

RICE UNIVERSITY

# Reasoning About Multi-stage Programs

by

Jun Inoue

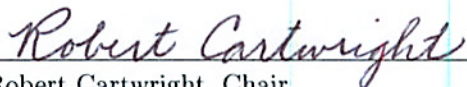
A THESIS SUBMITTED  
IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

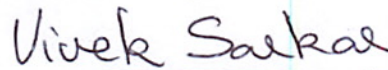
APPROVED, THESIS COMMITTEE:



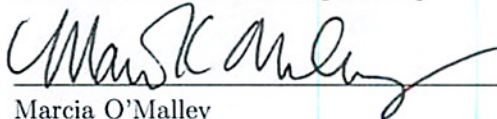
Walid Taha, Director  
Professor of Computer Science  
Halmstad University



Robert Cartwright, Chair  
Professor of Computer Science



Vivek Sarkar  
Professor of Computer Science and  
E.D. Butcher Chair in Engineering



Marcia O'Malley  
Associate Professor of Mechanical  
Engineering

Houston, Texas

April, 2012

## ABSTRACT

### Reasoning About Multi-stage Programs

by

Jun Inoue

Multi-stage programming (MSP) is a style of writing program generators—programs which generate programs—supported by special annotations that direct construction, combination, and execution of object programs. Various researchers have shown MSP to be effective in writing efficient programs without sacrificing genericity. However, correctness proofs of such programs have so far received limited attention, and approaches and challenges for that task have been largely unexplored. In this thesis, I establish formal equational properties of the multi-stage lambda calculus and related proof techniques, as well as results that delineate the intricacies of multi-stage languages that one must be aware of.

In particular, I settle three basic questions that naturally arise when verifying multi-stage functional programs. Firstly, can adding staging MSP to a language compromise the interchangeability of terms that held in the original language? Unfortunately it can, and more care is needed to reason about terms with free variables. Secondly, staging annotations, as the term “annotations” suggests, are often thought to be orthogonal to the behavior of a program, but when is this formally guaranteed to be the case? I give termination conditions that characterize when this guarantee holds. Finally, do multi-stage languages satisfy extensional facts, for example that functions agreeing on all arguments are equivalent? I develop a sound and complete

notion of applicative bisimulation, which can establish not only extensionality but, in principle, any other valid program equivalence as well. These results improve our general understanding of staging and enable us to prove the correctness of complicated multi-stage programs.

## Acknowledgments

I would like to thank my advisor Walid Taha for introducing me to this thesis topic, for his support throughout the work, for the numerous opportunities he gave me to meet respected people in the field, and most of all for his enthusiasm for my work. He has always taken as much pride in my work as I have. I thank my thesis committee members, Robert Cartwright, Vivek Sarkar, and Marcia O'Malley. I thank Robert Cartwright for his advice on specific matters relating to this thesis. I thank Vivek Sarkar for his time and his support for this work. I thank Edwin Westbrook for his direct contributions to this work. I thank Gregory Malecha for the hints he has left me with, which turned out to be essential for this work. I thank Mathias Ricken for his feedback that sharpened some of the results. I thank Ronald Garcia for their valuable comments on the papers that I have written on precursors for this thesis. I thank Yun “Angela” Zhu for her kind support for this work, and relaxing conversations that helped me move on.

Parts of this thesis was completed during my stay at Halmstad University, Sweden, where I was greeted with such hospitality. I thank Bertil Svensson and Eva Nestius for their assistance in vagary of concerns during my stay. Inputs from Paul Brauner, Bertil Svensson, Tony Larsson, and Veronica Gaspes have helped me improve the presentation of this material continually. This thesis would not have materialized without the help of Jan and Adam Duracz. I am humbled by the sheer amount of help I have received over the years from so many different people.

This research was supported in part by NSF grants CCF-0747431 and EHS-0720857. Completing this work would have not been possible without the support of Dr. Helen Gill of the National Science Foundation.

# Contents

Abstract	ii
Acknowledgments	iv
List of Figures	viii
List of Tables	ix
<b>1 Introduction</b>	<b>1</b>
1.1 Contributions . . . . .	4
<b>2 Multi-stage Programming</b>	<b>7</b>
2.1 Staging Annotations . . . . .	7
2.2 Example: Power . . . . .	9
<b>3 The <math>\lambda^U</math> Calculus: Syntax, Semantics, and Equational Theory</b>	<b>12</b>
3.1 Syntax and Operational Semantics . . . . .	13
3.2 Equational Theory . . . . .	18
3.3 Further Generalization of Axioms is Unsound . . . . .	25
3.4 Closing Substitutions Compromise Validity . . . . .	30
<b>4 Effects of Staging on Correctness</b>	<b>33</b>
4.1 Theorem Statement . . . . .	34
4.2 Example: Erasing Staged Power . . . . .	37
4.3 Why CBN Facts are Necessary for CBV Reasoning . . . . .	40

<b>5</b>	<b>Extensional Reasoning for <math>\lambda^V</math></b>	<b>45</b>
5.1	Proof by Bisimulation . . . . .	45
5.2	Example: Tying Loose Ends on Staged Power . . . . .	48
5.3	Soundness and Completeness of Applicative Bisimulation . . . . .	50
5.3.1	Overview . . . . .	50
5.3.2	The Proof . . . . .	52
<b>6</b>	<b>Case Study: Longest Common Subsequence</b>	<b>1</b>
6.1	The Code . . . . .	1
6.2	Purpose of Monadic Translation . . . . .	2
6.3	Correctness Proof . . . . .	5
<b>7</b>	<b>Related Works</b>	<b>18</b>
<b>8</b>	<b>Conclusion and Future Work</b>	<b>20</b>
	<b>Bibliography</b>	<b>22</b>
<b>A</b>	<b>OCaml</b>	<b>25</b>
<b>B</b>	<b>Coinduction</b>	<b>28</b>
<b>C</b>	<b>Proof Details</b>	<b>31</b>
C.1	Substitution . . . . .	31
C.2	Proofs for Operational Semantics . . . . .	32
C.2.1	Evaluation Contexts Compose . . . . .	32
C.2.2	Determinism, Irreducibility of Values, and Focus . . . . .	32
C.2.3	Equivalence of Open- and Closed-Term Observation . . . . .	34
C.3	Proofs for Equational Theory . . . . .	38
C.3.1	Confluence . . . . .	38

C.3.2 Standardization . . . . .	43
C.4 Proofs for Generalized Axioms . . . . .	53
C.5 Proofs for Extensionality . . . . .	54
C.6 Proofs for Soundness and Completeness of Applicative Bisimulation .	56
<b>D Summary of Notations</b>	<b>66</b>

# Figures

3.1	Syntax of $\lambda^U$ , parametrized in a set of constants $Const$ . . . . .	13
3.2	Operational semantics of $\lambda^U$ , parametrized in an interpretation (partial) map $\delta : Const \times Const \rightarrow \{v \in V_{cl}^0 : v \equiv \ v\ \}$ . . . . .	14
3.3	Parallel reduction. . . . .	21
3.4	Complete development. . . . .	22
4.1	Visualizations of the Erasure Theorem and the derived correctness lemma. . . . .	36
6.1	Unstaged longest common subsequence with helper functions. . . . .	16
6.2	Staged, memoized longest common subsequence. . . . .	17



Tables

3.1    Unsound generalizations of our axioms.  $\Omega$  is some divergent level-0  
term. The generalizations of  $\beta$  suppose CBN, generalizations of  $\beta_{\mathbf{v}}$   
suppose CBV, and the generalization of  $R_U$  is meant for both.    . . .    27

D.1    Summary of notations. . . . . 66

# Chapter 1

## Introduction

Generic programming style is a key to improving programmer productivity. Modern programming languages are often designed to help write generic code, but the relevant language features come with performance costs. For example, most languages can express a matrix multiplication function that works for matrices of integers as well as matrices of floating point numbers. This function need not be rewritten for every desired matrix element type, thus reducing development time if several different matrix types must be handled. But as a downside, the generic function is usually much slower than a specialized implementation that handles, say, only matrices of floating points. Thus, getting both genericity and good performance is difficult.

Multi-stage programming (MSP) solves this dilemma by allowing programmers to write code generators—programs which generate programs—which are themselves generic but produce specialized, efficient code. In the foregoing example, a multi-stage programmer can write a generator that takes a function for adding matrix elements (say, integer addition) and another for multiplying, and then returns specialized matrix multiplication code that has those functions inlined. The resulting code will perform as if the matrix element operations had been hard-coded.

Thus MSP delivers genericity and performance at the same time, which has been demonstrated in numerous studies [8, 6, 17, 11, 9]. However, few formal studies have considered how to prove, whether informally or rigorously, that generators written with MSP produce code that computes what it is supposed to compute. Instead,

much of the theoretical investigation has been on type systems [33, 21, 38, 20, 35, 36], which are primarily for ensuring basic safety properties, such as ensuring that the generated code does not crash.

This situation is somewhat of a paradox, because a key assumption behind the use of MSP is that it enhances performance while preserving the structure of the code, and that it therefore does not interfere much with reasoning [22, 6]. The power function is a good example of MSP preserving structure, presented here in MetaOCaml syntax. This function will be used as a running example in this thesis, for it is much simpler than the matrix multiplication example but still shows the essence.

```
let rec power  n x = if n = 1 then x else x * power (n-1) x
let rec genpow n x = if n = 1 then x else .<~x * ~(genpow (n-1) x)>.
let stpow n = .!.<fun z → ~(genpow n .<z>.)>.
```

The `power` function takes integers `n` and `x`, and returns `x` raised to the power of `n`, i.e.  $x^n$ . The `stpov` function is a staged replacement for `power` that takes `n` and produces a version of `power` specialized for that `n`. The `genpow` is a helper function that generates the body of the function that `stpov` produces.

To elaborate, the `power` function subsumes all functions that have the form `fun x → x*x*...*x` but incurs recursive calls each time it is called. There are three constructs called *staging annotations* used in `genpow` to eliminate this overhead by unrolling the recursion. Brackets `.<e>.` delay an expression `e`. An escape `~e` must occur within brackets and causes `e` to be evaluated without delay. The `e` should return a code value `.<e'>.`, and `e'` replaces `~e`. For example if `n = 2`, the `genpow n .<z>.` in `stpov` returns a delayed multiplication `.<z*z>..` This is an open term, containing the unbound (or undefined) variable `z`, but MetaOCaml allows manipulation of open terms while executing escaped expressions. Run `!.e` compiles and

runs the code generated by  $e$ , so `stpow 2` evaluates to the function `fun z → z*z`, which has no recursion. These annotations in MetaOCaml are hygienic (i.e., preserve static scoping [12]), but are otherwise like LISP’s `quasiquote`, `unquote`, and `eval` [25]. (See chapter 2 for a step-by-step introduction to staging.)

This example is typical of MSP usage, where a staged program `stpow` is meant as a drop-in replacement for the unstaged program `power`. Note that if we are given only `stpow`, we can reconstruct the unstaged program `power` by erasing the staging annotations from `stpow`—we say that `power` is the *erasure* of `stpow`. Given the similarity of these programs, if we are to verify `stpow`, we naturally expect `stpow`  $\approx$  `power` to hold for a suitable notion of program equivalence ( $\approx$ ) and hope to get away with proving that `power` satisfies whatever specifications it has, in lieu of `stpow`. We expect `power` to be easier to tackle, since it has no staging annotations and should therefore be amenable to conventional reasoning techniques designed for single-stage programs. But three key questions must be addressed before we can apply this strategy confidently:

*Conservativity.* Do all reasoning principles valid in a single-stage language carry over to its multi-stage extension?

*Conditions for Sound Erasure.* In the `power` example, staging seems to preserve semantics, but clearly this is not always the case: if  $\Omega$  is non-terminating, then  $\langle \Omega \rangle \not\approx \Omega$  for any sensible ( $\approx$ ). When do we know that erasing annotations preserves semantics?

*Extensional Reasoning.* How, in general, do we prove equivalences of the form  $e \approx t$ ? It is known that hygienic, purely functional MSP satisfies intensional equalities like  $\beta$  [32], but those equalities are too weak to prove such properties as extensionality (i.e.,

functions agreeing on all inputs are equivalent). Extensional facts are indispensable for reasoning about functions, like `stop` and `power`.

This thesis settles these questions, focusing on the untyped, purely functional case with hygiene. Types are not included, so as to avoid committing to the particulars of any specific type system, since there are multiple useful type systems for MSP [33, 35, 36]. This also ensures that the results apply to dynamically typed languages in which MSP is as interesting as in statically typed languages [12]. Hygiene is a widely accepted safety feature, and it ensures many of the nice theoretical properties of MSP, which makes it easy to reason about programs, and which we exploit in this study. Imperative MSP does not yet appear to be ready for an investigation like this. Types are essential for having a sane operational semantics without scope extrusion [20], but there is no decisive solution to this problem, and the jury is still out on the trade-offs involved in their designs.

## 1.1 Contributions

This thesis extends previous work on the call-by-name (CBN) multi-stage  $\lambda$  calculus,  $\lambda^U$  [32], to cover call-by-value (CBV) as well (chapter 3). In this calculus, we show the following results.

*Unsoundness of Reasoning Under Substitutions.* Unfortunately, the answer to the conservativity question is “no.” When one compares terms with free variables, it is customary to compare their values under various substitutions—e.g. we know  $0 * x$  is equivalent to (i.e. interchangeable with)  $0 * (x + 1)$  because substituting any value for  $x$  yields 0 on both sides (or type error, if we substitute a non-number for  $x$ ). However, because  $\lambda^U$  can express open term manipulation as seen in `genpow`

above, equivalences proved under closing substitutions are not always valid without substitution, for such a proof implicitly assumes that only closed terms are interesting. We illustrate clearly how this pathology occurs using the surprising fact  $(\lambda_.0) x \not\approx 0$ , and explain what can be done about it (Chapter 3.4). The rest of the paper will show that a lot can be achieved despite this drawback.

*Conditions for Sound Erasure.* We show that reductions of a staged term are simulated by equational rewrites of the term’s erasure. This gives simple termination conditions that guarantee erasure to be semantics-preserving (chapter 4). Considering CBV in isolation turns out to be unsatisfactory, and borrowing CBN facts is essential in establishing the termination conditions for CBV. Intuitively, this happens because annotations change the evaluation strategy, and the CBN equational theory subsumes reductions in all other strategies whereas the CBV theory does not.

*Soundness of Extensional Properties.* We give a sound and complete notion of applicative bisimulation [1, 15] for  $\lambda^U$ . Bisimulation gives a general extensional proof principle that, in particular, proves extensionality of  $\lambda$  abstractions. It also justifies reasoning under substitutions in some cases, limiting the impact of the non-conservativity result (chapter 5).

This thesis emphasizes general results about MSP, and the insights about semantics that can be gleaned from them. The ability to verify staged programs fall out from general principles, which will be demonstrated using the power function as a running example. As a substantial case study, chapter 6 presents a correctness proof of the longest common subsequence (LCS) algorithm. This is a sophisticated code generator that uses **let**-insertion coupled with continuation-passing style (CPS) and monadic memoization [31]. These features make an exact description of the generated

code hard to pin down, but our result on erasure makes such details irrelevant, and as the appendix shows, its proof is quite similar to that of the power example.

These results, in particular the correctness proof of LCS, demonstrate the core thesis of this study, namely: multi-stage programming makes it easier to achieve, simultaneously, not only genericity and performance, but also *correctness*.

# Chapter 2

## Multi-stage Programming

This chapter informally introduces multi-stage programming using the MSP language MetaOCaml [7]. This language extends the functional language OCaml with three annotation constructs, brackets, escape, and run, which direct how to split up a program’s execution into multiple stages. Their semantics is illustrated with minimal examples, and the behavior of the staged `power` function given in the introduction is followed in greater detail.

### 2.1 Staging Annotations

MetaOCaml is an extension of OCaml with three language constructs: brackets (`.<_>.`), escape (`.~`), and run (`.!`). MetaOCaml is impurely functional, but the focus of this thesis is the purely functional subset. Appendix A contains a brief summary of OCaml’s basic syntax and semantics that are required to navigate through the power example.

Brackets generate code. For example,

```
.<(fun x → x * x) 2>.
```

evaluates to a code value, in other words a parse tree, representing the expression `((fun x → x*x) 2)`, and not to the integer value 4. The type of this code value is `<int>`, read “code of `int`”.\* Just like most languages allow programmers to write

---

\*Actually, MetaOCaml assigns it the type `('a, int) code` where `'a` is an environment classifier



string literals by enclosing the string in double quotes (e.g. "Hello World"), and just like quasiquotation in LISP-like languages, `.<>.` gives a syntax for writing down code as a literal value.

Escapes allow parts of code values to be computed dynamically, similarly to string interpolation or `unquote` in LISP. For example:

```
let double x = .<~x * 2>. in
  double .<1 + 3>.
```

When the second line calls the `double` function, `.<1+3>.` gets substituted for `x` to produce the intermediate term `.<~(.<1+3>.) * 2>..` Then `.~` merges the code value `.<1+3>.` into the surrounding, which results in the code value `.<(1+3) * 2>.` being returned. The `x` inside `.~` can be any other expression that returns a code value of the right type. An `.~` can only appear (lexically) inside `.<_>..` The `.<_>.` and `.~` can be nested, but `.~` cannot be nested deeper than `.<_>..` The depth of `.<_>.` minus the depth of `.~` is called the *level*. Execution of escaped code is deferred like its surrounding if the nesting of `.<_>.` is strictly deeper (i.e. `level > 0`).

Finally, we have an analogue of LISP's `eval` function, which is run `(.!).` This construct takes a code value, compiles it into machine code (or byte code, depending upon the flavor of MetaOCaml being used), and immediately executes the machine (or byte) code. For example,

```
.! .<(fun x → x * x) 2>.
```

returns 4.

Informally, resolving escapes and generating a code value is called stage 0, while executing the generated code with `.!` is called stage 1. Stage 1 code may also create

---

[33]. Classifiers are orthogonal to the results presented here and are therefore omitted in all types.

and run another code value, which constitutes stage 2, but stages 2 and later are rarely useful.

