to T. By Lemma 13.89, assume without loss of generality that all $\mathcal{S},\mathcal{E}$-evaluation steps in $\vec{T_2}$ occur consecutively at the start of the sequence.

If $\vec{T_2}$ is an $\longmapsto^{\mathcal{E}}_{\mathcal{S}}$-reduction sequence, then by definition $C[T_2]$ is $\mathcal{S},\mathcal{E}$-evaluation normalizable. Otherwise, let j be the smallest integer such that $\vec{T_2}(j) \not\longmapsto^{\mathcal{E}}_{\mathcal{S}} \vec{T_2}(j+1)$. If $\vec{T_2}(j)$ were not an $\mathcal{S}$-evaluation normal form, then $\vec{T_2}(ar(\vec{T_2})) \equiv_\alpha T$ wouldn't be an $\mathcal{S}$-evaluation normal form either (by Lemma 13.119), contradicting the supposition; so $\vec{T_2}(j)$ must be an $\mathcal{S}$-evaluation normal form. By choice of j, $C[T_2] \longmapsto^{\mathcal{E}*}_{\mathcal{S}} \vec{T_2}(j)$; so again by definition, $C[T_2]$ is $\mathcal{S},\mathcal{E}$-evaluation normalizable.

For Condition 13.90(b), it suffices to show implication left-to-right; implication right-to-left follows by symmetry. Suppose T_3 is a minimal nontrivial $\mathcal{S}$-evaluation normal form, and $T_1 \longrightarrow^{\mathcal{E}*}_{\mathcal{S}} T_3$. Because T_3 is minimal nontrivial, any $\longrightarrow_{\mathcal{S}}$-reduction of T_3 is necessarily a top-level reduction (an $\longrightarrow^{\mathcal{S}}$-reduction), hence an $\longmapsto^{\mathcal{E}}_{\mathcal{S}}$-reduction; therefore, since T_3 is an $\mathcal{S}$-evaluation normal form, it is not $\longrightarrow_{\mathcal{S}}$-reducible. By Condition 13.90(b) (proven above), $T_2 \longmapsto^{\mathcal{E}}_{\mathcal{S}} T_2'$ for some $\mathcal{S}$-evaluation normal form T_2'; and by the Church–Rosser theorem, $T_2' \longrightarrow^*_{\mathcal{S}} T_3$. Let P be a minimal nontrivial poly-context satisfied by T_2'. Since T_2' is an $\mathcal{S}$-evaluation normal form, it is not $\longrightarrow^{\mathcal{S}}$-reducible (since $\longrightarrow^{\mathcal{S}} \subseteq \longmapsto^{\mathcal{E}}_{\mathcal{S}}$), nor is any term to which it $\longrightarrow_{\mathcal{S}}$-reduces; so T_3 satisfies some α-form of P, which is to say (since T_3 is itself minimal nontrivial) that T_3 itself is an α-form of P; and T_3 has arity zero, so P has arity zero, and $P = T_2'$. So T_2' must reduce to T_3 in zero steps; so $T_2 \longmapsto^{\mathcal{E}}_{\mathcal{S}} T_3$. ■

Chapter 14

Well-behavedness of vau calculi

14.0 Introduction

Preceding chapters have defined three principal $\mathring{\lambda}$-calculi: $\mathring{\lambda}_p$-calculus (§9.4), $\mathring{\lambda}C$-calculus (§11.3), and $\mathring{\lambda}S$-calculus (§12.3). Along the way there were also six other $\mathring{\lambda}$-calculi: two pure calculi preliminary to the primary pure calculus, $\mathring{\lambda}_e$-calculus (§9.1) and $\mathring{\lambda}_x$-calculus (§9.3); two pure calculi preliminary to the impure calculi, $\mathring{\lambda}_i$-calculus (§10.6) and $\mathring{\lambda}_r$-calculus (§10.7); and the regular variants of the impure calculi, $\mathring{\lambda}_r C$-calculus (§11.3) and $\mathring{\lambda}_r S$-calculus (§12.3.8).

This chapter explores the extent to which $\mathring{\lambda}$-calculi are well-behaved, in the sense conservatively called for by Plotkin's paradigm (§13.0): Church–Rosser-ness of $\longrightarrow^\bullet$, operational completeness of $\longrightarrow^*_\bullet$, and operational soundness of $=_\bullet$. Standardization of $\longrightarrow^*_\bullet$, being a means to an end, is not universally sought; and operational completeness/soundness are only meaningful for calculi that have an associated semantics.

14.1 Conformance of $\mathring{\lambda}$-calculi

This section checks all the $\mathring{\lambda}$-calculi against the criteria for SRSs and regular SRSs. Except for a technicality concerning $\mathring{\lambda}_e$-calculus, all the $\mathring{\lambda}$-calculi are SRSs, and all but the two principal impure calculi, $\mathring{\lambda}C$-calculus and $\mathring{\lambda}S$-calculus, are regular.

14.1.1 Terms and renaming functions

Most abstract constraints are imposed within numbered Assumptions; the exception is a miscellany of (mostly rather mundane) assumptions stated in prose in roughly the first two pages of §13.1.2 (prior to Lemma 13.16), and a few more concerning active and skew sets stated following Definition 13.34. Notably, among the unnumbered assumptions, $\equiv_\alpha$ is compatible; there is a countably infinite syntactic domain of variables, partially ordered by ancestry relation $\sqsubseteq$; $\mathcal{F}_\alpha$ is closed under finite composition

and includes $f_{\mathbf{id}}$; complement is unique, complement of a composition is the reverse-order composition of complements, and complement of complement is identity; the free set of a term is closed under ancestry; and the active set of a renaming f is closed under both f and its complement.

Numbered Assumptions about terms and renaming functions (*Terms* and $\mathcal{F}_\alpha$) are 13.7, 13.17, 13.19, 13.24, 13.35, and 13.36.

Of the nine $\natural$-calculi, four ($\natural_i$-, $\natural_r$-, $\natural_r C$-, and $\natural_r S$-calculus) share the term syntax and renaming functions of one or another of the three principal calculi; so there are five $\natural$-calculus syntaxes.

Various assumptions, both unnumbered and numbered, require the existence of variables, which on the face of it excludes $\natural_e$-calculus. To circumvent this technicality,[1] we simply add *pro forma* into $\natural_e$-calculus the usual domain of partial-evaluation variables (i.e., λ-calculus variables), associated renaming functions, and self-evaluating terms x and $\langle \natural x.T \rangle$. These additions have no impact on well-behavedness of the reduction relations of the calculus (such as Church–Rosser-ness), since, while introducing terms $\langle \natural x.T \rangle$, we have not introduced any schema that uses these structures; thus, terms x and $\langle \natural x.T \rangle$ remain, in the augmented $\natural_e$-calculus, merely somewhat eccentric passive data structures.

Of the unnumbered assumptions, most hold straightforwardly for all five syntaxes (with the additions to $\natural_e$-calculus). The choices of $\sqsubseteq$ and complements are discussed where those assumptions are introduced early in §13.1.2.

Assumption 13.7(a), though presented in §13.1.1, does not require separate verification since it is implied in §13.1.2 by Assumption 13.17(c).

Assumption 13.7(b) isn't implied by later assumptions, but is closely related to Assumption 13.17(c), and is readily seen to hold for all five syntaxes.

For Assumption 13.7(c), if the structures in the domain of a semantic variable don't contain any terms, the assumption holds trivially. If the structures in the domain are described using a context-free grammar, the assumption holds straightforwardly. The domain of values V in each calculus is described by prose but has context-free structure. Context-sensitive syntax constraints are listed explicitly after each syntax block to which they apply: (9.1), (9.13), (9.27), (10.1), (12.8), (12.28), (12.29). As observed following (9.1), of all the context-sensitive constraints, only one restricts which terms can be used in which contexts — that being the one that is also built into the abstract treatment (essentially, Lemma 13.16). The other context-sensitive constraints are all about keeping the bindings of an environment in sorted order by lookup key; and that doesn't create any mutual dependence between subterms, because the keys aren't in subterms. In, say, $\langle\!\langle s_1 \leftarrow T_1\ s_2 \leftarrow T_2 \rangle\!\rangle$, the sort keys s_k are part of minimal nontrivial poly-context $\langle\!\langle s_1 \leftarrow \Box_1\ s_2 \leftarrow \Box_2 \rangle\!\rangle$, into which arbitrary subterms T_k can be freely placed. So all the $\natural$-calculi satisfy the assumption.

[1] The technicality could, and presumably in the long run should, be repaired by rephrasing the assumptions, but was not a priority for the current document.

Assumptions 13.17, 13.19, 13.24, 13.35(a), 13.35(b), and 13.36 hold straightforwardly. Assumption 13.35(c) (if $\mathrm{Bind}(P) \cap \mathrm{Act}(f) = \{\}$ then $P \mid f$) is straightforward if one keeps in mind that proper descendants of $\mathrm{Bind}(P)$ cannot occur free in subterms $\vec{T}$ of $P[\vec{T}]$ (due to Lemma 13.16). Assumption 13.35(d), delineating sufficient conditions for α-renaming, is also straightforward.

Hence all five syntaxes (as amended) satisfy all the term and renaming assumptions of the abstract treatment.

14.1.2 Substitutive functions

Of the nine kinds of substitution defined for the $\mathfrak{l}$-calculi, four are used for renaming ($\mathcal{F}_\alpha$, supporting $\equiv_\alpha$), and six are used substitutively ($\mathcal{F}_\beta$, supporting $\longrightarrow^\bullet$). Only in partial-evaluation substitution —λ-calculus-style substitution— do the two uses coincide: substitutive $\Box[x_p \leftarrow T]$ doubles as a renaming function when T is restricted to partial-evaluation variables, $\Box[x_p \leftarrow x'_p]$. The substitutive kinds of substitution are $\Box[x_p \leftarrow T]$, $\Box[x_c \leftarrow C]$, $\Box[x_s \not\leftarrow]$, $\Box[x_g \leftarrow V]$, $\Box[x_g \not\leftarrow]$, and $\Box[[x_s, s] \leftarrow V]$. These are not substitutive functions per se, because they don't conform to Definition 13.49; they have α-renaming built into them, and various constraints required by the definition could be imposed in more than one way. This subsection shows how to perform these six modes of substitution using technically substitutive functions.

For $\Box[x_p \leftarrow T]$, for each $n \in \mathbb{N}$, let

$$f_p(x, T_1, P_1[\Box_1, P_2[\textstyle\sum_{k=1}^{n} \Box_{k+1}]], \vec{T_2}) = \begin{cases} T_1 \lfloor x \leftarrow P_2[\vec{T_2}] \rfloor & \text{if } x \in \mathit{PartialEvaluationVariables} \\ T_1 & \text{otherwise}, \end{cases} \tag{14.1}$$

restricted to cases, allowed under Condition 13.49(a), where $\mathrm{Bind}(P_1) = \{\}$; and let $f_{\mathrm{v}}(x, x') = x'$, restricted to $x' \notin (x \cap \mathit{PartialEvaluationVariables})$. (Defining $f_{\mathrm{v}}(x_p, x_p)$ would not prevent substitutiveness, but would disallow a β-rule using this f from eliminating the binding of the parameter, by Condition 13.91(e).) Then f_p is substitutive with variable transformation f_{v}; for Condition 13.49(e), $f_p(x, T, P_1[\Box_1, P_2[\sum_{k=1}^{n} \Box_{k+1}]])$ is constructed from T by replacing all free occurrences of x with P_2.

For $\Box[x_c \leftarrow C]$, (11.3), let $C = (P[\Box_1, \vec{P}])[\Box, \vec{T}]$ and

$$f_c(x, T, P[\Box_1, \vec{P}], \vec{T}) = \begin{cases} T \lfloor x \leftarrow (P[\Box_1, \vec{P}])[\Box, \vec{T}] \rfloor & \text{if } x \in \mathit{ControlVariables} \\ T & \text{otherwise}, \end{cases} \tag{14.2}$$

restricted to cases, allowed under Condition 13.49(a), where P is iso minimal nontrivial, $P[\Box_1, \vec{P}]$ is iso, and P binds no variables; and let $f_{\mathrm{v}}(x, x') = x'$. ($C$ non-binding is mentioned in §11.1; cf. Condition 13.91(e).) Then f_c is substitutive with variable transformation f_{v}.

For $\Box[x_s \not\leftarrow]$, (12.17), let

$$f_{\not s}(x, T, P) \;=\; \begin{cases} T\lfloor x \not\leftarrow \rfloor & \text{if } x \in \mathit{StateVariables} \\ T & \text{otherwise}\,, \end{cases} \tag{14.3}$$

restricted to cases, allowed under Condition 13.49(a), where P has arity one and doesn't bind any variables; if $x = [w_i]$ is a state variable, then $T\lfloor x \not\leftarrow \rfloor$ is defined; and for each child $[iw_i]$ of x occurring in T, either all occurrences of $[iw_i]$ are free in T, or all are bound at a single point in T.

α-renaming an arbitrary term T cannot always guarantee definition of $T\lfloor x \not\leftarrow \rfloor$, because $T\lfloor x \not\leftarrow \rfloor$ may be undefined due to free variables in T, which cannot be α-renamed. However, when T occurs within a redex of a schema that uses $\Box[x_s \not\leftarrow]$, the binding of x_s will be within the redex, so that α-renaming of the entire redex can always satisfy that restriction, and also the final restriction on redundantly named children of x (and function α does both).

Let

$$f_{\mathrm{v}}(x, x') \;=\; \begin{cases} x' & \text{if } x \notin \mathit{StateVariables} \\ x'\lfloor x \not\leftarrow \rfloor & \text{if } x \in \mathit{StateVariables} \text{ and } x' \neq x \\ \text{undefined} & \text{if } x \in \mathit{StateVariables} \text{ and } x' = x\,. \end{cases} \tag{14.4}$$

Then $f_{\not s}$ is substitutive with variable transformation f_{v}; note that Conditions 13.49(e) and 13.49(h) are trivial for $f_{\not s}$.

For $\Box[x_g \leftarrow V]$, (12.33), for each $n \in \mathbb{N}$, let

$$f_g(x, T, P_0, \vec{T}) \;=\; \begin{cases} T\lfloor x \leftarrow P[\vec{T}] \rfloor & \text{if } x \in \mathit{GetVariables} \\ T & \text{otherwise}\,, \end{cases} \tag{14.5}$$

restricted to cases, allowed under Condition 13.49(a), where P is an iso poly-context with arity n that minimally satisfies semantic polynomial "V" (the iso condition can be satisfied for any particular value V due to Assumption 13.7(c)); P_0 is an iso poly-context with arity $n+1$, with subexpression $P[\sum_{k=1}^{n} \Box_{k+1}]$ (noting that this is a sub*expression* but not necessarily a sub*term*, the specific relevance of the distinction being that the value in a stateful binding $x \leftarrow V$ is not a subterm), with $\mathrm{Bind}(P_0 \text{ at } \Box_1) = \{\}$, and with each $\mathrm{Bind}(P_0 \text{ at } \Box_{k+1}) = \mathrm{Bind}(P \text{ at } \Box_k)$; and $P[\vec{T}]$ is a term (hence, by choice of P, a value). Let $f_{\mathrm{v}}(x, x') = x'$, restricted to $x' \notin (x \cap \mathit{GetVariables})$. Then f_g is substitutive with variable transformation f_{v}.

For $\Box[x_g \not\leftarrow]$, (12.34), let

$$f_{\not g}(x, T, P) \;=\; \begin{cases} T\lfloor x \not\leftarrow \rfloor & \text{if } x \in \mathit{GetVariables} \\ T & \text{otherwise}\,, \end{cases} \tag{14.6}$$

restricted to cases, allowed under Condition 13.49(a), where P has arity one and doesn't bind any variables; and let $f_{\mathrm{v}}(x, x') = x'$, restricted to $x' \notin (x \cap \mathit{GetVariables})$. Then $f_{\not g}$ is substitutive with variable transformation f_{v}.

For $\Box[[x_s, s] \leftarrow V]$, (12.35), for each $n \in \mathbb{N}$ and $s \in \mathit{Symbols}$, let

$$f_s(x, T, P_0, \vec{T}) \;=\; \begin{cases} T\lfloor [x, s] \leftarrow P[\vec{T}] \rfloor & \text{if } x \in \mathit{StateVariables} \\ T & \text{otherwise}\,, \end{cases} \tag{14.7}$$

restricted to cases, allowed under Condition 13.49(a), where P is an iso poly-context with arity n that minimally satisfies semantic polynomial "V" (the iso condition can be satisfied for any particular value V due to Assumption 13.7(c)); P_0 is an iso poly-context with arity $n+1$, with subexpression $P[\sum_{k=1}^{n} \Box_{k+1}]$ (noting that this is a sub*expression* but not necessarily a sub*term*, the specific relevance of the distinction being that the value in a stateful binding $x \leftarrow V$ is not a subterm), with $\mathrm{Bind}(P_0 \text{ at } \Box_1) = \{\}$, and with each $\mathrm{Bind}(P_0 \text{ at } \Box_{k+1}) = \mathrm{Bind}(P \text{ at } \Box_k)$; and $P[\vec{T}]$ is a term (hence, by choice of P, a value). Let $f_{\mathrm{v}}(x, x') = x'$. Then f_s is substitutive with variable transformation f_{v}.

14.1.3 Calculus schemata

The schemata for the calculi are:

$\mathfrak{l}_e$-calculus (§9.1): Schemata 9.7, and for δ-rules, Schema 9.13.

$\mathfrak{l}_x$-calculus (§9.3): Schemata 9.25.

$\mathfrak{l}_p$-calculus (§9.4): Schemata 9.32.

$\mathfrak{l}_i$-calculus (§10.6): Schemata 10.6, amending $\mathfrak{l}_p$-calculus.

$\mathfrak{l}_r$-calculus (§10.7): Schemata 10.7, amending $\mathfrak{l}_i$-calculus.

$\mathfrak{l}C$-calculus (§11.3): Schemata 11.12 and 11.13, amending $\mathfrak{l}_i$-calculus.

$\mathfrak{l}_rC$-calculus (§11.3): Schemata 11.14 and 11.13, amending $\mathfrak{l}_r$-calculus.

$\mathfrak{l}S$-calculus (§12.3): **`$define!`** Schemata 12.36, set simplification Schemata 12.37, set bubbling-up Schema 12.38, symbol evaluation Schemata 12.39, get resolution Schemata 12.40, get simplification Schemata 12.41, get bubbling-up Schemata 12.42, state simplification Schemata 12.44, and state bubbling-up Schema 12.45.

$\mathfrak{l}_rS$-calculus (§12.3.8): **`$define!`** Schemata 12.36, symbol evaluation Schemata 12.39, and get resolution Schemata 12.40.

Note that singular evaluation contexts are defined by (11.8).

Just as the substitutions $\Box[x_p \leftarrow V]$ etc. had to be cast into the form required by the definition of substitutive function, so the calculi schemata have to be cast into the form required by the definition of SRS schema (Definition 13.96). This primarily means associating each schema with an α-normal set of SRS concrete schemata. For SRS concrete schemata (Definition 13.91), the most complex provisions to verify are those for hygiene, Conditions 13.91(e) and 13.91(f).

Additional properties of interest for individual schemata are:

- the left-hand side of each concrete schema is in general position (part of calculus regularity Condition 13.101(c)), and

- every term satisfying the left-hand side of each schema σ is reducible in $\longrightarrow^{\sigma}$ (calculus regularity Condition 13.101(g)).

$\mathfrak{l}_e$-calculus

Schemata 9.7 are relatively straightforwardly processed, since they contain no syntactic variables. Consider any one of these schemata, σ. Consider the set of all iso poly-contexts that minimally satisfy the left-hand side of σ. Each term satisfying the left-hand side of σ satisfies some such poly-context; and if it satisfies more then one such, then the poly-contexts satisfy each other (cf. Theorem 13.11) and differ only by permutation of their meta-variable indices. Restrict the set to one representative of each of these equivalence classes of poly-contexts (such as by sorting the meta-variable indices by an evaluation order); then each term satisfying the left-hand side of σ satisfies *exactly* one poly-context in the restricted set.

For three of the six schemata —9.7S, 9.7p, and 9.7a— the left-hand side is a semantic polynomial, and every term satisfying that polynomial is reducible in $\longrightarrow^{\sigma}$, and the choice of left-hand poly-context uniquely determines the right-hand side. Letting this left-hand poly-context be P_0 and $P_1 = \Box_1$ in (13.92), for those schemata we're done. The left-hand side of Schema 9.7γ isn't actually a semantic polynomial —it's a template for a countably infinite set of semantic polynomials, one for each possible number of operands m— but, by setting up one schema for each possible m, these schemata also have the properties of the first three, and we're done with them, too. Each term satisfying the left-hand side of any of these schemata satisfies exactly one of the concrete schemata, and there are no variables involved; therefore, the described concrete forms of the schemata are α-normal.

The last two schemata —9.7s and 9.7β— have additional constraints on them that preclude some terms that satisfy their respective left-hand sides. However, each left-hand poly-context determines unambiguously whether or not the additional constraint is satisfied (i.e., whether the lookup is defined for 9.7s, or whether the match is defined for 9.7β). So by converting every successful case from an SRS concrete schema back into an SRS schema —by replacing the meta-variables with semantic term-variables— we can convert each of these two constrained schemata into a countably infinite set of unconstrained schemata, each of which has an α-normal concrete form consisting of a single SRS concrete schema.

δ-rule Schema 9.13 is another technically multiple schema — one schema for each δ-form in each $\delta(o)$. Each of these schemata has the same properties as the first three schemata discussed above; the left-hand side is monic by Conditions 9.10(3) and 9.10(4), and the left-hand side determines the right-hand side by Condition 9.10(5).

$\mathfrak{l}_x$-calculus

Of the eight schemata in (9.25), six have no variables, so that the above reasoning

for $\natural_e$-calculus schemata applies, including the treatment of the γ-rule as a countably infinite set of schemata. The remaining schemata are (9.25β) and (9.25δ).

For the β-rule —the only schema in this calculus that does nontrivial substitution— use variants of concrete Schema 13.93, with poly-contexts P_V minimally satisfying semantic variable V:

$$[\text{combine } \langle \natural \text{x}.P_V[\Box_2] \rangle \ \Box_1] \ \longrightarrow \ f_p(\text{x}, \Box_2, P_0, \Box_1)\,, \tag{14.8}$$

where $P_0 = [\text{combine } \Box_1 \ P_V[\Box_2]]$ etc.; conformance to the definition of SRS concrete schema is mostly as outlined for (13.93). The left-hand side of the concrete schema is in general position, and every term satisfying the left-hand side of the schema is reducible.

The δ-rule for each δ-form in each $\delta(o)$ matches (13.92) with, again, trivial substitution and $P_1 = \Box_1$. The technicality that (13.92) requires an explicit variable that is trivially substituted for, which must then be avoided for variable hygiene concerns, is washed out by α-closure. The remaining bound-variable provisions of Condition 13.91(e) are satisfied by Condition 9.22(6), and free-variable Condition 13.91(f) by Conditions 9.22(6) and 9.22(7). The free-variable condition guarantees that every term satisfying the left-hand side of the schema is reducible in $\longrightarrow^\sigma$. General position is guaranteed possible by Condition 9.22(9), and α-normality is guaranteed possible by general position.

$\natural_p$-, $\natural_i$-, and $\natural_r$-calculus

Schema 9.32$\boldsymbol{\upsilon}$ can be split into an infinite set of schemata, one for each choice of minimal poly-context satisfying the left-hand side, similarly to symbol-lookup Schema 9.7s. All the other $\natural_p$-, $\natural_i$-, and $\natural_r$-calculus schemata are either minor modifications of $\natural_e$- or $\natural_x$-calculus schemata, or trivial transformations (or both).

$\natural C$- and $\natural_r C$-calculus

Of Schemata 11.12, the last three are trivial (noting that the throw bubbling-up rule should be treated as a set of schemata, one for each α-closure class of singular evaluation contexts E^s).

If *all* we wanted was an SRS (not a regular one), the catch garbage-collection schema, (11.12g), could be treated as an infinite set of SRS schemata, one for each α-closure class of terms T that satisfy the constraint (by having no free occurrences of x_c). In fact, each schema would have just one corresponding concrete schema which can be chosen for general position, and evidently every term satisfying the schema left-hand side is reducible in $\longrightarrow^\sigma$. The point on which regularity fails under that arrangement is that the left-hand side is not selective in $\longrightarrow^{\mathcal{S}}$.

The catch-catch simplification, (11.12cc), cannot be cast as an SRS schema (nor schemata) at all, because it violates the first clause of Condition 13.49(i), in the

definition of substitutive function: it may introduce free variable x_c into $T[x'_c \leftarrow x_c]$ when x_c didn't occur free in any structure drawn from the left-hand side.[2]

The catch bubbling-up schema, (11.12c), can be treated as a set of schemata, one for each α-closure class of singular evaluation contexts E^s. Each schema has an α-normal form consisting of a single SRS concrete schema, matching (13.92) with minimal nontrivial P_0 satisfied by E^s (as in (14.2)), and $P_1 = [\text{catch } x_c \ \square_1]$. General position can be arranged, and the hygienic α-renaming $T[x_c \leftarrow x'_c]$ is obviated by Condition 13.49(a). All terms satisfying the left-hand side of each schema are reducible in $\longrightarrow^\sigma$.

$\wr_r C$-calculus uses a subset of the same schemata, notably excluding non-SRS catch-catch simplification and non-regular garbage-collection.

$\wr S$- and $\wr_r S$-calculus

For ***$define!*** Schemata 12.36, the additional constraints —via function *definiend*— can be handled by splitting the rule into an infinite set of schemata, similarly to $\wr_p$-calculus symbol lookup (or the *definiend* constraint on the $\wr_p$-calculus vau rule), since any poly-context minimally satisfying the left-hand side determines unambiguously whether or not the constraints are satisfied. Symbol evaluation Schemata 12.39 is similar, as are get resolution Schemata 12.40 (noting that the substitutions in get resolution eliminate x_g, allowing it to be unbound by the right-hand side).

Set simplification Schemata 12.37 and set bubbling-up Schema 12.38 are trivial, since set is no a binding construct.

Of the three get simplification Schemata 12.41, empty-frame elimination (12.41?0) is trivial; get consolidation (12.41?2) is non-SRS because it violates substitutivity, similarly to catch-catch simplification and get-get concatenation (12.41??) is —despite the elaborate conditions placed on it— also trivial, as the conditions are only hygiene.

Of the two get bubbling-up Schemata 12.42, (12.42↑?) is trivial, its conditions being (like those of get-get concatenation) only hygiene. (12.42↑!?) also has a *lookup* constraint, which can be handled by splitting it into an infinite set of schemata, similarly to symbol evaluation schemata (noting, again, that the set construct is non-binding). An additional constraint on the second schema requires it to choose the leftmost binding request to bubble up through the set — which actually *simplifies* the situation for α-normalization, since without that additional constraint, some left-hand sides would be reducible by multiple choices of binding request, forcing a further multiplication of schemata to achieve SRS-hood of each schema, and precluding regularity (which requires the entire calculus to be α-normalizable).

Of the three state simplification Schemata 12.44, the first (state-state concatenation, 12.44$\sigma\sigma$) is trivial as its conditions are only hygiene; the second (garbage-

[2]This is a hygiene violation only when the substitution is considered separately from the schema in which it occurs, suggesting that some of the complexity of the abstract treatment may be an artifact of where the line is drawn between function and schema.

collection, 12.44σg) is SRS-able but inherently non-regular; and the third (empty-frame elimination, 12.44σ0) is trivial.

State bubbling-up Schema 12.45 is trivial, as its conditions are only hygiene, similarly to get bubbling-up.

$\mathring{\iota}_r S$-calculus uses a subset of the same schemata, notably excluding non-SRS get consolidation and non-regular garbage-collection.

14.1.4 Regularity

Only the two full impure $\mathring{\iota}$-calculi are expected to be irregular ($\mathring{\iota}C$- and $\mathring{\iota}S$-calculus); the other seven should be regular. This subsection confirms the latter expectations.

In addition to the constraints on individual schemata considered in §14.1.3, regularity requires a calculus to have an α-normal concrete form (Condition 13.101(b)) in which every left-hand side is both general-position and selective (Condition 13.101(c)), each substitutive function distributes over $\longrightarrow_{\mathcal{S}}$ and strictly over $\longrightarrow^{\mathcal{S}}$ (Condition 13.101(d)), and each substitutive function transforms each $\longrightarrow^S$-redex to an $\longrightarrow^S$-redex (Condition 13.101(e)).

α-normality and selectivity

These constraints compare schemata to each other.

Since each schema in our candidate regular calculi has an α-normal form, and every term satisfying the left-hand side of each schema σ is reducible in $\longrightarrow^\sigma$, for α-normality we only need to verify that no term can satisfy the left-hand sides of more than one SRS schema in a given calculus. As presented in Chapters 9–12, the schemata of the regular calculi are, in fact, mutually exclusive: no term can satisfy the left-hand side of more than one of them; however, when casting those schemata into SRS form, some of them were converted into *infinite sets* of SRS schemata, requiring a double-check that the schemata in each set are mutually exclusive. The double-check is straightforward, because in each case the split into multiple SRS schemata was *based* on mutually exclusive structures — α-closed classes of minimal satisfying poly-contexts for symbol lookup and similar (**`$vau`** (9.32$\boldsymbol{v}$), **`$define!`** (12.36), get resolution (12.40), and get-through-set bubbling-up (12.42↑!?)); α-closed classes of singular evaluation contexts for catch and throw bubbling-up; number of operands for γ-rules; primitive operator and δ-form for δ-rules. Non-α-normalizable alternatives were mentioned for the get-through-set bubbling-up schema (12.42↑!?), and alluded to for the get resolution schemata of (12.40), but were not implemented (and similarly, without remark, for get simplification Schema 12.41?2).

For selectivity, it suffices that each left-hand poly-contextual pattern cannot overlap with itself or any other in the calculus; checking this is greatly facilitated by the convention that all redex patterns require active terms.