Staging supports cross-stage persistence (CSP), which allows a value created at one stage to be used in any subsequent stage. For example, the following is a valid MetaOCaml program.

```
let two = 2 in

let double x = x * two in

  .<double 5>.
```

This program returns the code value `.<(fun x → x * 2) 5>.` which captures a user-defined value `double`. In the formal calculus, this capturing is expressed by substitution, as I just showed. In the implementation, the code value is represented as `.<□ 5>.` where `□` is a pointer to a function object.

MSP has some important features that distinguish it from most other metaprogramming systems. Unlike its LISP counterparts, staging annotations guarantee that the generated code is well-formed, whereas a LISP macro can generate nonsensical code fragments such as `(lambda 1)`. Staging also offers automatic hygiene like Scheme macros [12] but unlike traditional LISP macros. Staging also guarantees type safety for purely functional programs, including safety of the generated code [33].

## 2.2 Example: Power

Let us examine the power example from the introduction in more detail.

```
let rec power  n x = if n = 1 then x else x * power (n-1) x

let rec genpow n x = if n = 1 then x else .<~x * ~(genpow (n-1) x)>.

let stpow n = .!.<fun z → ~(genpow n .<z>.)>.
```

The `power` function is an (unstaged) implementation of exponentiation  $x^n$ . Adding staging annotations to `power` gives the code generator `genpow`. An invocation of the form

```
genpow n .<z>.
```

where  $n$  is some positive integer, produces a code value representing the computation that `power n z` performs, except with all the recursion unrolled and branching eliminated. For example:

- `genpow 1 .<z>.` just returns `.<z>..`
- `genpow 2 .<z>.` produces the intermediate term `.<~.<z>. * ~.<z>.>..` The left `.<z>.` comes from substituting the argument `.<z>.` for its placeholder `x` in the **else** branch of `genpow`, and the right `.<z>.` arose by executing the recursive call `genpow 1 .<z>..` The intermediate form evaluates to `.<z * z>..`, since escapes and brackets cancel out.
- `genpow 3 .<z>.` gives the intermediate term `.<~.<z>. * ~(.<z * z>.)>..` and eventually returns `.<z * (z * z)>..`

The `stpow` function is a wrapper to `genpow` that takes just the index  $n$  of the sequence element to compute and creates a function that maps  $x$  to  $x^n$ . The function call `stpow n` (for  $n \in \mathbb{Z}^+$ ) produces the intermediate term `!.<fun x → x*x*...*x>.` where  $n$  copies of  $x$  are multiplied. This intermediate form then evaluates to the (compiled) function `(fun x → x*x*...*x).` This generated function no longer wastes time resolving **if-then-else** or decrementing  $n$ ; it performs multiplication and nothing else, thereby being more efficient than `power`.

Staging thus enables programmers to write generic code without sacrificing performance. The staged function `stpow` is generic, in that it is able to handle all

positive-integer powers, but unlike `power`, the code that `stpow` generates does not pay a performance penalty every time it is called. Code generation costs are paid once and for all, and genericity no longer weighs us down thereafter.

The big uncertainty at this point is that, by adding staging annotations, we have modified the behavior of `power` in rather unconventional ways. How do we know that we have not broken the program in the course of applying staging? As a case in point, consider these declarations:

```
let rec loop () = loop () (* infinite loop *)
let f x = (let _ = loop () in 0)
let g x = (let _ = .<loop ()>. in 0)
```

Although `f` and `g` differ only by staging annotations—just like `stpow` and `power`—, they are not interchangeable: whereas `f` is nowhere defined, `g` is everywhere defined as 0.

The crux of this thesis is the identification of a bound on the impact that staging can have on a program’s semantics, with a rigorous theoretical underpinning. With the metatheory developed in this thesis, the mathematically inclined reader should be able to prove the correctness of not just `stpow` but the more complex, staged longest common subsequence algorithm (discussed in chapter 6). The metatheory will also benefit the more pragmatically inclined reader by giving general rules of thumb that ensure correctness while writing a multi-stage program or while reasoning about the behavior of such a program.

## Chapter 3

# The $\lambda^U$ Calculus: Syntax, Semantics, and Equational Theory

This section presents the multi-stage  $\lambda$  calculus  $\lambda^U$ . This is a simple but expressive formal language that models all possible uses of brackets, escape, and run in MetaOCaml’s purely functional core, sans types. In this calculus, MetaOCaml’s abstraction construct **fun**  $x \rightarrow e$  is modeled by  $\lambda x.e$ , local binding **let**  $x = e$  **in**  $t$  is modeled by function application  $(\lambda x.t) e$ , and recursive definition **let rec**  $f\ x = e$  is modeled by a fixed point combinator. The mapping of other constructs should be self-explanatory. The syntax and operational semantics of  $\lambda^U$  for both CBN and CBV are minor extensions of previous work [32] to allow arbitrary constants, which include first-order data like integers, booleans, and arrays thereof. The CBN equational theory is more or less as in [32], but the CBV equational theory is new.

**Notation.** A set  $S$  may be marked as CBV ( $S_v$ ) or CBN ( $S_n$ ) if its definition varies by evaluation strategy. The subscript is dropped in assertions and definitions that apply to both evaluation strategies. Syntactic equality ( $\alpha$  equivalence) is written ( $\equiv$ ). As usual,  $[e/x]t$  denotes the result of substituting  $e$  for  $x$  in an expression  $t$  in a capture-avoiding manner, while the set of free variables in  $e$  is written  $\text{FV}(e)$ . For a set  $S$ , we write  $S_{cl}$  to mean  $\{e \in S : \text{FV}(e) = \emptyset\}$ .

<i>Levels</i>	$\ell, m \in \mathbb{N}$	<i>Variables</i>	$x, y \in \text{Var}$	<i>Constants</i>	$c, d \in \text{Const}$
<i>Expressions</i>	$e, t, s \in E ::= c \mid x \mid \lambda x. e \mid e \ e \mid \langle e \rangle \mid \sim e \mid ! e$				
<i>Exact Level</i>	$\text{lv} : E \rightarrow \mathbb{N}$ where				
	$\text{lv } x \stackrel{\text{def}}{=} 0 \quad \text{lv } c \stackrel{\text{def}}{=} 0 \quad \text{lv}(e_1 \ e_2) \stackrel{\text{def}}{=} \max(\text{lv } e_1, \text{lv } e_2) \quad \text{lv}(\sim e) \stackrel{\text{def}}{=} \text{lv } e + 1$				
	$\text{lv}(\lambda x. e) \stackrel{\text{def}}{=} \text{lv } e \quad \text{lv}\langle e \rangle \stackrel{\text{def}}{=} \max(\text{lv } e - 1, 0) \quad \text{lv}(! e) \stackrel{\text{def}}{=} \text{lv } e$				
<i>Stratification</i>	$e^\ell, t^\ell, s^\ell \in E^\ell \stackrel{\text{def}}{=} \{e : \text{lv } e \leq \ell\}$				
<i>Values</i>	$u^0, v^0, w^0 \in V^0 ::= c \mid \lambda x. e^0 \mid \langle e^0 \rangle$ $u^{\ell+1}, v^{\ell+1}, w^{\ell+1} \in V^{\ell+1} ::= e^\ell$				
<i>Programs</i>	$p \in \text{Prog} \stackrel{\text{def}}{=} \{e^0 : \text{FV}(e^0) = \emptyset\}$				
<i>Contexts</i>	$C \in \text{Ctx} ::= \bullet \mid \lambda x. C \mid C \ e \mid e \ C \mid \langle C \rangle \mid \sim C \mid ! C$				

Figure 3.1 : Syntax of  $\lambda^U$ , parametrized in a set of constants  $\text{Const}$ .

### 3.1 Syntax and Operational Semantics

The syntax of  $\lambda^U$  is shown in Figure 3.1. A term is delayed when more brackets enclose it than do escapes, and a program must not have an escape in any non-delayed region. We track *levels* to model this behavior. A term's exact level  $\text{lv } e$  is its nesting depth of escapes minus brackets, and a program is a closed, exactly level-0 term. A level-0 value (i.e., a value in a non-delayed region) is a constant, an abstraction, or a code value with no un-delayed region. At level  $\ell > 0$  (i.e., inside  $\ell$  pairs of brackets), a value is any lower-level term. Throughout the thesis, “the set of terms with exact level at most  $\ell$ ”, written  $E^\ell$ , is a much more useful concept than “the set of terms with exact level equal to  $\ell$ ”. When we say “ $e$  has level  $\ell$ ” we mean  $e \in E^\ell$ , whereas “ $e$  has exact level  $\ell$ ” means  $\text{lv } e = \ell$ . A context  $C$  is a term with exactly one subterm replaced by a hole  $\bullet$ , and  $C[e]$  is the term obtained by replacing

*Evaluation Contexts* (Productions marked  $[\phi]$  apply only if the guard  $\phi$  is true.)

$$\begin{aligned} \text{(CBN)} \quad \mathcal{E}^{\ell,m} \in ECtx_{\mathbf{n}}^{\ell,m} ::= & \bullet[m = \ell] \mid \lambda x. \mathcal{E}^{\ell,m}[\ell > 0] \mid \langle \mathcal{E}^{\ell+1,m} \rangle \mid \sim \mathcal{E}^{\ell-1,m}[\ell > 0] \\ & \mid !\mathcal{E}^{\ell,m} \mid \mathcal{E}^{\ell,m} e^\ell \mid v^\ell \mathcal{E}^{\ell,m}[\ell > 0] \mid c \mathcal{E}^{\ell,m}[\ell = 0] \end{aligned}$$

$$\begin{aligned} \text{(CBV)} \quad \mathcal{E}^{\ell,m} \in ECtx_{\mathbf{v}}^{\ell,m} ::= & \bullet[m = \ell] \mid \lambda x. \mathcal{E}^{\ell,m}[\ell > 0] \mid \langle \mathcal{E}^{\ell+1,m} \rangle \mid \sim \mathcal{E}^{\ell-1,m}[\ell > 0] \\ & \mid !\mathcal{E}^{\ell,m} \mid \mathcal{E}^{\ell,m} e^\ell \mid v^\ell \mathcal{E}^{\ell,m} \end{aligned}$$

*Substitutable Arguments*  $a, b \in Arg ::= v^0$  (CBV)  $a, b \in Arg ::= e^0$  (CBN)

*Small-steps*  $e^\ell \rightsquigarrow_\ell t^\ell$  where:

<p>SS-<math>\beta</math></p> <p>(CBN)</p> <hr style="width: 100%;"/> $(\lambda x. e^0) t^0 \rightsquigarrow_\ell [t^0/x]e^0$	<p>SS-<math>\beta_v</math></p> <p>(CBV)</p> <hr style="width: 100%;"/> $(\lambda x. e^0) v^0 \rightsquigarrow_\ell [v^0/x]e^0$	<p>SS-<math>\delta</math></p> <p><math>(c, d) \in \text{dom } \delta</math></p> <hr style="width: 100%;"/> $c d \rightsquigarrow_\ell \delta(c, d)$
<p>SS-<math>E</math></p> <hr style="width: 100%;"/> $\sim \langle e^0 \rangle \rightsquigarrow_1 e^0$	<p>SS-<math>R</math></p> <hr style="width: 100%;"/> $! \langle e^0 \rangle \rightsquigarrow_0 e^0$	<p>SS-<math>C_{TX}</math></p> <hr style="width: 100%;"/> $e^m \rightsquigarrow_m t^m$ $\mathcal{E}^{\ell,m}[e^m] \rightsquigarrow_\ell \mathcal{E}^{\ell,m}[t^m]$

Figure 3.2 : Operational semantics of  $\lambda^U$ , parametrized in an interpretation (partial) map  $\delta : Const \times Const \rightarrow \{v \in V_{cl}^0 : v \equiv \|v\|\}$ .

the hole with  $e$ , with variable capture. Staging annotations use the same nesting rules as LISP's quasiquote and unquote [12], but we stress that they preserve scoping: e.g.,  $\langle \lambda x. \sim(\lambda x. \langle x \rangle) \rangle \equiv \langle \lambda x. \sim(\lambda y. \langle y \rangle) \rangle \not\equiv \langle \lambda y. \sim(\lambda x. \langle y \rangle) \rangle$ .

A term is unstaged if its annotations are erased in the following sense; it is staged otherwise. The **power** function is the erasure of **stpow** modulo  $\eta$  reduction.

**Definition 1** (Erasure). Define the erasure  $\|e\|$  by

$$\begin{aligned} \|x\| &\stackrel{\text{def}}{=} x & \|c\| &\stackrel{\text{def}}{=} c & \|\lambda x. e\| &\stackrel{\text{def}}{=} \lambda x. \|e\| & \|\sim e\| &\stackrel{\text{def}}{=} \|e\| \\ \|e_1 e_2\| &\stackrel{\text{def}}{=} \|e_1\| \|e_2\| & \|\langle e \rangle\| &\stackrel{\text{def}}{=} \|e\| & \|!e\| &\stackrel{\text{def}}{=} \|e\| \end{aligned}$$

The operational semantics is given in Figure 3.2; examples are provided below. Square brackets denote guards on grammatical production rules; for instance,  $ECtx_{\mathbf{n}}^{\ell,m} ::= \bullet[m = \ell] \mid \dots$  means  $\bullet \in ECtx_{\mathbf{n}}^{\ell,m}$  iff  $m = \ell$ . An  $\ell, m$ -evaluation context  $\mathcal{E}^{\ell,m}$  takes a level- $m$  redex and yields a level- $\ell$  term. Redex contractions are:  $\beta$  reduction at level 0,  $\delta$  reduction at level 0, run-bracket elimination (SS- $R$ ) at level 0, and escape-bracket elimination at level 1 (SS- $E$ ). CBN uses SS- $\beta$  and CBV uses SS- $\beta_v$ . All other rules are common to both evaluation strategies. The  $\delta$  reductions are specified by a partial map  $\delta : Const \times Const \rightarrow \{v \in V_{cl}^0 : v \equiv \|v\|\}$ , which is chosen according to what data types we would like to model in the calculus. This  $\delta$  should be undefined on ill-formed pairs like  $\delta(\mathbf{not}, 5)$ . Constant applications are assumed not to return staged terms.

Small-steps specify the behavior of deterministic evaluators. Every term decomposes in at most one way (see Appendix C.2.2 for a proof) as  $\mathcal{E}^{\ell,m}[t]$  where the redex (reducible expression)  $t$  is a level- $m$  term that must match the left-hand side of the bottom row of one of the rules SS- $\beta$  (for CBN), SS- $\beta_v$  (for CBV), SS- $\delta$ , SS- $E$ , and SS- $R$ . Replacing  $t$  by the right-hand side of the matching rule constitutes a small-step reduction (or just small-step). The small-step reduct so produced is unique if it exists. Intuitively, given a program  $p$ , an implementation of  $\lambda^U$  rewrites  $p$  by a series of small-steps until it is no longer possible to small-step. If the series of small-steps continues forever, then  $p$  is non-terminating; if the resulting term is a value, then that is the return value of  $p$ ; and if the resulting term is not a value,  $p$  is said to be stuck at that point and is considered to have died by an error.

Evaluation strategies—CBV vs. CBN—refer to the rules by which the redex to contract is chosen from any given expression. Basically, the rules for  $\lambda^U$  are: pick the outermost, leftmost, non-delayed redex. However, given a non-delayed function



application  $(\lambda x.e) t$ , CBN semantics chooses the whole application as the redex, whereas CBV semantics chooses the redex from  $t$  by recursively applying the same outermost-leftmost rule to  $t$ . Hence CBN immediately substitutes  $t$  for  $x$  without evaluating  $t$ , while CBV insists on evaluating  $t$  first. This difference is reflected in CBV and CBN evaluation contexts as follows. CBV evaluation contexts can place the hole inside the argument of a level-0 application, but CBN can do so only if the operator is a constant. At level  $> 0$ , both evaluation strategies simply walk over the syntax tree of the delayed term to look for escapes, including ones that occur inside the arguments of applications, hence both strategies' evaluation contexts can place holes in argument positions at level  $> 0$ .

**Notation.** We write  $\lambda_{\mathbf{n}}^U \vdash e \rightsquigarrow_{\ell} t$  for a CBN small-step judgment and  $\lambda_{\mathbf{v}}^U \vdash e \rightsquigarrow_{\ell} t$  for CBV. We use similar notation for  $(\Downarrow)$ ,  $(\Uparrow)$ , and  $(\approx)$  defined below. For any relation  $R$ , let  $R^+$  be its transitive closure and  $R^*$  its reflexive-transitive closure; let  $R^0$  be equality and let  $xR^ny$  mean  $\exists\{z_i\}_{i=0}^n$  such that  $x = z_0$ ,  $z_n = y$ , and  $\forall i. x_i R x_{i+1}$ . The metavariables  $a, b \in Arg$  will range over substitutable arguments, i.e.,  $e^0$  for CBN and  $v^0$  for CBV.

For example,  $p_1 \equiv (\lambda y.\langle 40 + y \rangle) (1 + 1)$  is a program. Its value is determined by  $(\rightsquigarrow_0)$ , which works like in conventional calculi. In CBN,  $\lambda_{\mathbf{n}}^U \vdash p_1 \rightsquigarrow_0 \langle 40 + (1 + 1) \rangle$ . The redex  $(1 + 1)$  is not selected for contraction because  $(\lambda y.\langle 40 + y \rangle) \bullet \notin ECtx_{\mathbf{n}}^{0,0}$ . In CBV,  $(\lambda y.\langle 40 + y \rangle) \bullet \in ECtx^{0,0}$ , so  $(1 + 1)$  is selected for contraction:  $\lambda_{\mathbf{v}}^U \vdash p_1 \rightsquigarrow_0 (\lambda y.\langle 40 + y \rangle) 2 \rightsquigarrow_0 \langle 40 + 2 \rangle$ .

Let  $p_2 \equiv \langle \lambda z.z (\sim[(\lambda \_.\langle z \rangle) 1]) \rangle$ , where we used square brackets  $[ ]$  as parentheses to improve readability. Let  $e^0$  be the subterm inside square brackets. In both CBN and CBV,  $p_2$  decomposes as  $\mathcal{E}[e^0]$ , where  $\mathcal{E} \equiv \langle \lambda z.z (\sim \bullet) \rangle \in ECtx^{0,0}$ , and  $e^0$  is a level-0 redex. Note the hole of  $\mathcal{E}$  is under a binder and the redex  $e^0$  is open, though  $p_2$  is

closed. The hole is also in argument position in the application  $z (\sim \bullet)$  even for CBN. This application is delayed by brackets, so the CBN/CBV distinction is irrelevant until the delay is canceled by  $!$ . Hence,  $p_2 \rightsquigarrow_0 \langle \lambda z.z (\sim \langle z \rangle) \rangle \rightsquigarrow_0 \langle \lambda z.z z \rangle$ .

As usual, this “untyped” formalism can be seen as dynamically typed. In this view,  $\sim$  and  $!$  take code-type arguments, where code is a distinct type from functions and base types. Thus  $\langle \lambda x.x \rangle 1$ ,  $\langle \sim 0 \rangle$ , and  $!5$  are all stuck. Stuckness on variables like  $x 5$  does not arise in programs for conventional languages because programs are closed, but in  $\lambda^U$  evaluation contexts can pick redexes under binders so this type of stuckness does become a concern. We will come back to this point in Chapter 3.4.

**Remark.** Binary operations on constants are modeled by including their partially applied variants. For example, to model addition we take  $Const \supseteq \mathbb{Z} \cup \{+\} \cup \{+_k : k \in \mathbb{Z}\}$  and set  $\delta(+, k) = +_k$ ,  $\delta(+_k, k') = (\text{the sum of } k \text{ and } k')$ . For example, in prefix notation,  $(+ 3 5) \rightsquigarrow_0 (+_3 5) \rightsquigarrow_0 8$ . Conditionals are modeled by taking  $Const \supseteq \{(), \text{true}, \text{false}, \text{if}\}$  and setting  $\delta(\text{if}, \text{true}) = \lambda a.\lambda b.a ()$  and  $\delta(\text{if}, \text{false}) = \lambda a.\lambda b.b ()$ . Then, for example, we have  $\text{if true } (\lambda_.1) (\lambda_.0) \rightsquigarrow_0 (\lambda a.\lambda b.a ()) (\lambda_.1) (\lambda_.0) \rightsquigarrow_0^* 1$ . Note that the rest of the thesis uses infix notation and displays conditionals as **if**  $e_1$  **then**  $e_2$  **else**  $e_3$  rather than **if**  $e_1$   $(\lambda_.e_2)$   $(\lambda_.e_3)$ .

**Definition 2** (Termination and Divergence). An  $e \in E^\ell$  *terminates* to  $v \in V^\ell$  at level  $\ell$  iff  $e \rightsquigarrow_\ell^* v$ , written  $e \Downarrow^\ell v$ . We write  $e \Downarrow^\ell$  to mean  $\exists v. e \Downarrow^\ell v$ . If no such  $v$  exists, then  $e$  *diverges* ( $e \Uparrow^\ell$ ). Note that divergence includes stuckness.

The operational semantics induces the usual notion of observational equivalence—a pair of terms are observationally equivalent precisely when replacing one by the other in any program is guaranteed not to change the observable outcome of the overall program. The “observable” outcome here consists of a) whether the program

terminates, and b) if the program terminates and returns a constant, which one is returned. Note that if the program returns a function, which function is returned is not observed.

**Definition 3** (Observational Equivalence).  $e \approx t$  iff for every  $C$  such that  $C[e], C[t] \in \text{Prog}$ ,  $C[e] \Downarrow^0 \iff C[t] \Downarrow^0$  holds and whenever one of them terminates to a constant, the other also terminates to the same constant.

Taha [32] used  $C[e], C[t] \in E^0$  in place of  $C[e], C[t] \in \text{Prog}$ , thus requiring equitermination under non-closing contexts (i.e. those which may not bind some free variables in  $e$  and  $t$ ). Intuitively,  $\text{Prog}$  is more accurate as we are interested only in executing programs, not terms. This does not represent any shift in semantics however, as these definitions coincide in MSP (see Appendix C.2.3).

The following lemma is useful for proving that certain terms diverge.

**Lemma 4.** If  $e^\ell \equiv \mathcal{E}^{\ell,m}[t^m] \rightsquigarrow_\ell^n v \in V^\ell$ , then  $t^m \rightsquigarrow_m^{n'} u \in V^m$  where  $n' \leq n$ . In particular, if  $t^m \Uparrow^m$  then  $\mathcal{E}^{\ell,m}[t^m] \Uparrow^\ell$ .

### 3.2 Equational Theory

The equational theory of  $\lambda^U$  is a proof system containing four inference rules: compatible extension ( $e = t \implies C[e] = C[t]$ ), reflexivity, symmetry, and transitivity. The CBN axioms are  $\lambda_{\mathbf{n}}^U \stackrel{\text{def}}{=} \{\beta, E_U, R_U, \delta\}$ , while CBV axioms are  $\lambda_{\mathbf{v}}^U \stackrel{\text{def}}{=} \{\beta_{\mathbf{v}}, E_U, R_U, \delta\}$ . Each axiom is shown below. If  $e = t$  can be proved from a set of axioms  $\Phi$ , then  $e$  and  $t$  are *provably equal* (under  $\Phi$ ), written  $\Phi \vdash e = t$ . We often omit the  $\Phi \vdash$  in definitions and assertions that apply uniformly to both CBV and CBN. Reduction is a term rewrite induced by the axioms:  $\Phi \vdash e \longrightarrow t$  iff  $e = t$  is derivable from the axioms by compatible extension alone.

Name	Axiom	Side Condition
$\beta$	$(\lambda x.e^0) t^0 = [t^0/x]e^0$	
$\beta_{\mathbf{v}}$	$(\lambda x.e^0) v^0 = [v^0/x]e^0$	
$E_U$	$\sim \langle e \rangle = e$	
$R_U$	$! \langle e^0 \rangle = e^0$	
$\delta$	$c d = \delta(c, d)$	$(c, d) \in \text{dom } \delta$

For example, axiom  $\beta_{\mathbf{v}}$  gives  $\lambda_{\mathbf{v}}^U \vdash (\lambda_{-}.0) 1 = 0$ . By compatible extension under  $\langle \bullet \rangle$ , we have  $\langle (\lambda_{-}.0) 1 \rangle = \langle 0 \rangle$ , in fact  $\langle (\lambda_{-}.0) 1 \rangle \longrightarrow \langle 0 \rangle$ . Note  $\langle (\lambda_{-}.0) 1 \rangle \not\gamma_0^{\sim} \langle 0 \rangle$  because brackets delay the application, but reduction allows all left-to-right rewrites by the axioms, so  $\langle (\lambda_{-}.0) 1 \rangle \longrightarrow \langle 0 \rangle$  nonetheless. Intuitively,  $\langle (\lambda_{-}.0) 1 \rangle \not\gamma_0^{\sim} \langle 0 \rangle$  because an evaluator does not perform this rewrite, but  $\langle (\lambda_{-}.0) 1 \rangle \longrightarrow \langle 0 \rangle$  because this rewrite is semantics-preserving and an optimizer is allowed to perform it.

Just like the plain  $\lambda$  calculus,  $\lambda^U$  satisfies the Church-Rosser property, so every term has at most one normal form (irreducible reduct) [32]. Terms are hence not provably equal when they have distinct normal forms. Church-Rosser also ensures that reduction and provable equality are more or less interchangeable, and when we investigate the properties of provable equality, we usually do not lose generality by restricting our attention to the simpler notion of reduction. This result had been known for CBN  $\lambda^U$  without constants, but a subsidiary contribution of this thesis is that this result extends to both CBN and CBV with constants and  $\delta$  reduction.

**Theorem 5** (Church-Rosser Property).  $e = e' \iff \exists t. e \longrightarrow^* t \longleftarrow^* e'$ .

Provable equality ( $=$ ) is an approximation of observational equivalence. The containment  $(=) \subset (\approx)$  is proper because  $(\approx)$  is not semi-decidable (since  $\lambda^U$  is Turing-complete) whereas  $(=)$  clearly is. There are several useful equivalences in  $(\approx) \setminus (=)$ ,

which we will prove by applicative bisimulation. Provable equality is nonetheless strong enough to discover the value of any term that has one, so the assertion “ $e$  terminates (at level  $\ell$ )” is interchangeable with “ $e$  reduces to a (level- $\ell$ ) value”.

**Theorem 6** (Soundness).  $(=) \subset (\approx)$ .

**Theorem 7.** If  $e \in E^\ell, v \in V^\ell$ , then  $e \Downarrow^\ell v \implies (e \longrightarrow^* v \wedge e = v)$  and  $e = v \in V^\ell \implies (\exists u \in V^\ell. u = v \wedge e \longrightarrow^* u \wedge e \Downarrow^\ell u)$ .

The rest of this section is devoted to describing the proofs of these properties. Readers who are not interested in the proofs should skip to Chapter 3.3. The Church-Rosser property has an equivalent formulation, confluence, which is more amenable to proof. Proofs that confluence and Church-Rosser imply each other can be found in standard textbooks on rewrite systems and programming language metatheory [24, 2].

**Theorem 8** (Confluence).  $t_1 \longleftarrow^* e \longrightarrow^* t_2 \implies \exists e'. t_1 \longrightarrow^* e' \longleftarrow^* t_2$ .

Confluence can be proved by the well-known Tait–Martin-Löf method. The stylized formulation used here is due to Takahashi [34] and is the same as the one used in Taha’s dissertation [32]. The method uses parallel reduction, written  $e \twoheadrightarrow t$ , which is like ordinary reduction but allows any number of independent redexes in  $e$  to be contracted at once. The main insight from Takahashi is that there is a maximal parallel reduction, the complete development  $e^*$ , which contracts all independent redexes in  $e$ . Because  $e \twoheadrightarrow t$  contracts a subset of those redexes, contracting the complement of that subset takes  $t$  to  $e^*$ , which does not depend on  $t$  (Takahashi’s property). Thus  $(\twoheadrightarrow)$  is strongly confluent, i.e. every  $t_1 \longleftarrow e \twoheadrightarrow t_2$  meets in one step,  $t_1 \twoheadrightarrow e^* \longleftarrow t_2$ . Strong confluence is easily seen to imply confluence.

$$\begin{array}{c}
\frac{}{c \xrightarrow{0} c} \text{ [PR-CONST]} \qquad \frac{}{x \xrightarrow{0} x} \text{ [PR-VAR]} \qquad \frac{e \xrightarrow{n} t}{\lambda x. e \xrightarrow{n} \lambda x. t} \text{ [PR-ABS]} \\
\\
\frac{e_1 \xrightarrow{n_1} t_1 \quad e_2 \xrightarrow{n_2} t_2}{e_1 \ e_2 \xrightarrow{n_1+n_2} t_1 \ t_2} \text{ [PR-APP]} \qquad \frac{(c, d) \in \text{dom } \delta}{c \ d \xrightarrow{1} \delta(c, d)} \text{ [PR-}\delta\text{]} \\
\\
\frac{e^0 \xrightarrow{n_1} t^0 \quad a \xrightarrow{n_2} b}{(\lambda x. e^0) \ a \xrightarrow{n_1+n_2\#(x, t^0)+1} [b/x]t^0} \text{ [PR-}\beta\text{]} \qquad \frac{e \xrightarrow{n} t}{\langle e \rangle \xrightarrow{n} \langle t \rangle} \text{ [PR-BRK]} \\
\\
\frac{e \xrightarrow{n} t}{\sim e \xrightarrow{n} \sim t} \text{ [PR-ESC]} \qquad \frac{e \xrightarrow{n} t}{\sim \langle e \rangle \xrightarrow{n+1} t} \text{ [PR-E]} \qquad \frac{e \xrightarrow{n} t}{!e \xrightarrow{n} !t} \text{ [PR-RUN]} \\
\\
\frac{e^0 \xrightarrow{n} t^0}{! \langle e^0 \rangle \xrightarrow{n+1} t^0} \text{ [PR-R]} \qquad (\xrightarrow{\phantom{n}}) \stackrel{\text{def}}{=} \bigcup_n (\xrightarrow{n})
\end{array}$$

$\#(x, e)$  is the number of occurrences of  $x$  in  $e$ .

Figure 3.3 : Parallel reduction.

Parallel reduction is defined in Figure 3.3. The subscript on  $(\xrightarrow{n})$  is complexity, which shows the number of ordinary, leftmost-outermost-first reductions it would take to mimic the parallel reduction. Complexity is used later to prove soundness, but for proving confluence it is dead weight, so we discard the complexity annotation and work with  $(\xrightarrow{\phantom{n}})$  for the time being.

As parallel reduction is just ordinary reduction with a different convention for counting the number of steps, its reflexive-transitive closure coincides with that of

$$\begin{array}{ll}
c^* \stackrel{\text{def}}{=} c & x^* \stackrel{\text{def}}{=} x \\
(\lambda x.e)^* \stackrel{\text{def}}{=} \lambda x.e^* & \langle e \rangle^* \stackrel{\text{def}}{=} \langle e^* \rangle \\
(\sim \langle e \rangle)^* \stackrel{\text{def}}{=} e^* & (\sim e)^* \stackrel{\text{def}}{=} \sim(e^*) \quad [\text{if } e \neq \langle e' \rangle] \\
(! \langle e^0 \rangle)^* \stackrel{\text{def}}{=} (e^0)^* & (! e)^* \stackrel{\text{def}}{=} ! e^* \quad [\text{if } e \neq \langle e^0 \rangle] \\
((\lambda x.e^0) a)^* \stackrel{\text{def}}{=} [a^*/x](e^0)^* & c d \stackrel{\text{def}}{=} \delta(c, d) \quad [\text{if } (c, d) \in \text{dom } \delta] \\
(e_1 e_2)^* \stackrel{\text{def}}{=} e_1^* e_2^* \quad [\text{if } e_1 e_2 \neq (\lambda x.e^0) a \text{ and } e_1 e_2 \equiv c d \implies (c, d) \notin \text{dom } \delta]
\end{array}$$

$e^*$  is the complete development of  $e$ .

Figure 3.4 : Complete development.

ordinary reduction:

**Lemma 9.**  $(\longrightarrow^*) = (\longrightarrow^*)$ .

*Proof.* Derivation rules for  $(\longrightarrow^*)$  subsume all reduction axioms and  $e \longrightarrow t \implies C[e] \longrightarrow C[t]$ , so  $(\longrightarrow^*) \subseteq (\longrightarrow^*)$ . For the reverse containment,  $e \longrightarrow t \implies e \longrightarrow^* t$  by straightforward induction on the parallel reduction judgment. Therefore,  $\forall n. e \longrightarrow^n t \implies e \longrightarrow^* t$  by induction on  $n$ .  $\square$

Thus for Theorem 8 it suffices to prove that  $(\longrightarrow)$  is confluent. As mentioned above,  $\lambda^U$ 's parallel reduction satisfies Takahashi's property, which ensures its confluence.

**Lemma 10** (Takahashi's Property).  $e \longrightarrow t \implies t \longrightarrow e^*$ .

*Proof.* Induction on  $e$ . See Appendix C.3.1 for details.  $\square$

**Proposition 11.**  $(\longrightarrow^*)$  is confluent.

*Proof.* This assertion reduces to Takahashi's property by lexicographical induction on the lengths of the departing parallel reductions (i.e., given  $t_1 \leftarrow^n e \rightarrow^m t_2$ , we induct on  $(n, m)$ ). See Appendix C.3.1 for details.  $\square$

Let us now consider soundness (Theorem 6). To prove soundness, we first prove correspondence between the values discovered by provable equality and by small-steps (i.e. Theorem 7), using reduction as a stepping stone.

**Proposition 12.** If  $e \in E^\ell$  and  $v \in V^\ell$  then  $e = v \iff \exists u \in V^\ell. e \rightarrow^* u = v$ .

*Proof.* For the forward direction, the Church-Rosser property guarantees that  $e \rightarrow^* v \implies \exists t \in E^\ell. e \rightarrow^* t \leftarrow^* v$  hence  $e \rightarrow^* t = v$ . The only remaining question is whether  $t \in V^\ell$ , but  $t$  is a reduct of a value so it must be a value as well (see Lemma 102). The converse follows from  $(\rightarrow^*) \subseteq (=)$ .  $\square$

**Lemma 13** (Compatibility of Reduction and Small-step Semantics). If  $e \in E^\ell$  and  $v \in V^\ell$  then  $e \rightarrow^* v \iff \exists u \in V^\ell. e \rightsquigarrow_\ell^* u \rightarrow^* v$ .

The  $(\iff)$  direction of this lemma is trivial. The  $(\implies)$  direction is proved via three lemmas that convert a parallel-reduction sequence

$$e_0^\ell \twoheadrightarrow e_1 \twoheadrightarrow \cdots \twoheadrightarrow e_n \equiv v^\ell$$

to a small-step sequence

$$e_0^\ell \rightsquigarrow_\ell t_1 \rightsquigarrow_\ell t_2 \rightsquigarrow_\ell \cdots \rightsquigarrow_\ell t_m \equiv u^\ell \twoheadrightarrow v^\ell.$$

The proofs of these lemma require the complexity annotations of  $(\twoheadrightarrow)$  as an induction measure.

**Lemma 14** (Transition). If  $e \in E^\ell$  and  $v \in V^\ell$  then  $e \xrightarrow{n} v \implies \exists u \in V^\ell. e \rightsquigarrow_\ell^* u \twoheadrightarrow v$ .



*Proof.* If  $e \in V^n$ , then just take  $u \stackrel{\text{def}}{=} e$ . Otherwise, induct on  $(n, e)$  under the lexicographical ordering with case analysis on the last rule used to derive the parallel reduction. See Appendix C.3.2 for details.  $\square$

**Lemma 15** (Permutation). If  $e, t, d \in E^\ell$  then  $e \twoheadrightarrow_n t \rightsquigarrow_\ell d \implies \exists t' \in E^\ell. e \rightsquigarrow_\ell^+ t' \twoheadrightarrow d$ .