Redex-invariance and distributivity

These constraints compare substitutive functions to schemata, and to each other.

In each redex pattern of each candidate regular calculus, a free variable cannot occur anywhere that would prevent substitution for that variable from commuting with reduction by the schema. Partial-evaluation variables can only be substituted for by values, which may alter which minimal satisfying poly-context is used to match the left-hand side of an SRS schema —and thereby alter which SRS concrete schema is used within the α-normal concrete form of that SRS schema— but will not alter which SRS schema can be used. Variable deletions don't happen in the candidate regular calculi. Control variables and get variables can only alter active subterms —throws and receives— which consequently don't overlap with value-subterm constraints (as noted above re selectivity); and receives do not occur in the redex patterns in any other capacity either, while throws occur only in the bubbling-up schema where substitution will neither affect nor be affected by their participation in the redex pattern. Besides establishing that substitution preserves reducibility in $\longrightarrow^{\mathcal{S}}$ (Condition 13.101(e)), this also means that distributivity of each substitutive f over the reduction relations (Condition 13.101(d), per Definition 13.65) is implied by distributivity of f over all the substitution functions (including itself; per Definition 13.56).

What remains is to verify that in each candidate regular calculus, each of the substitutive functions used distributes over all of the substitutive functions used (including itself). Of the six substitutive functions defined for the calculi, the deletions aren't used in the candidate regular calculi, and the mutable-to-immutable binding coercion isn't used at all, leaving only three substitutions actually used in the candidates: $\Box[x_p \leftarrow T]$, $\Box[x_c \leftarrow C]$, and $\Box[x_g \leftarrow V]$.[3] Pairwise distributivity between these three functions is straightforward.

14.2 Well-behavedness of $\natural$-calculi

The three results we want are Church–Rosser-ness of $\longrightarrow^{\mathcal{S}}$, operational completeness of $\longrightarrow^{*}_{\mathcal{S}}$, and operational soundness of $=_{\mathcal{S}}$. As remarked at the top of the chapter, completeness/soundness isn't meaningful for any calculus that doesn't have an associated semantics; and we are primarily concerned with the most general calculus associated with each $\natural$-semantics: $\natural_p$-, $\natural C$-, and $\natural S$-calculus. Seven out of nine $\natural$-calculi are regular SRSs, which implies Church–Rosser-ness for those seven (Theorem 13.104); but the completeness and soundness properties that come with regularity aren't what we had in mind even for $\natural_i$-calculus, because we wanted those correspondence results *with respect to the declared* $\bullet$*-semantics*. That is, we wanted $\longmapsto^{*}_{\bullet} \subseteq \longrightarrow^{*}_{\mathcal{S}}$

[3]Thus, no substitution in the regular calculi uses the domain of state variables — whose structural complexity contributes to the overall size of the abstract treatment; and no substitution in the regular calculi performs variable deletion — whose interaction with state variables motivates the provisions for variable transformations in the definition of substitutive functions.

and $=_{\mathcal{S}} \subseteq \simeq_\bullet$, where instead we have correspondences with an $\mathcal{S},\mathcal{E}$-evaluation relation $\longmapsto_{\mathcal{S}}^{\mathcal{E}*}$ provided $\mathcal{E}$ is $\mathcal{S}$-regular: $\longmapsto_{\mathcal{S}}^{\mathcal{E}*} \subseteq \longrightarrow_{\mathcal{S}}^{*}$ and $=_{\mathcal{S}} \subseteq \simeq_{\mathcal{S}}^{\mathcal{E}}$ (respectively, Definition 13.86 and Theorem 13.120).

In this section we obtain the intended soundness results. To do so, we assess in depth the extent of the mismatch between obtained $\mathcal{S},\mathcal{E}$-evaluation results and intended $\bullet$-semantics results, and provide supplementary proofs to bridge the gap where necessary. In particular we consider whether (or not) right-to-left evaluation order $\mathcal{E}$ is $\mathcal{S}$-regular for each of the seven regular λ-calculi $\mathcal{S}$, and —independently— whether each $\longmapsto_R^{\mathcal{E}*}$ is a subset of the associated $\longrightarrow_\bullet^*$, and whether each $\simeq_R^{\mathcal{E}}$ is a subset of the associated $\simeq_\bullet$, for each of the seven λ-calculi that share common syntax with $\bullet$-semantics (i.e., all but λ_e- and λ_x-calculus — noting that $R,\mathcal{E}$-evaluation doesn't require R to be an SRS at all, let alone regular).

In the supplementary results, we work with calculi that may violate selectivity —so that a term may be decomposed into $R,\mathcal{E}$-evaluation context and R-redex in more than one way— or α-normality — so that an R-redex may be reducible by more than one schema. These violations, in turn, cause difficulties for the abstract definitions of both $R,\mathcal{E}$-evaluation (Definition 13.86) and $R,\mathcal{E}$-standard reduction sequence (Definition 13.88). The immediate difficulty is that $R,\mathcal{E}$-evaluation produces a nondeterministic relation, trivially precluding useful correspondence with $\longmapsto_\bullet$. On a case-by-case basis, we customize that abstract definition by prioritizing the schemata, and requiring that an evaluation step exercise a schema of the highest priority possible, and the largest redex (thus, the shallowest possible evaluation context) possible within that priority. This, however, produces an anomaly in the abstract definition of $R,\mathcal{E}$-standard reduction sequence. According to that abstract definition, once the sequence stops performing evaluation steps (Criterion 13.88(b)), it has to descend recursively into proper subterms (Criterion 13.88(c)); but with the additional constraints on evaluation, it becomes possible for a non-evaluation reduction step to be on the term as a whole, rather than on a subterm. For this case, we extend the definition of $R,\mathcal{E}$-standard reduction sequence to allow top-level non-evaluation steps, in order of priority, just before descending recursively into the subterms; technically, this is another criterion added to the definition:[4]

Criterion 14.9 (Extending Definition 13.88)

$\vec{T_1}$ is a sequence of non-$R,\mathcal{E}$-evaluation $\longrightarrow^R$ steps in descending order of priority (using the priorities assigned to customize $R,\mathcal{E}$-evaluation), $\vec{T_2}$ is an $R,\mathcal{E}$-standard reduction sequence that does not include any $R,\mathcal{E}$-evaluation steps and does not include any $\longrightarrow^R$ steps, and $\vec{T} = \vec{T_1} \cdot \vec{T_2}$.

[4]In principle, these extensions could be incorporated directly into the abstract definitions of $R,\mathcal{E}$-evaluation and $R,\mathcal{E}$-standard reduction sequence in Chapter 13, Definitions 13.86 and 13.88, without changing any of the results in that chapter. However, the *proofs* in that chapter would have to take account of the extensions, and we prefer not to further complicate the abstract treatment there to handle a problem that only arises here.

Note that this extension does not invalidate Lemma 13.89, because it prepends top-level non-evaluations based on the absence of later evaluation steps *rather than* based on recursion via Criterion 13.88(c). The proof of the lemma blithely assumes that evaluation steps in the recursive phase can be shifted leftward to the end of the evaluation phase, which is true only because all intervening reductions are to other subterms; the new criterion violates this assumption, and must therefore assure that in its newly introduced case, the assumption isn't needed.

The intended completeness results, at least for the full calculi, $\longmapsto^*_\bullet \subseteq \longrightarrow^*_\bullet$, are generally straightforward and are not systematically pursued.

We assume that operational equivalence $\simeq_\bullet$ (defined following (8.14) in §8.3.2) uses the same notion of R-observable as does $\simeq^{\mathcal{E}}_R$ — which, until and unless stated otherwise, is the notion in Definition 13.90.

The schemata for the semantics are:

$\mathfrak{k}_i$-semantics (§10.3): Schemata 10.3. Evaluation contexts Syntax 10.2.

$\mathfrak{k}C$-semantics (§11.2): Schemata 11.6 and 11.7.

$\mathfrak{k}S$-semantics (§12.2): bubbling-up Schemata 12.19; symbol-evaluation Schema 12.20, amending $\mathfrak{k}_i$-semantics; ***$define!*** and lifting Schemata 12.22; and garbage-collection Schemata 12.23.

14.2.1 $\mathcal{S}$-regular evaluation order

It will emerge that strict right-to-left order works for four of the seven regular calculi, but fails $\mathcal{S}$-regularity for the other three ($\mathfrak{k}_e$-, $\mathfrak{k}_p$-, and $\mathfrak{k}_i$-calculus). Slightly adjusted evaluation orders are $\mathcal{S}$-regular for the other three calculi — but the adjustments are entirely unproblematic only for $\mathfrak{k}_e$-calculus, because we aren't trying to synch that calculus with a declared $\bullet$-semantics. Intended results for $\mathfrak{k}_i$-calculus, especially, should use the same evaluation order as $\mathfrak{k}C$- and $\mathfrak{k}S$-semantics, which conflicts with the adjustments for $\mathcal{S}$-regularity.

For $\mathcal{S}$-regularity of $\mathcal{E}$ (Definition 13.114), the left-hand side of each concrete schema must not contain any local irregularity, i.e., any unconstrained subterm with a constrained later sibling ("sibling" meaning another subterm of the same minimal nontrivial poly-context). No local irregularities can occur *within* a semantic variable based on V (which must match a value), which accounts for a large fraction of all degrees of freedom in the schemata. Note that some occurrences of V are themselves constraints on subterms, as in [combine $\langle \mathfrak{k}x.T\rangle$ V e], where the V constrains a position where the term syntax would allow an arbitrary term; while other occurrences of V merely reflect constraints of the term syntax, as in $[x_s, s] \leftarrow V$, where the V is not an allowable position for a meta-variable (but does represent a possible series of unconstrained subterms, contained within a poly-context minimally satisfying the V). In a related phenomenon, δ-rules are immune to local irregularities by way of Conditions 9.10(1)–9.10(2). Variables e, ω_s, and ω_p (environment, list of stateful bindings,

list of stateless bindings) also cannot have internal local irregularities, noting that e always occurs where the syntax would allow an arbitrary term, while ω_s and ω_p never do.

The $\wr_e$-calculus β-rule, (9.7β), constrains the first, second, and fourth subterms of the operative, but not the third (the body). Because it constrains *any* subterms of the operative frame, an operative frame in $\wr_e$-calculus is not a suspending context (as it fails the second clause in the definition of suspending context, Definition 13.80: a context surrounding an operative frame is not always decisive). Therefore, according to the definition of local irregularity (Definition 13.114), any $\mathcal{S}$-regular evaluation order must put the body last, after all the other subterms of an operative. The definition of $\mathcal{S}$-regular appears to provide an exception if the term (here, the operative) is an evaluation normal form — but this exception is illusory, because the presence of constraints on some of the subterms is exactly what makes the operative frame non-suspending and thereby prevents the term from being evaluation-normal. A deviation from right-to-left evaluation order just for this particular frame (which doesn't occur in the term syntax of any of the declared semantics anyway) does suffice for $\mathcal{S}$-regularity.

The $\wr_p$-calculus ϵ-rule, (9.32ϵ), constrains the first and third subterms of a combine frame, but not the second. This can be handled by another deviation from uniform right-to-left evaluation, observing that none of the other schemata leave *any* of the subterms of a combine unconstrained. However, in the impure calculi we will not have this option because, in order to avoid permuting side-effects during pair evaluation ((9.32p)), we will need the second subterm to precede the first. Consequently, our generic operational soundness result for regular SRSs, Theorem 13.120, isn't always useful to us for $\wr_i$- or $\wr_p$-calculus (depending on the intended use — though it is always useful for $\wr_r$-calculus).

$\wr S$-calculus set frames are unproblematic for right-to-left evaluation order, because the only subterm of a set that is ever constrained by a schema is the rightmost. (The unconstrained subterms of a set frame occur within the values on the right-hand sides of its stateful bindings.) $\wr S$-semantics doesn't specify ordering of the subterms of a set, because it doesn't need to: top-level sets are treated by the semantics as a special case, and subterm sets are either part of a redex or contained within an evaluation normal form.

14.2.2 $R, \mathcal{E}$-evaluation contexts

Three factors determine $R, \mathcal{E}$-evaluation contexts (Definition 13.83): the evaluation order $\mathcal{E}$, which determines which subterms of an $R, \mathcal{E}$-evaluation context are required to be R-evaluation normal forms; the $\longrightarrow^R$-suspending poly-contexts (specifically, the *compound* ones, i.e., those that contain meta-variables), which cannot occur above the meta-variable in an $R, \mathcal{E}$-evaluation context, and which determine the R-evaluation normal forms (Definition 13.81); and the decisively $\longrightarrow^R$-reducible contexts, which

are excluded from being $R,\mathcal{E}$-evaluation contexts.

The evaluation order is strictly right-to-left, as discussed in the preceding subsection, §14.2.1.

In each calculus, each decisively $\longrightarrow^R$-reducible context satisfies the left-hand side of some calculus schema. In order for these not to include any evaluation contexts of the declared semantics (Syntax 10.2), it is sufficient that the enumerated reduction relation $\longrightarrow^R$ be constrained such that whenever a term $E[T]$ satisfies nontrivial evaluation context E, if T is not an $\longrightarrow^R$-evaluation normal form then $E[T]$ is not an $\longrightarrow^R$-redex. For that, in turn, it is sufficient that each $\longrightarrow^R$-redex pattern be a poly-context each of whose meta-variables occurs in a subpattern constrained only to be a value (cf. δ-form Condition 9.10(2); in fact, this is more than sufficient, since an evaluation normal form only needs redexes to be contained within suspending contexts, whereas a value requires active terms to be contained even if they don't give rise to redexes). This sufficient condition is met by $\mathfrak{l}_r$-, $\mathfrak{l}_r C$-, and $\mathfrak{l}_r S$-calculus, but not by the other calculi that share the syntax needed for semantics evaluation contexts (i.e., not by $\mathfrak{l}_p$-, $\mathfrak{l}_i$-, $\mathfrak{l}C$-, or $\mathfrak{l}S$-calculus), as they do not impose the subterm-value constraints of Schemata 10.7.

It remains to identify the compound suspending poly-contexts, Definition 13.80.

To do this, one simply considers all compound minimal nontrivial poly-contexts (modulo α-closure and meta-variable indices), checking each such poly-context P against the left-hand side of each schema in the calculus to verify that (1) no schema left-hand side constrains subterms of P, since if one did that would prevent some $C[P]$ from being decisive, and (2) some schema left-hand side uses P, since that has to happen in order for $C[P]$ to be decisive when C isn't. Whenever a semantic variable based on V occurs in a subterm position on a left-hand side, the minimally satisfying poly-contexts will include some that constrain subterms of every kind of —inactive— poly-context that isn't specifically treated as if it were suspending by the definition of value; and this happens in the β-rule of every $\mathfrak{l}$-calculus; therefore, in each $\mathfrak{l}$-calculus, the only kinds of inactive compound poly-contexts that can possibly be suspending are operative frames and environment frames.

The garbage-collection schemata of the full impure $\mathfrak{l}$-calculi ($\mathfrak{l}C$- and $\mathfrak{l}S$-) constrain subterms of every compound minimal nontrivial poly-context whatsoever, so that in those two calculi there are no suspending contexts.

δ-rules do not constrain proper subterms of either operative frames or environment frames, by Condition 9.10(2). In all the $\mathfrak{l}$-calculi, excepting the garbage collection schemata, only $\mathfrak{l}_e$-calculus Schema 9.7β constrains any subterm of an operative frame; so operative frames are suspending in all the regular $\mathfrak{l}$-calculi except that one (and would be in the full impure $\mathfrak{l}$-calculi except for garbage-collection). No non-garbage-collection schema in any $\mathfrak{l}$-calculus constrains subterms of an environment frame (though parts of the environment frame itself are often constrained), so environment frames are suspending in all the non-garbage-collecting $\mathfrak{l}$-calculi that have them (the only one that doesn't have them being $\mathfrak{l}_x$-calculus).

Accordingly, the right-to-left $\mathfrak{l}_r, \mathcal{E}$-evaluation contexts are just the evaluation contexts of (10.2), and the singular $\mathfrak{l}_r, \mathcal{E}$-evaluation contexts are just the singular evaluation contexts of (11.8). The syntax of the impure $\mathfrak{l}$-calculi has additional compound minimal nontrivial poly-contexts, which are not included in Syntax 11.8: the $\mathfrak{l}C$-calculi have catch and throw frames, while the $\mathfrak{l}S$-calculi have state, set, get, and receive frames. Note that catch, throw, state, and get frames always have exactly one proper subterm, while set and receive frames may have embedded value subexpressions,[5] so that set and receive frames may have any number of subterms. Catch and throw frames occur on schema left-hand sides in (11.14), and their subterms are unconstrained; so catch and throw frames are suspending contexts in $\mathfrak{l}_rC$-calculus, although, even aside from garbage collection, catch frames are not suspending in $\mathfrak{l}C$-calculus since they do occur with constrained subterms in the catch-catch simplification Schema 11.12cc. State and set frames are non-suspending even in $\mathfrak{l}_rS$-calculus because they have constrained subterms in (12.40); get frames are suspending in $\mathfrak{l}_rS$-calculus but not in the full $\mathfrak{l}S$-calculus, because they have constrained subterms in (12.41); and receive frames are non-suspending in both state calculi because they never occur on any schema left-hand side. So the right-to-left $\mathfrak{l}_rC, \mathcal{E}$-evaluation contexts are also just the evaluation contexts of (10.2), while the right-to-left $\mathfrak{l}C$-, $\mathfrak{l}_rS, \mathcal{E}$-, and $\mathfrak{l}S, \mathcal{E}$-evaluation contexts are proper supersets thereof.

14.2.3 Pure $\mathfrak{l}$-calculi

Since $\mathfrak{l}_r$-calculus shares its set of evaluation contexts with $\mathfrak{l}_i$-semantics, and the enumerated relation $\longrightarrow^{\mathfrak{l}r}$ is exactly the base case of of $\longmapsto_{\mathfrak{l}i}$ (the case where $E = \Box$), by compatibility $\longmapsto_{\mathfrak{l}i} = \longmapsto^{\mathcal{E}}_{\mathfrak{l}r}$. Since right-to-left evaluation order is $\mathfrak{l}_r$-regular, $\mathfrak{l}_r$-calculus is subject to Theorem 13.120, and $=_{\mathfrak{l}r} \subseteq \simeq_{\mathfrak{l}i}$ (recalling $\simeq^{\mathcal{E}}_{\mathcal{S}}$ Definition 13.90, and the definition of $\simeq_\bullet$ following (8.14) in §8.3.2).

$\mathfrak{l}_i$- and $\mathfrak{l}_p$-calculus also share their evaluation contexts with $\mathfrak{l}_i$-semantics.

Lemma 14.10 Suppose $\mathfrak{l}_p$-calculus terms T, T_k, e.

If T is not $\longrightarrow^*_{\mathfrak{l}r}$-reducible to a value, then [eval T e] isn't $\longrightarrow^*_{\mathfrak{l}p}$-reducible to a value. If T_1 or T_2 is not $\longrightarrow^*_{\mathfrak{l}r}$-reducible to a value, then [combine T_1 T_2 e] isn't $\longrightarrow^*_{\mathfrak{l}p}$-reducible to a value. ■

[5] These are subexpressions but not sub*terms*, because the uppermost part of their syntax is embedded within the minimal nontrivial set or receive frame — including all the syntax elements that make it a value: operative and environment frames that would be suspending if they weren't embedded within another minimal nontrivial frame, but that in embedded form cannot be independently suspending even though other frames cannot bubble up through them. This seems to suggest that suspension may not be naturally a property of the minimal nontrivial poly-contexts themselves, as has been defined here (Definition 13.80); instead, suspension may be more naturally a property of particular meta-variable positions *within* a minimal nontrivial poly-context. The approach here attempted to generalize concepts from λ-calculus, where this distinction is not apparent since the only suspending poly-contexts have arity 1.

Proof. Both results simultaneously, by induction on number of reduction steps, with each step divided into cases by schema. ■

Theorem 14.11 (Operational soundness)
If $T_1 =_{\mathfrak{l}p} T_2$ then $T_1 \simeq_{\mathfrak{l}i} T_2$. ■

Proof. When broadening Schemata 10.7 and 10.6 to their $\mathfrak{l}_p$-calculus forms, Schemata 9.32, if the relaxed subterms (those required to be values by schema left-hand sides in $\mathfrak{l}_r$-calculus but unconstrained in $\mathfrak{l}_p$-calculus) can be reduced to values, then by Church–Rosser-ness, exercising these schemata before the subterms have actually been reduced to values cannot affect reduction of the whole term to a value. Supposing that no substitution is imposed from outside (which would uniformly affect the entire potential redex that we're considering), if any of the relaxed subterms cannot be reduced to values, then by Lemma 14.10, exercising these schemata still cannot affect reduction of the whole term to a value. Substitution affecting the entire potential redex is not affected by reductions of subterms of the potential redex (as might happen in an impure calculus if a subterm emitted a side-effect), so the interaction with external context is one-way (incoming); and this incoming external influence preserves the property that either the subterms are reducible to values or the whole isn't — especially, external substitution cannot *disable* reduction of any subterm to a value (verifiable by cases). So again, early exercise of the potential redex cannot result in a value unless waiting for value-subterms would have been possible. So, letting $\mathcal{E}_p$ be the identified $\mathfrak{l}_p$-regular evaluation order, $\simeq^{\mathcal{E}_p}_{\mathfrak{l}p} = \simeq^{\mathcal{E}_p}_{\mathfrak{l}r}$. Also, letting $\mathcal{E}_r$ be strictly right-to-left, $\simeq^{\mathcal{E}_p}_{\mathfrak{l}r} = \simeq^{\mathcal{E}_r}_{\mathfrak{l}r}$; as already established, $\simeq^{\mathcal{E}_r}_{\mathfrak{l}r} = \simeq_{\mathfrak{l}i}$; and since $\mathcal{E}_p$ is $\mathfrak{l}_p$-regular, $=_{\mathfrak{l}p} \subseteq \simeq^{\mathcal{E}_p}_{\mathfrak{l}p}$. In all, $=_{\mathfrak{l}p} \subseteq \simeq_{\mathfrak{l}i}$. ■

14.2.4 Control $\mathfrak{l}$-calculi

Since $\mathfrak{l}_r C$-calculus shares its set of evaluation contexts with $\mathfrak{l}C$-semantics, the only $\mathfrak{l}_r C, \mathcal{E}$-evaluation steps not already part of $\mathfrak{l}_r, \mathcal{E}$-evaluation are the bubblings up. Repeated calculus catch bubbling-up evaluation steps, via (11.12c), are necessarily equivalent to a single $\mathfrak{l}C$-semantics catch bubbling-up step via (11.6c); $\mathfrak{l}_r C, \mathcal{E}$-evaluation stops once a single catch reaches the top level, since catch is a suspending context in $\mathfrak{l}_r C$-calculus. Throw bubbling-up evaluation steps in $\mathfrak{l}_r C, \mathcal{E}$-evaluation are unsound in the sense of $\longmapsto^{\mathcal{E}}_{\mathfrak{l}rc} \not\subseteq \longmapsto_{\mathfrak{l}c}$, because $\mathfrak{l}C$-semantics doesn't have any corresponding schema for bubbling up a throw with no matching catch (Schemata 11.6) — but not unsound in the sense of $\simeq^{\mathcal{E}}_{\mathfrak{l}rc} \not\subseteq \simeq_{\mathfrak{l}c}$, because terms with an unmatched top-level throw cannot have any effect on $R, \mathcal{E}$-operational equivalence, due to the precondition excluding free variables in Definition 13.90 (which is why $\mathfrak{l}C$-semantics doesn't bother to provide a schema for this case).

While $\mathfrak{l}_r C, \mathcal{E}$-evaluation is semantically unsound (as opposed to operationally unsound), $\mathfrak{l}_r C, \mathcal{E}$-operational equivalence has a triviality problem. Because $\mathfrak{l}_r C$-calculus lacks garbage collection (Schema 11.12g), once a term T emits a side-effect, all further reducts are compound. (That is, if T has a top-level catch or throw, and $T \longrightarrow^*_{\mathfrak{l}rc} T'$, then T' has a top-level catch or throw.) Since R-observables are atomic (i.e., minimal nontrivial) under Definition 13.90, any $\mathfrak{l}_r C, \mathcal{E}$-evaluation step that involves a catch or throw schema can only affect $\simeq^{\mathcal{E}}_{\mathfrak{l}rc}$ through normalizability (Condition 13.90(a)); there is no observational difference (via Condition 13.90(b)) between $\simeq^{\mathcal{E}}_{\mathfrak{l}rc}$ and $\simeq^{\mathcal{E}}_{\mathfrak{l}r}$. The same applies to any subset of $\mathfrak{l}C$-calculus that omits garbage collection, up to and including full $\mathfrak{l}C$-calculus minus garbage collection. On the other hand, *including* garbage-collection renders the $R, \mathcal{E}$-evaluation relation $\longmapsto^{\mathcal{S}}_R$ uninteresting since there are no suspending contexts. When treating non-garbage-collecting extensions of $\mathfrak{l}_r C$-calculus, we therefore amend Definition 13.90 by broadening the sense of R-observable to allow a single surrounding catch frame provided it doesn't capture any variable in the (necessarily atomic) subterm.

Let $\mathfrak{l}C'$-calculus modify full $\mathfrak{l}C$-calculus by omitting garbage-collection Schema 11.12g, and let $\mathfrak{l}_r C'$-calculus modify $\mathfrak{l}C'$-calculus by imposing the value-subterm restrictions of $\mathfrak{l}_r$-calculus (Schemata 10.7). We are interested in the relationship between $\simeq^{\mathcal{E}}_{\mathfrak{l}c'}$ and $\simeq_{\mathfrak{l}c}$. We work up to $\mathfrak{l}C'$-calculus by increments.

In order for $R, \mathcal{E}$-evaluation to simulate $\mathfrak{l}C$-semantics bubblings up, R needs catch-catch and catch-throw simplifications ((11.12cc) and (11.12ct)). We therefore take, as the first increment, $\mathfrak{l}_r$-calculus plus bubbling-up Schemata 11.14 and catch-catch and catch-throw simplifications. For determinism, we prioritize bubbling up before simplification. Since catch frames are now non-suspending, $R, \mathcal{E}$-evaluation contexts can have a top-level catch frame — and $R, \mathcal{E}$-evaluation contexts change in no other way, since a catch surrounded by any singular $R, \mathcal{E}$-evaluation context would be decisively reducible (via either catch-catch simplification or catch bubbling-up). Other than lifting pure semantic steps (Schema 11.7), the only new $\mathcal{S}, \mathcal{E}$-evaluation steps simulate, using sequences of steps, the catch-catch and catch-throw semantics steps ((11.6cc) and (11.6ct)), so that $\simeq^{\mathcal{E}}_R \subseteq \simeq_{\mathfrak{l}c}$.

For the second increment, we add throw-throw simplification, (11.12tt), producing $\mathfrak{l}_r C'$-calculus. This has no effect on $R, \mathcal{E}$-operational equivalence, because if the outer throw has no matching catch, then the term has a free variable (and is therefore irrelevant to $\simeq^{\mathcal{E}}_R$), while if the outer throw *does* have a matching catch, the reduction order enforced by $R, \mathcal{E}$-evaluation will always cause the outer throw to bubble up and then be eliminated without ever applying the throw-throw simplification.

For the third and final increment, we remove the value-subterm constraints that distinguish $\mathfrak{l}_r$-calculus from $\mathfrak{l}_i$-calculus (cf. Schemata 10.7), producing $\mathfrak{l}C'$-calculus.

Lemma 14.12 Suppose R is the enumerated relation of the catch-catch, throw-throw, and catch-throw simplifications ((11.12cc), (11.12tt), and (11.12ct)).

(a) If $T_1 \longrightarrow_R T_2 \longrightarrow^*_{\mathit{lrc}} T_3$, then there exists T_2' such that $T_1 \longrightarrow^*_{\mathit{lrc}} T_2' \longrightarrow^*_R T_3$.

(b) If $T_2' \longrightarrow_R T_2$, $T_3' \longrightarrow_R T_3$, $T_2' \longrightarrow^*_{\mathit{lrc}} T_4'$, and $T_3' \longrightarrow^*_{\mathit{lrc}} T_4'$, then there exist T_2'', T_3'', and T_4'' such that
$T_4' \longrightarrow^*_{\mathit{lrc}} T_4''$,
$T_2 \longrightarrow^*_{\mathit{lrc}} T_2''$ and $T_4'' \longrightarrow^*_R T_2''$, and
$T_3 \longrightarrow^*_{\mathit{lrc}} T_3''$ and $T_4'' \longrightarrow^*_R T_3''$.
(See Figure 14.1.) ■

Proof. For (a), suppose $T_1 \longrightarrow_R T_2 \longrightarrow^*_{\mathit{lrc}} T_3$. Simplification $T_1 \longrightarrow_R T_2$ eliminates one of two adjacent catch/throw frames. Decorate the syntax of T_1 and T_2 by marking those two frames in T_1, and the one of them that remains in T_2. When taking a $\longrightarrow_{\mathit{lrc}}$ step, mark each frame in the result that is a copy of a marked frame. Let $\vec{T_2}$ be a $\longrightarrow_{\mathit{lrc}}$-reduction sequence from T_2 to T_3. Each term in $\vec{T_2}$ has one or more marked frames; the number of marked frames may be increased by any reduction step that involves substitution. Construct a $\longrightarrow_{\mathit{lrc}}$-reduction sequence $\vec{T_1}$ from T_1 as follows. Most of the terms in $\vec{T_1}$ have the same structure as those in $\vec{T_2}$ except that wherever a marked frame occurs in a term in $\vec{T_2}$, the corresponding term in $\vec{T_1}$ has two adjacent marked frames. When $\vec{T_2}$ performs a reduction step *other than* bubbling up a marked frame, $\vec{T_1}$ performs the same structural term transformation, which leaves each pair of marked frames intact (thought possibly copied multiple times by the transformation); marked pairs of frames remain adjacent since each such pair is either entire below or entirely outside the exercised redex pattern, so that the result of the $\vec{T_1}$ step corresponds to the result of the $\vec{T_2}$ step. When $\vec{T_2}$ bubbles up a marked frame, $\vec{T_1}$ performs two steps, bubbling up first the outer and then the inner of the two marked frames at the corresponding point in its term's structure; the result of the second bubbling up in $\vec{T_2}$ is a term corresponding to the result of the single bubbling up in $\vec{T_2}$. Let T_2' be the last term in $\vec{T_1}$; T_2' differs from T_3 (the last term in $\vec{T_2}$) only by having a pair of marked frames at each point where T_3 has a single marked frame. Simplifying the marked pairs of frames in T_2' yields $T_2' \longrightarrow^*_R T_3$.