*Proof.* Induction on  $n$  with case analysis on the last rule used to derive the parallel reduction. See Appendix C.3.2 for details.  $\square$

**Lemma 16** (Push Back). If  $e, t \in E^\ell$  and  $v \in V^\ell$  then  $e \twoheadrightarrow t \rightsquigarrow_\ell^+ v \implies \exists u \in V^\ell. e \rightsquigarrow_\ell^+ u \twoheadrightarrow v$ .

*Proof.* Let the length of the small-step sequence be  $n$ . Induct on  $n$ .

[If  $n = 1$ ] By Permutation  $\exists t' \in E^\ell. e \rightsquigarrow_\ell^+ t' \twoheadrightarrow v$ , so by Transition,  $\exists u \in V^\ell. t' \rightsquigarrow_\ell^* u \twoheadrightarrow v$ . Putting them together,  $e \rightsquigarrow_\ell^+ t' \rightsquigarrow_\ell^* u \twoheadrightarrow v$ .

[If  $n > 1$ ] By hypothesis,  $\exists d. e \twoheadrightarrow t \rightsquigarrow_\ell d \rightsquigarrow_\ell^{(n-1)} v$ . Permutation gives  $\exists t'. e \rightsquigarrow_\ell t' \twoheadrightarrow d \rightsquigarrow_\ell^{(n-1)} v$ . Then by IH  $\exists u \in V^\ell. t' \rightsquigarrow_\ell^+ u \twoheadrightarrow v$ , so  $e \rightsquigarrow_\ell t' \rightsquigarrow_\ell^+ u \twoheadrightarrow v$ .  $\square$

*Proof of Lemma 13.*

( $\implies$ ) If  $e \twoheadrightarrow^* v$  then  $e \twoheadrightarrow^n v$  for some  $n$  by Lemma 9. We wish to show  $\exists u \in V^\ell. \exists m \geq 0. e \rightsquigarrow_\ell^m u \twoheadrightarrow v$  by induction on  $n$ . If  $n = 0$  then  $u \stackrel{\text{def}}{=} v$ . If  $n > 0$  then  $\exists t \in E^\ell. e \twoheadrightarrow t \twoheadrightarrow^{n-1} v$  so by IH  $\exists u \in V^\ell. \exists m' \geq 0. e \twoheadrightarrow t \rightsquigarrow_\ell^{m'} u \twoheadrightarrow^* v$ . Then the conclusion follows from Transition if  $m' = 0$ , or from Push Back if  $m' > 0$ .

( $\impliedby$ ) Follows from  $(\rightsquigarrow_\ell) \subseteq (\twoheadrightarrow)$ .  $\square$

Combining Lemma 13 with Proposition 12, we see that provable equality and small-steps are compatible as well.

**Lemma 17** (Compatibility of Equational Theory and Small-step Semantics). If  $e \in E^\ell$  and  $v \in V^\ell$  then  $e = v \iff \exists u \in V^\ell. e \rightsquigarrow_\ell^* u = v$ .

*Proof.* First suppose  $e = v$ . Then by Proposition 12,  $\exists w \in V^\ell$  such that  $e \longrightarrow^* w = v$ . By Lemma 13,  $\exists u \in V^\ell. e \rightsquigarrow_\ell^* u \longrightarrow^* w$  so  $e \rightsquigarrow_\ell^* u = w = v$ . For the converse, suppose  $e \rightsquigarrow_\ell^* u = v$ . Then  $e = u = v$  because  $(\rightsquigarrow_\ell) \subseteq (\longrightarrow) \subseteq (=)$ .  $\square$

Theorem 7 is immediate from Lemma 13 and Lemma 17. The soundness theorem is also straightforward given Lemma 17.

*Proof of Soundness.* Let  $e, t, C$  be given such that  $e = t$  and  $C[e], C[t] \in \text{Prog}$ . Let us suppose that one of the plugged expressions terminates, say  $C[e] \Downarrow^0$ , and prove that the other also does. By definition,  $\exists v \in V^0. C[e] \rightsquigarrow_0^* v$  so using Lemma 17 and EQ-CTX,  $v = C[e] = C[t]$ . Then by Lemma 17 again,  $C[t] \Downarrow^0 u = v$  for some  $u \in V^0$ . Then since  $u = v \equiv c$ , by the Church-Rosser property  $u$  and  $c$  have a common reduct, which must be  $c$  itself. One can easily see that a reduct of a non-constant value is always non-constant (see Lemma 102), so  $u \equiv c$  is forced.  $\square$

### 3.3 Further Generalization of Axioms is Unsound

The equational theory presented above is not identical to Taha's [32], but generalizes rule  $E_U$  from  $\sim\langle e^0 \rangle = e^0$  to  $\sim\langle e \rangle = e$ . In this section we discuss the utility of this generalization and explain why other axioms cannot be generalized in the same manner.

The main use of the new, generalized  $E_U$  is to show that substitution preserves  $(\approx)$ . Thus, an equivalence proved on open terms hold for any closed instance. This fact plays an important role in the completeness proof of applicative bisimulation to

be presented later. This proposition is also somewhat surprising, considering that its converse fails in CBV—we will examine that issue in more detail in Chapter 3.4.

**Proposition 18.** If  $e \approx t$ , then  $[a/x]e \approx [a/x]t$  for any  $a, x$ .

*Proof.* Take  $\ell = \max(\text{lv } e, \text{lv } t)$ . Then

$$(\lambda x. \langle \langle \dots \langle e \rangle \dots \rangle \rangle) a \approx (\lambda x. \langle \langle \dots \langle t \rangle \dots \rangle \rangle) a$$

where  $e$  and  $t$  are each enclosed in  $\ell$  pairs of brackets. Both sides are level 0, so the  $\beta_{\mathbf{v}}$  rule applies and

$$\langle \langle \dots \langle [a/x]e \rangle \dots \rangle \rangle \approx \langle \langle \dots \langle [a/x]t \rangle \dots \rangle \rangle.$$

Escaping both sides  $\ell$  times gives

$$\sim \dots \sim \langle \langle \dots \langle [a/x]e \rangle \dots \rangle \rangle \approx \sim \dots \sim \langle \langle \dots \langle [a/x]t \rangle \dots \rangle \rangle.$$

Then applying the  $E_U$  rule  $\ell$  times gives  $[a/x]e \approx [a/x]t$ . The old  $E_U$  rule  $\sim \langle e^0 \rangle = e^0$  would apply only once here because the level of the  $\langle \langle \dots \langle [a/x]e \rangle \dots \rangle \rangle$  part increases—and that is why we need the generalized rule.  $\square$

It is natural to wonder why the other rules,  $\beta/\beta_{\mathbf{v}}$  and  $R_U$ , cannot be generalized to arbitrary levels, and why  $E_U$  is special. The reason is that generalizations of  $\beta/\beta_{\mathbf{v}}$  and  $R_U$  involve level change—moving a term from one level to another. Generally, moving a lower-level term to a higher level is benign, but the opposite direction, called demotion, is not. MSP type system researchers have long observed that unrestricted demotion is a type-unsafe operation [33, 36]. We show here that it is also unsound as an equational rule.

Table 3.1 shows generalized rules along with counterexamples that show their unsoundness. The left column names the rule that was generalized, the middle column shows the generalization, and the right column refutes it. Simply dropping level

Rule	Generalization	Counterexample	
$R_U$	$!\langle e \rangle = e$	$\langle !\langle \sim\Omega \rangle \rangle \neq \langle \sim\Omega \rangle$	(*1)
$\beta$	$(\lambda x.e^0) t = [t/x]e^0$	$\langle (\lambda x.\langle x \rangle) (\lambda y.\sim\Omega) \rangle \neq \langle \langle \lambda y.\sim\Omega \rangle \rangle$	(*2)
$\beta_v$	$(\lambda x.e^0) v^\ell = [v^\ell/x]e^0$	same as (*2)	(*3)
$\beta_v$	$(\lambda x.e^0) (\lambda y.t) = [(\lambda y.t)/x]e^0$	same as (*2)	(*4)
$\beta_v$	$(\lambda x.e^0) \langle t \rangle = [\langle t \rangle/x]e^0$	$\langle (\lambda x.\langle \langle x \rangle \rangle) \langle \sim\Omega \rangle \rangle \neq \langle \langle \langle \sim\Omega \rangle \rangle \rangle$	(*5)
$\beta$	$(\lambda x.e) t^0 = [t^0/x]e$	$\langle (\lambda x.\sim x) \langle e^0 \rangle \rangle \neq \langle \sim \langle e^0 \rangle \rangle$	(*6)
$\beta_v$	$(\lambda x.e) v^0 = [v^0/x]e$	same as (*6)	(*7)

Table 3.1 : Unsound generalizations of our axioms.  $\Omega$  is some divergent level-0 term. The generalizations of  $\beta$  suppose CBN, generalizations of  $\beta_v$  suppose CBV, and the generalization of  $R_U$  is meant for both.

constraints from  $R_U$  gives (\*1). In CBN  $\beta$ , relaxing the argument's level gives (\*2). In CBV  $\beta_v$ , simply removing the argument's constraint produces  $(\lambda x.e^0) v^\ell = [v^\ell/x]e^0$ , which is absurd; it subsumes CBN reduction (note  $V^1 = E^0$ ). More sensible attempts are (\*4) and (\*5), which keep the constraints on head term constructors. Generalizing the function in  $\beta$  and  $\beta_v$  gives (\*6) and (\*7), respectively.

Generalizations (\*1) through (\*5) fail because they involve demotion, which moves a term from one level to another. For example, the generalized rule in (\*1) puts  $e$  inside more brackets on the left-hand side than on the right-hand side. The counterexample exploits this mismatch by choosing an  $e$  that contains a divergent term enclosed in just enough escapes so that the divergence is forced on one side but not the other. More concretely, on the left-hand side  $!\langle \sim\Omega \rangle \in E^0$  so  $\langle !\langle \sim\Omega \rangle \rangle \in V^0$ . However on the right-hand side the  $\Omega$  is enclosed in fewer brackets and has  $\text{lv } \sim\Omega = 1$ ,

so  $\langle \sim \Omega \rangle \notin V^0$ ; in fact  $\langle \sim \bullet \rangle \in ECtx^{0,0}$  so assuming  $\Omega \rightsquigarrow_0 \Omega_1 \rightsquigarrow_0 \Omega_2 \rightsquigarrow_0 \dots$  we have  $\langle \sim \Omega \rangle \rightsquigarrow_0 \langle \sim \Omega_1 \rangle \rightsquigarrow_0 \langle \sim \Omega_2 \rangle \rightsquigarrow_0 \dots$ . We can formalize this general insight as follows.

**Definition 19** (Level Function). Define  $\Delta : Ctx \rightarrow \mathbb{Z}$  as follows.

$$\begin{aligned} \Delta \bullet &\stackrel{\text{def}}{=} 0 & \Delta(C\ e) &\stackrel{\text{def}}{=} \Delta C & \Delta \langle C \rangle &\stackrel{\text{def}}{=} \Delta C - 1 & \Delta(!C) &\stackrel{\text{def}}{=} \Delta C \\ \Delta(\lambda x.C) &\stackrel{\text{def}}{=} \Delta C & \Delta(e\ C) &\stackrel{\text{def}}{=} \Delta C & \Delta(\sim C) &\stackrel{\text{def}}{=} \Delta C + 1 \end{aligned}$$

**Proposition 20.**  $\forall C. \exists L(C) \in \mathbb{N}. \text{lv } e \geq L(C) \implies \text{lv } C[e] = \text{lv } e + \Delta C.$

*Proof.* Induction on  $C$ . See Appendix C.4 for details.  $\square$

Intuitively,  $\Delta C$  is the limiting value of  $\text{lv } C[e] - \text{lv } e$  as  $\text{lv } e \rightarrow \infty$ . This difference converges to a constant  $\Delta C$  because when  $e$  is sufficiently high-level, the deepest nesting of escapes in  $C[e]$  occurs within  $e$ . Then  $\text{lv } C[e] - \text{lv } e$  depends only on the number of brackets and escapes surrounding the hole of  $C$ .

**Theorem 21.** Any rewrite rule that has the form or subsumes a rule of the form  $C[e] \longrightarrow C'[e]$  with  $\Delta C \neq \Delta C'$  is unsound ( $\exists e. C[e] \not\approx C'[e]$ ).

The proof of this theorem relies on the fact that if  $e$  has enough escapes, the escapes dominate all the staging annotations in  $C$  and the term they enclose is given top priority during program execution. In more technical terms,  $\text{lv } C[e]$  grows unboundedly with  $\text{lv } e$  because of Proposition 20, and beyond a certain threshold  $C \in ECtx^{\ell, \ell - \Delta C}$ . Hence if, say,  $\Delta C > \Delta C'$  then by Lemma 94  $e$  is evaluated first under  $C'$  but not under  $C$ . Notice that this proof fails, as expected, if the  $e$  in  $C[e] \longrightarrow C'[e]$  is restricted to  $e^0$ .

**Lemma 22** (Context Domination).  $\text{size}(C) < \ell \implies \exists m. C \in ECtx^{\ell, m}$ .

*Proof.* Induction on  $C$ . See Appendix C.4 for details.  $\square$

**Lemma 23.**  $\Delta\mathcal{E}^{\ell,m} = \ell - m$ .

*Proof.* Straightforward induction on  $\mathcal{E}^{\ell,m}$ . □

*Proof of Theorem 21.* Take  $\ell \stackrel{\text{def}}{=} \max(L(C), L(C'), \text{size}(C) + 1, \text{size}(C') + 1)$  and  $e \equiv \underbrace{\sim \dots \sim}_{\ell \text{ times}} \Omega$ , where  $\Omega \in E^0$  and  $\Omega \uparrow^0$ . Then  $\text{lv } e = \ell$ ,  $e \uparrow^\ell$ ,  $\text{lv } C[e] = \ell + \Delta C$ , and  $\text{lv } C'[e] = \ell + \Delta C'$ . Without loss of generality,  $\Delta C > \Delta C'$ . By Lemma 22,  $C \in ECtx^{\ell+\Delta C, \ell}$  where the second superscript is known by Lemma 23. Then taking  $C_{\langle \dots \rangle} \stackrel{\text{def}}{=} \langle \langle \dots \langle \bullet \rangle \dots \rangle \rangle$  with  $\ell + \Delta C$  pairs of brackets,  $C_{\langle \dots \rangle}[C] \in ECtx^{0, \ell}$ , so Lemma 94 forces  $C_{\langle \dots \rangle}[C[e]] \uparrow^0$ . By contrast,  $\text{lv } C'[e] < \ell + \Delta C$ , so  $C_{\langle \dots \rangle}[C'[e]]$  is of the form  $\langle d^0 \rangle$ , hence  $C_{\langle \dots \rangle}[C'[e]] \Downarrow^0$ . □

Theorem 21 provides a quick sanity check for all equational rewrites. In particular, (\*1) through (\*5) above fail this test. Note that a sound rule can rewrite between contexts  $C$  and  $C'$  such that  $\text{lv } C[e] - \text{lv } e$  and  $\text{lv } C'[e] - \text{lv } e$  disagree for some  $e$ , as long as those  $e$  are all low-level. For example,  $E_U$  states  $\sim \langle e \rangle = e$ , but if  $e \in E^0$  then  $\text{lv } \sim \langle e \rangle - \text{lv } e = 1 \neq \text{lv } e - \text{lv } e$ . However, the differences of exact levels agree whenever  $\text{lv } e \geq 1$ , which is why Theorem 21 does not apply to  $E_U$ . Restricting the level of expressions that can plug level-mismatching holes may also ensure soundness; non-generalized  $R_U$  does this.

The entries (\*6) and (\*7) in Table 3.1 happen to pass the level function test. These rules have in a sense a dual problem: the substitutions in (\*6) and (\*7) inject extra brackets to locations that were previously stuck on a variable, whereas Theorem 21 injects extra escapes.

### 3.4 Closing Substitutions Compromise Validity

Here is a striking example of how reasoning in  $\lambda^U$  differs from reasoning in single-stage languages. Traditionally, CBV calculi admit the equational rule

$$(\beta_x) \quad (\lambda y.e^0) x = [x/y]e^0 .$$

Plotkin's seminal  $\lambda_V$  [27], for example, does so implicitly by taking variables to be values, defining  $x \in V$  where  $V$  is the set of values for  $\lambda_V$ . But  $\beta_x$  is *not* admissible in  $\lambda_V^U$ . For example, the terms  $(\lambda_{-}.0) x$  and  $0$  may seem interchangeable, but in  $\lambda_V^U$  they are distinguished by the program context  $\mathcal{E} \stackrel{\text{def}}{=} \langle \lambda x. \sim[(\lambda_{-}. \langle 1 \rangle) \bullet] \rangle$ :

$$\langle \lambda x. \sim[(\lambda_{-}. \langle 1 \rangle) ((\lambda_{-}.0) x)] \rangle \Uparrow^0 \quad \text{but} \quad \langle \lambda x. \sim[(\lambda_{-}. \langle 1 \rangle) 0] \rangle \Downarrow^0 \langle \lambda x.1 \rangle . \quad (1)$$

(Once again, we are using  $[ ]$  as parentheses to enhance readability.) The term on the left is stuck because  $x \notin V^0$  and  $x \not\rightarrow_0$ . Intuitively, the value of  $x$  is demanded before anything is substituted for it. If we apply a substitution  $\sigma$  that replaces  $x$  by a value, then  $\sigma((\lambda_{-}.0) x) = \sigma 0$ , so the standard technique of reasoning under closing substitutions is unsound. Note the  $\beta_x$  redex itself need not contain staging annotations; thus, adding staging to a language can compromise some existing equivalences, i.e., staging is a non-conservative language extension.

The problem here is that  $\lambda_V^U$  can evaluate open terms. Some readers may recall that  $\lambda_V$  *reduces* open terms just fine while admitting  $\beta_x$ , but the crucial difference is that  $\lambda^U$  *evaluates* (small-steps) open terms under program contexts whereas  $\lambda_V$  never does. Small-steps are the specification for implementations, so if they can rewrite an open subterm of a program, implementations must be able to perform that rewrite as well. By contrast, reduction is just a semantics-preserving rewrite, so implementations may or may not be able to perform it.