The strategy for (b) is illustrated by Figure 14.1. Terms T_2', T_3', T_4', T_2, and T_3 and the reductions between them are given; the other terms and reductions are deduced — first the reduction to T_4'', then the reductions from T_4'' to T_2'' and T_3'', and finally the reductions from T_2 and T_3 to T_2'' and T_3''.

Suppose $T_2' \longrightarrow_R T_2$, $T_3' \longrightarrow_R T_3$, $T_2' \longrightarrow^*_{\mathit{lrc}} T_4'$, and $T_3' \longrightarrow^*_{\mathit{lrc}} T_4'$. Simplification $T_2' \longrightarrow_R T_2$ collapses entire consecutive blocks of frames in T_2' (frames nested each in the next with nothing between) each to an individual frame in T_2. Let $\vec{T_2'}$ be a $\longrightarrow_{\mathit{lrc}}$-reduction sequence from T_2' to T_4'. By decorating the syntax of T_2', and propagating these decorations through each step of $\vec{T_2'}$, we can identify which groups of frames in T_4' correspond to each collapsed frame in T_2. At each step of the sequence, a matched group of frames might be multiplied by copying, changing the number of matched groups in the term (if the group occurs within the V of a substitution $\Box[x_p \leftarrow V]$,

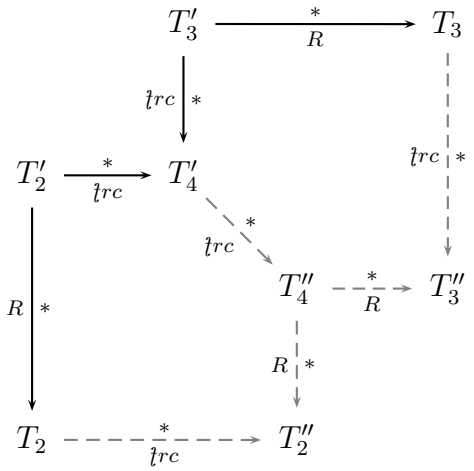


Figure 14.1: Elements of Lemma 14.12(b)

or within the E of a substitution $\Box[x_c \leftarrow E]$), and an evaluation context might be inserted between frames in any matched group (either because a frame in the group (not the innermost of the group) is a catch that bubbles up, or because a frame in the group (not the innermost) is a throw whose binding catch bubbles up). However, each matched group remains intact in that

- any copying must copy all the frames in the group, or none of them, and
- no suspending context or redex pattern can be inserted between any two frames of the group.

(Both of these rely on the subterm-value constraints of $\wr_r$ Schemata 10.7). So throughout the sequence, it is always still possible for all the non-outermost frames of any one matched group to be bubbled up until the entire group is consecutive again; and particularly, this is still true in T_4'. We wish to show that, in a finite number of steps, one can bubble up all of the non-outermost frames of all these matched groups, producing a term in which every matched group of frames is consecutive. This can be done in two phases.

Call a frame *outstanding* if it belongs to a matched group, and it isn't the outermost frame in its group, and the frame immediately outside it isn't part of its group. (Per the above observations, every outstanding frame is immediately surrounded by a singular evaluation context.)

In the first phase, do all required catch bubblings up, in *bottom-up order*. As long as there is any outstanding catch in the term, choose any outstanding catch such that no other outstanding catch occurs within the nearest catch surrounding this one; and perform a single bubbling up of this outstanding catch. This single bubbling up makes copies of the singular evaluation context that is bubbled up through — but

all but one of these copies is inserted just inside a matching throw, and therefore does not increase the total number of outstanding catches. One copy of the singular evaluation context is inserted just below the outstanding catch that bubbles up — and it's possible that that outstanding catch was consecutive with another catch below it in the same group, so that the catch below it becomes outstanding; but in that case, we can then bubble up that next catch down, and so on until we have exhausted all consecutive catches further down in this group. The entire operation has reduced, by exactly one, the total number of singular evaluation contexts stacked above outstanding catches; so by induction, this first phase will terminate after a finite number of such operations, leaving a term with no outstanding catches.

In the second phase, repeatedly choose any outstanding throw, and bubble it up. Since throw bubbling-up merely deletes the singular evaluation context directly above it, each such step decreases the size of the term, and does not introduce any outstanding catches; so this phase terminates after a finite number of steps, leaving a term in which every matched group of frames is consecutive.

Moreover, by using the same techniques for reasoning about $T_3' \longrightarrow_R^* T_3$ and $T_3' \longrightarrow_{\not\iota rc}^* T_4'$, we can also identify matched groups of frames in T_4' that correspond to collapsed frames in T_3; call these T_3-*groups*, as opposed to the T_2-*groups* corresponding to collapsed frames in T_2. Using the same techniques as above for bubbling up from T_4', but taking into account T_3-groups as well as T_2-groups, we can derive a term by bubbling up from T_4' such that every T_2-group is consecutive *and* every T_3-group is consecutive. Call this term T_4''.

Each T_2-group in T_4'' can be reduced via $\longrightarrow_R^*$ to a single collapsed frame (in a unique way, up to $\equiv_\alpha$, since $\longrightarrow_R$ evidently —by cases— has the diamond property, Definition 13.63). Call the result of doing all these reductions T_2''. Similarly, call the result of reducing all T_4'''s T_3-groups T_3''.

It only remains to show that $T_2 \longrightarrow_{\not\iota rc}^* T_2''$ (from which $T_3 \longrightarrow_{\not\iota rc}^* T_3''$ follows by symmetry). Let $\vec{T_2''}$ be a $\longrightarrow_{\not\iota rc}$-sequence from T_2' to T_4''. We would like this sequence to have the twin properties that

- if $\vec{T_2''}(k)$ contains a T_2-outstanding catch (that is, a catch outstanding relative to a T_2-group), then $\vec{T_2''}(k) \longrightarrow_{\not\iota rc} \vec{T_2''}(k+1)$ is a bubbling up of a T_2-outstanding catch; and

- if $\vec{T_2''}(k)$ does not contain a T_2-outstanding catch, but does contain a T_2-outstanding throw, then $\vec{T_2''}(k) \longrightarrow_{\not\iota rc} \vec{T_2''}(k+1)$ is a bubbling up of a T_2-outstanding throw.

Suppose $\vec{T_2''}$ has these two properties. We can straightforwardly construct a $\longrightarrow_{\not\iota rc}$-sequence $\vec{T_2}$ from T_2 to T_2'', as follows. For each term that has nothing T_2-outstanding, construct a corresponding term for $\vec{T_2}$ by collapsing all the T_2-groups. The terms of $\vec{T_2}$ are just these corresponding terms; the first and last terms of $\vec{T_2}$ are T_2 corresponding

to T_2', and T_2'' corresponding to T_4''. Suppose $\vec{T}_2(k)$ corresponds to $\vec{T}_2''(j)$, and $\vec{T}_2(k+1)$ corresponds to $\vec{T}_2''(i)$ (the next term in $\vec{T}_2''$ after $\vec{T}_2''(j)$ that doesn't have anything T_2-outstanding). Then, straightforwardly (by cases), $\vec{T}_2(k) \longrightarrow_{\ell rc} \vec{T}_2(k+1)$. So $\vec{T}_2$ is a $\longrightarrow_{\ell rc}$-sequence from T_2 to T_2'', and we're done.

Finally, suppose $\vec{T}_2''$ does not have these two properties. Then we describe how to repeatedly modify it until it does.

Suppose $\vec{T}_2''(k)$ contains at least one T_2-outstanding catch, but $\vec{T}_2''(k) \longrightarrow_{\ell rc} \vec{T}_2''(k+1)$ is not a bubbling up of a T_2-outstanding catch; and suppose $\vec{T}_2''(k)$ is the last term in the sequence that has both of these properties. Choose a T_2-outstanding catch in $\vec{T}_2''(k)$ such that no other outstanding catch occurs within the nearest catch surrounding this one; and modify the sequence by inserting immediately after $\vec{T}_2''(k)$ a bubbling up of this T_2-outstanding catch. Modify each subsequent term of the sequence by this same bubbling up, until the point where that T_2-outstanding frame was already being bubbled up (after which, later terms in the sequence don't have to be modified; and there must come a point in the sequence where this happens, since the sequence eventually arrives at T_4'' that has no T_2-outstanding frames). Some pairs of consecutive terms later in the sequence are no longer related by a single $\longrightarrow_{\ell rc}$ step, because the single step that used to relate those two terms (before at least one of the terms was modified) has been copied multiple times by the modification, so that to get from the left-hand of these terms to the right-hand one, one needs to perform that original step on each of the subterm copies. If the original step was a bubbling up of a T_2-outstanding frame, or if the original step did not introduce any new T_2-outstanding catch, then simply insert, at that point in the sequence, however many intermediate steps are necessary to perform the original step on all the copies; in either of these cases, the added steps do not violate the ordering that we're trying to introduce into the sequence. Suppose the original step was not a bubbling up of a T_2-outstanding frame, and it *did* introduce a new T_2-outstanding catch. (In this case, the original step must have been a bubbling up of the outermost catch in a T_2-group.) Then the original step, that we now need to do on multiple copies, must have been immediately followed by consecutive steps that eliminated all T_2-outstanding catches — because we chose $T_2''(k)$ to be the last term in the sequence that failed to immediately address a T_2-outstanding catch. Treat this entire subsequence —the original step followed by the consecutively following elimination of T_2-outstanding catches— as a unit, and introduce copies of the entire subsequence, one copy of the subsequence after another.

After these revisions, we have another $\longrightarrow_{\ell rc}$-sequence from T_2' to T_4'', which we again call $\vec{T}_2''$. $\vec{T}_2''(k)$ is the same as it was in the previous version, and $\vec{T}_2''(k) \longrightarrow_{\ell rc} \vec{T}_2''(k+1)$ is a bubbling up of a T_2-outstanding catch. The step after this one, $\vec{T}_2''(k+1) \longrightarrow_{\ell rc} \vec{T}_2''(k+2)$, is the only step from $\vec{T}_2''(k)$ on that might possibly fail to immediately address a T_2-outstanding catch; and another revision such as we just

did will push it out further, until eventually, when we have bubbled up all the T_2-outstanding catches in the term (a finite process, as observed earlier), we will have a revised sequence $\vec{T_2''}$ in which we have decremented the number of steps that fail to immediately address a T_2-outstanding catch. By induction, we can revise $\vec{T_2''}$ so that T_2-outstanding catches are always immediately addressed, satisfying the first of the two properties we want of $\vec{T_2''}$.

Suppose $\vec{T_2''}$ always immediately addresses T_2-outstanding catches. Suppose $\vec{T_2''}(k)$ contains at least one T_2-outstanding throw, but $\vec{T_2''}(k) \longrightarrow_{\wr rc} \vec{T_2''}(k+1)$ is not a bubbling up of a T_2-outstanding throw; and suppose $\vec{T_2''}(k)$ is the last term in the sequence that has both of these properties. Modify the sequence by inserting immediately after $\vec{T_2''}(k)$ a bubbling up of some T_2-outstanding throw. Modify each subsequent term of the sequence by this same bubbling up, until the point where that T_2-outstanding frame was already being bubbled up. Some pairs of consecutive terms later in the sequence are now identical, because the step from one to the next was exercising a redex in part of the term that has been deleted by the throw bubbling up; simply delete the second term of any such consecutive pair, decrementing the length of the sequence. We again have a $\longrightarrow_{\wr rc}$-reduction sequence from T_2' to T_4''; again call it $\vec{T_2''}$. No new T_2-outstanding catches have been introduced, so the revised sequence always addresses T_2-outstanding catches immediately. $\vec{T_2''}(k)$ is the same as it was in the previous version, and $\vec{T_2''}(k) \longrightarrow_{\wr rc} \vec{T_2''}(k+1)$ is a bubbling up of a T_2-outstanding throw. The step after this one, $\vec{T_2''}(k+1) \longrightarrow_{\wr rc} \vec{T_2''}(k+2)$, is the only step from $\vec{T_2''}(k)$ on that might possibly fail to immediately address a T_2-outstanding frame; and as before we can continue to insert bubblings up of T_2-outstanding throws, pushing the failing step further and further to the right, until we have eliminated all the T_2-outstanding throws before it, and we have a revised sequence $\vec{T_2''}$ with a decremented number of steps that violate the intended properties. By induction, we can revise $\vec{T_2''}$ until it always immediately addresses T_2-outstanding catches and then T_2-outstanding throws, allowing us to construct $\vec{T_2}$ and establish $T_2 \longrightarrow^*_{\wr rc} T_2''$. ■

Theorem 14.13 (Church–Rosser-ness)
$\longrightarrow^{\wr rc'}$ is Church–Rosser.
$\longrightarrow^{\wr c'}$ is Church–Rosser.
$\longrightarrow^{\wr c}$ is Church–Rosser. ■

Proof. Let R be as in Lemmata 14.12. R is evidently Church–Rosser (as already remarked: by cases, $\longrightarrow_R$ has the diamond property, Definition 13.63); and since $\wr_r C$-calculus is a regular SRS, $\longrightarrow^{\wr rc}$ is Church–Rosser. Suppose $T_1 \longrightarrow^*_{\wr rc'} T_2$ and $T_1 \longrightarrow^*_{\wr rc'} T_3$. By Lemma 14.12(a), let $T_1 \longrightarrow^*_{\wr rc} T_2' \longrightarrow^*_R T_2$ and $T_1 \longrightarrow^*_{\wr rc} T_3' \longrightarrow^*_R T_3$. Since $\longrightarrow^{\wr rc}$ is Church-Rosser, let $T_2' \longrightarrow^*_{\wr rc} T_4'$ and $T_3' \longrightarrow^*_{\wr rc} T_4'$. By

Lemma 14.12(b), let $T_2 \longrightarrow^*_{\mathfrak{l}rc} T_2''$, $T_3 \longrightarrow^*_{\mathfrak{l}rc} T_3''$, $T_4'' \longrightarrow^*_R T_2''$, and $T_4'' \longrightarrow^*_R T_3''$. Since R is Church–Rosser, let $T_2'' \longrightarrow^*_R T_4$ and $T_3'' \longrightarrow^*_R T_4$. Then $T_2 \longrightarrow^*_{\mathfrak{l}rc} T_2'' \longrightarrow^*_R T_4$ and $T_3 \longrightarrow^*_{\mathfrak{l}rc} T_3'' \longrightarrow^*_R T_4$; so $\longrightarrow^{\mathfrak{l}rc'} = (\longrightarrow^{\mathfrak{l}rc} \cup R)$ is Church-Rosser.

$\longrightarrow^{\mathfrak{l}c'}$ differs from $\longrightarrow^{\mathfrak{l}rc'}$ only by relaxing the value-subterm constraints of (10.7). Given a term containing a potential redex due to this relaxation, there are three kinds of reductions *alternative* to exercising that redex; in each of these three cases, we verify that the result of exercising the potential redex before the alternative can also be obtained if the alternative is exercised first.

(1) The alternative is a reduction within a subterm (of the potential redex) — which manifestly *cooperates* (Definition 13.63) with exercise of the potential redex.

(2) The alternative is exercise of a containing redex — which could impose a substitutive transformation on the potential redex, or could make multiple copies of the potential redex (when substituting it into something else), either of which cooperates with (possibly multiple, parallel) exercise of the potential redex.

(3) The alternative is bubbling up of a frame emitted by a subterm, moving into the potential redex pattern (and possibly through the pattern in a single step, depending on which of the four relaxed schemata is involved) — in which case, the frame can always be bubbled up further, as necessary, so that it passes entirely through the potential redex pattern, either making copies of the pattern (if the frame is a catch), each of which can be exercised later, or deleting the pattern so that it no longer matters whether the pattern was exercised beforehand.

$\longrightarrow^{\mathfrak{l}c}$ differs from $\longrightarrow^{\mathfrak{l}c'}$ only by the addition of garbage collection. Let Q be the enumerated relation of garbage-collection Schema 11.12g; then $\longrightarrow^*_Q$ cooperates (Definition 13.63) with $\longrightarrow^?_{\mathfrak{l}c}$ (by cases). Church–Rosser-ness of $\longrightarrow^{\mathfrak{l}c}$ follows by induction. ■

Definition 14.14 A *side-effect-ful value* (over $\mathfrak{l}C$-calculus) is a term of the form $C[V]$, where C is a (possibly trivial) composition of catch frames and throw frames. ■

Lemma 14.15 Suppose $\mathfrak{l}C$-calculus terms T, T_k, e.

If T is not $\longrightarrow^*_{\mathfrak{l}rc}$-reducible to a side-effect-ful value, then [eval T e] isn't $\longrightarrow^*_{\mathfrak{l}c'}$-reducible to a side-effect-ful value.

If T_2 is not $\longrightarrow^*_{\mathfrak{l}rc}$-reducible to a side-effect-ful value,

or

T_1 is not $\longrightarrow^*_{\mathfrak{l}rc}$-reducible to a side-effect-ful value, and T_2 is not reducible to a side-effect-ful value with a free throw (i.e., a throw outside the value whose variable is free in the term),

then

[combine T_1 T_2 e] isn't $\longrightarrow^*_{\mathfrak{l}c'}$-reducible to a side-effect-ful value. ■

Proof. As for Lemma 14.10. ■

Theorem 14.16 (Standard normalization)
Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_c$.

If there exists a $\mathfrak{l}_r C'$-evaluation normal form N such that $T \longrightarrow^*_{\mathfrak{l}rc'} N$, then there exists a $\mathfrak{l}_r C'$-evaluation normal form N' such that $T \longmapsto^{\mathcal{E}*}_{\mathfrak{l}rc'} N'$. ■

Proof. By regularity (and Theorem 13.118), $T_1 \longrightarrow^*_{\mathfrak{l}rc} T_2$ iff there exists a $\mathfrak{l}_r C, \mathcal{E}$-standard reduction sequence from T_1 to T_2. If the definition of $\mathfrak{l}_r C, \mathcal{E}$-standard reduction sequence were changed by not treating catch frames as suspending contexts, this would not perturb the standard reduction sequences at all, because the fact that a catch frame is suspending only matters to the definition of $\mathfrak{l}_r C, \mathcal{E}$-evaluation when the catch frame occurs at the top level of the term (similarly to Schemata 11.6cc and 11.6ct) — and a standard reduction sequence would then simply descend recursively into the sole subterm of the catch anyway. The definition of $\mathfrak{l}_r C', \mathcal{E}$-standard reduction sequence further differs from this by the fact that simplification redex patterns become decisively reducible; however, because $\mathfrak{l}_r C', \mathcal{E}$-evaluation prefers bubbling up to simplification, this too does not perturb the standard reduction sequences: once the simplifiable pattern occurs at the top level, where evaluation would be able to simplify it, standard order can simply recurse into the sole subterm.

Given any $\longrightarrow_{\mathfrak{l}rc'}$-reduction sequence from T to N, we can find another from T to N in which all the simplifications are done last (by Lemma 14.12(a)). By the above reasoning, we can then put the non-simplifying prefix of this sequence into standard order, without introducing any simplifications to do so; call this standard prefix $\vec{T}$ Assume without loss of generality that $\vec{T}$ puts all evaluation steps consecutively at its start, and that all the recursive subterm standard-reduction sequences do likewise (by Lemma 13.89). Consider the earliest point in $\vec{T}$ where evaluation would permit a simplification (catch-catch or catch-throw). Modify $\vec{T_1}$ by inserting the simplification at this point; the rest of $\vec{T_1}$ remains the same (other than the simplification having been made), the rest of $\vec{T_1}$ is still a standard reduction sequence, it still puts all evaluation steps before all non-evaluation steps, and the rest of $\vec{T_1}$ is no longer than it was before. Repeat this with the next point where a simplification could occur as an evaluation step, and continue to repeat it until every such point in $\vec{T}$ is simplified; this is a finite process, since any given term can only be simplified finitely many times. In the final revision of $\vec{T}$, let N' be the result of the last evaluation step, and suppose N' isn't an evaluation normal form. Since it isn't an evaluation normal form, there is some evaluation step possible from N'. The evaluation step from N' can't be a simplification, because if it were, the above construction would have inserted the simplification, so that the next step would be an evaluation step (contradicting the assumption); and the evaluation step from N' can't be a $\longrightarrow_{\mathfrak{l}rc}$ step, because if it

were, the original, non-simplifying version of $\vec{T}$ would have exercised it next, and by the above construction it would still be exercised next. So, by reductio ad absurdum, N' is an evaluation normal form. ■

Theorem 14.17 (Operational soundness)
If $T_1 =_{\mathfrak{l}c} T_2$ then $T_1 \simeq_{\mathfrak{l}c} T_2$. ■

Proof. Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_c$.

$\simeq^{\mathcal{E}}_{\mathfrak{l}rc'} \subseteq \simeq_{\mathfrak{l}c}$ was established in the discussion at the top of this subsection (§14.2.4, preceding Lemmata 14.12); so for soundness, we want to show that $=_{\mathfrak{l}rc'} \subseteq \simeq^{\mathcal{E}}_{\mathfrak{l}rc'}$. Suppose $T_1 =_{\mathfrak{l}rc'} T_2$.

Suppose $T_1 \longmapsto^{\mathcal{E}*}_{\mathfrak{l}rc'} N$, and N is a $\mathfrak{l}_r C'$-evaluation normal form. By Church–Rosser-ness (Theorem 14.13), let T' such that $N \longrightarrow^*_{\mathfrak{l}rc'} T'$ and $T_2 \longrightarrow^*_{\mathfrak{l}rc'} T'$. Since $N \longrightarrow^*_{\mathfrak{l}rc'} T'$ and N is a $\mathfrak{l}_r C'$-evaluation normal form, T' is a $\mathfrak{l}_r C'$-evaluation normal form (by Lemma 13.82). Since $T_2 \longrightarrow^*_{\mathfrak{l}rc'} T'$ and T' is a $\mathfrak{l}_r C'$-evaluation normal form, there exists a $\mathfrak{l}_r C'$-evaluation normal form T'' such that $T_2 \longmapsto^{\mathcal{E}*}_{\mathfrak{l}rc'} T''$ (by Theorem 14.16).

Further, if N is an observable, then $T'' = N$ (by Church–Rosser-ness, Theorem 14.13).

The extension of the result by relaxing the value-subterm constraints of Schemata 10.7 is similar to the proof of Theorem 14.11. Note that as the subterms of a $\longrightarrow^{\mathfrak{l}i}$-redex are reduced to side-effect-ful values, from right to left, their side-effect frames can be bubbled up out of the redex — which either deletes the $\longrightarrow^{\mathfrak{l}i}$-redex (if one of the side-effect-ful values has a free throw frame), or leaves a $\longrightarrow^{\mathfrak{l}r}$-redex surrounded by side-effect frames. A difference from that proof is that here, a substitution $\Box[x_c \leftarrow C]$ imposed from outside can transform a side-effect-ful value into a term that cannot be reduced to a side-effect-ful value; however, this requires a free throw surrounding the value, since the matching catch frame has to be outside the $\longrightarrow^{\mathfrak{l}i}$-redex, and this case is not unsound because bubbling up the free throw would delete the $\longrightarrow^{\mathfrak{l}i}$-redex.

Finally, addition of garbage collection matters to $R, \mathcal{E}$-operational equivalence only in that it obviates the need to allow a garbage-collectable catch frame in the definition of R-observable; otherwise, garbage collection perturbs neither normalizability (Condition 13.90(a)) nor observation (Condition 13.90(b)). So $=_{\mathfrak{l}c} \subseteq \simeq_{\mathfrak{l}c}$. ■

14.2.5 State $\mathfrak{l}$-calculi

$\mathfrak{l}_r S, \mathcal{E}$-evaluation contexts are a proper superset of those of $\mathfrak{l}S$-semantics, because in $\mathfrak{l}_r S$-calculus, state, set, and receive frames are non-suspending.

Most of the anomalies associated with this supersetting will be self-correcting, for various reasons explained below; but one requires some intervention to redress it. A set frame is a minimal nontrivial poly-context that may have an arbitrarily large arity — with only one subterm that we actually *want* to be able to reduce via $R, \mathcal{E}$-evaluation, but possibly any number of others that we nevertheless *can* reduce via $R, \mathcal{E}$-evaluation, according to the definition of that relation (Definition 13.86). For example, frame $P = [\text{state } [[x_s, s] \leftarrow \langle \natural_0.\Box_2 \rangle] \ \Box_1]$ would belong to the usual right-to-left $\mathcal{E}$, being a minimal nontrivial poly-context with its meta-variable indexed right-to-left; and $R, \mathcal{E}$-evaluation would reduce first the subterm at $\Box_1$, then the one at $\Box_2$; but $\natural S$-semantics would only reduce the subterm at $\Box_1$. The $\langle \natural_0.\Box_2 \rangle$ here is not a suspending poly-context because *in this case* it isn't a poly-context at all — it's just a syntax fragment embedded within non-suspending minimal nontrivial P.

The straightforward fix is to further adjust the definitions of $R, \mathcal{E}$-evaluation relation and R-evaluation normal form, for all state calculi (including $\natural_r S$-calculus, which will have to be done carefully to avoid invalidating the general theorems for regular SRSs from Chapter 13). The $R, \mathcal{E}$-evaluation relation is simply forbidden to descend into any but the first subterm of a set frame, which causes evaluation of the set term to stop when evaluation of its first (i.e., rightmost) subterm stops. An R-evaluation normal form is then a term such that all R-redexes occur *either* within a suspending poly-context *or* within a later subterm of a set frame, which causes a term to be $R, \mathcal{E}$-evaluable iff it isn't an R-evaluation normal form (if, that is, $\mathcal{E}$ is a regular evaluation order).[6] These adjustments don't invalidate the major theorems from Chapter 13, because from state calculus $\mathcal{S}$ with regular evaluation order $\mathcal{E}$, we could construct a new calculus $\mathcal{S}'$ on an extended term syntax, replacing each set frame of $\mathcal{S}$ with a compound poly-context that actually puts the subterms of each $\mathcal{S}$ set frame into specially introduced $\mathcal{S}'$ poly-contexts so that all but the first subterm are suspended. The schemata of $\mathcal{S}'$ then treat these compound set structures as units (bubbling up the whole structure, etc.); so $\mathcal{S}'$ is regular, and the theorems applied to $\mathcal{S}'$ under the old definitions give the theorems applied to $\mathcal{S}$ under the new definitions.

Because $\natural_r S$-calculus lacks state and set bubbling-up schemata, a state or set frame occurring in a singular evaluation context is not a decisively $\longrightarrow^{\natural_r s}$-reducible pattern, and consequently such nestings (and compositions involving them) are $\natural_r S, \mathcal{E}$-evaluation contexts; however, addition of state and set bubbling-up schemata eliminates these anomalies by making the pattern decisively reducible, leaving state and set frames in $\natural_r S, \mathcal{E}$-evaluation contexts only in certain top-level arrangements (for example, $[\text{state } [w_s] \ [\text{set } [\![\omega_s]\!] \ E[\Box]]]$, but not $[\text{state } [w_s] \ [\text{set } [\![\omega_s]\!] \ [\text{state } [w'_s] \ E[\Box]]]]$ since the latter contains a decisively reducible pattern). $\natural S$-semantics provides for all of these arrangements that don't involve get and could potentially affect operational equivalence; get is omitted from the term syntax of $\natural S$-semantics, and some top-level

[6] This is an ad hoc deployment of the idea that suspension should be a property of positions in poly-contexts, rather than a property of the poly-contexts themselves, as raised earlier in this chapter in Footnote 5.

arrangements are omitted since they guarantee free variables, thereby rendering them irrelevant to operational equivalence (for example, [set $[\![\omega_s]\!]$ $E[\Box]$]).

Get frames are suspending in $\wr_r S$-calculus. If concatenations and empty-frame eliminations are added (particularly, get-get concatenation, (12.41??); state-state concatenation, (12.44$\sigma\sigma$); and empty get elimination, (12.41?0)), get frames become non-suspending; but due to get resolution and get bubbling-up (both included in $\wr_r S$-calculus) and the new schemata, there will still be no difficulty with $R, \mathcal{E}$-evaluation contexts involving get frames. An empty get frame is decisively reducible. If a nonempty get frame occurs at the top level of a term, it presents a free get variable, and thereby renders the term irrelevant to operational equivalence. If a nonempty get frame occurs in a nontrivial $R, \mathcal{E}$-evaluation context, then either it participates in a decisively reducible pattern (via either get bubbling-up or get elimination), or it presents a free state variable (because state-state concatenation assures that only a free state variable would prevent the get from resolving) and thereby renders the term irrelevant to operational equivalence.

Receive frames don't occur on the left-hand side of any $\wr S$-calculus schema short of garbage collection (which, as noted before, is pathological because all possible poly-contexts occur on the left-hand sides of garbage-collection concrete schemata). Consequently, receive frames can occur freely in $R, \mathcal{E}$-evaluation contexts for all state $\wr$-calculi variants. However, this does not impact operational equivalence, because a receive frame occurring in an $R, \mathcal{E}$-evaluation context would guarantee a free variable (since either it would not occur within a matching get frame, or the matching get frame would not occur within a matching state frame; a get frame within a matching state frame would always present a redex pattern, so that the receive could not occur in an $R, \mathcal{E}$-evaluation context).

Let $\wr S'$-calculus modify full $\wr S$-calculus by omitting garbage-collection, (12.44σg); and $\wr_r S'$-calculus modify $\wr S'$-calculus by imposing the value-subterm restrictions of $\wr_r$-calculus, (10.7).

For state calculi without garbage collection, the definition of R-observable is broadened to allow a state frame, and within it a set frame, provided they don't capture any variable in the atomic subterm (similarly to the allowance of a catch frame in observables in non-garbage-collecting control calculi, §14.2.4).

For deterministic $R, \mathcal{E}$-evaluation for calculi intermediate between $\wr_r S$-calculus and $\wr S$-calculus, give priority first to empty-frame elimination and get consolidation, then set bubbling-up, then state bubbling-up, then all $\longrightarrow_{\wr rs}$ schemata (which includes get resolution and get bubbling-up), and lastly concatenation (set-set, get-get, and state-state). (Garbage collection will be introduced last, and by that time we won't be bothering with $R, \mathcal{E}$-evaluation anymore.)

Theorem 14.18 (Church–Rosser-ness)
$\longrightarrow^*_{\wr rs'}$ is Church–Rosser.
$\longrightarrow^*_{\wr s'}$ is Church–Rosser.
$\longrightarrow^*_{\wr s}$ is Church–Rosser. ■

Proof. $\wr_r S$-calculus is regular, so $\longrightarrow^{\wr rs}$ is Church–Rosser.

Let R_1 be the enumerated relation of the concatenation, consolidation, and empty-frame elimination schemata — (12.37), (12.41), and (12.44) except (12.44σg). R_1 is Church–Rosser (by cases). Any $T_1 \longrightarrow_{\wr rs} T_2$ will not be disabled by an $\longrightarrow_{R_1}$-reduction of T_1 (though it might make multiple copies of the R_1-redex, so that replicating the single $\longrightarrow_{R_1}$-reduction of T_1 may require multiple $\longrightarrow_{R_1}$-reductions of T_2; also by cases). The only case where $T_1 \longrightarrow_{R_1} T_2$ might be significantly interfered with by a $\longrightarrow_{\wr rs}$-reduction of T_1 is when the $\longrightarrow_{R_1}$ step is a get consolidation, and the $\longrightarrow_{\wr rs}$ is a get resolution that eliminates the first of the two binding requests that the R_1 consolidates; but if the get resolution is failure, then additional get resolutions would eventually eliminate the second request as well, while if the get resolution is success, further requests can be resolved or bubbled up through the set until the second request is disposed of. So $\longrightarrow^{\wr rs \cup R_1}$ is Church–Rosser.

Let R_2 be the enumerated relation of the state and set bubbling-up schemata — (12.45) and (12.38). R_2 itself is Church–Rosser (by cases). If $T_1 \longrightarrow_{\wr rs \cup R_1} T_2$, then any state/set bubbling-up that could be done to T_1 either wouldn't disable that step, or could be followed by additional state/set/get bubbling-up to re-enable the step (by cases for the schemata of $\longrightarrow^{\wr rs \cup R_1}$); so by induction, $\longrightarrow^*_{R_2}$ and $\longrightarrow^*_{\wr rs \cup R_1}$ cooperate (Definition 13.63), and again by induction, $\longrightarrow^{\wr rs \cup R_1 \cup R_2}$ is Church–Rosser.

$\longrightarrow^{\wr rs \cup R_1 \cup R_2}$ is $\longrightarrow^{\wr rs'}$, differing from $\longrightarrow^{\wr s'}$ only by relaxation of value-subterm constraints of $\wr_r$-calculus Schemata 10.7. This relaxation can be handled substantially as in Theorem 14.13, noting that bubbling up is simpler here than in the control calculi, because it always occurs without disruption of the potential redex pattern that it passed through (whereas in the control calculi, a throw bubbling-up destroyed everything in its path). Addition of garbage collection for full $\wr S$-calculus is also handled as in Theorem 14.13. ■

Definition 14.19 A *side-effect-ful value* (over $\wr S$-calculus) is a term of the form $C[V]$, where C is a (possibly trivial) composition of state, set, and get frames. ■

Lemma 14.20

If T is not $\longrightarrow^*_{\wr rs'}$-reducible to a side-effect-ful value, then [eval T e] isn't $\longrightarrow^*_{\wr s'}$-reducible to a side-effect-ful value. ■

Proof. Where Lemmata 14.10 and 14.15 included the corresponding condition on combine frames, here it is sufficiently complex that we have deferred it to the proof: In order for [combine T_1 T_2 e] to be reducible to a side-effect-ful value, it is necessary that T_2 be reducible to a side-effect-ful value, and that T_1 be reducible to a side-effect-ful value *given* the side-effects of T_2; that is, T_2 reduces to $C_2[V_2]$, C_2 bubbles up through the combine frame to C'_2 ([combine T_2 $C_2[V_2]$ e] $\longrightarrow^*_{rs'}$ C'_2[[combine T_2 V_2 e]]), and $C'_2[T_1]$ reduces to a side-effect-ful value. This dovetails neatly into the result for

[eval T e] since, in Schema 9.32p, reduction of $T = (T_1 \ . \ T_2)$ to a side-effect-ful value includes possible propagation of side-effects from T_2 to T_1.

The proof proceeds similarly to the earlier lemmata. ■

Lemma 14.21 Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_s$.

If $T_1 \longrightarrow_{\wr rs'} T_2$, $T_1 \not\longmapsto^{\mathcal{E}}_{\wr rs'} T_2$, and T_1 is not a $\wr_r S'$-evaluation normal form, then T_2 isn't either. ■

Proof. Let T, P, and P_2 be as in the proof of Lemma 13.119. If P and P_2 are both $\wr_r S$-redex patterns, they are mutually selective, and the proof completes as before. Otherwise, at least one of them is not a $\wr_r S$-redex pattern (hence, is either a state/set bubbling-up, state/set/get concatenation or consolidation, or empty-frame elimination).

In order for a $\wr_r S$- and a non-$\wr_r S$-redex pattern to overlap, the $\wr_r S$-redex pattern has to be for get bubbling-up or get resolution. Empty-frame resolution would have to be P (the evaluation-step redex), since that would have higher priority; and an empty frame can't participate in get resolution, so the overlapping $\wr_r S$-redex would have to be a bubbling up of the empty frame — which would leave the empty frame still an evaluation-step redex. Get consolidation would also have to be P; bubbling up the consolidatable get frame leaves it still an evaluation redex, while resolving one of the requests in the previous consolidatable get might leave it no longer consolidatable but would still leave either some sort of evaluation redex (at least, some get bubbling-up or get resolution would be possible). A state/set bubbling-up would also be P, and then P_1 would have to be get resolution — in which case, if the state/set bubbling-up didn't produce some other evaluation redex further out (such as a further bubbling up of the same frame), there would certainly be a get bubbling-up of the inner frame of the disrupted get resolution. A get concatenation would have lower priority that bubbling up or resolution, so the concatenation would be P_2, and after the concatenation there would still be a bubbling up or resolution possible.

Finally, suppose P and P_2 are both non-$\wr_r S$-redex patterns. An empty state/set frame P could bubble up via P_2 but would still be an evaluation redex pattern; and when a empty-frame elimination redex P overlaps with a concatenation redex P_2, exercising P_2 is actually indistinguishable from exercising P (up to α-renaming), so that we can simply assume the evaluation step was the one taken. Get consolidation P could overlap with get concatenation P_2, but after concatenation there will still be a consolidation possible. Two concatenations could overlap —three consecutive frames, where P is the outer two and P_2 is the inner two— but after the inner concatenation, there will still be a concatenable pair of frames. ■

Lemma 14.22 Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_s$.

$T_1 \longrightarrow^*_{\wr rs} T_2$ iff there exists a $\wr_r S', \mathcal{E}$-standard $\longrightarrow_{\wr rs}$-reduction sequence from T_1 to T_2. ■

Proof. By regularity (and Theorem 13.118), $T_1 \longrightarrow^*_{\wr rs} T_2$ iff there exists a $\wr_r S, \mathcal{E}$-standard reduction sequence from T_1 to T_2.

Suppose the definition of $\wr_r S, \mathcal{E}$-standard reduction sequence were changed by not treating get frames as suspending contexts. A standard reduction might be perturbed by this, because a get subterm, which would have been skipped over during initial $R, \mathcal{E}$-evaluation, is now included in that phase of standard reduction. No reduction of this subterm can have any effect on reduction of the rest of the term (which is why get frames were suspending in the first place); and reductions to the rest of the term can only affect the subterm by making copies of it (during β-reduction) and by substituting into it, all of which must occur, by construction of $R, \mathcal{E}$-evaluation, *before* the subterm would be reduced even if the get frame weren't suspending. In a standard reduction sequence from before de-suspending the get frame, the (possibly trivial) subsequence reducing that subterm must have consisted of a (possibly trivial) $R, \mathcal{E}$-evaluation subsequence followed by standard reductions of subterms; and when de-suspending the get frame, this $R, \mathcal{E}$-evaluation subsequence can be moved to an earlier point in the overall sequence — which is trivially easy since the subsequence doesn't interact with anything that was done between where it moves from and where it moves to, and doesn't increase the length of the overall sequence. Also, this evaluation subsequence of the subterm standard reduction is moved to a point somewhere in the evaluation prefix of the standard reduction of the overall term — and if the evaluation subsequence of the subterm failed to reduce the subterm to an evaluation normal form, then moving this subsequence into the overall-evaluation prefix means that any overall-evaluation steps that used to *follow* that point are now no longer evaluation steps (because they reside in contexts that are no longer evaluation contexts; for example, in a term

$$[\text{combine } T_1 \ [\text{get } [\![\omega_g]\!] \ T_2] \ e] \,, \tag{14.23}$$

evaluation steps can reduce T_1 if the get is suspending, but when the get isn't suspending, no evaluation step can reduce T_1 unless T_2 is first reduced to an evaluation normal form). However, these now-disqualified evaluation steps are all contained within subterms (such as T_1 in the example) that cannot have any effect on reduction of their context (because as long as T_2 isn't a value, side-effects can't bubble up from T_1); therefore, the evaluation prefix of each of these subterms can be moved *rightward* to the appropriate non-evaluation position in the standard reduction sequence, just as smoothly as the evaluation prefix of the get subterm was moved leftward.

So $T_1 \longrightarrow^*_{\wr rs} T_2$ iff there exists an $R, \mathcal{E}$-standard $\longrightarrow_{\wr rs}$-reduction sequence from T_1 to T_2 under the modified definition with non-suspending get frames.

Given an $R, \mathcal{E}$-standard reduction sequence under this modified definition, further perturbation can produce a $\wr_r S', \mathcal{E}$-standard reduction sequence: the further change of definition is that additional redex patterns will prevent any subsequent reductions from qualifying as evaluation steps for the term as a whole. However, since the sequence still uses only $\longrightarrow_{\wr rs}$-reduction steps, these subsequent reductions can only

be subterm reductions, which can be shifted rightward into the recursion phase of the standard sequence. ■

Lemma 14.24 Let $\mathcal{E}$ be the strictly right-to-left evaluation order of Ч_s.

If there exists a $\text{ɬ}_r S'$-evaluation normal form N such that $T_1 \longrightarrow_{\text{ɬ}rs} T_2 \longmapsto^{\mathcal{E}*}_{\text{ɬ}rs'} N$, then there exists a $\text{ɬ}_r S'$-evaluation normal form N' such that $T_1 \longmapsto^{\mathcal{E}*}_{\text{ɬ}rs'} N'$. ■

Proof. Suppose N is a $\text{ɬ}_r S'$-evaluation normal form, $T_1 \longrightarrow^*_{\text{ɬ}rs} T_2$, and $\vec{T_2}$ is a $\text{ɬ}_r S', \mathcal{E}$-evaluation sequence from T_2 to N. Let n be the number of non-$\longrightarrow_{\text{ɬ}rs}$ steps in $\vec{T_2}$. We will show that there exists a $\text{ɬ}_r S', \mathcal{E}$-evaluation sequence from T_1 to some $\text{ɬ}_r S'$-evaluation normal form with at most n non-$\longrightarrow_{\text{ɬ}rs}$ steps. Proceed by induction on n.

Suppose $n = 0$. Then $T_1 \longrightarrow^*_{\text{ɬ}rs} N$; so let $\vec{T}$ be a $\text{ɬ}_r S'$-standard reduction sequence from T_1 to N, by Lemma 14.22. Assume without loss of generality that $\vec{T}$ has all of its evaluation steps at its start, by Lemma 13.89. Let N' be the result of the last evaluation step in $\vec{T}$; then N' must be a $\text{ɬ}_r S'$-evaluation normal form, by Lemma 14.21.

Suppose $n \geq 1$, and the proposition holds for all smaller n. If $T_1 \longrightarrow_{\text{ɬ}rs} T_2$ is a $\text{ɬ}_r S', \mathcal{E}$-evaluation step, the result follows immediately by Lemma 13.89 and Criterion 13.88(b); so assume $T_1 \not\longmapsto^{\mathcal{E}}_{\text{ɬ}rs'} T_2$. It suffices to consider the case where the first step of $\vec{T_2}$ is a non-$\longrightarrow_{\text{ɬ}rs}$ step, by Lemma 14.22; so assume that. The first step of $\vec{T_2}$ is, in order of priority, either an empty-frame elimination or get consolidation, a set bubbling-up, a state bubbling-up, or a state/set/get concatenation.

There are two cases, depending on whether or not the evaluation-step redex pattern exercised at T_2 already exists at T_1.

Case 1: the evaluation-step redex pattern exercised at T_2 does not exist at T_1. This requires that the T_2 evaluation redex pattern be created, in whole or in part, by the non-evaluation step $T_1 \longrightarrow_{\text{ɬ}rs} T_2$. A β-rule step could only create the whole of an evaluation redex pattern, or the inner frame of a bubbling-up or concatenation evaluation redex pattern, if the β-rule step were itself an evaluation step. A get bubbling-up could only create a concatenation evaluation redex pattern if the get bubbling-up were itself an evaluation step. So $T_1 \longrightarrow_{\text{ɬ}rs} T_2$ must be either a get bubbling-up or a get resolution. A non-evaluation get bubbling-up could bubble up a get frame that is itself an empty-frame elimination or get consolidation redex pattern; the two actions could then be done in the opposite order, $T_1 \longmapsto^{\mathcal{E}}_{\text{ɬ}rs'} T_1' \longrightarrow_{\text{ɬ}rs} \vec{T_2}(2)$, and by the inductive hypothesis on n, we'd be done. A get resolution could *create* an empty get frame, by eliminating the last binding request in the frame; but if the get resolution isn't an evaluation step, then the empty-frame elimination wouldn't be an evaluation step either (because any redex pattern that would prevent the get resolution from being an evaluation step would be decisively reducible and strictly above the empty frame, so that the high priority of empty-frame elimination would

never come into play). Similarly, a get bubbling-up could *create* a get concatenation redex pattern, but if the get bubbling-up isn't an evaluation step, then the get concatenation (which has lowest priority) wouldn't be either. There is no way that a state/set bubbling-up evaluation step could have been enabled by a non-evaluation get bubbling-up or get resolution.

Case 2: the evaluation-step redex pattern exercised at T_2 already exists at T_1. This doesn't require that the two redex patterns be non-overlapping; but if they do overlap, exercising the non-evaluation redex doesn't perturb the evaluation redex pattern. If they *don't* overlap, then exercising the evaluation redex first will neither eliminate nor multiply the other redex (considering the possibilities for a non-$\longrightarrow_{\natural rs}$ redex); so $T_1 \longmapsto^{\mathcal{E}}_{\natural rs'} T_1' \longrightarrow_{\natural rs} \vec{T_2}(2)$, and by the inductive hypothesis on n, we're done. Suppose the redex patterns do overlap. Then the evaluation redex pattern of T_2 is an *evaluation* redex in T_1 (by cases). If the non-evaluation step were a get bubbling-up, it would have perturbed the evaluation redex pattern; this possibility was covered in Case 1. Suppose the non-evaluation step is a get resolution. The evaluation step isn't a concatenation (of the two frames immediately above the get, in this case), because then the get resolution would have been an evaluation step, having higher priority than concatenation. The evaluation step isn't an empty-frame elimination, because a get can't be resolved by an *empty* state/set frame. The only possibility is that the evaluation step is a state/set bubbling-up; suppose this. $\vec{T_2}$ must eventually dispose of the get frame involved in resolution $T_1 \longrightarrow_{\natural rs} T_2$, either by eliminating it (if it is empty in T_2), or by bubbling it up until it meets the frame above it. Let $T_1 \longmapsto^{\mathcal{E}}_{\natural rs'} T_1'$. We describe how to construct, by modifying $\vec{T_2}$, a $\longrightarrow_{\natural rs'}$-reduction sequence $\vec{T_1'}$ from T_1' to N with at most $n-1$ non-$\longrightarrow_{\natural rs}$ steps, in which all the non-$\longrightarrow_{\natural rs}$ steps are evaluation steps; the result then follows from the inductive hypothesis on n. As the first two steps from T_1', bubble up the get frame and then resolve it once; since the surrounding state/set frame is then in the position achieved by the first step of $\vec{T_2}$, and the get resolution has performed its substitution as in $T_1 \longrightarrow_{\natural rs} T_2$, these two steps from T_1' result in a term that differs from $\vec{T_2}(2)$ only by the fact that the get frame has been bubbled up. At some point later in $\vec{T_2}$, this get frame must be bubbled up to this position, or eliminated (if empty). Adjust each term in the sequence up to that point by shifting the get frame to its bubbled-up position. The only sort of $\longrightarrow_{\natural rs'}$ steps that might be spoiled by this shift are get concatenations with a frame below the shifted frame; but these cannot actually happen in $\vec{T_2}$, because it consists only of evaluation steps, and concatenation would not be the evaluation step since it has lower priority than either empty-frame elimination or get bubbling-up. ■

Lemma 14.25 Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathbb{H}_s$, and $R = (\longrightarrow_{\natural rs'} - \longrightarrow_{\natural rs})$.

If there exists a $\natural_r S'$-evaluation normal form N such that $T_1 \longrightarrow_R T_2 \longmapsto^{\mathcal{E}}_{\natural rs'} N$,

then there exists a $\wr_r S'$-evaluation normal form N' such that $T_1 \longmapsto^{\mathcal{E}}_{\wr r s'} N'$. ■

Proof. Suppose N is a $\wr_r S'$-reduction normal form, $T_1 \longrightarrow_R T_2$, and $\vec{T}$ is a $\wr_r S', \mathcal{E}$-evaluation sequence from T_2 to N. Let S be the enumerated relation of bubbling-up Schema 12.42↑? (get bubbling-up through a singular evaluation context (Syntax 11.8)). Let Q be the set difference of $\longrightarrow_{\wr r}$ minus S. Let q be the number of $\longrightarrow_Q$ steps in $\vec{T}$, r the number of $\longrightarrow_R$ steps in $\vec{T}$, and s the number of $\longrightarrow_S$ steps in $\vec{T}$. We will show that there exists a $\wr_r S', \mathcal{E}$-evaluation sequence from T_1 to some $\wr_r S'$-evaluation normal form with at most q $\longrightarrow_Q$ steps. Suppose this proposition holds for all smaller q (which is vacuously true when $q = 0$); holds for this q for all smaller r; and holds for this q and r for all smaller s.

If $T_1 \equiv_\alpha N$, the result is immediate; so suppose T_1 is not a $\wr_r S'$-evaluation normal form. If $T_1 \longmapsto_R T_2$ is a $\wr_r S', \mathcal{E}$-evaluation step, again the result is immediate (with sequence $\langle T_1, T_2\rangle \cdot \vec{T}$); so suppose it isn't. Then T_2 isn't a $\wr_r S'$-evaluation normal form either (by Lemma 14.21); so $ar(\vec{T}) \geq 2$. Let P_1 be the redex pattern exercised in $T_1 \longrightarrow_R T_2$, and P_2 the redex pattern exercised in the first step of $\vec{T}$.

Case 1: P_1 and P_2 are independent of each other (i.e., both already occur non-overlapping in T_1). Then exercising P_2 in T_1 is still an evaluation step; call it $T_1 \longmapsto^{\mathcal{E}}_{\wr r s'} T_1'$. This step makes some number of copies of the subterm containing P_1. If zero copies are made, $T_1' \equiv_\alpha \vec{T}(2)$, and we're done; suppose at least one copy is made. If exactly one copy is made, $T_1' \longrightarrow_R \vec{T}(2)$, and by the inductive hypothesis (on q, r, or s) we're done; suppose at least two copies are made. Then the first step of $\vec{T}$ is a $\longrightarrow_Q$ step (since all the schemata that can multiply subterms are in Q); and from T_1', one can exercise all the copies of P_2, $T_1' \longrightarrow^+_R \vec{T}(2)$. The result then follows from the inductive hypothesis on q.

Case 2: P_1 and P_2 are not independent, and $T_1 \longrightarrow_R T_2$ is an empty-frame elimination. The first step of $\vec{T}$ can't be an empty-frame elimination or get consolidation, because the P_k would be independent; and it can't be any other kind of $\longrightarrow_{\wr r s'}$ step, because for any of those redex patterns P_2 to be non-independent of the empty-frame redex pattern P_1, $T_1 \longrightarrow_R T_2$ would have to be an evaluation step (since it has higher priority). So this can't happen.

Case 3: P_1 and P_2 are not independent, and $T_1 \longrightarrow_R T_2$ is a get consolidation. The first step of $\vec{T}$ can't be an empty-frame elimination, state/set bubbling up, state/set concatenation, or $\longrightarrow_{\wr r}$ step, because the P_k would be independent. If the first step of $\vec{T}$ were a get consolidation, then to be non-independent it would have to be on the same get frame as $T_1 \longrightarrow_R T_2$; and by the determinism built into the get consolidation Schema 12.41?2, $T_1 \longrightarrow_R T_2$ would be an evaluation step. The only remaining possibilities are that the first step of $\vec{T}$ is a get bubbling-up, get resolution,

or get concatenation; but a non-independent get consolidation $T_1 \longrightarrow_R T_2$ would have to be on the get frame of the following evaluation step, in which case $T_1 \longrightarrow_R T_2$ would be an evaluation step.

Case 4: P_1 and P_2 are not independent, and $T_1 \longrightarrow_R T_2$ is a state/set bubbling-up. In order for the P_k to be non-independent, the first step of $\vec{T}$ cannot be a get empty-frame elimination, get consolidation, get bubbling-up, $\longrightarrow_{\wr r}$ step, or get concatenation. If the first step of $\vec{T}$ were a state/set bubbling-up or state/set concatenation, then $T_1 \longrightarrow_R T_2$ would be an evaluation step. The remaining possibility is that the first step of $\vec{T}$ is an empty-frame elimination; but then it must be eliminating the same frame that was bubbled up in $T_1 \longrightarrow_R T_2$, so that simply eliminating the frame at T_1 would be an evaluation step $T_1 \longrightarrow_R \vec{T}(2)$, and we're done.

Case 5: P_1 and P_2 are not independent, and $T_1 \longrightarrow_R T_2$ is a state/set/get concatenation. Neither of the frames in P_1 is an empty frame, because if either of them were, then $T_1 \longrightarrow_R T_2$ would be an empty-frame elimination step, and therefore an evaluation step. Consider subcases depending on the form of the first step of $\vec{T}$.

Case 5a: the first step of $\vec{T}$ is a concatenation. Then, for the P_k to be non-independent, one of the two frames in P_2 must be the concatenation of the two frames in P_1; and this cannot be the outer frame of P_2, because if it were, $T_1 \longrightarrow_R T_2$ would have to be an evaluation step. So there are three consecutive frames in T_1, of which P_1 is made up of the inner two, and $T_1 \longrightarrow_R^2 \vec{T}(2)$ concatenates these three frames into one. Let $T_1 \longmapsto^{\mathcal{E}}_{\wr rs'} T_1'$; this evaluation step must be concatenating the outer two of the three frames. Then concatenating the inner two frames gives $T_1' \longrightarrow_R \vec{T}(2)$; and the result follows from the inductive hypothesis on q.

Case 5b: the first step of $\vec{T}$ is a $\longrightarrow_{\wr r}$ step. Then the P_k would have to be independent.

Case 5c: the first step of $\vec{T}$ is a get resolution. Then, for the P_k to be non-independent, one of the two frames in P_2 must be the concatenation of the frames in P_1; and if that were the outer frame of P_2, then $T_1 \longrightarrow_R T_2$ would be an evaluation step. So the inner, get frame of P_2 is the concatenation of the frames in P_1. Because the get resolution Schemata 12.40 always resolve the leftmost binding request, and get concatenation Schema 12.41?? preserves the ordering of binding requests, and both frames of P_1 are nonempty (as noted earlier), it must be that the binding request resolved from T_2 is in the *outer* frame of P_1, and therefore that binding request can be resolved from T_1. Furthermore, that resolution must be an evaluation step from T_1, after which the concatenation could still be done. $T_1 \longmapsto^{\mathcal{E}}_{\wr rs'} T_1' \longrightarrow_R \vec{T}(2)$, and the result follows from the inductive hypothesis on q.

Case 5d: the first step of $\vec{T}$ is a get bubbling up through a set. Then, for the P_k to be non-independent, the get frame in P_2 must be the concatenation of the frames in P_1. Because of the form of the get-through-set bubbling-up Schema 12.42↑!?, only

the leftmost binding request of P_2 bubbles up in the evaluation step from T_2; and as already noted, the outer frame of P_1 is not empty, so that an evaluation step from T_1 would bubble up this same binding request through the surrounding set. Let $T_1 \longmapsto^{\mathcal{E}}_{\ell rs'} T_1'$. In T_1', the two get frames just inside the set are almost P_1, except that the outer get frame is missing the one binding request that was bubbled up; so the only difference between T_1' and $\vec{T}(2)$ is the concatenation of those two get frames. Therefore, $T_1' \longrightarrow_R \vec{T}(2)$, and the result follows from the inductive hypothesis on q.

Case 5e: the first step of $\vec{T}$ is an $\longrightarrow_S$ step (a get bubbling up through a singular evaluation context (Syntax 11.8)). Then, for the P_k to be non-independent, the get frame in P_2 must be the concatenation of the frames in P_1. An evaluation step from T_1 must bubble up the outer of the two get frames of P_1; and subsequent evaluation steps will continue to bubble up that frame as long as there are singular evaluation contexts directly above it. Let $T_1 \longrightarrow_S^+ T_1'$ bubble up this frame through as many singular evaluation contexts as are available. $T_1 \longmapsto^{\mathcal{E}+}_{\ell rs'} T_1'$. In T_1', the bubbled-up get frame has either reached the top level of the term, or it is immediately inside a state, set, or get frame.

Case 5e(1): in T_1', the outer frame of P_1 has reached the top level. Then further evaluation steps from T_1' will bubble up the *inner* frame of P_1 until it is again adjacent to the outer frame. Call this term T_1''; $T_1' \longrightarrow_S^+ T_1''$, and $T_1' \longmapsto^{\mathcal{E}+}_{\ell rs'} T_1''$. The evaluation step from T_1'' must concatenate the two frames, $T_1'' \longmapsto^{\mathcal{E}}_{\ell rs'} T_2'$. Sequence $\vec{T}$ must similarly bubble up the concatenated get frame of P_2 until it reaches the top level, producing a term $\vec{T}(k) \equiv_\alpha T_2'$, and we have $T_1 \longmapsto^{\mathcal{E}+} N$.

Case 5e(2): in T_1', the outer frame of P_1 is immediately inside a state frame. Either those two frames admit a get resolution (failure), or they don't. If they don't, the scenario proceeds as in 5e(1). Suppose they do admit a get resolution, $T_1' \longmapsto^{\mathcal{E}}_{\ell rs'} T_1''$. Sequence $\vec{T}$ must similarly bubble up the concatenated get frame in P_2 until it reaches that state frame, and then resolve the same binding request; let $\vec{T}(k)$ be the term achieved by these steps. Then the same term (up to $\equiv_\alpha$) can be reached from T_1'' by bubbling up the inner frame of P_1 and concatenating it with (what remains of) the outer frame of P_1, $T_1'' \longrightarrow_S^+ \cdot \longrightarrow_R T_2' \equiv_\alpha \vec{T}(k)$. The result follows from the inductive hypothesis on q (because the step from $\vec{T}(k-1)$ to $\vec{T}(k)$ is a get resolution, which is a $\longrightarrow_Q$ step).

Case 5e(3): in T_1', the outer frame of P_1 is immediately inside a set frame. Since the outer frame of P_1 is not empty, the evaluation step from T_1' will be either a get resolution or a get-through-set bubbling-up, either of which is a $\longrightarrow_Q$. The scenario proceeds as in 5e(2).

Case 5e(4): in T_1', the outer frame of P_1 is immediately inside a get frame. This will be similar to 5e(1), with parallel evaluation sequences being traced through until they reconverge, except that here the parallel sequences will be more complex.

The evaluation step from T_1' is concatenation of the outer frame of P_1 with the get

frame above it (call that frame P_0), and if their concatenation creates any new opportunities for get consolidation, subsequent evaluation steps are those consolidations until the concatenated frame has been consolidated as far as possible. The newly concatenated and consolidated get frame is now either at the top level or just below a state frame (because otherwise the get frame that was already in this position would have been able to bubble up, and would therefore have prevented all the evaluation steps we've already taken to this point). From this point, some number of the newly arrived binding requests (that came originally from the outer frame of P_1) might be resolved by evaluation steps (only if there is a state frame just above the get frame), and then further evaluation steps will begin to bubble up the inner frame of P_1. Once it reaches the get frame above it, another evaluation step will concatenate the get frames; if the concatenation crease any new opportunities for consolidations, those consolidations are performed as further evaluation steps; and if any of the binding requests after consolidation can be resolved against a surrounding state frame, those resolutions are evaluation steps and should be done as well. Call the result of this entire sequence of evaluations T_1''. $T_1' \longmapsto^{\mathcal{E}+}_{\natural rs'} T_2'$.