Implementations of  $\lambda_V^U$  including MetaOCaml have no runtime values, or data

structures, representing the variable  $x$ —they implement  $x \notin V^0$ . They never perform  $(\lambda_.0) x \rightsquigarrow_0 0$ , for if they were forced to evaluate  $(\lambda_.0) x$ , then they would try to evaluate the  $x$  as required for CBV and throw an error. Some program contexts in  $\lambda^U$  do force the evaluation of open terms, e.g., the  $\mathcal{E}$  given above. We must then define a small-step semantics with  $(\lambda_.0) x \not\rightsquigarrow_0 0$ , or else we would not model actual implementations, hence we must reject  $\beta_x$ , for it is unsound for  $(\approx)$  in such a small-step semantics. In other words, lack of  $\beta_x$  is an inevitable consequence of the way practical implementations behave.

Even in  $\lambda_V$ , setting  $x \in V$  is technically a mistake because  $\lambda_V$  implementations typically do not have runtime representations for variables either. But in  $\lambda_V$ , whether a given evaluator implements  $x \in V$  or  $x \notin V$  is unobservable. Small-steps on a  $\lambda_V$  program (which is closed by definition) never contract open redexes because evaluation contexts cannot contain binders. Submitting programs to an evaluator will never tell if it implements  $x \in V$  or  $x \notin V$ . Therefore, in  $\lambda_V$ , there is always no harm in pretending  $x \in V$ . A small-step semantics with  $x \in V$  gives the same  $(\approx)$  as one with  $x \notin V$ , and  $\beta_x$  is sound for this  $(\approx)$ .

Now, the general, more important, problem is that reasoning under substitutions is unsound, i.e.,  $\forall \sigma. \sigma e \approx \sigma t \not\Rightarrow e \approx t$ . The lack of  $\beta_x$  is just an example of how this problem shows up in reasoning. We stress that the real challenge is this more general problem with substitutions because, unfortunately,  $\beta_x$  is not only an illustrative example but also a tempting straw man. Seeing  $\beta_x$  alone, one may think that its unsoundness is some idiosyncrasy that can be fixed by modifying the calculus. For example, type systems can easily recover  $\beta_x$  by banishing all stuck terms including  $\beta_x$  redexes. But this little victory over  $\beta_x$  does not justify reasoning under substitutions, and how or whether we can achieve the latter is a much more difficult question. It is



unclear if any type systems justify reasoning under substitutions in general, and it is even less clear how to prove that.

Surveying which refinements (including, but not limited to the addition of type systems) for  $\lambda^U$  let us reason under substitutions and why is an important topic for future study, but it is beyond the scope of this work. Here, we focus instead on showing that we can achieve a lot without committing to anything more complicated than  $\lambda^U$ . In particular, we will show with applicative bisimulation (chapter 5) that the lack of  $\beta_x$  is not a large drawback after all, as a refined form of  $\beta_x$  can be used instead:

$$(C\beta_x) \quad \lambda x.C[(\lambda y.e^0) x] = \lambda x.C[[x/y]e^0] ,$$

with the side conditions that  $C[(\lambda y.e^0) x], C[[x/y]e^0] \in E^0$  and that  $C$  does not shadow the binding of  $x$ . Intuitively, given just the term  $(\lambda y.e^0) x$ , we cannot tell if  $x$  is well-leveled, i.e., bound at a lower level than its use, so that a value is substituted for  $x$  before evaluation can reach it.  $C\beta_x$  remedies this problem by demanding a well-leveled binder. As a special case,  $\beta_x$  is sound for any subterm in the erasure of a closed term—that is, the erasure of any self-contained generator.

## Chapter 4

### Effects of Staging on Correctness

This chapter presents the Erasure Theorem and its utility in proving  $e \approx \|e\|$ , i.e. that staging annotations in a given term are irrelevant to what it computes. This theorem shows that, in essence, the only aspect of a term's meaning that staging annotations can alter is termination. Hence the main question for the correctness of staging is whether the staged term and its erasure agree on termination behavior. This insight provides termination conditions that guarantee  $e \approx \|e\|$ .

The theorem also demonstrates that the effects of erasure are more subtle in CBV than in CBN. In CBV,  $\beta$  reduction is invalid and only its restriction to  $\beta_v$  is sound, but staging can ostensibly make any CBN  $\beta$  reduction  $(\lambda x.e^0) t^0 \longrightarrow [t^0/x]e^0$  appear as CBV by forcing the argument to be a value, like  $(\lambda x.e^0) \langle t^0 \rangle \longrightarrow [\langle t^0 \rangle/x]e^0$ . Erasing annotations could invalidate such reductions in CBV; by contrast, CBN's  $\beta$  reduction is robust against erasure. For this reason, CBN is easier to reason about, and we derive a simpler correctness condition for CBN. In this respect, this result shows that MSP is a particularly good match for lazy languages like Haskell.

Uses of a very similar theorem was pioneered by Yang [37] for languages that omit run, but his version would make guarantees only if the code generated by a staged program is used in a context that has no further staging. For this reason, Yang's result had limited applicability to reasoning about multi-stage programs used in multi-stage languages. By recasting his ideas in terms of the equational theory, this thesis makes them work with arbitrary contexts in  $\lambda^U$ .

## 4.1 Theorem Statement

The theorem statement differs for CBN and CBV, with CBV being more subtle. We will present CBN first. Intuitively, all that staging annotations do is to describe and enforce an evaluation strategy—they may enforce CBN, CBV, or some other strategy that the programmer wants. But CBN reduction can simulate any strategy because it allows the redex to be chosen from anywhere.\* Thus, erasure commutes with CBN reductions (Figure 4.1(a)). The same holds for provable equalities.

**Lemma 24.**  $e \in E^\ell \implies \|e\| \in E^\ell$ .

*Proof.* Straightforward induction on  $e$ . □

**Lemma 25.**  $\| \|t\|/x \|e\| \equiv \|[t/x]e\|$ .

*Proof.* Straightforward induction on  $e$ . □

**Lemma 26.** If  $\|e\| \longrightarrow t$  then  $t \equiv \|t\|$ .

*Proof.* Straightforward induction on the reduction judgment using Lemma 25. □

**Theorem 27** (CBN Erasure). If  $\lambda_{\mathbf{n}}^U \vdash e \longrightarrow^* t$  then  $\lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* \|t\|$ . Also, if  $\lambda_{\mathbf{n}}^U \vdash e = t$  then  $\lambda_{\mathbf{n}}^U \vdash \|e\| = \|t\|$ .

*Proof.* By induction on the length of the reduction, we only need to prove this theorem for one-step reduction  $e \longrightarrow t$ . Decomposing this reduction as  $C[r] \longrightarrow C[d]$  where  $r$  is a redex, all that needs to be shown is  $\|r\| \longrightarrow^* \|d\|$ . For then  $\|C[r]\| \equiv$

---

\*This only means that reductions under exotic evaluation strategies are semantics-preserving rewrites under CBN semantics. CBN evaluators may not actually perform such reductions unless forced by staging annotations.

$(\|C\|)(\|r\|) \longrightarrow^* (\|C\|)(\|d\|) \equiv \|C[d]\|$ , where  $\|C\|$  is defined by adding  $\|\bullet\| \equiv \bullet$  to the rules for erasing terms.

[If  $r \equiv \sim\langle d \rangle$  or  $!\langle d \rangle$ ]  $\|r\| \equiv \|d\|$ .

[If  $r \equiv (\lambda x.e_1) e_2$  for some  $e_1, e_2 \in E^0$ ]  $\|r\| \equiv (\lambda x.\|e_1\|) \|e_2\| \longrightarrow [\|e_2\|/x]\|e_1\| \equiv [e_1/x]e_2 \equiv \|d\|$ , where the reduction holds by Lemma 24 and the syntactic equality following the reduction is by Lemma 25.

[If  $r \equiv c d$  for some  $(c, d) \in \text{dom } \delta$ ] Recall the assumption that  $\|\delta(c, d)\| \equiv \delta(c, d)$ .

Thus erasure commutes with CBN reduction. The statement involving  $(=)$  follows by the Church-Rosser property.  $\square$

This theorem gives useful insights on what staging annotations can or cannot do in CBN. For example, staging preserves return values up to erasure if those values exist, while any term has a more terminating erasure unless the former's external interface (i.e., set of possible return values) contains staging.

**Corollary 28.** If  $u, v \in V^0$  and  $(\lambda_{\mathbf{n}}^U \vdash e \longrightarrow^* u \wedge \lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* v)$ , then  $v \equiv \|v\|$  and  $\lambda_{\mathbf{n}}^U \vdash \|u\| = v$ .

**Corollary 29.** If  $\lambda_{\mathbf{n}}^U \vdash e \Downarrow^\ell \|v\|$ , then  $\lambda_{\mathbf{n}}^U \vdash \|e\| \Downarrow^\ell \|v\|$ .

How does Theorem 27 help prove equivalences of the form  $e \approx \|e\|$ ? The theorem gives a simulation of reductions from  $e$  by reductions from  $\|e\|$ . If  $e$  reduces to an unstaged term  $\|t\|$ , then simulating that reduction from  $\|e\|$  gets us to  $\| \|t\| \|$ , which is just  $\|t\|$ ; thus  $e \longrightarrow^* \|t\| \longleftarrow^* \|e\|$  and  $e = \|e\|$ . Amazingly, this witness  $\|t\|$  can be *any* reduct of  $e$ , as long as it is unstaged! In fact, by Church-Rosser, any  $t$  with  $e = \|t\|$  will do. So staging is correct (i.e., semantics-preserving, or  $e \approx \|e\|$ ) if we can find this  $\|t\|$ . As we will show in Chapter 4.2, this search boils down to a termination check on the generator.

$$\begin{array}{ccc}
\lambda_{\mathbf{n}}^U \vdash e \longrightarrow^* t & \lambda_{\mathbf{v}}^U \vdash e \longrightarrow^* t & \lambda_{\mathbf{v}}^U \vdash e \stackrel{\lambda_{\mathbf{n}}^U}{\top} \equiv c \\
\parallel\!-\!\parallel \downarrow & \parallel\!-\!\parallel \downarrow & \parallel \quad \parallel \\
\lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* \|t\| & \lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* \|t\| & \lambda_{\mathbf{v}}^U \vdash \|e\| \equiv d \\
\text{(a) CBN erasure.} & \text{(b) CBV erasure.} & \text{(c) CBV correctness lemma.}
\end{array}$$

Figure 4.1 : Visualizations of the Erasure Theorem and the derived correctness lemma.

**Lemma 30** (CBN Correctness).  $(\exists t. \lambda_{\mathbf{n}}^U \vdash e = \|t\|) \implies \lambda_{\mathbf{n}}^U \vdash e = \|e\|$ .

CBV satisfies a property similar to Theorem 27, but the situation is more subtle. Staging modifies the evaluation strategy in CBV as well, but not all of them can be simulated in the erasure by CBV reductions, for  $\beta_{\mathbf{v}}$  reduces only a subset of  $\beta$  redexes. For example, if  $\Omega \in E^0$  is divergent, then  $(\lambda_{-}.0) \langle \Omega \rangle \longrightarrow 0$  in CBV, but the erasure  $(\lambda_{-}.0) \Omega$  does not CBV-reduce to 0 since  $\Omega$  is not a value. However, it is the case that  $\lambda_{\mathbf{n}}^U \vdash (\lambda_{-}.0) \Omega \longrightarrow 0$  in CBN. In general, erasing CBV reductions gives CBN reductions (Figure 4.1(b)).

**Theorem 31** (CBV Erasure). If  $\lambda_{\mathbf{v}}^U \vdash e \longrightarrow^* t$  then  $\lambda_{\mathbf{n}}^U \vdash \|e\| \longrightarrow^* \|t\|$ . Also, if  $\lambda_{\mathbf{v}}^U \vdash e = t$  then  $\lambda_{\mathbf{n}}^U \vdash \|e\| = \|t\|$ .

This theorem has similar ramifications as the CBN Erasure Theorem, but with the caveat that they conclude in CBN despite having premises in CBV. In particular, if  $e$  is CBV-equal to an erased term, then  $e = \|e\|$  in CBN.

**Corollary 32.**  $(\exists t. \lambda_{\mathbf{v}}^U \vdash e = \|t\|) \implies \lambda_{\mathbf{n}}^U \vdash e = \|e\|$ .

CBN equalities given by this corollary may at first seem irrelevant to CBV programs, but in fact if we show that  $e$  and  $\|e\|$  CBV-reduce to constants, then the CBN equality can be safely cast to CBV equality. Figure 4.1(c) summarizes this reasoning. Given  $e$ , suppose we found some  $c, d$  that satisfy the two horizontal CBV equalities.

Then from the top equality, Corollary 32 gives the left vertical one in CBN. As CBN equality subsumes CBV equality, tracing the diagram counterclockwise from the top right corner gives  $\lambda_{\mathbf{n}}^U \vdash c = d$  in CBN. Then the right vertical equality  $c \equiv d$  follows by the Church-Rosser property in CBN. Tracing the diagram clockwise from the top left corner gives  $\lambda_{\mathbf{v}}^U \vdash e = \|e\|$ .

**Lemma 33** (CBV Correctness). If  $\lambda_{\mathbf{v}}^U \vdash e = c$  and  $\lambda_{\mathbf{v}}^U \vdash \|e\| = d$ , then  $\lambda_{\mathbf{v}}^U \vdash e = \|e\|$ .

Thus, we can prove  $e = \|e\|$  in CBV by showing that each side terminates to some constant, *in CBV*. Though we borrowed CBN facts to derive this lemma, the lemma itself leaves no trace of CBN reasoning.

## 4.2 Example: Erasing Staged Power

This section demonstrates how to apply the Erasure Theorem to **stpow**. First, some technicalities: MetaOCaml's constructs are interpreted in  $\lambda^U$  in the standard manner, e.g., **let**  $x = e$  **in**  $t$  stands for  $(\lambda x.t) e$  and **let rec**  $f x = e$  stands for **let**  $f = \Theta(\lambda f.\lambda x.e)$  where  $\Theta$  is some fixed-point combinator. We assume the *Const* set in  $\lambda^U$  has integers and booleans, with a suitable definition of  $\delta$ . For conciseness, we treat top-level bindings **genpow** and **stpow** like macros, so  $\|\mathbf{stpow}\|$  is the erasure of the recursive function to which **stpow** is bound with **genpow** inlined, not the erasure of a variable named **stpow**.

As a caveat, we might want to prove  $\mathbf{stpow} \approx \mathbf{power}$  but this goal is not quite right. The whole point of **stpow** is to process the first argument without waiting for the second, so it can disagree with **power** when partially applied, e.g.,  $\mathbf{stpow} \ 0 \ \uparrow^0$  but  $\mathbf{power} \ 0 \ \Downarrow^0$ . We sidestep this issue for now by concentrating on positive arguments, and discuss divergent cases in Chapter 5.2.

To prove  $k > 0 \implies \mathbf{stpow} \ k = \mathbf{power} \ k$  for CBN, we only need to check that the code generator  $\mathbf{genpow} \ k$  terminates to some  $\cdot\langle\|e\|\rangle\cdot$ ; then the  $\cdot!$  in  $\mathbf{stpow}$  will take out the brackets and we have the witness required for Lemma 30. To say that something terminates to  $\cdot\langle\|e\|\rangle\cdot$  roughly means that it is a two-stage program, which is true for almost all uses of MSP that we are aware of. This use of the Erasure Theorem is augmented by the observation  $\|\mathbf{stpow}\| = \mathbf{power}$ —these functions are not syntactically equal, the former containing an  $\eta$  redex.

**Lemma 34.**  $\lambda_n^U \vdash \|\mathbf{stpow}\| = \mathbf{power}$

*Proof.* Contract the  $\eta$  expansion by (CBN)  $\beta$ . □

**Proposition 35** (Erasing CBN Power).  $\forall k \in \mathbb{Z}^+. \lambda_n^U \vdash \mathbf{stpow} \ k = \mathbf{power} \ k$ .

*Proof.* Induction on  $k$  gives some  $e$  s.t.  $\mathbf{genpow} \ k \cdot\langle x \rangle\cdot = \cdot\langle\|e\|\rangle\cdot$ , so

$$\begin{aligned} \mathbf{stpow} \ k &= \cdot!\cdot\langle\mathbf{fun} \ x \rightarrow \cdot\tilde{\cdot}(\mathbf{genpow} \ k \cdot\langle x \rangle\cdot)\rangle\cdot \\ &= \cdot!\cdot\langle\mathbf{fun} \ x \rightarrow \cdot\tilde{\cdot}\langle\|e\|\rangle\cdot\rangle\cdot \\ &= \cdot!\cdot\langle\mathbf{fun} \ x \rightarrow \cdot\|e\|\rangle\cdot \\ &= \mathbf{fun} \ x \rightarrow \cdot\|e\|\cdot \end{aligned}$$

hence  $\mathbf{stpow} \ k = \|\mathbf{stpow}\| \ k = \mathbf{power} \ k$  by Lemmas 30 and 34. □

The proof for CBV is similar, but we need to fully apply both  $\mathbf{stpow}$  and its erasure to confirm that they both reach some constant. The beauty of Lemma 33 is that we do not have to know what those constants are. Just as in CBN, the erasure  $\|\mathbf{stpow}\|$  is equivalent to  $\mathbf{power}$ , but note this part of the proof uses  $C\beta_x$ .

**Lemma 36.**  $\lambda_n^U \vdash \|\mathbf{stpow}\| \approx \mathbf{power}$

*Proof.* Contract the  $\eta$  expansion by  $C\beta_x$ . □

**Proposition 37** (Erasing CBV Power). For  $k \in \mathbb{Z}^+$  and  $m \in \mathbb{Z}$ ,  $\lambda_v^U \vdash \mathbf{stpow} \ k \ m \approx \mathbf{power} \ k \ m$ .