Sequence $\vec{T}$ necessarily bubbles up the concatenated get frame of P_2 until it reaches P_0; concatenates it with P_0; consolidates the concatenated frame as much as possible; and then resolves and binding requests that can be resolved against a state frame (if there is one) surrounding the concatenated consolidated frame. This produced a term $\vec{T}(k) \equiv_\alpha T_2'$, and we're done.

Case 5f: the first step of $\vec{T}$ is a state/set bubbling-up. This is similar to 5e(1). For the P_k to be non-independent, the inner frame of P_2 must be the concatenation of the frames in P_1. An evaluation step from T_1 must bubble up the outer of the two frames of P_1; and subsequent evaluation steps will continue to bubble up that frame as long as they can. Then, if its bubbling up was stopped by another frame of the same type, the next evaluation step will concatenate them. Subsequent evaluation steps will bubble up the inner frame of P_1 until it meets with the outer frame (or the concatenation of the outer frame with the like frame that it encountered), and another evaluation step will concatenate them. The resulting term is $\equiv_\alpha$ to a term that must occur in $\vec{T}$, and we're done.

Case 5g: the first step of $\vec{T}$ is a get consolidation. Then, for the P_k to be non-independent, the single frame in P_2 must be the concatenation of the frames in P_1. If either of the two frames in P_1 were empty, then concatenation $T_1 \longrightarrow_R T_2$ would also be an empty-frame elimination step, and therefore an evaluation step; so neither frame in P_1 is empty. If any get consolidation is possible on either one of the two separate frames in P_1, evaluation steps from T_1 would do that first. Allowing for any get consolidation evaluation steps from T_1, and any further consolidation of P_2 in $\vec{T}$, this case proceeds substantially as 5e (including subcases for interaction of the get frame(s) with whatever they encounter after possibly bubbling up.

Case 5h: the first step of $\vec{T}$ is an empty-frame elimination. Then, for the P_k to be

non-independent, P_2 must be the empty concatenation of the frames in P_1; so both frames in P_1 are empty, and two evaluation steps from T_1 will eliminate them both, $T_1 \longmapsto^{\mathcal{E}2}_{\wr rs'} \vec{T}(2)$. ■

Theorem 14.26 (Standard normalization)

Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_s$.

If there exists a $\wr_r S'$-evaluation normal form N such that $T \longrightarrow^*_{\wr rs'} N$, then there exists a $\wr_r S'$-evaluation normal form N' such that $T \longmapsto^{\mathcal{E}*}_{\wr rs'} N'$. ■

Proof. Follows from Lemmata 14.24 and 14.25, by induction on the length of a $\wr_r S'$-evaluation sequence from T to N. ■

Theorem 14.27 (Operational soundness)

If $T_1, T_2 \in \mathfrak{H}_{\mathrm{ss}}$ and $T_1 =_{\wr s} T_2$, then $T_1 \simeq_{\wr s} T_2$. ■

Proof. Let $\mathcal{E}$ be the strictly right-to-left evaluation order of $\mathfrak{H}_s$.

It was established in the discussion at the top of the subsection (§14.2.5, preceding Theorem 14.18) that, although some $\wr_r S', \mathcal{E}$-evaluation contexts are not allowed for in the $\wr_r S$-semantics schemata, those not allowed for cannot have any impact on operational equivalence. Suppose $T \in \mathfrak{H}_{ss}$, $\mathrm{FV}(T) = \{\}$, N is a $\wr_r S'$-evaluation normal form, and $T \longmapsto^{\mathcal{E}}_{\wr rs'} N$. We claim that $N \in \mathfrak{H}_{ss}$ and there exists a term N', differing from N only by the possible presence of some empty frames that are eliminable by $\longmapsto^{\mathcal{E}}_{\wr rs'}$ but not $\longmapsto_{\wr s}$, such that $T \longmapsto^*_{\wr s} N'$.

The syntax extensions of $\mathfrak{H}_s$ beyond $\mathfrak{H}_{\mathrm{ss}}$ are get/receive Syntax 12.28 and environment Syntax 12.29. The extended forms of environment terms cannot be introduced by reduction if they were not already present in the term. The only way get/receive syntax could be introduced is through symbol evaluation Schema 12.39s*s*; and since this would have to happen in an evaluation context, and here the term has no free variables, the get will necessarily be resolved and the receive eliminated by evaluation steps, so that $N \in \mathfrak{H}_{\mathrm{ss}}$.

For $T \longmapsto^*_{\wr s} N'$, it suffices that any $\wr_r S', \mathcal{E}$-evaluation step taken from T will be the start of a sequence of $\wr_r S', \mathcal{E}$-evaluation steps that simulates a sequence of $\wr S$-semantics steps, up to elimination of empty frames. The fact that the $\wr S$-semantics steps do not eliminate the empty frames causes no difficulty, as they are bubbled up out of the way of any calculus step they could otherwise interfere with; note also that if the term reduces to an observable (rather than merely to an evaluation normal form), $\wr S$-semantics will eventually remove any empty frames via (12.23g0) and (12.23g0!). Any $\wr_r S', \mathcal{E}$-evaluation step from T is straightforwardly the start of simulating a single $\longmapsto_{\wr s}$ step, up to elimination of empty frames.

$=_{\mathring{l}rs'} \subseteq \simeq^{\mathcal{E}}_{\mathring{l}rs'}$ is obtained as in the proof of $\mathring{l}C$ operational soundness (Theorem 14.17), from Church–Rosser-ness (Theorem 14.18), standard normalization (Theorem 14.26), and Lemma 13.82. The extension of this result by relaxing the value-subterm constraints of Schemata 10.7 is similar to the proof of Theorem 14.11.

The relationship between $\longmapsto^*_{\mathring{l}s'}$ and $\longmapsto^{\mathcal{E}*}_{\mathring{l}s'}$ is at this point still only up to empty-frame eliminations, which makes no difference to the normalizability condition of operational equivalence (Condition 13.90(a)), but does create some discrepancies in which observables are reduced to. Adding garbage collection, (12.44σg), doesn't alter normalizability (although it would do so if we hadn't adjusted the definitions to suppress evaluation of later subterms of a set), and does allow $\longmapsto^{\mathcal{E}*}_{\mathring{l}s}$ to arrive at the *same* observables as $\longmapsto^*_{\mathring{l}s}$ (i.e., the definition of *observable* no longer has to be extended to allow state and set frames around the atomic term). ■

Chapter 15

The theory of fexprs is (and isn't) trivial

15.0 Introduction

Likely the most well-known modern paper to prominently address fexprs is Mitchell Wand's 1998 "The Theory of Fexprs is Trivial" ([Wa98]). Not only did that paper demonstrate the trivialization of theory that, in the vocabulary of this dissertation, can result from the introduction of fexprs into an implicit-evaluation calculus, but it also presented a compellingly plausible intuitive argument that this trivialization of theory is an inherent consequence of the introduction of fexprs into λ-calculus.

Despite the rather alarming appearance of contradiction between the central thesis of [Wa98] and the major well-behavedness results on $\mathfrak{f}$-calculi developed in this dissertation, there is no actual contradiction. Instead, there are major discrepancies of terminology and technical approach that create an illusion of contradiction where none exists. The "fexprs" investigated by [Wa98] are a dramatically different facility than the "fexprs" investigated by this dissertation; and, partly entangled with that difference of terminology, [Wa98] assumes an isomorphism between source expressions and computational states, while this dissertation does not.

This chapter explores the depth and consequences of these discrepancies.

15.1 Encapsulation and computation

The difference between fexprs in [Wa98] and fexprs in this dissertation lies in what kinds of operands they are able to deconstruct. Parameterizing the central result of [Wa98] for this difference, we have the semi-formal proposition

Proposition 15.1 Let $\mathcal{S}$ be the set of all objects that can be deconstructed by passing them as operands to fexprs in object language $\mathcal{L}$. Then the theory of $\mathcal{L}$-operational equivalence of objects in $\mathcal{S}$ is trivial. ■

What we mean by *deconstruction* is that when the object (i.e., object-language expression) is passed as an operand to the fexpr, the fexpr can then completely analyze all the salient features of the object. If there is any salient difference between two objects $S_1, S_2 \in \mathcal{S}$, then a fexpr in $\mathcal{L}$ can distinguish between them, and therefore they are not $\mathcal{L}$-operationally equivalent — which is all the proposition says. To the credit of [Wa98], this statement is obvious once understood.[1]

Based on the role of such deconstructible objects in Kernel, and their traditional role in Lisps, we call them *S-expressions* (*S* either for *Source*, *Syntax*, or —traditionally— *Symbolic*).

In [Wa98], the fexprs considered are capable of deconstructing all possible computational entities in the formal system (a modified λ-calculus); therefore, all operational equivalences between any computational entities whatsoever in the formal system are trivial. One might reasonably call this property of the formal system —that all terms are S-expressions— *full reflection*. Were this terminology introduced into the paper —a minor and non-structural change; for example, the word *fexpr* doesn't even occur in the paper's abstract— the paper might defensibly have been titled "The Theory of Reflection is Trivial", and its overall import would have been so changed that there might have been no serious seeming of contradiction between that document and this one.

In this dissertation, the fexprs considered are only able to deconstruct certain kinds of entities (only certain kinds of entities are S-expressions). Fexprs here are unable to deconstruct either encapsulated objects —i.e., environments or compound operatives— or computational states —i.e., *active terms*, as in (9.27), (11.1), (12.8), (12.28).

From the point of view of the object language $\mathcal{L}$, active terms are an artifact of the auxiliary language we are using to model the semantics of $\mathcal{L}$. Programs in $\mathcal{L}$ can only access objects, which is why reflection requires a means of *reification* (cf. §1.2.4): in order for programs to access computational state, the state has to be manifested in the form of an object. If the object language itself has no encapsulated objects —that is, if all objects are S-expressions— then all operational equivalences between objects are trivial. Since traditional semantics has focused exclusively on operational equivalences between objects (to put it another way, semantic equivalences between syntax expressions), this would mean that all operational equivalences whatsoever are trivial. This situation may be suggestive of the observation, in the conclusion of [Wa98], that "To get non-trivial theories, one needs to find weaker reflection principles" — but whether that observation applies here depends on whether one views encapsulated objects as a withholding of the power of fexprs from those objects (making fexprs

[1] Heuristically, the profundity of a truth is proportional to how difficult it was to discover the first time, divided by how obvious it is once successfully explained. While some truths only have to be uncovered to become obvious, others may have a lucid explanation that is even more difficult to find that the truth itself was. If there exists a lucid explanation, the difficulty of finding it should be counted in the numerator of the heuristic.

weaker), or as an extension of the object-language to data structures beyond the purview of fexprs (making the language stronger).

By a straightforward understanding of *object language*, when the object language is Kernel, or any typical Lisp, all expressions in the object language are unencapsulated. That is, encapsulated objects cannot be read from a Lisp source file: they only arise when those source objects are evaluated. Following a denotational approach to programming language semantics, in which syntax values are mapped to semantic values, syntax structure is exactly what fexprs are able to deconstruct, and all encapsulated objects are semantic values that are not syntax. Therefore, if the denotational mapping is to be described by a term-rewriting calculus, the term set of the calculus must be a proper superset of the term set of the object language (the "syntax"). The term-rewriting approach is more natural for a Lisp, since Lisp characteristically equates the domains of data (denoted values) and program (denoting values) — so that the denotational model itself is no longer useful, but its dichotomy between semantics and syntax lingers in the distinction between objects that fexprs can and cannot deconstruct.

The possibility of *active* terms in the calculus —that is, terms that are neither language input (denoting) nor language output (denoted)— does not occur in [Wa98]. The conclusion of [Wa98] brushes close to this possibility when acknowledging that the semantics of fexprs will ultimately require something beyond the purview of fexprs: "we care not only when two terms have the same behavior, but we may also care just what that behavior is!". The bridge from that sentiment to active terms is that the missing behavioral element can be incorporated into the term syntax of the modeling calculus. For this dissertation, that bridge was arrived at through the implicit/explicit evaluation distinction discussed in §1.2.3.

Once active terms are included in the toolkit of available modeling techniques, even a fully reflective formal system can be modeled by a calculus with a nontrivial formal equational theory.

15.2 Nontrivial models of full reflection

What we intend by *fully reflective* is not merely that all objects are S-expressions, which would in itself guarantee that all operational equivalences between objects are trivial, but that the object language is a *small-step term-rewriting system* in which all terms are S-expressions. This is the situation in [Wa98], and the situation of which we observe that the semantics of the object language can be modeled by a calculus with a nontrivial formal equational theory.

To illuminate the principles involved, we develop a nontrivial modeling calculus for the fully reflective formal system of [Wa98], then a second variant modeling calculus, and consider the prospects for (but do not try to construct) a third variant. The fully reflective system to be modeled is presented here under the name *W-semantics*.

The first modeling calculus tries to make the smallest possible modification to W-semantics that will afford a technically nontrivial theory. We call this *W-calculus*. W-calculus demonstrates in pure form the fundamental enabling principle for nontriviality in calculi modeling full reflection, unadorned by any attempt to localize rewriting or strengthen the theory.

The second modeling calculus, *$\mathcal{W}$-calculus*, relents from the stark minimalism of W-calculus by eliminating unbounded-depth evaluation contexts from its redex patterns.

The third natural step in upgrading the modeling calculus would be to allow lazy subterm reduction, analogous to the removal of value-subterm constraints when upgrading from $\mathfrak{k}_r$-calculus to $\mathfrak{k}_i$-calculus (§10.7). In particular, we would like to allow a β-reduction to be performed without first reducing the body of the operator to a calculus normal form. Unfortunately, there are significant technical challenges in arranging this, which will be described and briefly discussed in §15.2.4.

15.2.1 W-semantics

The fully reflective formal system, W-semantics, is now the object language. All the terms of W-semantics are S-expressions. The syntax of S-expressions is

W-semantics.

Syntax (S-expressions):

$$\begin{array}{lll} x \in \textit{Variables} & & \\ S ::= x \mid (\lambda x.S) \mid (S\,S) \mid (\textbf{fexpr}\ S) \mid (\textbf{eval}\ S) & \text{(Sexprs)} & \\ V ::= (\lambda x.S) \mid (\textbf{fexpr}\ V) & \text{(Values)}\,. & \end{array} \tag{15.2}$$

Relation $\longmapsto_W$ is a deterministic non-compatible binary relation on S-expressions. Besides the usual substitution borrowed from λ-calculus, [Wa98] uses a Mogensen–Scott encoding on an arbitrary S-expression S when it occurs as an operand to a fexpr, to "reify" it homomorphically into an irreducible S-expression $\lceil S \rceil$ that can be queried to analyze the structure of the original S.

W-semantics.

Syntax (evaluation contexts):

$$E \ ::= \ \Box \mid (E\,T) \mid ((\lambda x.T)\,E) \mid (\textbf{fexpr}\ E) \mid (\textbf{eval}\ E) \qquad \text{(Evaluation contexts)}$$

Auxiliary functions:

$$\begin{array}{rcl} \lceil x_0 \rceil & = & (\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.(x_1\,x_0)))))) \\ \lceil (S_1\,S_2) \rceil & = & (\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.((x_2\,\lceil S_1 \rceil)\,\lceil S_2 \rceil)))))) \\ \lceil (\lambda x_0.S) \rceil & = & (\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.(x_3\,(\lambda x_0.\lceil S \rceil))))))) \\ \lceil (\textbf{fexpr}\ S) \rceil & = & (\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.(x_4\,\lceil S \rceil)))))) \\ \lceil (\textbf{eval}\ S) \rceil & = & (\lambda x_1.(\lambda x_2.(\lambda x_3.(\lambda x_4.(\lambda x_5.(x_5\,\lceil S \rceil)))))) \end{array} \qquad (15.3)$$

Schemata:

$$\begin{array}{rcll} E[((\lambda x.S)\,V)] & \longmapsto & E[S[x \leftarrow V]] & (\beta\text{-reduction}) \\ E[((\textbf{fexpr}\ V)\,S)] & \longmapsto & E[(V\,\lceil S \rceil)] & (\text{reification}) \\ E[(\textbf{eval}\ \lceil S \rceil)] & \longmapsto & E[S] & (\text{reflection})\,. \end{array}$$

Definition 15.4 S_1, S_2 are *W-contextually equivalent*, denoted $S_1 \simeq_W S_2$, if for every context C such that $C[S_1], C[S_2] \in \mathit{Sexprs}$,
there exists V_1 such that $C[S_1] \longmapsto_W^* V_1$
iff there exists V_2 such that $C[S_2] \longmapsto_W^* V_2$. ■

The paper's main result is

Theorem 15.5 (Triviality)
$S_1 \simeq_W S_2$ iff $S_1 \equiv_\alpha S_2$. ■

Proof. Straightforward; see [Wa98]. ■

15.2.2 W-calculus

In the modeling calculi, which practice explicit evaluation, no reductions $S_1 \longmapsto_W S_2$ will be admitted, since they are all reductions of inactive terms. Instead, we introduce an active frame $[\mathcal{E}\,\Box]$, and all of our reduction schemata require an active redex. The $\mathcal{E}$ frame is a declaration of intent to reduce the framed term to a value. Since W-contextual equivalence, the relation of primary interest, is defined using values in the sense of (15.2), the modeling calculi keep its exact definition intact, including the requirement that a value cannot contain any active subterms. In W-semantics, $\lceil S \rceil$ is irreducible; in the calculi, $[\mathcal{E}\,\lceil S \rceil] \longrightarrow_\bullet^* \lceil S \rceil$.

The entire point of W-calculus is that, for technical nontriviality, all we have to do is lift $\longmapsto_W$ to active terms.

W-calculus.

Syntax (terms, extending W-semantics):

$$\begin{array}{lll} I ::= x \mid (\lambda x.T) \mid (T\,T) \mid (\mathbf{fexpr}\ T) \mid (\mathbf{eval}\ T) & & \text{(Inactive)} \\ A ::= [\mathcal{E}\,T] & & \text{(Active)} \\ T ::= A \mid I & & \text{(Terms)} \end{array} \tag{15.6}$$

Schemata:

$$\begin{array}{lll} [\mathcal{E}\,S_1] \longrightarrow [\mathcal{E}\,S_2] & \text{if } S_1 \longmapsto_W S_2 & (\bot) \\ [\mathcal{E}\,V] \longrightarrow V & & (V)\,. \end{array}$$

Keep in mind that we are primarily interested in compatible relation $\longrightarrow_W$ (the compatible closure of $\longrightarrow^W$).

Theorem 15.7 (Correspondence)

$S_1 \longmapsto^*_W S_2$ iff $[\mathcal{E}\,S_1] \longrightarrow^*_W [\mathcal{E}\,S_2]$.

$S \longmapsto^*_W V$ iff $[\mathcal{E}\,S] \longrightarrow^+_W V$. ■

Proof. Immediate. ■

Theorem 15.8 (Church–Rosser-ness)
$\longrightarrow^W$ is Church–Rosser. ■

Proof. Straightforward, by induction on the number of active frames $[\mathcal{E}\,\Box]$ in the term (which is non-increasing across $\longrightarrow_W$ steps). The case of a single active frame is immediate from the fact that $\longmapsto_W$ is deterministic. For nested $\mathcal{E}$ frames, all inner $\mathcal{E}$ frames have to be reduced to an S-expression before the outer frame can be reduced (because Schema $(\bot)$ requires its redex subterm to be an S-expression). ■

Definition 15.9 W-calculus terms T_1, T_2 are *W-contextually equivalent*, denoted $T_1 \simeq_W T_2$, if for every context C,
there exists V_1 such that $C[T_1] \longrightarrow^*_W V_1$
iff there exists V_2 such that $C[T_2] \longrightarrow^*_W V_2$. ■

There is no need to use some different notation for this relation than for the W-semantics contextual equivalence of Definition 15.4, because they agree on the domain of the more restricted relation (by Theorem 15.7).

Theorem 15.10 (Nontriviality)
There exist $T_1 \not\simeq_W T_2$. ■

Proof. Let $S \longmapsto_W V$. Then $[\mathcal{E}\,S] \longrightarrow^2_W V$. $[\mathcal{E}\,S] \not\equiv_\alpha V$. By Church–Rosser-ness, for all C' and V', $C'[[\mathcal{E}\,S]] \longrightarrow^*_W V'$ iff $C'[V] \longrightarrow^*_W V'$. ■

15.2.3 $\mathcal{W}$-calculus

We prefer not to embed unbounded-depth evaluation contexts into the redex patterns on the left-hand sides of our calculus schemata (cf. §11.3). To avoid this, in $\mathcal{W}$-calculus we instead propagate $\mathcal{E}$ frames downward through evaluation contexts until they reach the local redex. The term syntax is unchanged from W-calculus.

$\mathcal{W}$-calculus.

Schemata:

$$\begin{array}{rcll}
[\mathcal{E}\,[\mathcal{E}\,T]] & \longrightarrow & [\mathcal{E}\,T] & (\mathcal{E}) \\
[\mathcal{E}\,(\lambda x.T)] & \longrightarrow & (\lambda x.T) & (\lambda) \\
[\mathcal{E}\,(T_1\,T_2)] & \longrightarrow & [\mathcal{E}\,([\mathcal{E}\,T_1]\,T_2)] & (p) \\
[\mathcal{E}\,((\lambda x.T_1)\,T_2)] & \longrightarrow & [\mathcal{E}\,((\lambda x.T_1)\,[\mathcal{E}\,T_2])] & (p\lambda) \\
[\mathcal{E}\,(\textbf{fexpr}\ T)] & \longrightarrow & (\textbf{fexpr}\ [\mathcal{E}\,T]) & (\mathbf{f}) \\
[\mathcal{E}\,(\textbf{eval}\ T)] & \longrightarrow & [\mathcal{E}\,(\textbf{eval}\ [\mathcal{E}\,T])] & (\mathbf{e}) \\
[\mathcal{E}\,((\lambda x.S)\,V)] & \longrightarrow & [\mathcal{E}\,S[x \leftarrow V]] & (\beta\text{-reduction}) \\
[\mathcal{E}\,((\textbf{fexpr}\ V)\,S)] & \longrightarrow & [\mathcal{E}\,(V\,\lceil S\rceil)] & (\text{reification}) \\
[\mathcal{E}\,(\textbf{eval}\ \lceil S\rceil)] & \longrightarrow & [\mathcal{E}\,S] & (\text{reflection})\,.
\end{array} \tag{15.11}$$

The three schemata of W-semantics are adapted intact, only replacing the evaluation contexts with active frames. The other six schemata serve only to redistribute $\mathcal{E}$ labels across the term; call them *$\mathcal{E}$-distribution schemata*. Let $\longrightarrow^{\mathcal{W}d}$ be the enumerated relation of the $\mathcal{E}$-distribution schemata, and $\longrightarrow^{\mathcal{W}c} = (\longrightarrow^{\mathcal{W}} - \longrightarrow^{\mathcal{W}d})$. Let $d(T)$ be the term that results from reducing T via $\longrightarrow_{\mathcal{W}d}$ so that there are $\mathcal{E}$ labels on all, and only, those subterms where the labels might be useful.

$$\begin{array}{rcl}
d(x) & = & x \\
d((\lambda x.T)) & = & (\lambda x.d(T)) \\
d((T_1\,T_2)) & = & (d(T_1)\,d(T_2)) \\
d((\textbf{fexpr}\ T)) & = & (\textbf{fexpr}\ d(T)) \\
d((\textbf{eval}\ T)) & = & (\textbf{eval}\ d(T)) \\
d([\mathcal{E}\,x]) & = & [\mathcal{E}\,x] \\
d([\mathcal{E}\,(\lambda x.T)]) & = & (\lambda x.d(T)) \\
d([\mathcal{E}\,(T_1\,T_2)]) & = & \begin{cases} [\mathcal{E}\,(d(T_1)\,d([\mathcal{E}\,T_2]))] & \text{if } d(T_1) \text{ has the form } (\lambda x.T) \\ [\mathcal{E}\,(d([\mathcal{E}\,T_1])\,d(T_2))] & \text{otherwise} \end{cases} \\
d([\mathcal{E}\,(\textbf{fexpr}\ T)]) & = & (\textbf{fexpr}\ d([\mathcal{E}\,T])) \\
d([\mathcal{E}\,(\textbf{eval}\ T)]) & = & [\mathcal{E}\,(\textbf{eval}\ d([\mathcal{E}\,T]))] \\
d([\mathcal{E}\,[\mathcal{E}\,T]]) & = & d([\mathcal{E}\,T])\,.
\end{array} \tag{15.12}$$

Lemma 15.13

$T \longrightarrow^*_{\mathcal{W}d} d(T)$.

If $T_1 \longrightarrow_{\mathcal{W}d} T_2$, then $d(T_1) \equiv_\alpha d(T_2)$.

$\longrightarrow^{\mathcal{W}d}$ is Church–Rosser. ■

Proof. Straightforward. ■

Not only does d collapse the term set of $\mathcal{W}$-calculus into a set of equivalence classes closed under $\longrightarrow_{\mathcal{W}d}$, but $\longrightarrow_{\mathcal{W}c}$ respects those classes. Let $T_1 \longrightarrow_{\mathcal{W}'} T_2$ iff $d(T_1) \longrightarrow_{\mathcal{W}c} \cdot \longrightarrow^*_{\mathcal{W}d} d(T_2)$; then

Lemma 15.14

If $T_1 \longrightarrow_{\mathcal{W}c} T_2$, then $d(T_1) \longrightarrow_{\mathcal{W}'} d(T_2)$.

If $T_1 \longrightarrow^*_W T_2$, then $T_1 \longrightarrow^*_{\mathcal{W}'} T_2$.

If $T_1 \longrightarrow^*_{\mathcal{W}'} T_2$, then there exists T_2' such that $T_1 \longrightarrow^*_W T_2'$ and $T_2' =_{\mathcal{W}d} T_2$. ■

Proof. By cases. ■

Theorem 15.15 (Correspondence)

$S_1 \longmapsto^*_W S_2$ iff $[\mathcal{E}\, S_1] \longrightarrow^*_{\mathcal{W}'} [\mathcal{E}\, S_2]$.

$S \longmapsto^*_W V$ iff $[\mathcal{E}\, S] \longrightarrow^+_{\mathcal{W}} V$. ■

Proof. Lemma 15.14 and Theorem 15.7. ■

Theorem 15.16 (Church–Rosser-ness)

$\longrightarrow^{\mathcal{W}}$ is Church–Rosser. ■

Proof. Lemma 15.14 and Theorem 15.8.

Definition 15.17 $\mathcal{W}$-calculus terms T_1, T_2 are *$\mathcal{W}$-contextually equivalent*, denoted $T_1 \simeq_{\mathcal{W}} T_2$, if for every context C,
there exists V_1 such that $C[T_1] \longrightarrow^*_{\mathcal{W}} V_1$
iff there exists V_2 such that $C[T_2] \longrightarrow^*_{\mathcal{W}} V_2$. ■

Theorem 15.18 (Nontriviality)

$T_1 \simeq_W T_2$ iff $T_1 \simeq_{\mathcal{W}} T_2$.

There exist $T_1 \not\simeq_{\mathcal{W}} T_2$. ■

Proof. Lemma 15.14 and Theorem 15.10. ■

15.2.4 Lazy subterm reduction

$\mathcal{W}$-calculus puts eager constraints on redex subterms in five positions:

- the β-reduction operator must be an S-expression;
- the β-reduction operand must be a value;
- the reification operator must be a value;
- the reification operand must be an S-expression; and
- the reflection body must be a Mogensen–Scott encoded S-expression.

β-reduction, being the most complex case, is both most difficult and of most interest. The β-reduction operand constraint cannot safely be tampered with.

The reification/reflection constraints can be relaxed without serious difficulty. The reification operator can simply replace V with T. Reification operand and reflection body require additional machinery, because if we don't insist on having an S-expression at the time of reification/reflection, we need an active frame to tell us what do with an S-expression when we (hopefully) derive it later. Suppose two new active frames: $[\mathcal{M}\,\Box]$ declaring intent to encode the framed term, and $[\mathcal{O}\,\Box]$ declaring intent to decode the framed term. We could replace the reification and reflection schemata with

$$\begin{array}{rcll}
[\mathcal{E}\,((\mathbf{fexpr}\ T_1)\,T_2)] & \longrightarrow & [\mathcal{E}\,(T_1\,[\mathcal{M}\,T_2])] & \text{(reification)} \\
[\mathcal{E}\,(\mathbf{eval}\,T)] & \longrightarrow & [\mathcal{E}\,[\mathcal{O}\,T]] & \text{(reflection)} \\
[\mathcal{O}\,[\mathcal{M}\,T]] & \longrightarrow & T & \text{(cancel)} \\
[\mathcal{M}\,S] & \longrightarrow & \lceil S \rceil & (\mathcal{M}) \\
[\mathcal{O}\,\lceil S \rceil] & \longrightarrow & S & (\mathcal{O})\,.
\end{array} \tag{15.19}$$

With some additional tedium, we could also break up the monolithic $(\mathcal{M})$ and $(\mathcal{O})$ schemata so that they descend by increments into the syntactic structure of the term, according to the five cases in the definition of the encoding (in (15.3)). We won't pursue that option, though; lazy β-operators are a much bigger prize, if we could manage them.

There is a serious problem with lazy reduction of β-reduction operators. The Mogensen–Scott encoding doesn't commute with substitution for free variables in the term being encoded: encoding a free variable and then substituting for it will produce a different result, in general, than substituting for the free variable and then encoding. The question of which to do first only matters when an active term, whose reduction would entail encoding a subterm with a free variable x, occurs within a β-reduction redex whose operator binds x. This type of situation cannot occur when reducing a term of the form $[\mathcal{E}\,S]$, so it cannot affect the correspondence between $S \longmapsto^*_W V$ and $[\mathcal{E}\,S] \longrightarrow^*_\bullet V$; but such situations matter for Church–Rosser-ness. For example, in a lazy-β calculus,

$$\begin{array}{lll}
[\mathcal{E}\,((\lambda x.[\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,x)])\,V)] & \longrightarrow_\bullet^+ & [\mathcal{E}\,((\lambda x.\lceil x\rceil)\,V)] \\
 & \longrightarrow_\bullet^+ & (\lceil x\rceil)[x \leftarrow V] \\
[\mathcal{E}\,((\lambda x.[\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,x)])\,V)] & \longrightarrow_\bullet^+ & [\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,V)] \\
 & \longrightarrow_\bullet^+ & \lceil V\rceil\,.
\end{array} \tag{15.20}$$

For Church–Rosser-ness, one or the other of these orders must be disallowed. $\mathcal{W}$-calculus disallows substitution before encoding, by requiring the operator of a β-reduction to have no remaining active subterms (note the role of bottom-up elimination of $\mathcal{E}$ frames in the inductive proof of Theorem 15.8).

With some rather baroque provisions in the calculus, one could allow lazy-operator β-reduction while still putting encoding before substitution. When a substitution is applied to any active frame, the active frame could intercept and suspend the substitution as a binding, until such time as the framed term becomes an S-expression — something like this:

$$\begin{array}{lll}
[\mathcal{E}\,((\lambda x.[\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,x)])\,V)] & \longrightarrow_\bullet & \\
[\mathcal{E}\,[\![x \leftarrow V]\!]\,((\mathbf{fexpr}\ (\lambda y.y))\,x)] & \longrightarrow_\bullet^+ & [\mathcal{E}\,[\![x \leftarrow V]\!]\,\lceil x\rceil] \\
 & \longrightarrow_\bullet & (\lceil x\rceil)[x \leftarrow V]\,.
\end{array} \tag{15.21}$$

The complexity of this approach corresponds to its undermining of the uniformity of λ-style substitution.

A much simpler approach, with a possibly-surmountable deficiency in operational completeness, is to refuse to encode an S-expression as long as it has any free variables. This favors the substitution-before-encoding order; but it also refuses to complete some mundane reductions. For example, terms

$$\begin{array}{r}
[\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,x)] \\
(\lambda x.[\mathcal{E}\,((\mathbf{fexpr}\ (\lambda y.y))\,x)])
\end{array} \tag{15.22}$$

cannot be reduced to values, because the x that occurs free in the reification operand will never be substituted for, and therefore its encoding will never be allowed to proceed. Since the first term has the form $[\mathcal{E}\,S]$, by failing to reduce it to a value, the calculus fails operational completeness. It might be possible to devise an alternative concept to serve in place of the usual notion of operational completeness, involving suitable surrounding contexts akin to those used in operational equivalence.

Both these approaches have evident drawbacks. An entirely different approach to the problem is to reassess the use of a Mogensen–Scott encoding. It is unclear (at the current time and to the current author) how much of the problem is due to the essential nature of full reflection, and how much of it is due to particular properties of the encoding. One might gain insight into this balance by replacing, or modifying, the encoding.

15.3 Abstraction contexts

Besides [Wa98], another modern paper that touches on the misbehavior of fexprs is

John C. Mitchell's 1993 "On Abstraction and the Expressive Power of Programming Languages" ([Mi93]). Unlike [Wa98], which is centrally concerned with fully reflective fexprs (though not fexprs in general), [Mi93] only brings in fexprs to make a peripheral point about the main topic of interest, which is *abstraction-preserving* transformations between languages.[2] The peripheral point being made is that there exist languages that have no abstraction, so that they cannot be the codomain of any abstraction-preserving transformation unless the domain doesn't have any abstraction either; Lisp with fexprs is the paper's recommended canonical example of a language with no abstraction.

Here too is an appearance of contradiction with the current work. The starting point of of this dissertation was that fexprs should provide increased abstractive power (§1.1); and the thesis claims that fexprs can subsume traditional abstractions (§1.3). Again, however, there is no actual contradiction, only a seeming that results from differences in terminology and technical approach. As is typical of traditional treatments of programming language semantics, [Mi93] focuses on compiled programming languages — and consequently, as was noted earlier of the denotational approach, the only objects considered are *S-expressions* in the sense defined here in §15.1. The non-abstractiveness result in [Mi93] is therefore technically a statement only about S-expressions in the presence of fexprs, and is technically very close to Proposition 15.1.

The key definition in [Mi93] is that of an *abstraction context*.

Definition 15.23 Suppose language $\mathcal{L}$ with big-step semantic relation $\longmapsto_{\mathcal{L}}$.

T_1 and T_2 are *observationally equivalent* in $\mathcal{L}$, denoted $T_1 \simeq_{\mathcal{L}} T_2$, if for every context C and observable O,
$C[T_1] \longmapsto_{\mathcal{L}} O$
iff $C[T_2] \longmapsto_{\mathcal{L}} O$.

C hides the difference between T_1 and T_2 in $\mathcal{L}$ if $T_1 \not\simeq_{\mathcal{L}} T_2$ and $C[T_1] \simeq_{\mathcal{L}} C[T_2]$.

C is an *abstraction context* in $\mathcal{L}$ if C hides the difference between some terms T_1 and T_2 in $\mathcal{L}$. ■

C hiding the difference between T_1 and T_2 implies that for all C' and O', $C'[C[T_1]] \longmapsto_{\mathcal{L}} O'$ iff $C'[C[T_2]] \longmapsto_{\mathcal{L}} O'$. An abstraction-preserving transformation $\theta\colon \mathcal{L} \to \mathcal{L}'$ is a homomorphism from $\mathcal{L}$ programs to $\mathcal{L}'$ programs that preserves both observational equivalence and observational inequivalence ($T_1 \simeq_{\mathcal{L}} T_2$ iff $\theta(T_1) \simeq_{\mathcal{L}'} \theta(T_2)$). θ is "abstraction-preserving" in that whenever C hides the difference between T_1 and T_2 in $\mathcal{L}$, preserving both $\simeq_{\mathcal{L}}$ and $\not\simeq_{\mathcal{L}}$ across the homomorphism guarantees that $\theta(C)$ hides the difference between $\theta(T_1)$ and $\theta(T_2)$. Fexprs come into the situation because, owing to the fact that all programs considered by the paper are S-expressions, Lisp with fexprs has *no abstraction contexts*. Hence the point made by the paper, that

[2]The actual terminology in the paper is "abstraction-preserving reduction", which we avoid since we're heavily invested in a different use of the word *reduction*.

if θ is abstraction-preserving, and $\mathcal{L}'$ is Lisp with fexprs, then $\mathcal{L}$ doesn't have any abstraction contexts (because all $\mathcal{L}$ contexts map into non-abstraction $\mathcal{L}'$ contexts).

Abstraction in Kernel is an interpretation-time phenomenon, involving objects that cannot be deconstructed by fexprs. Recall from Chapter 5 that encapsulation of operatives and environments were key to supporting Kernel hygiene. Formally, in $\mathfrak{l}_p$-calculus the minimal abstraction contexts are operative frames, environment frames, eval frames, and combine frames.

15.4 λ-calculus as a theory of fexprs

One other point that may further illuminate the situation concerns the way calculi are used to model the semantics of Lisp.

Traditionally, λ-calculus expressions are taken as directly modeling Lisp programs — which is to say, Lisp S-expressions. This is essentially a compiled approach: a Lisp source expression

```
(($lambda (x) (* x x)) (+ 2 3))
```
(15.24)

would be mapped directly to a λ-calculus expression

$$((\lambda x.((* \, x) \, x)) \, ((+ \, 2) \, 3)) \, . \tag{15.25}$$

The mapping presumes that the operand will be evaluated. If we wanted to extend the modeling technique to support fexprs, a mechanically minimal change to the calculus might somehow *suppress* evaluation of the otherwise evaluable subterm $((+ \, 2) \, 3)$, an essentially implicit-evaluation impulse (§1.2.3) whose consequences have been considered both via quotation (§8.4.1) and via full reflection (§15.1), which are complementary views of the same effect.

A naive mapping from Lisp to $\mathfrak{l}_p$-calculus would map (15.24) to something like

[eval `(($lambda (x) (* x x)) (+ 2 3))` e_0] . (15.26)

Note that most of this term is an S-expression in the technical sense of this chapter: an expression that can be completely deconstructed if it is passed as an operand to a fexpr. Under favorable conditions (stable bindings, §5.3)), though, the same degree of deduction might be possible here as in the non-fexpr case of (15.25); and in that case, there *is* a part of $\mathfrak{l}_p$-calculus available whose term-reduction properties —in cases where we can ignore dynamic environments— are exactly like those of the unadulterated λ-calculus:

$$\begin{aligned}[\text{combine } \langle \mathfrak{l}x.[\text{combine } [\text{combine } * \; x \; \langle\!\rangle] \; x \; \langle\!\rangle] \rangle \\ [\text{combine } [\text{combine } + \; 2 \; \langle\!\rangle] \; 3 \; \langle\!\rangle] \\ \langle\!\rangle] \, .\end{aligned} \tag{15.27}$$

None of the subterms here are S-expressions: they belong to a part of $\mathfrak{l}_p$-calculus that is fully encapsulated, and therefore has no obstacle to a nontrivial theory. A

key insight here is that this particular corner of $\mathfrak{f}_p$-calculus is, in fact, isomorphic to unadulterated λ-calculus, complete with the same reduction schemata —namely, the β-rule and nothing else— and with the same equational theory. (This may be more obvious for $\mathfrak{f}_x$-calculus, §9.3, which has no environments at all; but since we are considering a subset of terms that doesn't involve symbols, there would be no way to access the contents of dynamic environments anyway, and we can simply ignore them.) So all the equational strength that we expected of our modeling λ-calculus *before* we added fexprs is still available in the modeling $\mathfrak{f}_p$-calculus after, to be used in those cases where we can prove it is safe to do so. $\mathfrak{f}_p$-calculus conservatively extends this λ-isomorphic subcalculus. The isomorphism is

$$\begin{array}{rcl} \theta(x) & = & x \\ \theta(c) & = & c \\ \theta((T_1\, T_2)) & = & [\text{combine } \theta(T_1)\ \theta(T_2)\ \langle\!\langle\rangle\!\rangle] \\ \theta((\lambda x.T)) & = & \langle \mathfrak{f} x.\theta(T)\rangle\,. \end{array} \tag{15.28}$$

There is an insight here into the way both calculi (λ and $\mathfrak{f}_p$) function when modeling Lisp, in the fact that a λ-expression —which conventionally one would think of as modeling an applicative— is mapped isomorphically to a $\mathfrak{f}$-expression that evidently models an operative. As noted, the image of this isomorphism is still used exactly as the co-image used to be, but not as often: it only comes into play when we can compile away all of the trappings of evaluation. The evaluation machinery occupies the whole rest of $\mathfrak{f}_p$-calculus beyond the pale of the isomorphic image, and the *absence* of that machinery from λ-calculus is the reason we had trouble using λ-calculus as a model of fexpr-based situations that are intrinsically concerned with evaluation. Since we can only do without that machinery in $\mathfrak{f}_p$-calculus when working with pure fexprs — and we are using the isomorphic image in the same way that we had used its co-image (i.e., λ-calculus) — a reasonable view of the situation is that the combiners of λ-calculus always were, essentially, fexprs: they don't do anything special to bring about evaluation of their operands, after all (because they can't, as that machinery is outside the purview of λ-calculus), and as soon as it becomes necessary to consider the distinction between operatives and applicatives, λ-expressions take the operative part. The association simply wasn't apparent when distinction never had to be considered, and the predominant object-language combiners with the same name were applicatives.

Overall, $\mathfrak{f}_p$-calculus has three parts: a set of terms representing S-expressions (pairs, symbols, and constants), with a trivial equational theory; a subcalculus representing pure fexpr-call structure, which has isomorphically the equational theory of λ-calculus; and machinery for evaluation, connecting the first two components and determining what mixture of the weak and strong theories of those components will bear on a given situation.

Chapter 16

Conclusion

16.0 Introduction

Fexprs can form the basis for a simple, well-behaved Scheme-like language, subsuming traditional abstractions without a multi-phase model of evaluation.

This chapter offers a Big Picture view of the dissertation's support for the thesis.

16.1 Well-behavedness

The most notorious criticism of fexprs, in recent years, has been the assertion by [Wa98] that "The Theory of Fexprs is Trivial"; Chapter 15 addressed this directly, clarifying that the theoretical scenario investigated by that paper uses fully reflective fexprs, whereas the fexprs in Kernel are not fully reflective in the relevant technical sense.[1] While removing the absolute condemnation of trivialization, however, this does not in itself remove the broader concern of ill-behavior, of which trivialization is only the most extreme theoretical form. Practical ill-behavior was central to the deprecation of fexprs circa 1980 ([Pi80]). The most extreme form of practical ill-behavior of fexprs occurs in dynamically scoped Lisps, which dominated mainstream Lisp at the time of the deprecation, but (ironically) began to be phased out of the mainstream a few years later. The historical evolution of Lisp combiner constructors, including dynamic versus static scope, was reviewed in §3.3. Practical measures to minimize and mitigate the ill-behavior of fexprs were discussed in Chapter 5, and their deployment was described in Chapter 7. On the theoretical side, the feasibility of well-behaved modeling calculi for Lisp with fexprs, with Church–Rosser and Plotkin's Correspondence Theorems, was established by Chapters 9–14.

[1] This is the refutation in Chapter 15 that matters directly to the thesis. It was also demonstrated, in §15.2, that a well-behaved modeling calculus can exist even when the object language itself is fully reflective and therefore as badly behaved as possible; but that demonstration was offered to further illuminate the principles involved. The fact that the object language *isn't* as badly behaved as possible is rather more to the point.

16.2 Simplicity

The strategy used here to achieve simplicity is to put fexprs at the center of the evaluation model, as the primary driving mechanism of all Lisp function application. The core element of all combiner construction is therefore the operative constructor, `$vau`. Peripheral derivation of applicatives from these underlying operatives is managed via `wrap` and `unwrap`, whose orthogonality to `$vau` and to the operative function-call mechanism (theoretically, the β-rule) facilitates deductions about separation of evaluation concerns from operative calls. The basic combiner treatment is described in Chapter 4. Purely subjective aspects of its simplicity are demonstrated through use in Part I, especially Chapter 7; a relatively concrete practical manifestation of its simplicity, through the relative size of its meta-circular evaluator, is studied in Chapter 6. The significance of cleanly separating argument evaluation concerns from operative calls is, as brought out by §15.4, that the weaknesses of the equational theory occur in the parts of it that deal with evaluation, while the part of the theory that deals with operative calls alone is isomorphic to the theory of λ-calculus.

The thesis claims simplicity of the programming language — not simplicity of some associated modeling calculus, let alone simplicity of some particular proof of well-behavedness of some associated modeling calculus. It therefore has no *direct* bearing on the thesis that the well-behavedness of $\mathfrak{k}$-calculi is established here by an exceedingly un-simple theoretical treatment (most especially Chapter 13). Given that simplicity remains a subjective property, though, it may be of some value for the thesis to note that the first proof of a result is likely to be complicated *because* it is the first proof, regardless of whether a simpler proof is eventually formulated. A topical example is that, while Church and Rosser published their proof of what is now called the Church–Rosser Theorem in 1936 ([ChuRo36]), a *simple* proof of the theorem, due to Martin-Löf, only appeared three and a half decades later.[2]

Simplicity of the calculi themselves is of somewhat greater indirect relevance since it is only one stage removed from the object language (while simplicity of the proofs for the calculi is two stages removed). The impure calculi's use of multiple classes of variables, with multiple distinct forms of substitution, is an unfamiliar complication from the pure λ-calculus; however, the multiplicity of substitution functions is recommended here as a means of *simplifying* the treatment of impurity. It is suggested that Felleisen's impure calculi were complicated by their attempt to fit impure calculus behaviors into the mold of the β-reduction of λ-calculus (see primarily §8.3.3); the alternative forms of substitution in the impure $\mathfrak{k}$-calculi allow them to bubble up side-effect frames without the characteristic structural churning that occurs in Felleisen's calculi.[3] A possible target for future work (as the general principles in-

[2] Cf. Footnote 11 of Chapter 13.

[3] Felleisen had the enviable disadvantage of being first, making the alternative approach to bubbling up in $\mathfrak{k}$-calculi another example (to whatever extent it has merit) of time delay from a first treatment to the development of a simpler approach.

volved become better understood) is the development of some unified treatment of the different classes of variables, though it isn't clear at this time whether that unification would actually simplify the situation, or just clarify it.[4]

16.3 Subsuming traditional abstractions

The fact that fexprs can do what macros can do is not very deep, although it is key to the underlying motivation for pursuing fexprs — abstractive power (§1.1). Practical replacement of macros by fexprs was addressed in Chapter 7.

16.4 A closing thought

Fexprs can form the basis for a simple, well-behaved Scheme-like language, subsuming traditional abstractions without a multi-phase model of evaluation.

[4]There are some bemusing similarities between the problem of unifying the classes of variables in λ-calculi, and the problem of developing a TOE in physics. Traditionally there are four forces in physics, versus four classes of variables (though this is somewhat historically selective, given the unification of the electroweak force around the time of Klop's dissertation). More suggestively, one of the forces is clearly not like the others (gravity), in ways that seem to give it a peculiarly central role in the overall structure of the universe; while one of the classes of variables is clearly not like the others (λ-calculus variables, a.k.a. partial-evaluation variables), and plays a peculiarly central role in the overall structure of computation.

Appendices

Appendix A

Complete source code for the meta-circular evaluators

This appendix provides complete source code for the six meta-circular evaluators of Chapter 6. In comments, the six are referred to shortly as `vanilla` (vanilla Scheme), `template` (naive template macros), `procedural` (naive procedural macros), `hygienic` (hygienic macros), `single` (single-phase macros), and `kernel`.

In their executable form, the source definitions for the evaluators are distributed amongst 26 source files, such that no definition occurs in more than one file, and for each evaluator a master file selects just the definitions it needs by *`load`*ing a subset of the 26. This appendix simplifies the organization somewhat for human readers, repeating or regrouping some definitions to present a small number of logically coherent sets of alternatives.

The high-level code, all of which appeared in Chapter 6, is conservative in its use of Kernel features that differ behaviorally from Scheme; no first-class operatives are used, and compound definiends are used only as the parameter trees of *`$lambda`*-expressions. The low-level code observes no such restrictions; first-class operatives and compound definiends are used at convenience, and the encapsulated object-language types are implemented using Kernel's *`make-encapsulation-type`* device ([Shu09, §8 (Encapsulations)]).

A.1 Top-level code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; one-phase algorithms (vanilla, single, kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! interpreter
   ($lambda () (rep-loop (make-initial-env))))

($define! rep-loop
   ($lambda (env)
      (display ">>> ")
      (write (mceval (read) env))
      (newline)
      (rep-loop env)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; two-phase algorithms (template, procedural, hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! interpreter
   ($lambda () (rep-loop (make-initial-macro-env) (make-initial-env))))

($define! rep-loop
   ($lambda (macro-env env)
      (display ">>> ")
      (write (mceval (preprocess (read) macro-env) env))
      (newline)
      (rep-loop macro-env env)))
```

A.2 mceval

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; non-gensym scheme algorithms (vanilla, template) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mceval
   ($lambda (expr env)
      ($cond ((symbol? expr)  (lookup expr env))
             ((pair? expr)    (mceval-combination expr env))
             (#t  expr))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; gensym scheme algorithms (procedural, hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mceval
   ($lambda (expr env)
      ($cond ((mc-symbol? expr)  (lookup expr env))
             ((pair? expr)       (mceval-combination expr env))
             (#t expr))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; non-scheme algorithms (single, kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mceval
   ($lambda (expr env)
      ($cond ((mc-symbol? expr)  (lookup expr env))
             ((pair? expr)       (combine (mceval (car expr) env)
                                          (cdr expr)
                                          env))
             (#t expr))))
```

A.3 Combination evaluation (high-level)

```
;
; Most of each algorithm is presented in a single block, except for
; a block common to all algorithms (presented at the end), and a
; small block that is only added for the "procedural" algorithm.
;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; scheme algorithms (single, template, procedural, hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mceval-combination
   ($lambda ((operator . operands) env)
      ($cond ((if-operator? operator)      (mceval-if operands env))
             ((define-operator? operator)  (mceval-define operands env))
             ((lambda-operator? operator)  (mceval-lambda operands env))
             (#t  (mc-apply (mceval     operator env)
                            (map-mceval operands env))))))
```

```
($define! if-operator?     ($make-tag-predicate $if))
($define! define-operator? ($make-tag-predicate $define!))
($define! lambda-operator? ($make-tag-predicate $lambda))

($define! mceval-if
   ($lambda ((test consequent alternative) env)
      ($if (mceval test env)
           (mceval consequent env)
           (mceval alternative env))))

($define! mceval-define
   ($lambda ((definiend definition) env)
      (match! env definiend (mceval definition env))))

($define! mceval-lambda
   ($lambda ((ptree body) env)
      (make-applicative
         ($lambda arguments
            ($let ((env  (make-mc-environment env)))
               (match! env ptree arguments)
               (mceval body env))))))

($mc-define! apply (make-applicative mc-apply))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive procedural macro algorithm (procedural) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($mc-define! gensym (make-applicative gensym))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; single-phase macro algorithm (single) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! combine
   ($lambda (combiner operands env)
      ($if (mc-operative? combiner)
           (mc-operate combiner operands env)
           (mc-apply combiner (map-mceval operands env)))))
```

```
($mc-define! $if
   (make-operative
      ($lambda ((test consequent alternative) env)
         ($if (mceval test env)
              (mceval consequent env)
              (mceval alternative env)))))

($mc-define! $define!
   (make-operative
      ($lambda ((definiend definition) env)
         (match! env definiend (mceval definition env)))))

($mc-define! $lambda
   (make-operative
      ($lambda ((ptree body) env)
         (make-applicative
            ($lambda arguments
               ($let ((env  (make-mc-environment env)))
                  (match! env ptree arguments)
                  (mceval body env)))))))

($mc-define! $macro
   (make-operative
      ($lambda ((parameters names body) static-env)
         (make-operative
            ($lambda (operands dynamic-env)
               (mceval
                  ($let ((local-env  (make-mc-environment static-env)))
                     (match! local-env parameters operands)
                     (gensyms! local-env names)
                     (transcribe body local-env))
                  dynamic-env))))))

($define! gensyms!
   ($lambda (env names)
      ($cond ((mc-symbol? names)  (match! env names (gensym)))
             ((pair? names)
                 (gensyms! env (car names))
                 (gensyms! env (cdr names))))))
```

```
($define! transcribe
   ($lambda (body env)
      ($cond ((mc-symbol? body)  (lookup body env))
             ((pair? body)
                 (cons (transcribe (car body) env)
                       (transcribe (cdr body) env)))
             (#t  body))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; kernel algorithm (kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! combine
   ($lambda (combiner operands env)
      ($if (mc-operative? combiner)
           (mc-operate combiner operands env)
           (combine (mc-unwrap combiner) (map-mceval operands env) env))))

($mc-define! wrap   (make-applicative mc-wrap))
($mc-define! unwrap (make-applicative mc-unwrap))
($mc-define! eval   (make-applicative mceval))

;
; make-applicative must precede the common block (presented below).
;

($define! make-applicative
   ($lambda (meta-appv)
      (mc-wrap
         (make-operative
            ($lambda (operands #ignore)
               (apply meta-appv operands))))))

;
; the rest of this block must follow the common block (presented below).
;

($mc-define! $if
   (make-operative
      ($lambda ((test consequent alternative) env)
         ($if (mceval test env)
              (mceval consequent env)
              (mceval alternative env)))))
```

```
($mc-define! $define!
   (make-operative
      ($lambda ((definiend definition) env)
         (match! env definiend (mceval definition env)))))

($mc-define! $vau
   (make-operative
      ($lambda ((ptree eparam body) static-env)
         (make-operative
            ($lambda (operands dynamic-env)
               ($let ((local-env  (make-mc-environment static-env)))
                  (match! local-env ptree operands)
                  (match! local-env eparam dynamic-env)
                  (mceval body local-env)))))))

;;;;;;;;;;;;;;;;;;
; all algorithms ;
;;;;;;;;;;;;;;;;;;
;
; This block must precede all (other) calls to $mc-define!,
; since they add to ground-environment.
;

($define! ground-environment (make-mc-environment))

($define! make-initial-env
   ($lambda ()
      (make-mc-environment ground-environment)))

($define! map-mceval
   ($lambda (operands env)
      (map ($lambda (expr) (mceval expr env))
           operands)))

($mc-define! <?    (make-applicative <? ))
($mc-define! <=?   (make-applicative <=?))
($mc-define! =?    (make-applicative =? ))
($mc-define! >=?   (make-applicative >=?))
($mc-define! >?    (make-applicative >? ))
($mc-define! +     (make-applicative +  ))
($mc-define! -     (make-applicative -  ))
($mc-define! *     (make-applicative *  ))
($mc-define! /     (make-applicative /  ))
($mc-define! cons  (make-applicative cons))
```

A.4 Preprocessing (high-level)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; all two-phase algorithms (template, procedural, hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! preprocess
   ($lambda (expr macro-env)
      ($if ($and? (pair? expr)
                  (define-macro-operator? (car expr)))
           (preprocess-define-macro! (cdr expr) macro-env)
           (expand expr macro-env))))

($define! define-macro-operator?
          ($make-tag-predicate $define-macro))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive macro algorithms (template, procedural) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! make-initial-macro-env
   ($lambda ()
      (make-empty-macro-env ($lambda (x) x))))

($define! expand
   ($lambda (expr macro-env)
      ($if (pair? expr)
           (check-for-macro-call
              (map ($lambda (expr) (expand expr macro-env))
                   expr)
              macro-env)
           expr)))

($define! check-for-macro-call
   ($lambda (expr macro-env)
      ($if (symbol? (car expr))
           ($let ((x  (macro-lookup (car expr) macro-env)))
              ($if (symbol? x)
                   expr
                   (expand (apply-macro x (cdr expr))
                           macro-env)))
           expr)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive template macro algorithm (template) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! preprocess-define-macro!
   ($lambda (((name . parameters) #ignore template)
             macro-env)
      (macro-match! macro-env name (make-macro parameters template))))

($define! make-macro
   ($lambda (parameters template)
      ($lambda operands
         ($let ((macro-env  (make-empty-macro-env ($lambda (x) x))))
            (macro-match! macro-env parameters operands)
            (transcribe template macro-env)))))

($define! transcribe
   ($lambda (template macro-env)
      ($cond ((symbol? template)
                  (macro-lookup template macro-env))
             ((pair? template)
                  (cons (transcribe (car template) macro-env)
                        (transcribe (cdr template) macro-env)))
             (#t  template))))

($define! apply-macro apply)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive procedural macro algorithm (procedural) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! preprocess-define-macro!
   ($lambda ((name definition) macro-env)
      (macro-match! macro-env name
         (mceval definition (make-initial-env)))))

($define! apply-macro mc-apply)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; hygienic macro algorithm (hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! preprocess-define-macro!
   ($lambda (((name . parameters) #ignore template)
             macro-env)
      (macro-match! macro-env name
         (make-macro parameters template macro-env))))

($define! macro? applicative?)

($define! apply-macro
   ($lambda (macro operands macro-env)
      (macro operands macro-env)))

($define! expand
   ($lambda (expr macro-env)
      ($cond ((pair? expr)
                  ($let ((x (macro-lookup (car expr) macro-env)))
                     ($if (macro? x)
                          (apply-macro x (cdr expr) macro-env)
                          (map ($lambda (x) (expand x macro-env))
                               expr))))
             ((mc-symbol? expr)
                 (macro-lookup expr macro-env))
             (#t  expr))))


($define! ground-macro-env (make-empty-macro-env ($lambda (x) x)))

($define! make-initial-macro-env
   ($lambda ()
      (make-child-macro-env ground-macro-env)))


($define! make-lambda    ($make-tag-prefixer $lambda))
($define! make-let-macro ($make-tag-prefixer $let-macro))
```

```
($mc-macro-define! $let-macro
   ($lambda ((((name . ptree) #ignore template) body) macro-env)
      ($let ((macro      (make-macro ptree template macro-env))
             (macro-env  (make-child-macro-env macro-env)))
         (macro-match! macro-env name macro)
         (expand body macro-env))))

($mc-macro-define! $lambda
   ($lambda ((ptree body) macro-env)
      ($let ((macro-env  (make-child-macro-env macro-env)))
         ($let ((ptree  (rename-ptree! ptree macro-env)))
            (make-lambda ptree (expand body macro-env))))))

($define! rename-ptree!
   ($lambda (ptree macro-env)
      ($cond ((mc-symbol? ptree)
                  ($let ((gs  (gensym)))
                     (macro-match! macro-env ptree gs)
                     gs))
             ((pair? ptree)
                 (cons (rename-ptree! (car ptree) macro-env)
                       (rename-ptree! (cdr ptree) macro-env)))
             (#t  ptree))))

($define! make-macro
   ($lambda (parameters template static-menv)
      ($lambda (operands dynamic-menv)

         ($define! new-menv (make-child-macro-env dynamic-menv))

         ($define! subst-menv
            (make-empty-macro-env
               ($lambda (s)
                  ($let ((gs  (gensym)))
                     (macro-match! subst-menv s gs)
                     (macro-match! new-menv
                                   gs (macro-lookup s static-menv))
                     gs))))

         (macro-match! subst-menv parameters operands)
         ($let ((expr  (transcribe template subst-menv)))
            (expand expr new-menv)))))
```

```
($define! transcribe
   ($lambda (template subst-menv)
      ($cond ((mc-symbol? template)
                  (macro-lookup template subst-menv))
             ((pair? template)
                  (cons (transcribe (car template) subst-menv)
                        (transcribe (cdr template) subst-menv)))
             (#t  template))))
```

A.5 Evaluation (low-level)

```
;;;;;;;;;;;;;;;;;;
; all algorithms ;
;;;;;;;;;;;;;;;;;;

($define! $mc-define!
   ($vau (ptree expr) env
      ($let ((expr  (eval expr env)))
         (match! ground-environment ptree expr))))

;
; environments
;
($provide! (make-mc-environment  lookup  match!)