*Proof.* We stress that this proof works entirely with CBV equalities; we have no need to deal with CBN once Lemma 33 is established. By induction on  $k$ , we prove that  $\exists e. \mathbf{genpow} \ k \ .\langle x \rangle. = .\langle \|e\| \rangle.$  and  $[m/x]\|e\| \Downarrow^0 m'$  for some  $m' \in \mathbb{Z}$ . We can do so without explicitly figuring out what  $\|e\|$  looks like. The case  $k = 1$  is easy; for  $k > 1$ , the returned code is  $.\langle x * \|e'\| \rangle.$  where  $[m/x]\|e'\|$  terminates to an integer by inductive hypothesis, so this property is preserved. Then

$$\begin{aligned} \mathbf{stpow} \ k \ m &= .!. \langle \mathbf{fun} \ x \rightarrow .\sim(\mathbf{genpow} \ k \ .\langle x \rangle.) \rangle. m \\ &= .!. \langle \mathbf{fun} \ x \rightarrow \|e\| \rangle. m \\ &= [m/x]\|e\| = m' \in \mathit{Const}. \end{aligned}$$

Clearly  $\mathbf{power} \ k \ m$  terminates to a constant. By Lemma 36,  $\|\mathbf{stpow}\| \ k \ m$  also yields a constant, so by Lemma 33,  $\mathbf{stpow} \ k \ m = \|\mathbf{stpow}\| \ k \ m \approx \mathbf{power} \ k \ m$ .  $\square$

These proofs illustrate our answer to the erasure question in the introduction. Erasure is semantics-preserving if the generator terminates to  $\langle \|e\| \rangle$  in CBN, or if the staged and unstaged terms terminate to constants in CBV. Showing the latter requires propagating type information and a termination assertion for the generated code. Type information would come for free in a typed system, but it can be easily emulated in an untyped setting. Hence we see that correctness of staging generally reduces to termination not just in CBN but also in CBV—in fact, the correctness proof is essentially a modification of the termination proof.



### 4.3 Why CBN Facts are Necessary for CBV Reasoning

So far, we have let erasure map CBV equalities to the superset of CBN equalities and performed extra work to show that the particular CBN equalities we derived hold in CBV as well. One might instead try to find a subset of CBV reductions that erase to CBN reductions, which is essentially how Yang [37] handled CBV erasure. In this section we will show that this alternative approach does work, but only in simple cases.

As discussed before, the problem with erasing CBV reductions is that the argument in a  $\beta_v$  redex may have a divergent erasure. To eliminate this case, we might restrict  $\beta_v$  to a “careful” variant with a side condition, like

$$(\beta_{v\Downarrow}) \quad (\lambda x.e^0) v^0 = [v^0/x]e^0 \text{ provided } \lambda_v^U \vdash \|v^0\| \Downarrow^0.$$

If we define a new set of axioms  $\lambda_{v\Downarrow}^U \stackrel{\text{def}}{=} \{\beta_{v\Downarrow}, E_U, R_U, \delta\}$  then reductions (hence equalities) under this axiom set erase to CBV reductions. But  $\beta_{v\Downarrow}$  is much too crude. It prohibits contracting redexes of the form  $(\lambda y.e^0) \langle x \rangle$  (note  $x \Uparrow^0$ ), which are ubiquitous—a function as simple as `stpow` already contains one.

Observe that the erasure  $x$  of the argument in a  $\beta_x$  redex would terminate under a substitution. As discussed in Chapter 3.4, introducing a substitution to equalities can be a point of no return, but let us plow ahead and worry about that problem later. Allowing substitutions in the check  $\|v^0\| \Downarrow^0$  for  $\beta_{v\Downarrow}$  gives a refined  $\lambda_{v\Downarrow}^U$ :

**Definition 38** (Careful Equality). Let  $\sigma : \text{Var} \xrightarrow{\text{fin}} V^0$  be a substitution by not necessarily closed values. A CBV provable equality is *careful modulo*  $\sigma$ , written  $\lambda_{v\Downarrow}^U/\sigma \vdash e = t$ , iff it can be deduced from the axioms  $E_U, R_U, \delta$ , and

$$(\beta_{v\Downarrow}/\sigma) \quad (\lambda x.e^0) v^0 = [v^0/x]e^0 \text{ provided } \lambda_v^U \vdash \sigma\|v^0\| \Downarrow^0$$

through the inference rules of reflexivity, symmetry, transitivity, and *constrained* com-

patible extension:  $C[e] = C[t]$  can be deduced from  $e = t$  iff all variables captured (bound) by  $C$  are fresh for  $\sigma$ . A careful equality is a careful reduction iff its derivation uses only constrained compatible extension.

Careful reductions erase to  $\lambda_{\mathbf{v}}^U$ -equality, albeit under substitutions. The conclusion is equality and not reduction because the simulation of  $\beta_{\mathbf{v}\Downarrow}$  reduction on  $(\lambda x.e^0) v^0$  needs reverse reductions: we have  $\|v^0\| \Downarrow^0 u^0$  for some  $u^0$ , so  $(\lambda x.\|e^0\|) \|v^0\| \longrightarrow^* (\lambda x.\|e^0\|) u^0 \longrightarrow [u^0/x]\|e^0\| \longleftarrow^* [\|v^0\|/x]\|e^0\|$ .

$$\begin{array}{ccc} \lambda_{\mathbf{v}\Downarrow}^U/\sigma \vdash e & \longrightarrow^* & t \\ \sigma\|-\| \downarrow & & \downarrow \sigma\|-\| \\ \lambda_{\mathbf{v}}^U \vdash \sigma\|e\| & = & \sigma\|t\| \end{array}$$

**Theorem 39** (Careful Erasure).  $\lambda_{\mathbf{v}\Downarrow}^U/\sigma \vdash e \longrightarrow^* t \implies \lambda_{\mathbf{v}}^U \vdash \sigma e = \sigma t$  and  $\lambda_{\mathbf{v}\Downarrow}^U/\sigma \vdash e = t \implies \lambda_{\mathbf{v}}^U \vdash \sigma\|e\| = \sigma\|t\|$ .

*Proof.* By induction on the length of the reduction, it suffices to prove  $\lambda_{\mathbf{v}\Downarrow}^U/\sigma \vdash e \longrightarrow t \implies \lambda_{\mathbf{v}}^U \vdash \sigma e = \sigma t$ . First consider primitive reductions, i.e. those which derive from an axiom directly and without applying any inference rule.

If  $e \longrightarrow t$  is an  $E_U$ ,  $R_U$ , or  $\delta$  reduction, then  $\lambda_{\mathbf{v}}^U \vdash e \longrightarrow t$  by the same argument as Theorem 27. Then repeating the argument in Proposition 18 gives  $\lambda_{\mathbf{v}}^U \vdash \sigma e \longrightarrow \sigma t$ .

The interesting case is  $\beta_{\mathbf{v}\Downarrow}/\sigma$  reduction, where we have  $(\lambda x.e^0) v^0 \longrightarrow [v^0/x]e^0$  and  $\lambda_{\mathbf{v}}^U \vdash \sigma\|v^0\| \Downarrow^0 u^0$ , and we must show  $\lambda_{\mathbf{v}}^U \vdash \sigma((\lambda x.\|e^0\|) \|v^0\|) = \sigma[\|v^0\|/x]\|e^0\|$ . By Theorem 7 we get  $\lambda_{\mathbf{v}}^U \vdash \sigma\|v^0\| = u^0$ , so

$$\begin{aligned} \lambda_{\mathbf{v}}^U \vdash \sigma((\lambda x.\|e^0\|) \|v^0\|) &= (\lambda x.\sigma\|e^0\|) u^0 \\ &= [u^0/x](\sigma\|e^0\|) \\ &= [\sigma\|v^0\|/x](\sigma\|e^0\|) \\ &\equiv \sigma([\|v^0\|/x]\|e^0\|) \end{aligned}$$

where we used the fact that by Barendregt's variable convention [3],  $x$  is fresh for  $\sigma$ , so we can freely commute  $\sigma$  with the binder  $\lambda x$ . The rightmost term is just  $\sigma\llbracket v^0/x \rrbracket e^0\rrbracket$  by Lemma 25.

For a careful equality derived by constrained compatible extension  $\lambda_{\mathbf{v}\downarrow}^U/\sigma \vdash C[e] \longrightarrow C[t]$ , note

$$\sigma(\llbracket C[e] \rrbracket) \equiv \sigma(\llbracket C \rrbracket \llbracket e \rrbracket) \equiv (\sigma \llbracket C \rrbracket) [\sigma \llbracket e \rrbracket]$$

and likewise for  $C[t]$ , using the constraint that all bindings in  $C$  are fresh for  $\sigma$ . Since we have already shown  $\lambda_{\mathbf{v}}^U \vdash \sigma \llbracket e \rrbracket = \sigma \llbracket t \rrbracket$ , we have

$$\lambda_{\mathbf{v}}^U \vdash (\sigma \llbracket C \rrbracket) [\sigma \llbracket e \rrbracket] = (\sigma \llbracket C \rrbracket) [\sigma \llbracket t \rrbracket]$$

by (unconstrained) compatible extension. □

Now, can we justify the substitution in careful equalities? In the **power** example, the expected way to use the generator is **stpow**  $n$   $k$ , where  $n, k \in \mathbb{Z} \subseteq \text{Const}$ . This application reduces to

$$.!.<\mathbf{fun} \ x \rightarrow .\tilde{(\text{genpow } n \ .<\mathbf{x}>.)}>. \ k$$

Note two things: **genpow**  $n$   $.<\mathbf{x}>.$  reduces carefully for  $[k/\mathbf{x}]$ , and there is a  $k$  waiting outside the  $.!.<\dots>.$  to be substituted. All that it takes to justify reasoning under  $[k/\mathbf{x}]$  is to show that  $k$  is eventually pulled into the body of the **fun**  $\mathbf{x} \rightarrow \dots$  by  $\beta_{\mathbf{v}}$ .

**Lemma 40.** The following implication holds, as well as its evident generalization to  $n$  values and variables.

$$\lambda_{\mathbf{v}\downarrow}^U/[v/x] \vdash e = \langle \llbracket t \rrbracket \rangle \implies \lambda_{\mathbf{v}}^U \vdash !\langle \lambda x. \tilde{e} \rangle v = (\lambda x. \llbracket e \rrbracket) v.$$

*Proof.* Let  $\sigma \equiv [v/x]$ . The premise  $\lambda_{\mathbf{v}\downarrow}^U/\sigma \vdash e = \langle \llbracket t \rrbracket \rangle$  separately implies two halves of the desired equality. Firstly, it implies  $\lambda_{\mathbf{v}}^U \vdash e = \langle \llbracket t \rrbracket \rangle$  because  $\lambda_{\mathbf{v}\downarrow}^U/\sigma$  is a restriction

of  $\lambda_{\mathbf{v}}^U$ . Therefore

$$\lambda_{\mathbf{v}}^U \vdash !\langle \lambda x. \tilde{e} \rangle v = !\langle \lambda x. \tilde{\langle \|t\| \rangle} \rangle v = (\lambda x. \|t\|) v = [v/x] \|t\|.$$

Secondly, the same premise implies by Theorem 39 that

$$\lambda_{\mathbf{v}}^U \vdash [v/x] \|t\| = [v/x] \|e\| = (\lambda x. \|e\|) v.$$

Pasting together these chains of equalities proves the lemma.  $\square$

*Alternative Proof of Proposition 37.* By induction on  $k$ , there exists some  $e$  such that  $\lambda_{\mathbf{v}\downarrow}^U/[m/\mathbf{x}] \vdash \mathbf{genpow} \ k. \langle \mathbf{x} \rangle. = .\langle \|e\| \rangle.;$  apply Lemma 40.  $\square$

When it works, careful erasure is more convenient than the approach that exploits CBN; but careful erasure does not handle nested binders well and is unsuitable for certain generators, including the longest common subsequence example in chapter 6. Writing  $\mathbf{let} \ x = e \ \mathbf{in} \ t$  as a shorthand for  $(\lambda x. t) \ e$  as usual,

$$.!. \langle \mathbf{let} \ y = 0 \ \mathbf{in} \ \mathbf{let} \ x = y \ \mathbf{in} \ .\tilde{((\lambda z. z) \ .\langle x+y \rangle.})} \rangle.$$

is clearly equivalent to its erasure. To prove this fact, we might observe that

$$\lambda_{\mathbf{v}}^U \vdash [0/y](\mathbf{let} \ x = y \ \mathbf{in} \ (\lambda z. z) \ (x+y)) = [0/y](\mathbf{let} \ x = y \ \mathbf{in} \ x+y)$$

and expect

$$\lambda_{\mathbf{v}\downarrow}^U/[0/y] \vdash \mathbf{let} \ x = y \ \mathbf{in} \ .\tilde{((\lambda z. z) \ .\langle x+y \rangle.})} = \mathbf{let} \ x = y \ \mathbf{in} \ x+y$$

but this does not hold. For  $(\lambda z. z) \ .\langle x+y \rangle.$  to reduce carefully, we need  $[0, 0/x, y]$  and not  $[0/y]$ ; however,

$$\lambda_{\mathbf{v}\downarrow}^U/[0, 0/x, y] \vdash \mathbf{let} \ x = y \ \mathbf{in} \ .\tilde{((\lambda z. z) \ .\langle x+y \rangle.})} = \dots$$

is incorrect. The  $x$  in the proof system  $\lambda_{\mathbf{v}\downarrow}^U/[0, 0/x, y]$  must be distinct from the  $x$  bound in the object term, or else the proof system would violate  $\alpha$  equivalence.

The problem here is that we must reason under different substitutions in different

scopes, and it is tricky to propagate the results obtained under  $\lambda_{\mathbf{v}\Downarrow}^U/\sigma$  to an outer context where some variables in  $\sigma$  may have gone out of scope. While it may not be possible to pull off the bookkeeping, we find ourselves fighting against hygiene rather than exploiting it. For this reason restricting CBV reductions is unsatisfactory, and appealing to CBN reasoning results in a much simpler approach to handling nested binders.

## Chapter 5

### Extensional Reasoning for $\lambda^V$

This section presents applicative bisimulation [1, 15], a well-established tool for analyzing higher-order functional programs. Bisimulation is sound and complete for  $(\approx)$ , and justifies  $C\beta_x$  (Chapter 3.4) and extensionality, allowing us to handle the divergence issues ignored in Chapter 4.2.

#### 5.1 Proof by Bisimulation

Intuitively, for a pair of terms to applicatively bisimulate, they must both terminate or both diverge, and if they terminate, their values must bisimulate again under experiments that examine their behavior. In an experiment, functions are called, code values are run, and constants are left untouched. Effectively, this is a bisimulation under the transition system consisting of evaluation ( $\Downarrow$ ) and experiments. If  $eRt$  implies that either  $e \approx t$  or  $e, t$  bisimulate, then  $R \subseteq (\approx)$ .

**Definition 41** (Relation Under Experiment). Given a relation  $R \subseteq E \times E$ , let  $\tilde{R} \stackrel{\text{def}}{=} R \cup (\approx)$ . For  $\ell > 0$  set  $u R_{\dagger}^{\ell} v$  iff  $u \tilde{R} v$ . For  $\ell = 0$  set  $u R_{\dagger}^0 v$  iff either:

- $u \equiv v \in \text{Const}$ ,
- $u \equiv \lambda x.e$  and  $v \equiv \lambda x.t$  for some  $e, t$  s.t.  $\forall a. ([a/x]e) \tilde{R} ([a/x]t)$ , or
- $u \equiv \langle e \rangle$  and  $v \equiv \langle t \rangle$  for some  $e, t$  s.t.  $e \tilde{R} t$ .

**Definition 42** (Applicative Bisimulation). An  $R \subseteq E \times E$  is an *applicative bisimulation* iff every pair  $(e, t) \in R$  satisfies the following: let  $\ell = \max(\text{lv } e, \text{lv } t)$ ; then for any

finite substitution  $\sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  we have  $\sigma e \Downarrow^\ell \iff \sigma t \Downarrow^\ell$ , and if  $\sigma e \Downarrow^\ell u \wedge \sigma t \Downarrow^\ell v$  then  $u R_\dagger^\ell v$ .

**Theorem 43.** Given  $R \subset E \times E$ , define  $R^\bullet \stackrel{\text{def}}{=} \{(\sigma e, \sigma t) : e R t, (\sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg})\}$ . Then  $R \subseteq (\approx)$  iff  $R^\bullet$  is an applicative bisimulation.

This is our answer to the extensional reasoning question in the introduction: this theorem shows that bisimulation can in principle derive *all* valid equivalences, including all extensional facts. Unlike in single-stage languages [1, 18, 15],  $\sigma$  ranges over non-closing substitutions, which may not substitute for all variables or may substitute open terms. Closing substitutions are unsafe since  $\lambda^U$  has open-term evaluation. But for CBV, bisimulation gives a condition under which substitution is safe, i.e., when the binder is at level 0 (in the definition of  $\lambda x.e R_\dagger^0 \lambda x.t$ ). In CBN this is not an advantage as  $\forall a.[a/x]e \tilde{R}[a/x]t$  entails  $[x/x]e \tilde{R}[x/x]t$ , but bisimulation is still a more approachable alternative to  $(\approx)$ .

The importance of the substitution in  $\lambda x.e R_\dagger^0 \lambda x.t$  for CBV is best illustrated by the proof of extensionality, from which we get  $C\beta_x$  introduced in Chapter 3.4.

**Proposition 44.** If  $e, t \in E^0$  and  $\forall a. (\lambda x.e) a \approx (\lambda x.t) a$ , then  $\lambda x.e \approx \lambda x.t$ .

*Proof.* Take  $R \stackrel{\text{def}}{=} \{(\lambda x.e, \lambda x.t)\}^\bullet$ . To see that  $R$  is a bisimulation, fix  $\sigma$ , and note that  $\sigma \lambda x.e, \sigma \lambda x.t$  terminate to themselves at level 0. By Barendregt's variable convention [3],  $x$  is fresh for  $\sigma$ , thus  $\sigma \lambda x.e \equiv \lambda x.\sigma e$  and  $\sigma \lambda x.t \equiv \lambda x.\sigma t$ . We must check  $[a/x]\sigma e \approx [a/x]\sigma t$ : by assumption and by Proposition 18 we get  $\sigma[a/x]e \approx \sigma[a/x]t$ , and one can show that  $\sigma$  and  $[a/x]$  commute modulo  $(\approx)$  (see Appendix C.5). Hence by Theorem 43,  $\lambda x.e \approx \lambda x.t$ .  $\square$

**Corollary 45** (Soundness of  $C\beta_x$ ). If  $C[(\lambda y.e^0) x], C[[x/y]e^0] \in E^0$  and  $C$  does not bind  $x$ , then  $\lambda x.C[(\lambda y.e^0) x] \approx \lambda x.C[[x/y]e^0]$ .

*Proof.* Apply both sides to an arbitrary  $a$  and use Proposition 44 with  $\beta/\beta_v$ .  $\square$

The proof of Proposition 44 would have failed in CBV had we defined  $\lambda x.e \ R_{\dagger}^0 \lambda x.t \iff e \tilde{R}t$ , without the substitution. For when  $e \equiv (\lambda_.0) \ x$  and  $t \equiv 0$ , the premise  $\forall a.[a/x]e \approx [a/x]t$  is satisfied but  $e \not\approx t$ , so  $\lambda x.e$  and  $\lambda x.t$  do not bisimulate with this weaker definition. The binding in  $\lambda x.e \in E^0$  is guaranteed to be well-leveled, and exploiting it by inserting  $[a/x]$  in the comparison is strictly necessary to get a complete (as in “sound and complete”) notion of bisimulation.

Howe’s method [18] is used to prove Theorem 43, but adapting this method to  $\lambda^U$  is surprisingly tricky because  $\lambda^U$ ’s bisimulation must handle substitutions inconsistently: in Definition 42 we cannot restrict our attention to  $\sigma$ ’s that substitute away any particular variable, but in Definition 41, for  $\lambda x.e \ R_{\dagger}^0 \lambda x.t$ , we must restrict our attention to the case where substitution eliminates  $x$ . Proving Theorem 43 entails coinduction on a self-referential definition of bisimulation; however, Definition 41 refers not to the bisimulation whose definition it is a part of, but to a different bisimulation that holds only under substitutions that eliminate  $x$ . To solve this problem, we recast bisimulation to a family of relations indexed by a set of variables to be eliminated, so that the analogue of Definition 41 can refer to a different member of the family. Theorem 43 is then proved by mutual coinduction.

**Remark.** Extensionality is a common addition to the equational theory for the plain  $\lambda$  calculus, usually called the  $\omega$  rule [26, 19]. But unlike  $\omega$  in the plain  $\lambda$  calculus,  $\lambda^U$  functions must agree on open-term arguments as well. This is no surprise since  $\lambda^U$  functions do receive open arguments during program execution. However, we know of no specific functions that fail to be equivalent because of open arguments. Whether extensionality can be strengthened to require equivalence only under closed arguments is an interesting open question.



**Remark.** The only difference between Definition 42 and applicative bisimulation in the plain  $\lambda$  calculus is that Definition 42 avoids applying closing substitutions. Given that completeness can be proved for this bisimulation, it seems plausible that the problem with reasoning under substitutions is the only thing that makes conservativity fail. Hence it seems that for *closed* unstaged terms,  $\lambda^U$ 's ( $\approx$ ) could actually coincide with that of the plain  $\lambda$  calculus. Such a result would make a perfect complement to the Erasure Theorem, for it lets us completely forget about staging when reasoning about an erased program. We do not have a proof of this conjecture, however. Conservativity is usually proved through a denotational semantics, which is notoriously difficult to devise for hygienic MSP. It will at least deserve separate treatment from this paper.

## 5.2 Example: Tying Loose Ends on Staged Power

In Chapter 4.2, we sidestepped issues arising from the fact that `stpow`  $0 \uparrow^0$  whereas `power`  $0 \Downarrow^0$ . If we are allowed to modify the code, this problem is usually easy to avoid, for example by making `power` and `genpow` terminate on non-positive arguments. If not, we can still persevere by finessing the statement of correctness. The problem is partial application, so we can force `stpow` to be fully applied before it executes by stating `power`  $\approx \lambda n.\lambda x.\text{stpow } n \ x$ .

**Lemma 46.** Let  $e' \approx_{\uparrow} t'$  mean  $e' \approx t' \vee (\sigma e' \uparrow^{\ell} \wedge \sigma t' \uparrow^{\ell})$  where  $\ell = \max(\text{lv } e', \text{lv } t')$ . For a fixed  $e, t$ , if for every  $\sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  we have  $\sigma e \approx_{\uparrow} \sigma t$ , then  $e \approx t$ .

*Proof.* Notice that  $\{(e, t)\}^{\bullet}$  is an applicative bisimulation. □

**Proposition 47** (CBN `stpow` is Correct).  $\lambda_n^U \vdash \text{power} \approx \lambda n.\lambda x.\text{stpow } n \ x$ .

*Proof.* We just need to show  $\forall e, t \in E^0. \text{power } e \ t \approx_{\uparrow} \text{stpov } e \ t$ , because then  $\forall e, t \in E^0. \forall \sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma(\text{power } e \ t) \approx_{\uparrow} \sigma(\text{stpov } e \ t)$ , whence  $\text{power} \approx \lambda n. \lambda x. \text{stpov } n \ x$  by Lemma 46 and extensionality. So fix arbitrary, potentially open,  $e, t \in E^0$ , and split cases on the behavior of  $e$ . As evident from the following argument, the possibility that  $e, t$  contain free variables is not a problem here.

[If  $e \uparrow^0$  or  $e \Downarrow^0 u \notin \mathbb{Z}^+$ ] Both  $\text{power } e \ t$  and  $\text{stpov } e \ t$  diverge.

[If  $e \Downarrow^0 m \in \mathbb{Z}^+$ ] Using Proposition 35,  $\text{power } e = \text{power } m \approx \text{stpov } m = \text{stpov } e$ ,  
so  $\text{power } e \ t \approx \text{stpov } e \ t$ . □

**Proposition 48** (CBV  $\text{stpov}$  is Correct).  $\lambda_v^U \vdash \text{power} \approx \lambda n. \lambda x. \text{stpov } n \ x$ .

*Proof.* By the same argument as in CBN, we just need to show  $\text{power } u \ v \approx_{\uparrow} \text{stpov } u \ v$  for arbitrary  $u, v \in V^0$ .

[If  $u \notin \mathbb{Z}^+$ ] Both  $\text{power } u \ v$  and  $\text{stpov } u \ v$  get stuck at **if**  $n = 0$ .

[If  $u \in \mathbb{Z}^+$ ] If  $u \equiv 1$ , then  $\text{power } 1 \ v = v = \text{stpov } 1 \ v$ . If  $u > 1$ , we show that the generated code is strict in a subexpression that requires  $v \in \mathbb{Z}$ . Observe that  $\text{genpow } u \ .\langle x \rangle. \Downarrow^0 \ .\langle e \rangle.$  where  $e$  has the form  $\ .\langle x * t \rangle.$  For  $[v/x]e \Downarrow^0$  it is necessary that  $v \in \mathbb{Z}$ . It is clear that  $\text{power } u \ v \Downarrow^0$  requires  $v \in \mathbb{Z}$ . So either  $v \notin \mathbb{Z}$  and  $\text{power } u \ v \uparrow^0$  and  $\text{stpov } u \ v \uparrow^0$ , in which case we are done, or  $v \in \mathbb{Z}$  in which case Proposition 37 applies. □

**Remark.** Real code should not use  $\lambda n. \lambda x. \text{stpov } n \ x$ , as it re-generates and recompiles code upon every invocation. Application programs should always use  $\text{stpov}$ , and one must check (outside of the scope of verifying the function itself) that  $\text{stpov}$  is always eventually fully applied so that the  $\eta$  expansion is benign.

### 5.3 Soundness and Completeness of Applicative Bisimulation

In this section, we will prove Theorem 43. As noted above, the soundness proof of applicative bisimulation (the harder half of Theorem 43) is an adaptation of Howe's [18], and the main issue is to remove the original method's reliance on being able to substitute away free variables. We begin with an overview to motivate the main difference from Howe's formulation, namely our choice to index the bisimilarity relation by sets of variables. This overview will be more technical than the Remark in Chapter 5.1.

#### 5.3.1 Overview

We focus on CBV in this informal overview. In single-stage calculi, observational equivalence is typically the greatest (under inclusion) consistent congruence, i.e., the greatest context-respecting ( $e \sim t \implies C[e] \sim C[t]$ ) equivalence that is a strict subset of  $E \times E$ . Howe [18] gives a framework for showing that the union of all bisimulations ( $\sim$ ) is a nontrivial congruence, from which it follows that  $(\sim) \subseteq (\approx)$ . It is easy to prove that  $(\sim)$  is an equivalence, but not that it is context respecting. For this harder step, Howe defines an auxiliary relation  $e \hat{\sim} t$ , called the *precongruence candidate*, which holds iff  $e$  can be transformed into  $t$  by one bottom-up pass of replacing successively larger subterms  $e'$  of  $e$  by some  $t'$  such that  $e' \sim t'$ :

$$\begin{array}{c}
 \frac{x \sim t}{x \hat{\sim} t} \quad \frac{c \sim t}{c \hat{\sim} t} \quad \frac{e_1 \hat{\sim} s_1 \quad e_2 \hat{\sim} s_2 \quad s_1 \ s_2 \sim t}{e_1 \ e_2 \hat{\sim} t} \quad \frac{e \hat{\sim} s \quad \lambda x.s \sim t}{\lambda x.e \hat{\sim} t} \\
 \frac{e \hat{\sim} s \quad \langle s \rangle \sim t}{\langle e \rangle \hat{\sim} t} \quad \frac{e \hat{\sim} s \quad \tilde{s} \sim t}{\tilde{e} \hat{\sim} t} \quad \frac{e \hat{\sim} s \quad !s \sim t}{!e \hat{\sim} t}
 \end{array}$$

where  $s$  ranges over  $E$ . A deep understanding of how  $e \hat{\sim} t$  works is unnecessary for now. The point is that it is derived by repeatedly replacing subterms  $e'$  of  $e$  with  $t$  such that  $e' \sim t$ , in a way that makes  $(\hat{\sim})$  a context-respecting superset of  $(\sim)$ . Howe proves that  $(\hat{\sim})$  is also a bisimulation, and concludes that  $(\hat{\sim}) = (\sim)$  as  $(\sim)$  contains all bisimulations, hence that  $(\sim)$  respects contexts.

As a part of showing that  $(\hat{\sim})$  is a bisimulation, we need to prove that  $(\lambda x.e) e_a \hat{\sim} (\lambda x.t) t_a$  and  $(\lambda x.e) e_a \Downarrow^0 u$  implies  $\exists v. (\lambda x.t) t_a \Downarrow^0 v$  and  $u \hat{\sim}_{\dagger}^0 v$ , where all terms and values are level 0. Inducting on the number of steps  $(\lambda x.e) e_a$  takes to terminate, we can get the inductive hypothesis

$$e_a \Downarrow^0 u' \qquad t_a \Downarrow^0 v' \qquad u' \hat{\sim}_{\dagger}^0 v'$$

and  $(\lambda x.e) u' \xrightarrow{0}_{\circ} [u'/x]e \Downarrow^0 u$ , where  $[u'/x]e$  terminates in fewer steps than  $(\lambda x.e) u'$ . If we could show  $[u'/x]e \hat{\sim} [v'/x]t$  then the conclusion follows from inductive hypothesis. To prove  $[u'/x]e \hat{\sim} [v'/x]t$ , we seem to need  $u' \hat{\sim} v'$ . After all, assuming  $x \in \text{FV}(e)$ , if rewriting subterms of  $[u'/x]e$  by  $(\sim)$  gets us  $[v'/x]t$ , then rewriting subterms of  $u'$  by  $(\sim)$  should get us  $v'$ .

But herein lies the problem. What if  $u', v'$  are  $\lambda$ 's, say  $\lambda x.e', \lambda x.t'$  respectively? Because we must exploit well-leveled bindings as discussed above, from  $u' \hat{\sim}_{\dagger}^0 v'$  we get only\*  $\forall a. [a/x]e' \hat{\sim} [a/x]t'$ . This cannot possibly imply  $e' \hat{\sim} t'$  if  $(\hat{\sim}) = (\sim) = (\approx)$  really holds, for then we have the counterexample  $e' \equiv (\lambda_.0) x, t' \equiv 0$ . If we had  $e' \hat{\sim} t'$  then  $\lambda x.e' \hat{\sim} \lambda x.t'$  would follow since  $(\hat{\sim})$  respects contexts, but not having  $e' \hat{\sim} t'$ , we cannot seem to extract  $u' \hat{\sim} v'$  from  $u' \hat{\sim}_{\dagger}^0 v'$  alone.

In Howe's setting, which prohibits open-term evaluation, this problem does not arise because everything is compared under closing substitutions. He defines  $e \hat{\sim} t$  to

---

\*We can ignore the tilde in Definition 41 here since we are expecting  $(\hat{\sim}) = (\sim) = (\approx)$ .

hold iff  $\forall \text{closing } \sigma$  the  $\sigma e$  and  $\sigma t$  satisfy certain conditions, so the conditional assertion  $\forall a. [a/x]e' \hat{\sim} [a/x]t'$  that only assures  $(\hat{\sim})$  under  $[a/x]$  coincides with  $e' \hat{\sim} t'$ . In such a setting defining  $\lambda x.e' \hat{\sim}_\dagger \lambda x.t'$  as  $e' \hat{\sim} t'$  works fine, whereas in  $\lambda^U$  it is unsatisfactory because  $\forall a. [a/x]e' \approx [a/x]t' \not\Rightarrow e' \approx t'$ .

To solve this problem, we generalize bisimilarity to a family of relations  $e \sim_X t$  indexed by sets of variables  $X$ , which hold iff  $\sigma e \sim \sigma t$  under all substitutions with  $\text{dom } \sigma \supseteq X$ . Then relations under experiment are redefined to  $\lambda x.e' \hat{\sim}_{\dagger X}^0 \lambda x.t' \iff e' \sim_{X \cup x} t'$ , and  $\lambda x.e' \hat{\sim}_X \lambda x.t'$  is refined to

$$\frac{e^0 \hat{\sim}_X s \quad \lambda x.s \sim_{X \setminus \{x\}} t}{\lambda x.e^0 \hat{\sim}_{X \setminus \{x\}} t}$$

Then the family  $(\hat{\sim}_X)$  respects contexts with diminishing indices, i.e.,  $e \hat{\sim}_X t \implies \forall C. \exists Y \subseteq X. C[e] \hat{\sim}_Y C[t]$ . In particular,  $u' \hat{\sim}_{\dagger X}^0 v'$  gives  $e' \hat{\sim}_{X \cup \{x\}} t'$ , which implies  $\lambda x.e' \hat{\sim}_X \lambda x.t'$  i.e.  $u' \hat{\sim}_X v'$ , and the rest of the proof goes smoothly.

### 5.3.2 The Proof

We now move on to the formal presentation of the proof. The following applies to both CBV and CBN. To simplify the proof, we will mostly use observational order rather than the full observational equivalence.

**Definition 49** (Observational Order).  $e \lesssim t$  iff for every  $C$  such that  $C[e], C[t] \in \text{Prog}$ ,  $C[e] \Downarrow^0 \implies C[t] \Downarrow^0$  holds and whenever  $C[e]$  terminates to a constant, then other terminates to the same constant.

**Remark.** Note  $(\approx) = (\lesssim) \cap (\gtrsim)$ .

We define indexed applicative bisimilarity coinductively using  $\nu \overline{R_X}. f \overline{R_X}$ , which denotes the greatest fixed point of a monotonic function  $f$  from families of relations

to families of relations. The basis for the existence of such a fixed point and the associated coinduction principle ( $\forall S. S \subseteq f S \implies S \subseteq \nu R.f R$ ) is reviewed in Appendix B.

**Notation.** A sequence, or family, of mathematical objects  $x_i$  indexed by a set  $I$  is written  $\overline{x_i^{i \in I}}$ . The superscript binds the index variable  $i$ . The superscript may be abbreviated like  $\overline{x_i^i}$  or omitted if the intention is clear. Let  $X, Y$  range over  $\wp_{\text{fin}} \text{Var}$ , the set of all finite subsets of  $\text{Var}$ . Let  $\overline{R_X}$  denote a family of relations  $R_X$  indexed by  $X \in \wp_{\text{fin}} \text{Var}$ . Union, inclusion, or other operations between families are done point-wise, e.g.,  $\overline{R_X} \subseteq \overline{S_X}$  denotes  $\forall X. R_X \subseteq S_X$ . Let the signature  $\sigma : X | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  mean that  $\sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  and  $\text{dom } \sigma \supseteq X$ , i.e.,  $\sigma$  substitutes for at least the variables in  $X$ . For a relation  $R$ , let  $R^{-1}$  denote  $\{(e, t) | t R e\}$ .

**Definition 50** (Indexed Applicative Bisimilarity). Define *indexed applicative similarity*  $(\overline{\lesssim_X})$ , *indexed applicative bisimilarity*  $(\overline{\sim_X})$ , and auxiliary relations as follows.

$$\begin{aligned}
u \{\overline{R_X^X}\}^0 v &\stackrel{\text{def}}{\iff} \begin{cases} u \equiv \lambda x.e \implies (v \equiv \lambda x.t \wedge e R_{\{x\}} t) \\ u \equiv \langle e \rangle \implies (v \equiv \langle t \rangle \wedge e R_{\emptyset} t) \\ u \equiv c \implies v \equiv c \end{cases} \\
u \{\overline{R_X^X}\}^{\ell+1} v &\stackrel{\text{def}}{\iff} u R_{\emptyset} v \\
e [R]_X t &\stackrel{\text{def}}{\iff} \forall \sigma : X | \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \text{ let } \ell = \max(\text{lv } e, \text{lv } t) \text{ in} \\
&\quad \sigma e \Downarrow^\ell u \implies (\sigma t \Downarrow^\ell v \wedge u \{\overline{R_X^X}\}^\ell v) \\
\overline{\lesssim_X} &\stackrel{\text{def}}{=} \nu \overline{R_X^X}. \overline{[R]_X^X} \\
\overline{\sim_X} &\stackrel{\text{def}}{=} \nu \overline{R_X^X}. \overline{[R]_X^X} \cap \overline{[R^{-1}]_X^{-1}}^X
\end{aligned}$$

Note that  $\{-\}^\ell$  maps a family of relations  $\overline{R_X}$  to a single relation  $\{\overline{R_X}\}^\ell$ , whereas  $\overline{[-]_X}$  maps a family  $\overline{R_X}$  to a family  $\overline{[R]_X}$ .