   ($define! (encapsulate
              #ignore
              decapsulate)  (make-encapsulation-type))
   ;
   ; The encapsulated value is a pair whose car is a list of local
   ; bindings, and whose cdr is a list of the contents of the parents.
   ;

   ($define! make-mc-environment
      ($lambda x
         (encapsulate (cons ()
                            (map decapsulate x)))))
```

```
($define! lookup
   ($lambda (symbol env)

      ($define! get-binding
         ($lambda (tree)
            ($let ((binding  (assoc symbol (car tree))))
               ($if (pair? binding)
                    binding
                    ($let ((bindings  (filter pair?
                                              (map get-binding
                                                 (cdr tree)))))
                       ($if (pair? bindings)
                            (car bindings)
                            ()))))))

      ($let ((binding  (get-binding (decapsulate env))))
         ($if (null? binding)
              ($sequence
                 (display "Dying horribly due to unbound symbol ")
                 (display symbol)
                 (newline))
              #inert)
         (cdr binding))))

($define! bind!
   ($lambda (tree symbol value)
      ($let ((binding  (assoc symbol (car tree))))
         ($if (pair? binding)
              (set-cdr! binding value)
              (set-car! tree (cons (cons symbol value)
                                   (car tree)))))))
```

```
   ($define! tree-match!
      ($lambda (binding-tree parameter-tree operand-tree)
         ($cond ((mc-symbol? parameter-tree)
                     (bind! binding-tree parameter-tree operand-tree))
                ((pair? parameter-tree)
                     (tree-match! binding-tree
                                  (car parameter-tree)
                                  (car operand-tree))
                     (tree-match! binding-tree
                                  (cdr parameter-tree)
                                  (cdr operand-tree))))))
                ; no error-handling,
                ; so no other cases needed

   ($define! match!
      ($lambda (env parameter-tree operand-tree)
         (tree-match! (decapsulate env)
                      parameter-tree
                      operand-tree))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; primitive-lambda algorithms (all but kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; applicatives
;
($provide! (make-applicative  mc-apply)

   ($define! (encapsulate
              #ignore
              decapsulate)  (make-encapsulation-type))
   ;
   ; The encapsulated value is a meta-language applicative
   ; that expects to be applied to the argument list.
   ;

   ($define! make-applicative
      ($lambda (action)
         (encapsulate action)))

   ($define! mc-apply
      ($lambda (appv args)
         (apply (decapsulate appv) args))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; non-scheme algorithms (single, kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; operatives
;
($provide! (mc-operative?  make-operative  mc-operate)

   ($define! (encapsulate
              mc-operative?
              decapsulate)  (make-encapsulation-type))
   ;
   ; The encapsulated value is a meta-language applicative whose
   ; arguments should be the operand list and the dynamic environment.
   ;

   ($define! make-operative
      ($lambda (action)
         (encapsulate action)))

   ($define! mc-operate
      ($lambda (appv args env)
         ((decapsulate appv) args env))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; gensym algorithms (procedural, hygienic, single) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; gensyms
;
($provide! (gensym  mc-symbol?)

   ($define! (encapsulate
              gensym?
              decapsulate)  (make-encapsulation-type))

   ($define! get-unique-ticket
      ($letrec ((self     (get-current-environment))
                (counter  0))
         ($lambda ()
            ($set! self counter (+ counter 1))
            counter)))
```

```
   ($define! gensym
      ($lambda ()
         (encapsulate (get-unique-ticket))))

   ($define! mc-symbol?
      ($lambda (x)
         ($or? (symbol? x) (gensym? x)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; vanilla algorithm (vanilla) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mc-symbol? symbol?)

($define! $make-tag-predicate
   ($vau (tag) #ignore
      ($lambda (x) (eq? x tag))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive template macro algorithm (template) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! mc-symbol? symbol?)

($define! $make-tag-predicate
   ($vau (tag) #ignore
      ($lambda (x) (eq? x tag))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; naive procedural macro algorithm (procedural) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! $make-tag-predicate
   ($vau (tag) #ignore
      ($lambda (x) (eq? x tag))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; hygienic macro algorithm (hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! $make-tag-predicate
   ($vau (tag) #ignore
      ($lambda (x) (eq? x tag))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; kernel algorithm (kernel) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; applicatives
;
($define! (mc-wrap #ignore mc-unwrap) (make-encapsulation-type))
```

A.6 Preprocessing (low-level)

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; two-phase algorithms (template, procedural, hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; macro environments
;
($provide! (make-empty-macro-env
            make-child-macro-env
            macro-match!
            macro-lookup)

   ($define! (encapsulate
               #ignore
               decapsulate)  (make-encapsulation-type))
   ;
   ; The encapsulated value is a pair, whose car is a list of bindings,
   ; and whose cdr is either a default-behavior applicative, or the
   ; value encapsulated by the parent.
   ;

   ($define! make-empty-macro-env
      ($lambda (default-behavior)
         (encapsulate (cons () default-behavior))))

   ($define! make-child-macro-env   ; for hygienic macros
      ($lambda (parent)
         (encapsulate (cons () (decapsulate parent)))))
```

```
   ($define! bind!
      ($lambda (lss symbol value)
         (set-car! lss (cons (cons symbol value)
                             (car lss)))))

   ($define! match!
      ($lambda (lss parameter-tree operand-tree)
         ($cond ((mc-symbol? parameter-tree)
                   (bind! lss parameter-tree operand-tree))
                ((pair? parameter-tree)
                   (match! lss
                           (car parameter-tree)
                           (car operand-tree))
                   (match! lss
                           (cdr parameter-tree)
                           (cdr operand-tree))))))

   ($define! macro-match!
      ($lambda (macro-env parameter-tree operand-tree)
         (match! (decapsulate macro-env)
                 parameter-tree
                 operand-tree)))

   ($define! macro-lookup
      ($lambda (symbol macro-env)

         ($define! aux
            ($lambda (lss)
               ($let ((binding  (assoc symbol (car lss))))
                  ($cond ((pair? binding)  (cdr binding))
                         ((pair? (cdr lss))  (aux (cdr lss)))
                         (#t  ((cdr lss) symbol))))))

        (aux (decapsulate macro-env)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; hygienic macro algorithm (hygienic) ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

($define! $make-tag-prefixer
   ($vau (tag) #ignore
      ($lambda x (cons tag x))))
```

```
($define! $mc-macro-define!
   ($vau (name expr) env
      ($let ((expr  (eval expr env)))
         (macro-match! ground-macro-env name expr))))
```

Appendix B

Compilation of Kernel programs

This appendix comments briefly on how Kernel programs might be compiled for fast execution.

The thesis makes no direct claim about fast execution. If Kernel were inherently unoptimizable, that would cast doubt on the well-behavedness claim of the thesis; but the existence and promotion of stable special cases, which are therefore optimizable, was addressed in Chapter 5. Nevertheless, fast execution was an important factor in the historical debate between macros and fexprs, and so a serious discussion of fexprs begs the question of how execution speed using fexprs compares to that using macros.

Expansion of preprocessed macros takes *zero execution time*, exactly because preprocessing is, by definition, strictly prior to execution. It does not necessarily follow that preprocessing a macro will cause the program that uses it to execute faster than if the macro were handled lazily, during execution; on the contrary, macro expansion to a larger executable image may actually penalize execution time, due to caching. However, one would still like to know that fexprs *can* be inlined (the usual term for preprocessing-time replacement of a compound-combiner call with a customized transcription of the body of the combiner), so that a compiler has the freedom to choose whether inlining is appropriate; and, moreover, macro "expansion" may be Turing-powerful, and so does not necessarily result in a larger executable image than would a fexpr-based solution.

Inlining a hygienic applicative call is a well-understood technique, and inlining a hygienic operative call should be comparable. The principal outstanding question, then, is to what extent one can inline calls to an *un*hygienic fexpr.

Fexprs that subsume the purposes of macros (and that are therefore central to our comparison) are typically unhygienic just to the extent that each such fexpr, after some bounded internal computation to derive suitable expressions from its operands, uses its dynamic environment to evaluate the derived expressions. (The conversion of macros into fexprs of this kind was described in §7.3.) This is a very well-behaved sort of bad hygiene, in that symbols in the captured operands are ultimately evaluated

in the static environment of the source-code region from which they were captured.[1] Assuming that the fexpr's own static and local environments are stable, an optimizing compiler can then readily inline the fexpr, using embedded combiners and gensyms to maintain hygiene. For example, given the derivation of hygienic binary ***$or?*** (from §5.1),

```
($define! $or?
   ($vau (x y) env
      ($let ((temp  (eval x env)))
         ($if temp temp (eval y env))))),
```
(B.1)

a combination (***$or?*** ⟨x⟩ ⟨y⟩) could be rewritten as

```
($sequence
   ($define! ⟨g⟩ ⟨x⟩)
   ($if ⟨g⟩ ⟨g⟩ ⟨y⟩)),
```
(B.2)

where ⟨g⟩ is a unique symbol generated for the particular instance of combination-inlining.

One cannot casually use a ***$let*** in the inlined code, because that would change the properties of the dynamic environment of ⟨y⟩ for purposes of mutation. If the body of the fexpr to be inlined is sufficiently tame in its use of local variables —if, for example, it doesn't capture its own local variable-names and doesn't perform exotic environment-mutations— then one can safely store the local variables in the dynamic environment, as in the above example (B.2). Local variables that cannot safely be handled this way would be more troublesome, though in principle one might still perform inlining by maintaining a variable in the dynamic environment whose value is itself an explicitly simulated local environment of the fexpr.

An advanced, but still manageable, inlining transformation may be possible if the macro/fexpr is *tail-recursive*. A tail-recursive combiner calls itself only through a *tail call*, that is, a call whose result will immediately become the result of the caller. The advantage of a tail call is that it does not require storage of any deferred actions by the caller: the continuation provided for the result of the caller is provided by the caller for the result of the tail call. Deep tail-recursion only requires a single continuation, rather than a deep stack of continuations; and it is possible, in principle, for a language implementation to support unbounded depths of tail recursion, even

[1] It appears that, with some effort, one might formulate a weaker hygiene condition that this case would satisfy (something about the relationship between the source-code position of a symbol and the environment through which its value is actually observed). The effort of formulation was not compellingly justified for this dissertation, since after it one would still have a hygiene condition that cannot be guaranteed in general. Formulation of a similar, but more sophisticated, hygiene condition would be integral to the development of an environment-guarding facility, which was discussed in Footnote 9 of §5.2.3.

though it has only finite memory space. Such an implementation is said to be *properly tail-recursive*.

In a strictly hygienic Lisp, continuations are the only obstacle to properly tail-recursive implementation. However, if first-class combiners can capture their dynamic environments, it is possible that arbitrarily deep tail recursion, while requiring only a bounded continuation stack, may require an arbitrarily deep chaining of *environments*, as the local environment of each recursive call might contain a link back to the local environment of the preceding recursive call. This is why traditional, dynamically scoped Lisps weren't properly tail recursive; part of why modern Lisps (following the precedent of Scheme) are statically scoped; and why Kernel pointedly allows compound combiners to *not* capture their dynamic environments, via atom `#ignore`, and then encourages its use by building it into ***$lambda***.

However, when a fexpr captures its dynamic environment, and then calls itself recursively, it will most likely make the recursive call *in* its dynamic environment. This is because the likely reason for capturing its dynamic environment in the first place is so that the operands, perhaps somewhat transformed, can be evaluated therein; and in a recursive call, the intended environment for operand evaluation is the same as was intended by the recursing caller. Here, for example, is a derivation of variadic ***$or?***:

```
($define! $or?
   ($vau x e
      ($cond ((null? x)          #f)
             ((null? (cdr x))    (eval (car x) e))
             ((eval (car x) e)   #t)
             (#t                 (apply (wrap $or?) (cdr x)
                                        e)))))
```
(B.3)

The only remaining difficulty in recognizing that a call to ***$or?*** can be inlined is then recognizing that its recursion follows the syntactic structure of the operand. Once this is determined, the entire recursive sequence of calls can be inlined; for example, a combination `(`***$or?*** ⟨x⟩ ⟨y⟩`)` would be inlined (with the obvious optimization of eliminating needless tests) as

```
(($cond (⟨x⟩  #t)
        (#t  ($cond (⟨y⟩  #t)
                    (#t  #f)))).
```
(B.4)

Commonly, any local variables within the fexpr will disappear during inlining, because they usually refer to parts of the operand tree, and operand manipulations will be handled during inlining.

It isn't even necessary, for inlining of a recursive-fexpr call, that the operand list of the call be acyclic. Given a fexpr that recursively traverses a cyclic operand list,

one can splice the inline expansion into the cyclic structure. For example (using the cyclic-structure notation from SRFI-38, [Dil03]), an expression

```
($or? . #0=(⟨x⟩ ⟨y⟩ . #0#)),                                    (B.5)
```

which is a cyclic list of three pairs with a cycle length of 2, would be inlined as

```
#0=($cond (⟨x⟩ #t)
          (#t ($cond (⟨y⟩ #t)                                   (B.6)
                     (#t #0#)))).
```

Splicing the expansion into the structure of the operand list avoids an infinite loop at expansion time (though the code might still cause an infinite loop at execution time, if neither evaluation of ⟨x⟩ nor evaluation of ⟨y⟩ produces a time-dependent result).

Appendix C

The letter vau

In choosing a name and glyphs (lower- and upper-case, per §8.2) for the constructor of operatives, practical and aesthetic criteria were used (aesthetics here being, in part, a means to the practical goal of mnemonic utility):

- The name should provide continuity with the traditional name *lambda* of the constructor of applicatives. Therefore the name of a letter was to be chosen, also conveniently providing glyphs.
- Its upper- and lower-case glyphs should be immediately distinguishable from other common mathematical characters (mainly, Greek Roman and English letters); while, at the same time, they should fit stylistically with the Greek alphabet as usually typeset in mathematics (respectively upper- and lower-case), so that their intended roles in the notional conventions —which require case associations— are immediately recognized without cognitive dissonance.

Combiner constructors with call syntax similar to `$lambda` (or λ) are often given names that are variants of lambda — as Interlisp NLAMBDA for operative construction, or Felleisen's λ_v for call-by-value applicative construction ([Fe91]). However, using a variant of lambda would imply a central role for `$lambda` that it doesn't have in the current work. The possibility was considered of using the letter corresponding to λ in another (by preference, even older) alphabet, but the choices were mostly uninspiring.[1] It was decided to use instead a different classical Greek letter, providing continuity with lambda through the choice of alphabet rather than through the choice of lineage.

Choosing a letter not cognate to λ loses the specific mnemonic association of the choice with λ-calculus; so, to compensate, a letter was sought that would have

[1]The most interesting relative of λ identified, because of its additional connection to hacker culture (both an asset and a liability), was *lambe*, which is J.R.R. Tolkien's elvish letter (tengwa) for the sound of λ. (Re the cultural significance, see [Ra03, "Elvish"].) However, the name *lambe* is so similar in spelling to *lambda* as to foster confusion, and the glyphic representation of lambe, $\mathfrak{T}$, looks like a relative of τ rather than λ.

mnemonic association with the new purpose for which it was to be used. Considered particularly likely were classical Greek cognates to the letters O (mnemonic for *Operative*), or S or F (mnemonic for either word in *Special Form*).

- The classical Greek letter properly corresponding to the initial O in *Operative* is omicron. However, omicron's glyphs are the same as modern English O, so that when embedded in a modern English document it fails to look like classical Greek, undermining its association with λ.

- The other classical Greek letter corresponding to English O is omega. Omega has the sound of long O ("mega O"), which doesn't properly correspond to the initial short O ("micro O") of the word *Operative*. Moreover, lower- and upper-case omega are associated with traditional uses in related mathematics — notably, lower-case omega for the countable set of nonnegative integers, upper-case omega for an unnormalizable term in λ-calculus— that would be misleading in the current work.

- The classical Greek letter corresponding to S is sigma. Lower-case sigma is already being used in the realm of λ-calculi for an entirely different purpose (σ-capabilities, §8.3.3.2).

The usual, 24-letter classical Greek alphabet,

α	A	alpha	ι	I	iota	ρ	P	rho
β	B	beta	κ	K	kappa	σ	Σ	sigma
γ	Γ	gamma	λ	Λ	lambda	τ	T	tau
δ	Δ	delta	μ	M	mu	υ	Υ	upsilon
ϵ	E	epsilon	ν	N	nu	ϕ	Φ	phi
ζ	Z	zeta	ξ	Ξ	xi	χ	X	chi
η	H	eta	o	O	omicron	ψ	Ψ	psi
θ	Θ	theta	π	Π	pi	ω	Ω	omega,

(C.1)

has no letter corresponding to F. That alphabet was used in Hellenistic times for writing *text*.

Ancient alphabets, though, were also often used as numerals — the first letter representing 1, the second 2, and so on. The ancient Greeks also devised another, more sophisticated system of letter-numerals: the first nine letters represented integers 1–9; the next nine letters represented multiples of ten, 10–90; and the last nine letters represented multiples of a hundred, 100–900. Thus, a sequence of 1–3 letters could represent any integer up to 999. (For larger numbers, a mark added to a numeral would multiply its value by a thousand). Of course, this system requires 27 letters. They used for the purpose two letters that had been dropped from the

written language some time before, and one invented symbol (that may or may not have been descended from a pre-Greek letter):

1	α	A	alpha	10	ι	I	iota	100	ρ	P	rho	
2	β	B	beta	20	κ	K	kappa	200	σ	Σ	sigma	
3	γ	Γ	gamma	30	λ	Λ	lambda	300	τ	T	tau	
4	δ	Δ	delta	40	μ	M	mu	400	υ	Υ	upsilon	
5	ϵ	E	epsilon	50	ν	N	nu	500	ϕ	Φ	phi	(C.2)
6			vau	60	ξ	Ξ	xi	600	χ	X	chi	
7	ζ	Z	zeta	70	o	O	omicron	700	ψ	Ψ	psi	
8	η	H	eta	80	π	Π	pi	800	ω	Ω	omega	
9	θ	Θ	theta	90			koppa	900			sampi.	

The invented symbol, sampi, was simply tacked onto the end of the sequence. The letters vau and koppa, which had existed in early versions of the Greek alphabet, were retained as numerals in the alphabetic positions they had inherited from the Phoenicians, and recognizably in the positions that the cognate Roman letters still occupy in modern English: vau, corresponding to *F*, in the sequence *DEF* (delta epsilon vau); koppa, corresponding to *Q*, in the sequence *OPQRST* (omicron pi koppa rho sigma tau).

Kappa and koppa (or qoppa) are cognate to, respectively, Phoenician letters kaph and koph (qoph), which represent sounds that are meaningfully distinct in Semitic languages such as Phoenician, but are not meaningfully distinct in Indo-European languages such as Greek. To the Greeks, therefore, both letters effectively represented the same sound, and they dropped the second of the two, koppa, at an early stage in the development of their alphabet.

The story of vau is more tangled.[2] *Vau* (or *wau*, *waw*, *vav*) is the name of the sixth letter in the Phoenician alphabet, with approximately the sound of *w*. The Greeks, needing letters for vowel sounds (written vowels are needed for Indo-European but not for Semitic languages), requisitioned the form of the Semitic vau to represent the vowel sound of *u*. However, the resulting vowel wasn't placed in the position of its ancestor, sixth, because there was a lingering need for the sound of *w*; instead it was tacked onto the end of the alphabet, after tau (i.e., upsilon), and the sound of *w* retained the sixth position, but took on a new form (on whose origins there are multiple theories). Eventually, the new form of the sixth letter stabilized on something much like "F", and in that form it is commonly called *digamma*, describing (so we

[2]The literature about vau is tangled, too. Readily available accounts of vau routinely contradict each other on important points. I assembled the brief composite account here after consulting a number of sources, most of which had little of substance to say on the subject and far less, I eventually concluded, that could be relied upon. My most trusted particular sources (after careful comparison and contrast) were, uncoincidentally, also the most recent — two 1971 Encyclopædia Britannica articles ("F" and "Alphabet") and (believe it or not) several Wikipedia articles (notably "stigma (letter)", "digamma", "waw (letter)"; `http://en.wikipedia.org/`, observed 16 December 2005). A good representative of earlier sources is [Men69].

are told) its shape that resembles two overlapping gammas. Alternatively, a number of authors have freely called the Greek letter *vau*.[3] The name *vau* is preferred for the current work, and for Kernel, on the grounds that *digamma* doesn't convey a suitable sense of atomicity (rather, it suggests a composite device, derived from gamma).[4]

In a further twist to the story, since medieval times the numeral vau has often been given the written form "ϛ" (even when being called *digamma*, a name allegedly descriptive of a different shape[5]). Symbol ϛ, properly called *stigma*, is a medieval ligature for the sequence sigma-tau; and in fact the Greek numeral six has also sometimes been written as "στ". The stigma glyph is also often mistaken for a variant form of sigma.[6]

Observing the situation's remarkably high level of muddlement, especially that the Ⅎ-like symbol is commonly called *digamma* while both names *vau* and *digamma* have multiple forms, it was judged reasonable to treat the written form of vau for the current work as an open question, with initially in play the full variety of early historical forms for the letter. Some of the more interesting of these forms (all of these dating from, as it happens, circa 600 BCE) are

Cretan

Attic

Corinthian

Chalcidian . (C.3)

We want a glyph that *doesn't* look like lambda ("Λ"), which rules out the Cretan; and we want a glyph that doesn't look like "F", which rules out the Chalcidian. The modern associations of the symbol "**?**" prevent it from looking like a letter, let alone a Greek letter. We therefore take the Corinthian backwards-Ⅎ as a starting point. To provide the required case distinction and stylistic compatibility with modern mathematical notation, we suppose that, if "ꟻ" were put through the same evolutionary

[3]This could be taken as simply retaining the name of the ancestral Semitic letter. The Wikipedia article on "digamma" remarked —very plausibly, given the other literature I've observed— that it simply isn't known what the ancient Greeks called the letter; but, having said that, the article also suggested that they most likely called it *vau*.

[4]The choice of *vau* over occasional competing spelling *wau* (the suspected ancient Greek spelling is vau-alpha-upsilon, which settles the second and third letters, but begs the question of the first) follows in part the same selective principle, as English letter *v* conveys a more atomic sense than English letter *w* (called "double *u*" and written as double *v*).

[5]However, by numerical value regardless of glyphic representation, digamma is twice gamma.

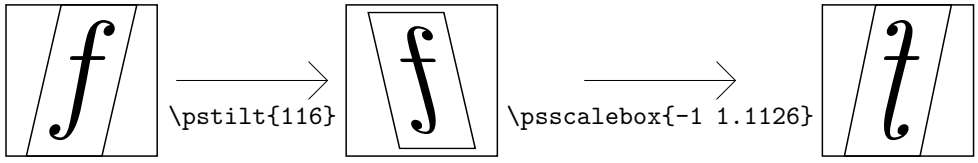
[6]In a final twist that somehow isn't as surprising as it ought to be, the Wikipedia article on "stigma" noted that *stigma* is also an alternative name for an obsolete Cyrillic letter. . . *koppa*, derived from the Greek *qoppa*.

transformation as "F", it would develop into a left-right reflection of the modern English F — which premise also has the practical advantage of letting us typeset our vau glyphs (both cases) by transforming readily available F glyphs.

For the mechanics of typesetting vau glyphs in LaTeX 2_ε (the medium of the dissertation), we use the PSTricks package commands `\psscalebox` and `\pstilt`. The upper-case vau is a simple left-to-right reflection of text-roman "F":

```
\newcommand{\Vau}{{\psscalebox{-1 1}{\textrm{F}}}} .
```
(C.4)

Because lower-case vau transforms the slanted mathematical form of lower-case F, "f" (rather than the vertical text-roman form, "f"), it can't be a simple left-right reflection, because the transformed letter needs to slant in the same direction as other mathematical letters (whereas simple left-right reflection would yield "ɟ"). To produce the proper effect, we use PSTricks command `\pstilt` to wrench the stem of the "f" backwards without reversing the arcs at top and bottom of the glyph, and then use `\psscalebox` to reflect the entire glyph left-right (and correct for vertical compression during the tilt):



(C.5)

.

(The leftward tilt is 26 degrees, mapping +90 (the positive y axis) to +116. The horizontal scale factor of 1.1126 is one over the cosine of 26 degrees. Why 26 degrees? Because it looks better than 25 or 27.)

The currently implemented LaTeX 2_ε code for lower-case vau (including my own clumsy code to correct the width of the letter) is

```
\newlength{\vauwidth}
\newcommand{\vau}{%
\settowidth{\vauwidth}{\ensuremath{f}}%
{\hspace*{-.4583\vauwidth}%
{\psscalebox{-1 1.1126}{\pstilt{116}{\ensuremath{f}}}}%
\hspace*{-.4583\vauwidth}} .
```
(C.6)

Because this code suffers from the common problem of failing to respond to font-size changes in math mode, the common task of embedding lower-case vau in mathematical super- and subscripts is streamlined by a separate command

```
\newcommand{\vauscript}{\mbox{\scriptsize \vau}} .
```
(C.7)

Bibliography

[AbSu85] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, New York: McGraw-Hill, 1985.

The first edition of the Wizard Book. ([Ra03, "Wizard Book"].) Mostly superseded by [AbSu96].

[AbSu96] Harold Abelson and Gerald Jay Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*, Second Edition, New York: MIT Press, 1996. Available (verified October 2009) at URL:
`http://mitpress.mit.edu/sicp/sicp.html`

The second edition of the Wizard Book. ([Ra03, "Wizard Book"].)

[Bac78] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", *Communications of the ACM* 21 no. 8 (August 1978), pp. 613–641.

Augmented form of the 1977 ACM Turing Award lecture, which proposes the functional programming paradigm, and coins the term "von Neumann bottleneck".

[Bare84] Hendrik Pieter Barendregt, *The Lambda Calculus: Its Syntax and Semantics* [*Studies in Logic and the Foundations of Mathematics* 103], Revised Edition, Amsterdam: North Holland, 1984.

The bible of lambda calculus.

[Baw88] Alan Bawden, *Reification without Evaluation*, memo 946, MIT AI Lab, June 1988. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

Explains, criticizes, and endeavors to improve on, the decomposition of 3-LISP by Friedman and Wand ([FrWa84, WaFr86]). Confirms that reflective procedures in [FrWa84] don't really need continuations, making them ordinary fexprs, which he then dismisses as an obviously bad idea. He goes on to show that the tower of meta-interpreters introduced in [WaFr86] doesn't really need continuations either (and would be better off without them). He proposes a

language called *Stepper* in which interpreter level shifts are handled by a mechanism orthogonal to continuations.

[Baw99] Alan Bawden, "Quasiquotation in Lisp", *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (*PESP'99*), San Antonio, Texas, January 22–23, 1999, pp. 88–99.

Analyzes advantages of quasiquotation, addresses splicing and nesting, and reviews the history of the technology.

[Baw00] Alan Bawden, "First-class Macros Have Types", *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages* [Boston, Massachusetts, USA, January 19–21, 2000], 2000, pp. 133–141. Available (verified October 2009) at URL:
`http://people.csail.mit.edu/alan/mtt/`

An implementation is available at the web site.

[Be69] James Bell, "Transformations: The Extension Facility of Proteus", in [ChrSh69], pp. 27–31.

Transformations encompass both macros and procedures; but their generality works by selecting properties, rather than unifying them.

[Bl94] Simon Blackburn, *The Oxford Dictionary of Philosophy*, New York: Oxford University Press, 1994.

[BrMo62] R.A. Brooker and D. Morris, "A General Translation Program for Phrase Structure Languages", *Journal of the ACM* 9 no. 1 (January 1962), pp. 1–10.

[Cam85] Martin Campbell-Kelly, "Christopher Strachey, 1916–1975 — A Biographical Note", *Annals of the History of Computing* 7 no. 1 (January 1985), pp. 19–42.

[Car63] Alfonso Caracciolo di Forino, "Some Remarks on the Syntax of Symbolic Programming Languages", *Communications of the ACM* 6 no. 8 (August 1963), pp. 456–460.

A lucid articulation and advocacy of the principle of grammar adaptability. Well thought out, and still worth reading.

[Che69] T.E. Cheatham, Jr., "Motivation for Extensible Languages" (followed by group discussion), in [ChrSh69], pp. 45–49.

The pro-extensibility complement to [McI69].

[Chr69] Carlos Christensen, "Extensible Languages Symposium — Chairman's Introduction", in [ChrSh69], p. 2.

[Chr88] Henning Christiansen, "Programming as language development", *datalogiske skrifter* no. 15, Roskilde University Centre, February, 1988.

[ChrSh69] Carlos Christensen and Christopher J. Shaw, editors, *Proceedings of the Extensible Languages Symposium*, Boston Massachusetts, May 13, 1969 [*SIGPLAN Notices* 4 no. 8 (August 1969)].

[Chu32/33] Alonzo Church, "A Set of Postulates for the Foundation of Logic" (two papers), *Annals of Mathematics* (2), 33 no. 2 (April 1932), pp. 346–366; and (2), 34 no. 4 (October 1933), pp. 839–864.

Church's logic that contains lambda calculus as a subset. The definition of substitution in this paper is corrected by [Kle34] to prevent variable capture.

[Chu36] Alonzo Church, "An Unsolvable Problem of Elementary Number Theory", *American Journal of Mathematics* 58 no. 2 (April 1936), pp. 345–363.

[Chu41] Alonzo Church, *The Calculi of Lambda-Conversion*, Annals of Mathematics Studies, Princeton: Princeton University Press, 1941.

77 thrilling pages. Heavy reading, but could be much worse.

[Chu71] Alonzo Church, "Logic, History of", *Encyclopædia Britannica*, 1971.

[ChuRo36] Alonzo Church and J.B. Rosser, "Some Properties of Conversion", *Transactions of the American Mathematical Society* 39 no. 3 (May 1936), pp. 472–482.

[Cl85] William Clinger, editor, *The Revised Revised Report on Scheme, or An Uncommon Lisp*, memo 848, MIT Artificial Intelligence Laboratory, August 1985. Also published as Computer Science Department Technical Report 174, Indiana University, June 1985. Available (verified April 2010) at URL: `http://publications.csail.mit.edu/ai/`

This is the first of the *RxRS*s with a huge pile of authors (for two perspectives, see [ReCl86, Introduction], [SteGa93, §2.11.1]). There is, as one might expect from a committee, almost nothing in the way of motivation. `LABELS` has disappeared in favor of a family of `LET` variants. The data types are elaborated, notably the column of numeric types, but also the other familiar ones such as ports and vectors, so that R2R Scheme looks a lot like the current language. There is a nifty little special form `REC` for creating a procedure that can name itself; `(REC x (lambda ...))` is equivalent to `(LETREC ((x (lambda ...))) x)`.

There is also, as one would not normally expect from a committee, a verse about lambda modeled on J.R.R. Tolkien's verse about the Rings of Power.

[ClRe91a] William Clinger and Jonathan Rees, "Macros that work", *POPL '91*: *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, Orlando, Florida, January 21–23, 1991, pp. 155–162.

Draws together the best of previous work reconciling macros with hygiene. The introduction extensively discusses the various problems that can arise with macros.

[ClRe91b] William Clinger and Jonathan Rees, editors, "The Revised[4] Report on the Algorithmic Language Scheme", *Lisp Pointers* 4 no. 3 (1991), pp. 1–55. Available (verified October 2009) at URL:
`http://www.cs.indiana.edu/scheme-repository/doc.standards.html`

[Cu29] H.B. Curry, "An Analysis of Logical Substitution", *American Journal of Mathematics* 51 no. 3 (July 1929), pp. 363–384.

[DaMyNy70] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard, "Common Base Language", Norwegian Computing Center Forskningveien 1 B, Oslo 3, Norway, Publication No. S-22 (Revised edition of publication S-2), October 1970. Available (verified October 2009) at URL:
`http://www.fh-jena.de/~kleine/history/history.html`

The original version was (apparently; I haven't seen it) from May 1968.

[Dij72] E.W. Dijkstra, "Notes on Structured Programming", in O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, *Structured Programming* [*A.P.I.C. Studies in Data Programming* 8], New York: Academic Press, 1972, pp. 1–82.

[Dil03] Ray Dillinger, "External Representation for Data with Shared Structure", *SRFI-38*, finalized 2 April 2003. Available (verified October 2009) at URL:
`http://srfi.schemers.org/srfi-38/`

[DoGhLe04] Daniel J. Dougherty, Silvia Ghilezan, and Pierre Lescanne, "Characterizing strong normalization in a language with control operators", *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, August 24–26, 2004, Verona, Italy, pp. 155–166. Available (verified October 2009) at URL:
`http://web.cs.wpi.edu/~dd/publications/#a_fest07`

[Dy92] Freeman Dyson, review of *Genius: The Life and science of Richard Feynman* by James Gleick (New York: Simon and Schuster, 1992); in *Physics Today*, November 1992, p. 87.

[Fe87] Matthias Felleisen, *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*, Ph.D. Dissertation, TR226, Computer Science Department, Indiana

University, 5 August 1987. Available (verified April 2010) at URL:
`http://www.ccs.neu.edu/scheme/pubs/#felleisen87`

[Fe88] Matthias Felleisen, "The Theory and Practice of First-Class Prompts", *POPL '88: Conference Record of the Fifteenth Annual ACM Conference on Principles of Programming Languages*, San Diego, California, January 10–13, 1988, pp. 180–190.

[Fe91] Matthias Felleisen, "On the Expressive Power of Programming Languages", *Science of Computer Programming* 17 nos. 1–3 (December 1991) [Selected Papers of ESOP '90, the 3rd European Symposium on Programming], pp. 35–75. A preliminary version appears in Neil D. Jones, editor, *ESOP '90: 3rd European Symposium on Programming* [Copenhagen, Denmark, May 15–18, 1990, *Proceedings*] [*Lecture Notes in Computer Science* 432], New York: Springer-Verlag, 1990, pp. 134–151. Available (verified October 2009) at URLs:
`http://www.ccs.neu.edu/scheme/pubs/#scp91-felleisen`
`http://www.cs.rice.edu/CS/PLT/Publications/Scheme/`

His formal criterion for expressiveness is closely related to Landin's notion of syntactic sugar and Kleene's formal definition of eliminable symbols in a formal system.

[FeFr89] Matthias Felleisen and Daniel P. Friedman, "A Syntactic Theory of Sequential State", *Theoretical Computer Science* 69 no. 3 (18 December 1989), pp. 243–287.

Provides some good insights into the nature of calculi.

[FeFrKoDu87] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba, "A Syntactic Theory of Sequential Control", *Theoretical Computer Science* 52 no. 3 (1987), pp. 205–237.

[FeHi92] Matthias Felleisen and Robert Hieb, "The Revised Report on the Syntactic Theories of Sequential Control and State", *Theoretical Computer Science* 103 no. 2 (September 1992), pp. 235–271. Available (verified December 2009) at URL:
`http://www.ccs.neu.edu/scheme/pubs/#tcs92-fh`

[Fl10] Matthew Flatt and PLT, *Reference: PLT Scheme*, version 4.2.5. Available (verified April 2010) at URL:
`http://download.plt-scheme.org/doc/html/reference/index.html`

[FrWa84] Daniel P. Friedman and Mitchell Wand, "Reification: Reflection without Metaphysics", *Proceedings of the 1984 ACM Conference on Lisp and Functional Programming*, 1984, pp. 348-355.

Reflective procedures are what might be called 'three-argument fexprs' — operand list, dynamic environment, and *continuation*; but the reflective capacity of the language actually comes from mixing these with an erosion of the encapsulation of the environment data type. Their language is called *Brown*, and includes neither the rearrangement of evaluation and quotation embodied in Smith's work by 2-LISP, [Sm84], nor Smith's infinite tower of interpreters. The tower is reintroduced in [WaFr86].

[Ga89] Richard P. Gabriel, editor, "Draft Report on Requirements for a Common Prototyping System", *SIGPLAN Notices* 24 no. 3 (March 1989), p. 93ff (independently paginated; only the first page also has issue pagination).

[Ga91] Richard P. Gabriel, "LISP: Good News, Bad News, How to Win Big", *AI Expert* 6 no. 6 (June 1991), pp. 30–39. Available (verified April 2010) at URL:
`http://www.dreamsongs.com/WIB.html`

[GaSt90] Richard P. Gabriel and Guy L. Steele Jr., "Editorial: The Failure of Abstraction", *Lisp and Symbolic Computation* 3 no. 1 (January 1990), pp. 5–12.

Lists three advantages of abstraction: abbreviation, opacity, and locality. Details nine failures of abstraction.

The first six are "failures of human spirit to push the concept of abstraction to its maximum extent": (1) During modification, locality disappears. (2–4) Lack of control, communication, and process abstractions. (5) These omissions cause programs to be written at varying levels of abstraction. (6) Failure to abstract over time.

The last three are "failures of the people who design and use programming languages": (7) Lack of support for documentation and other mechanisms for learning about an abstraction. (8) Abstractions are often selected for performance reasons. (9) Difficulty of reapplying abstractions to new problems.

[GiSu1880] Sir W.S. Gilbert and Sir Arthur Sullivan, *The Pirates of Penzance; or, The Slave of Duty*, 1880. Available (verified April 2010) at URL:
`http://math.boisestate.edu/gas/pirates/html/`

[GoSmAtSo04] Dina Q. Goldin, Scott A. Smolka, Paul C. Attie, and Elaine L. Sonderegger, "Turing machines, transition systems, and interaction" *Information*

and Computation 194 no. 2 (1 November 2004), pp. 101–128. Available (verified April 2010) at URL:
`http://www.cs.aub.edu.lb/pa07/files/pubs.html`

[Gra93] Paul Graham, *On Lisp*, Practice Hall, 1993. Available (verified April 2010) at URL:
`http://www.paulgraham.com/onlisp.html`

[Gri90] Timothy G. Griffin, "A Formulae-as-Types Notion of Control", *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, United States, 1990, pp. 47–58.

[Gua78] Loretta Rose Guarino, "The Evolution of Abstraction in Programming Languages, *CMU-CS-78-120*, Department of Computer Science, Carnegie-Mellon University, Pennsylvania, 22 May 1978.

[Ha63] Timothy P. Hart, *MACRO Definitions for LISP*, memo 57, MIT AI Lab, October 1963. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

[Hof79] Douglas R. Hofstadter, *Gödel, Escher, Bach: An Eternal Golden Braid*, New York: Vintage Books, 1979.

[Hu00] John Hughes, "Generalizing monads to arrows", *Science of Computer Programming* 37 nos. 1–2 (May 2000), pp. 67–111.

His overloading of the term "arrow" in a categorical setting is reprehensible. Instead of adjunctions (thus, monads) he's using Freyd categories.

[Ic71] Jean D. Ichbiah, "Extensibility in SIMULA 67", in [Sc71], pp. 84–86.

[KeClRe98] Richard Kelsey, William Clinger, and Jonathan Rees, editors, "Revised^5 Report on the Algorithmic Language Scheme", 20 February 1998. Available (verified April 2010) at URL:
`http://groups.csail.mit.edu/mac/projects/scheme/index.html`

[KeRi78] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Englewood Cliffs: Prentice-Hall, 1978.

The Old Testament. (See [Ra03, "Old Testament"].)

[Kle34] S.C. Kleene, "Proof by Cases in Formal Logic", *Annals of Mathematics* (2), 35 no. 3 (July 1934), pp. 529–544.

Incidental to the main point of the paper, which is to show that proof by cases is possible under the axioms of [Chu32/33], adjusts the definition of substitution to avoid variable capture.

[Kle52] S.C. Kleene, *Introduction to Metamathematics*, Princeton, N.J.: Van Nostrand, 1952.

This excellent book is (as of January 2003) still in print, by North-Holland, in its thirteenth impression. Corrections were made through the seventh impression in 1971.

[KleRo35] S.C. Kleene and J.B. Rosser, "The Inconsistency of Certain Formal Logics", *Annals of Mathematics* (2), 36 no. 3 (July 1935), pp. 630–636.

[Kli72] Morris Kline, *Mathematical Thought from Ancient Through Modern Times*, New York: Oxford University Press, 1972.

[Kl80] Jan Willem Klop, *Combinatory Reduction Systems*, Ph.D. Thesis, University of Utrecht, 1980. Also, Mathematical Centre Tracts 127. Available (verified December 2009) at URL:
`http://web.mac.com/janwillemklop/Site/Bibliography.html`

[KoFrFeDu86] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba, "Hygienic macro expansion", *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986, pp. 151–159.

[Kr01] Shriram Krishnamurthi, "Linguistic Reuse", Ph.D. Dissertation, Rice University, May 2001. Available (verified April 2010) at URL:
`http://www.cs.rice.edu/CS/PLT/Publications/Scheme/#diss`

[La64] P.J. Landin, "The mechanical evaluation of expressions", *Computer Journal* 6 no. 4 (January 1964), pp. 308–320.

[La00] P.J. Landin, "My Years with Strachey", *Higher-Order and Symbolic Computation* 13 no. 1/2 (April 2000), pp. 75–76.

[Li93] C.H. Lindsey, "A History of ALGOL 68", *SIGPLAN Notices* 28 no. 3 (March 1993) [Preprints, *ACM SIGPLAN Second History of Programming Languages Conference*, Cambridge, Massachusetts, April 20–23, 1993], pp. 97–132.

[Lo1690] John Locke, *An Essay Concerning Human Understanding*, 1690. Available (verified April 2010) at URL:
`http://humanum.arts.cuhk.edu.hk/Philosophy/Locke/echu/`

[McC60] John McCarthy, "Recursive Functions of Symbolic Expressions and Their Computation by Machine", *Communications of the ACM* 3 no. 4 (April 1960), pp. 184–195.

This is the original reference for Lisp.

[McC78] John McCarthy, "History of LISP", *SIGPLAN Notices* 13 no. 8 (August 1978) [Preprints, *ACM SIGPLAN History of Programming Languages Conference*, Los Angeles, California, June 1–3, 1978], pp. 217–223.

Covers development of the basic ideas, through 1962.

Erratum: The second and third sections are both numbered "2".

[McC+60] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park, and S. Russell, *Lisp I Programmer's Manual*, Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1, 1960. Available (verified April 2010) at URL:
`http://www.softwarepreservation.org/projects/LISP/`

[McC+62] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin, *LISP 1.5 Programmer's Manual*, Cambridge, Massachusetts: The MIT Press, 1962. Available (verified April 2010) at URL:
`http://www.softwarepreservation.org/projects/LISP/`

The second edition was 1965, same authors.

[McI60] M. Douglas McIlroy, "Macro Instruction Extensions of Compiler languages", *Communications of the ACM* 3 no. 4 (April 1960), pp. 214–220.

[McI69] M. Douglas McIlroy, "Alternatives to Extensible Languages" (followed by group discussion), in [ChrSh69], pp. 50–52.

The symposium had two papers back to back, roughly *pro* and *con* extensibility. The *pro* position was taken by T.E. Cheatham, [Che69]; while *con* was taken in this paper (by McIlroy, who was as close to a founding father as the extensible-languages movement had). The mood of the paper isn't exactly negative, though; he actually agrees with Cheatham on a couple of big points, one being that "the effort in extensible languages *is* going to lay bare the fundamentals of programming and, therefore, from an academic standpoint, the effort is highly justified."

[Men69] Karl Menninger, *Number Words and Number Symbols*, translated by Paul Broneer from the revised German edition, Cambridge, Massachusetts: The MIT Press, 1969.

[Mer90] N. David Mermin, "What's wrong with these equations?", in: *Boojums All The Way Through: Communicating Science in a Prosaic Age*, Cambridge University Press, 1990, pp. 68–73.

Eloquently promotes three rules for treatment of displayed equations:

1. Number all displayed equations (so that anyone who later reads the document can refer to them).
2. Refer to a displayed equation by a descriptive phrase as well as by number (so that the reader can make sense of a referring sentence without having to first decode its references).
3. Treat each displayed equation as part of the prose in which it occurs, ending the equation with a punctuation mark as appropriate (so the reader passes smoothly through the equation as part of the text, rather than stepping discontinuously into and then out of it as they would if it were a table or figure separated from the text).

[Mi93] John C. Mitchell, "On Abstraction and the Expressive Power of Programming Languages", *Science of Computer Programming* 212 (1993) [Special issue of papers from Symposium on Theoretical Aspects of Computer Software, Sendai, Japan, September 24–27, 1991], pp. 141–163. Also, a version of the paper appeared in: Takayasu Ito and Albert R. Meyer, editors, *Theoretical Aspects of Computer Science: International Conference TACS'91* [Sendai, Japan, September 24–27, 1991] [*Lecture Notes in Computer Science* 526], Springer-Verlag, 1991, pp. 290–310. Available (verified April 2010) at URL:
`http://theory.stanford.edu/people/jcm/publications.htm#typesandsemantics`

[MitGnuScheme] MIT Scheme 9.0.1 Reference. Available (verified April 2010) at URL:
`http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/index.html`

[Mose70] Joel Moses, *The Function of FUNCTION in LISP; or, Why the FUNARG Problem Should be Called the Environment Problem*, memo 199, MIT AI Lab, June 1970. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

[Moss00] Peter D. Mosses, "A Foreword to 'Fundamental Concepts in Programming Languages'", *Higher-Order and Symbolic Computation* 13 no. 1/2 (April 2000), pp. 7–9.

[Mu91] Robert Muller, "M-LISP: Its Natural Semantics and Equational Logic", in *Proceedings of the ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics Based Program Manipulation*, June 1991, pp. 234–242. Available (verified April 2010) at URL:
`http://www.cs.bc.edu/~muller/research/papers.html#pepm91`

Superseded by [Mu92], but the later paper doesn't specifically address fexprs (and, incidentally, beware the bibliography of the later paper, as it's riddled with errors).

[Mu92] Robert Muller, "M-LISP: A Representation-Independent Dialect of LISP with Reduction Semantics", *ACM Transactions on Programming Languages and Systems* 14 no. 4 (October 1992), pp. 589–616. Available (verified April 2010) at URL:
`http://www.cs.bc.edu/~muller/research/papers.html#toplas`

Caveat: the bibliography of this paper is riddled with errors.

A revised and expanded version of, and supersedes, [Mu91]; however, the earlier paper specifically discusses fexprs, which this paper does not.

[Pi80] Kent M. Pitman, "Special Forms in Lisp", *Proceedings of the 1980 ACM Conference on Lisp and Functional Programming*, 1980, pp. 179–187. Available (verified January 2010) at URL:
`http://www.nhplace.com/kent/Papers/Special-Forms.html`

This is one of those papers that gets cited a lot, by papers within its particular clique, because it carefully and clearly develops some basic conclusions that everyone later wants to take as given. In a nutshell: fexprs are badly behaved (second opinion: they're ugly, too), so future Lisps should use macros instead.

[Pi83] Kent M. Pitman, *The revised MacLisp Manual* (Saturday evening edition), MIT Laboratory for Computer Science Technical Report 295, May 21, 1983.

[Pla86] P.J. Plauger, "Which tool is best?", *Computer Language* 3 no. 7 (July 1986), pp. 15–17, 19.

The premier of Plauger's regular column, "Programming on Purpose".

[Plo75] Gordon D. Plotkin, "Call-by-name, call-by-value, and the λ-calculus", *Theoretical Computer Science* 1 no. 2 (December 1975), pp. 125–159. Available (verified April 2010) at URL:
`http://homepages.inf.ed.ac.uk/gdp/publications/`

[Plo81] Gordon D. Plotkin, "A structural approach to operational semantics", Technical Report DAIMI FN-19, Aarhus University, 1981. A transcribed version is available (verified April 2010) at URL:
`http://homepages.inf.ed.ac.uk/gdp/publications/`

[Qu61] W.V. Quine, *from a logical point of view*, Second Edition, revised, New York: Harper & Row, 1961.

A collection of nine "logico-philosophical essays".

[Ra03] Eric S. Raymond, *The Jargon File*, version 4.4.7, 29 December 2003. Available (verified April 2010) at URL:
`http://www.catb.org/~esr/jargon/`

[ReCl86] Jonathan Rees and William Clinger, editors, "The Revised[3] Report on the Algorithmic Language Scheme", *SIGPLAN Notices* 21 no. 12 (December 1986), pp. 37–43. Available (verified April 2010) at URL:
`http://www.cs.indiana.edu/scheme-repository/doc.standards.html`

The second of the *RxRS*s authored by a committee. Introduces a high-level statement on language-design principles in the *Introduction*, which has been passed on to all the *RxRS*s since.

[Rey93] John C. Reynolds, "The discoveries of continuations", *Lisp and Symbolic Computation* 6 nos. 3/4 (1993), pp. 233–247. Available (verified April 2010) at URL:
`ftp://ftp.cs.cmu.edu/user/jcr/histcont.pdf`

[Rey72] John C. Reynolds, "Definitional Interpreters for Higher-Order Programming Languages", *Proceedings of the ACM Annual Conference*, 1972, (vol. 1) pp. 717–740. Reprinted in *Higher-Order and Symbolic Computation* 11 no. 4 (December 1998), pp. 363–397.

Unifies several previous attempts to define language semantics under the name "meta-circular evaluator".

He describes static scoping, not by that name, and says —forcefully— that everyone agrees that well-designed languages must work that way.

[Rog87] Hartley Rogers, Jr., *Theory of Recursive Functions and Effective Computability*, Cambridge, Massachusetts: The MIT Press, 1987.

[Ros82] J. Barkley Rosser, "Highlights of the history of the lambda-calculus", *Proceedings of the 1982 ACM Conference on Lisp and Functional Programming*, 1982, pp. 216–225.

[Sa69] Jean E. Sammet, *Programming Languages: History and Fundamentals*, Englewood Cliffs: Prentice-Hall, 1969.

[Sc71] Stephen A. Schuman, *Proceedings of the International Symposium on Extensible Languages*, Grenoble, France, September 6–8, 1971 [*SIGPLAN Notices* 6 no. 12 (December 1971)].

[Share58] J. Strong, J. Olsztyn, J. Wegstein, O. Mock, A. Tritter, and T. Steel, "The Problem of Programming Communication With Changing Machine: A Proposed Solution" (Report of the Share Ad-Hoc Committee on Universal Languages), *Communications of the ACM* 1 no. 8 (August 1958), pp. 12–18.

The first of two parts.

[ShiWa05] Olin Shivers and Mitchell Wand, "Bottom-up β-reduction: uplinks and λ-DAGs", in Moogly Sagiv, editor, *Programming Languages and Systems: 14th European Symposium on Programming, ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4–8, 2005, Proceedings* [*Lecture Notes in Computer Science 3444*], Springer-Verlag, 2005. Available (verified April 2010) at URL:
`http://www.ccs.neu.edu/home/wand/pubs.html#ShiversWand:ESOP-05`
Expanded version: BRICS Technical Report RS-04-38, Department of Computer Science, University of Århus. Available (verified April 2010) at URL:
`http://www.brics.dk/RS/04/38/index.html`

[Sho95] R.J. Shoenfield, "The mathematical work of S.C. Kleene", *The Bulletin of Symbolic Logic* 1 no. 1 (March 1995), pp. 9–43.

[Shu03a] John N. Shutt, "Monads for programming languages", technical report WPI-CS-TR-03-21, Worcester Polytechnic Institute, Worcester, MA, June 2003. Available (verified April 2010) at URL:
`http://www.cs.wpi.edu/Resources/techreports.html`

[Shu03b] John N. Shutt, "Recursive Adaptable Grammars", M.S. Thesis, WPI CS Department, 10 August 1993, emended 16 December 2003. Available (verified April 2010) at URL:
`ftp://ftp.cs.wpi.edu/pub/projects_and_papers/theory/`

[Shu08] John N. Shutt, "Abstractive Power of Programming Languages: Formal Definition", technical report WPI-CS-TR-08-01, Worcester Polytechnic Institute, Worcester, MA, March 2008, emended 26 March, 2008. Available (verified April 2010) at URL:
`http://www.cs.wpi.edu/Resources/techreports.html`

[Shu09] John N. Shutt, "Revised^{-1} Report on the Kernel Programming Language", technical report WPI-CS-TR-05-07, Worcester Polytechnic Institute, Worcester, MA, March 2005, amended 29 October 2009. Available (verified June 2010) at URL:
`http://www.cs.wpi.edu/Resources/techreports.html`

[Shu10] John N. Shutt, *Fexprs as the basis of Lisp function application; or,* **`$vau`** *: the ultimate abstraction*, Ph.D. Dissertation, WPI CS Department, 2010.

[Sm84] Brian Cantwell Smith, "Reflection and Semantics in Lisp", *POPL '84: Conference Record of the ACM Conference on Principles of Programming Languages*, Salt Lake City, Utah, January 15–18, 1984, pp. 23–35.

3-LISP lambda expressions have a keyword `simple` or `reflect`. Reflective procedures take three parameters: operands, dynamic environment, and continuation. This paper is short on technical explanation.

[Sp+07] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten, "Revised^6 Report on the Algorithmic Language Scheme", 26 September 2007. Available (verified April 2010) at URL:
`http://www.r6rs.org/`

[Sta75] Thomas A. Standish, "Extensibility in Programming Language Design", *SIGPLAN Notices* 10 no. 7 (July 1975) [*Special Issue on Programming Language Design*], pp. 18–21.

A retrospective survey of the subject, somewhat in the nature of a postmortem. The essence of Standish's diagnosis is that the extensibility features required an expert to use them. He notes that when a system is complex, modifying it is complex. (He doesn't take the next step, though, of suggesting that some means should be sought to reduce the complexity of extended systems.)

He classifies extensibility into three types: *paraphrase* (defining a new feature by showing how to express it with pre-existing features — includes ordinary procedures as well as macros); *orthophrase* (adding new facilities that are orthogonal to what was there — think of adding a file system to a language that didn't have one); and *metaphrase* (roughly what would later be called "reflection").

[Ste76] Guy Lewis Steele Jr., *LAMBDA: The Ultimate Declarative*, memo 379, MIT AI Lab, November 1976. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

[Ste78] Guy Lewis Steele, *RABBIT: A Compiler for SCHEME*, memo 474, MIT AI Lab, May 1978. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

[Ste90] Guy Lewis Steele Jr., *Common Lisp: The Language*, 2nd Edition, Digital Press, May 1990. Available (verified April 2010) at URL:
`http://www.supelec.fr/docs/cltl/cltl2.html`

[SteGa93] Guy L. Steele Jr. and Richard P. Gabriel, "The Evolution of Lisp", *SIGPLAN Notices* 28 no. 3 (March 1993) [Preprints, *ACM SIGPLAN Second History of Programming Languages Conference*, Cambridge, Massachusetts, April 20–23, 1993], pp. 231–270.

[SteSu76] Guy Lewis Steele Jr. and Gerald Jay Sussman, *LAMBDA: The Ultimate Imperative*, memo 353, MIT AI Lab, March 10, 1976. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

[SteSu78a] Guy Lewis Steele Jr. and Gerald Jay Sussman, *The Revised Report on SCHEME: A Dialect of LISP*, memo 452, MIT Artificial Intelligence Laboratory, January 1978. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

The last 12 of 34 pages are endnotes. Contrast with the original Scheme report ([SuSt75]): `EVALUATE` has been removed, replaced by `ENCLOSE` that takes an expression and a *representation* of an environment. There are macros. The multiprocessing is still there, but the synchronization primitive has been removed and they say (p. 24) it was a mistake because it synchronized lexically instead of dynamically. There are also fluid bindings.

[SteSu78b] Guy Lewis Steele Jr. and Gerald Jay Sussman, *The Art of the Interpreter; or, The Modularity Complex*, memo 453, MIT Artificial Intelligence Laboratory, May 1978. Available (verified April 2010) at URL:
`http://publications.csail.mit.edu/ai/`

Incrementally evolves a meta-circular evaluator starting from the most primitive LISP, analyzing difficulties at each state and augmenting/modifying accordingly. Focus is on support for incremental development of modular systems.

See [SteGa93, §2.8].

[Stra67] Christopher Strachey, "Fundamental Concepts in Programming", Lecture notes for International Summer School on Computer Programming, Copenhagen, August 7 to 25, 1967.

These notes have been cited commonly when crediting Strachey with coining the term *polymorphism* as it applies to programming languages. The discussion of polymorphism is on page 10.

Strachey later turned these notes into a paper, which remained unpublished until it appeared as [Stra00].

[Stra00] Christopher Strachey, "Fundamental Concepts in Programming Languages", *Higher-Order and Symbolic Computation* 13 no. 1/2 (April 2000), pp. 11–49.

This is Strachey's paper based on his lectures, [Stra67]. On the history of the paper, see [Moss00].

[SuSt75] Gerald Jay Sussman and Guy Lewis Steele Jr., *Scheme: An Interpreter for Extended Lambda Calculus*, memo 349, MIT Artificial Intelligence Laboratory, December 1975.

The revised^0 report on Scheme. The actual description of the language is only through page 5.

[Ta99] Walid Taha, "Multi-Stage Programming: Its Theory and Applications", Ph.D. Dissertation, Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999. Available (verified April 2010) at URL:
`http://www.cs.rice.edu/~taha/publications.html`

[To88] M. Tofte, "Operational semantics and polymorphic type inference", Ph.D. Thesis, University of Edinburgh, 1988. Available (verified April 2010) at URL:
`http://www.itu.dk/people/tofte/publ/`

Big step operational semantics. (Contrast [WrFe94].)

[Tu37] Alan Turing, "Computability and λ-Definability", *Journal of Symbolic Logic* 2 no. 4 (December 1937), pp. 153–163.

[vW65] Adriann van Wijngaarden, "Orthogonal design and description of a formal language", Technical Report MR 76, Mathematisch Centrum, Amsterdam, 1965. Available (verified April 2010) at URL:
`http://www.fh-jena.de/~kleine/history/history.html`

This paper is the origin of the term *orthogonal* in programming-language design (although that term occurs only in the title), and of W-grammars. It was an early element of the Algol X (later Algol 68) development process; see [Li93].

[Wa98] Mitchell Wand, "The Theory of Fexprs is Trivial", *Lisp and Symbolic Computation* 10 no. 3 (May 1998), pp. 189–199. Available (verified April 2010) at URL:
`http://www.ccs.neu.edu/home/wand/pubs.html#Wand98`

This paper is a useful elaboration of the basic difficulty caused by mixing fexprs with implicit evaluation in an equational theory. Along the way, the author makes various sound observations. "We care," he notes in the concluding section, "not only when two terms have the same behavior, but we may also care just what that behavior is."

[WaFr86] Mitchell Wand and Daniel P. Friedman, "The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower", *Lisp and Symbolic Computation* 1 no. 1 (1988), pp. 11–37.

Sequel to [FrWa84], adds a reflective tower to Brown.

[Wegb80] Ben Wegbreit, *Studies in Extensible Programming Languages* [*Outstanding Dissertations in the Computer Sciences*], New York: Garland Publishing, Inc., 1980.

A reprint of *ESD-TR-70-297*, Harvard University, May 1970.

Shows that (1) ECFL membership is Turing decidable, and (2) a large subset of ECFLs are context-sensitive; but whether all ECFLs are context-sensitive is left an open question.

[Wegn69] Peter Wegner, Chair, "Panel on the Concept of Extensibility", in [ChrSh69], pp. 53–54.

[WrFe94] Andrew K. Wright and Matthias Felleisen, "A Syntactic Approach to Type Soundness", *Information and Computation* 155 no. 1 (November 1994), pp. 38–94. Available (verified April 2010) at URL:
`http://www.ccs.neu.edu/scheme/pubs/#ic94-wf`

Small step operational semantics. (Contrast [To88].)

[Za03] Richard Zach, "Hilbert's Program", 2003, in *Stanford Encyclopedia of Philosophy*, Metaphysics Research Lab, Stanford University. Available (verified April 2010) at URL:
`http://plato.stanford.edu/entries/hilbert-program/`