Indexed applicative bisimilarity agrees with the simpler notion of indexed applica-

tive mutual similarity, which is the symmetric reduction of  $\overline{(\lesssim_X)}$ . We will use these notions interchangeably.

**Proposition 51.** Define applicative mutual similarity as  $\overline{(\sim'_X)} \stackrel{\text{def}}{=} \overline{(\lesssim_X)} \cap \overline{(\gtrsim_X)}$ . Then  $\overline{(\sim_X)} = \overline{(\sim'_X)}$ .

*Proof.* See Appendix C.6 for details.  $\square$

As discussed above, the main idea is that indexed applicative bisimilarity should be a re-definition of observational equivalence. However, indexed applicative bisimilarity coincides not with observational equivalence but an indexed variant thereof. At each index  $X$ , the relation  $(\lesssim_X)$  asserts  $(\lesssim)$  under substitutions whose domains contain  $X$ . Then, whereas Howe proved  $(\lesssim) \subseteq (\lesssim)$ , we prove  $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$ .

**Definition 52** (Indexed Observational Order and Equivalence). Define  $e \approx_X t \stackrel{\text{def}}{\iff} \forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e \approx \sigma t$  and  $e \lesssim_X t \stackrel{\text{def}}{\iff} \forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e \lesssim \sigma t$ .

To prove  $\overline{(\lesssim_X)} = \overline{(\lesssim_X)}$ , we show mutual containment. The harder direction  $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$  (soundness of indexed bisimilarity) derives from the fact that  $\overline{(\lesssim_X)}$  is context-respecting, in the following adapted sense.

**Definition 53.** An indexed family of relations  $\overline{R_X}$  *respects contexts with diminishing indices* iff we have  $\forall i. e_i R_X t_i \implies (\tau \overline{e_i}) R_Y (\tau \overline{t_i})$  where  $Y = X \setminus \{x\}$  if  $\tau \overline{e_i} \equiv \lambda x. e^0$  and  $Y = X$  otherwise.

**Lemma 54.**  $e \lesssim_X t \iff \forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e \lesssim_{\emptyset} \sigma t$ .

*Proof.* Straightforward; See Lemma 108 in Appendix C.6.  $\square$

**Theorem 55** (Soundness of Indexed Applicative Bisimulation).  $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$ , therefore  $\overline{(\sim_X)} \subseteq \overline{(\approx_X)}$ .

*Proof.* We will show below that  $\overline{(\hat{\lesssim}_X)}$  respects contexts with diminishing indices. Suppose  $e \lesssim_X t$  and let a  $\sigma : X|Var \xrightarrow{\text{fin}}$  and a context  $C$  be given such that  $C[\sigma e], C[\sigma t] \in \text{Prog}$ . Then

$$\begin{array}{ll}
\sigma e \lesssim_{\emptyset} \sigma t & \text{by Lemma 54} \\
C[\sigma e] \lesssim_{\emptyset} C[\sigma t] & \text{by context-respecting property} \\
\left. \begin{array}{l} C[\sigma e] \Downarrow^0 \implies C[\sigma t] \Downarrow^0 \\ \text{and } \forall c. C[\sigma e] \Downarrow^0 c \implies C[\sigma t] \Downarrow^0 c \end{array} \right\} & \text{because } (\lesssim_X) = [\lesssim]_X \\
\sigma e \lesssim \sigma t & \text{because } C \text{ is arbitrary} \\
e \lesssim_X t & \text{because } \sigma \text{ is arbitrary}
\end{array}$$

Therefore,  $(\lesssim_X) \subseteq (\lesssim_X)$  and  $(\sim_X) = (\gtrsim_X) \cap (\lesssim_X) \subseteq (\gtrsim_X) \cap (\lesssim_X) = (\approx_X)$ .  $\square$

To prove that  $(\lesssim_X)$  respects contexts with diminishing indices, Howe's precongruence candidate is modified for indexed relations as follows.

**Definition 56** (Term Constructor). Let  $\tau$  range over multi-contexts (contexts with zero or more holes) of the forms  $x$ ,  $c$ ,  $(\lambda x.\bullet)$ ,  $(\bullet\bullet)$ ,  $\langle\bullet\rangle$ ,  $\sim\bullet$ , and  $!\bullet$ . The two holes in  $\bullet\bullet$  can be plugged by different expressions.

Using the notation introduced above for sequences and families, we write  $\tau \overline{e_i}^{i \in I}$  (using some finite set of positive integers  $I$ ) for a term that is formed by plugging the holes of  $\tau$  by immediate subterms  $\overline{e_i}$ ; for example, when  $\tau \equiv \bullet\bullet$ , then  $\tau \overline{e_i}^{i \in \{1,2\}} \equiv e_1 e_2$ . If  $\tau \equiv x$  or  $c$ , then the index set  $I$  is empty.

**Definition 57** (Indexed Precongruence Candidate). Given a family of relations  $\overline{R_X}$ , define the *indexed precongruence candidate*  $\widehat{R}_X$  by the following rules.

$$\frac{\tau \overline{e_i} \text{ not of form } \lambda x.e^0 \quad \forall i. e_i \widehat{R}_X s_i \quad \tau \overline{s_i} R_X t}{\tau \overline{e_i} \widehat{R}_X t} \qquad \frac{e^0 \widehat{R}_X s \quad (\lambda x.s) R_{X \setminus \{x\}} t}{(\lambda x.e^0) \widehat{R}_{X \setminus \{x\}} t}$$



Proposition 58, Proposition 59 (iv) and Lemma 60 imply  $\overline{(\widehat{\lesssim}_X)} = \overline{(\lesssim_X)}$ , so by Proposition 59 (ii), it follows that  $\overline{(\lesssim_X)}$  respects contexts with diminishing indices.

**Proposition 58.** Indexed applicative similarity is a monotonic family of precongruences:

- (i)  $(\lesssim_X)$  is reflexive for every  $X$ .
- (ii)  $(\lesssim_X)$  is transitive for every  $X$ .
- (iii)  $\overline{(\lesssim_X)}$  is monotonic in  $X$  (i.e.,  $X \subseteq Y \implies (\lesssim_X) \subseteq (\lesssim_Y)$ ).

*Proof.* The proofs for (i) and (ii) are adapted from [18].

- (i) Define  $(\equiv_X)$  to be syntactic equality for every  $X$ . Clearly  $(\equiv_X) \subseteq [\equiv_X]$ , so by coinduction  $\overline{(\equiv_X)} \subseteq \overline{(\lesssim_X)}$  in the product lattice  $\prod_X \wp(E \times E)$ . Therefore,  $\forall X. (\equiv) \subseteq (\lesssim_X)$ .
- (ii) Define  $R \circ S \stackrel{\text{def}}{=} \{(e, t) : \exists d. e R d S t\}$ . Take any triple  $e, d, t$  such that  $e \lesssim_X d \lesssim_X t$ , and let  $\sigma : X | \text{Var} \xrightarrow{\text{fin}} A, \ell$  be given. Then  $\sigma e \Downarrow^\ell v \implies \sigma d \Downarrow^\ell w \implies \sigma t \Downarrow^\ell u$  and  $v \{\overline{\lesssim_X}\}^\ell w \{\overline{\lesssim_X}\}^\ell u$ . The last assertion is equivalent to  $v \{\overline{\lesssim_X \circ \lesssim_X}\}^\ell u$ , so  $e [\overline{\lesssim_X \circ \lesssim_X}] t$ . Then by coinduction  $\overline{(\lesssim_X \circ \lesssim_X)} \subseteq \overline{(\lesssim_X)}$ .
- (iii) Suppose  $e \lesssim_X t$  and  $X \subseteq Y$ . Any  $\sigma : Y | \text{Var} \xrightarrow{\text{fin}} A$  also satisfies  $\sigma : X | \text{Var} \xrightarrow{\text{fin}} A$ , so if  $\sigma, \sigma t \in E^\ell$  then  $\sigma e \Downarrow^\ell v \implies \sigma t \Downarrow^\ell u$  where  $v \{\overline{\lesssim_X}\}^\ell u$ . Thus  $e \lesssim_Y t$ .  $\square$

**Proposition 59** (Basic Properties of the Indexed Precongruence Candidate). Let  $\overline{R_X}$  be a family of preorders that is monotone in  $X$ , i.e., each  $R_X$  is a preorder and  $X \subseteq Y \implies R_X \subseteq R_Y$ . Then

- (i)  $\widehat{R}_X$  is reflexive for every  $X$ .

- (ii)  $\widehat{R_X}$  respects contexts with diminishing indices.
- (iii)  $e\widehat{R_X}sR_Xt \implies e\widehat{R_X}t$  at each  $X$ .
- (iv)  $\overline{R_X} \subseteq \widehat{R_X}$ .
- (v)  $\widehat{R_X}$  is monotonic in  $X$ .

*Proof.*

- (i) Trivial induction on  $e$  shows  $e\widehat{R_X}e$ .
- (ii) By reflexivity of  $R_X$ , derivation rules for  $\widehat{R_X}$  subsume this assertion.
- (iii) Straightforward induction on  $e$  using (i) and transitivity of  $R_X$ .
- (iv) Apply (i) to (iii).
- (v) Straightforward induction on  $e$  using monotonicity of  $\overline{R_X}$  shows

$$(e\widehat{R_X}t \wedge X \subseteq Y) \implies e\widehat{R_Y}t. \quad \square$$

**Lemma 60.**  $e \widehat{\lesssim_X} t \implies e [\widehat{\lesssim_X}]t$ .

*Proof.* Fix a  $\sigma$  and an  $\ell$ , and assume  $\sigma e \rightsquigarrow_\ell^n v$ . Then show  $\sigma t \Downarrow^\ell u \wedge v \{\widehat{\lesssim_X}\}^\ell u$  by lexicographic induction on  $(n, e)$  with case analysis on the form of  $e$ . See Appendix C.6 for details.  $\square$

To prove the completeness of indexed bisimilarity  $(\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)})$ , we show  $\overline{(\lesssim_X)} \subseteq \overline{[\lesssim]_X}$  and coinduct. While proving  $\overline{(\lesssim_X)} \subseteq \overline{[\lesssim]_X}$ , it is necessary to convert  $(\lesssim)$  to  $(\lesssim_\emptyset)$ . For example, when  $e \lesssim_X t$ ,  $\sigma e \Downarrow^0 \langle e' \rangle$  then we can easily show  $\sigma t \Downarrow^0 \langle t' \rangle$  with  $e' \lesssim t'$ , but we cannot immediately conclude the  $e' \lesssim_\emptyset t'$  that we need for  $\langle e' \rangle \{\overline{(\lesssim_X)}\}^0 \langle t' \rangle$ . The argument given in Proposition 18, which hinges on the

new, generalized  $E_U$  rule, allows us to perform this conversion from  $(\lesssim)$  to  $(\lesssim_\emptyset)$ . We restate Lemma 61 here for  $(\lesssim)$ .

**Lemma 61.**  $\forall \sigma : Var \xrightarrow{\text{fin}} Arg. e \lesssim t \implies \sigma e \lesssim \sigma t.$

*Proof.* Use the same argument as Proposition 18.  $\square$

**Lemma 62.** For every  $X$ ,  $(\lesssim) \subseteq (\lesssim_X)$ . In particular,  $(\lesssim) = (\lesssim_\emptyset)$ . Likewise,  $(\approx) \subseteq (\approx_X)$  and  $(\approx) = (\approx_\emptyset)$ .

*Proof.* If  $e \lesssim t$ , then  $\sigma e \lesssim \sigma t$  for every  $\sigma : X | Var \xrightarrow{\text{fin}} Arg$  by Lemma 61, so  $e \lesssim_X t$ . Therefore  $(\lesssim) \subseteq (\lesssim_X)$ . When  $X = \emptyset$ , the reverse containment  $(\lesssim_\emptyset) \subseteq (\lesssim)$  also holds: the  $(\lesssim_\emptyset)$  relation implies  $(\lesssim)$  under any substitution, including the empty substitution. Hence  $(\lesssim) = (\lesssim_\emptyset)$ . The statement for  $(\approx)$  follows immediately.  $\square$

**Theorem 63** (Completeness of Indexed Applicative Bisimulation).  $\overline{(\lesssim_X)} = \overline{(\lesssim_X)}$  and  $\overline{(\approx_X)} = \overline{(\approx_X)}$ .

*Proof.* By Theorem 55, only  $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$  and  $\overline{(\approx_X)} \subseteq \overline{(\approx_X)}$  need to be proved. Suppose  $e \lesssim_X t$  and fix a  $\sigma : X | Var \xrightarrow{\text{fin}} Arg$  and an  $\ell$ . By definition  $\sigma e \lesssim \sigma t$  so  $\sigma e \Downarrow^\ell v \implies \sigma t \Downarrow^\ell u$ ; we will show that if these  $v, u$  exist then  $v \{\overline{(\lesssim_X)}\}^\ell u$ .

[If  $\ell > 0$ ] Because  $v \approx \sigma e \lesssim \sigma t \approx u$ , by Lemma 62 it follows that  $v \lesssim_\emptyset u$ .

[If  $\ell = 0$ ] Split cases by the form of  $v$ .

[If  $v \equiv \lambda x.e'$ ] If  $u$  were of the form  $\langle d \rangle$ , then the context  $\langle \sim \bullet \rangle$  would distinguish  $v$  and  $u$  because  $\langle \sim \lambda x.e' \rangle$  is stuck while  $\langle \sim \langle d \rangle \rangle \Downarrow^0 \langle d \rangle$ . If  $u$  were a constant, then the trivial context  $\bullet$  would distinguish  $u$  and  $v$ . Therefore,  $u \equiv \lambda x.t'$  for some  $t' \in E^0$ . For any  $a \in Arg$ , the equivalence  $v \lesssim u$  guarantees  $[a/x]e' \lesssim v \ a \lesssim u \ a \lesssim [a/x]t'$  so using Lemma 62,  $e' \lesssim_{\{x\}} t'$ .

[If  $v \equiv \langle e' \rangle$ ] By the same argument as above,  $u \equiv \langle t' \rangle$ . Then, since  $e' \in E^0$ , we

have  $e' \approx !\langle e' \rangle \lesssim !\langle t' \rangle \approx t'$ , so by Lemma 62,  $e' \lesssim_{\emptyset} t'$ .

[If  $v \equiv c$ ]  $u \equiv c$ , for otherwise the trivial context  $\bullet$  would distinguish  $u$  and  $v$ .

It follows that  $e [\lesssim]_X t$ , so  $\overline{(\lesssim_X)} \subseteq \overline{[\lesssim]_X}$ . By coinduction,  $\overline{(\lesssim_X)} \subseteq \overline{(\lesssim_X)}$ . Therefore,  $(\approx_X) = (\gtrsim_X) \cap (\lesssim_X) = (\gtrsim_X) \cap (\lesssim_X) = (\sim_X)$  for each  $X$ .  $\square$

Finally, from Theorems 55 and 63, we can prove the soundness and completeness of non-indexed applicative bisimulations.

*Proof of Theorem 43.* Let us first prove soundness (if  $R^\bullet$  is a non-indexed bisimulation, then  $R \subseteq (\approx)$ ). Given a relation  $R$ , define an indexed family  $\overline{R_X}$  by  $eR_X t \stackrel{\text{def}}{\iff} \forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}. (\sigma e)R(\sigma t)$ , and set  $\forall X. \tilde{R}_X \stackrel{\text{def}}{=} R_X \cup (\approx_X)$ . Note  $R^\bullet = R_\emptyset$ . Definition 42 states that  $R^\bullet$  is a (non-indexed) applicative bisimulation precisely when  $R^\bullet \subseteq [\tilde{R}]_\emptyset \cap [\tilde{R}^{-1}]_\emptyset^{-1}$ . When this containment holds, we claim that  $\forall X. \tilde{R}_X \subseteq [\tilde{R}]_\emptyset \cap [\tilde{R}^{-1}]_\emptyset^{-1}$  follows. Suppose  $e\tilde{R}_X t$  for some  $e, t, X$ , let  $\ell = \max(\text{lv } e, \text{lv } t)$ , and let  $\forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  be given. Then we have

$$\exists u. \sigma e \Downarrow^\ell u \iff \exists v. \sigma t \Downarrow^\ell v \text{ and if } u, v \text{ exist then } u \{\overline{\tilde{R}_X}\}^\ell v \wedge v \{\overline{\tilde{R}_X^{-1}}\}^\ell u \quad (*8)$$

as follows. First, we have  $\sigma e \tilde{R}_\emptyset \sigma t$  from  $e\tilde{R}_X t$ , hence either  $\sigma e R_\emptyset \sigma t$  or  $\sigma e \approx_\emptyset \sigma t$ .

[If  $\sigma e R_\emptyset \sigma t$ ]  $R_\emptyset = R^\bullet \subseteq [\tilde{R}]_\emptyset \cap [\tilde{R}^{-1}]_\emptyset^{-1}$ , so  $(*8)$  is immediate.

[If  $\sigma e \approx_\emptyset \sigma t$ ] By Theorems 55 and 63  $(\approx_\emptyset) = (\sim_\emptyset)$ , so  $\sigma e \Downarrow^\ell u$  iff  $\sigma t \Downarrow^\ell v$  and  $u \{\overline{\sim_X}\}^\ell v$ . But  $\overline{(\sim_X)} \subseteq \overline{\tilde{R}_X}$ , so  $(*8)$  follows.

Therefore, it follows that  $\overline{\tilde{R}_X} \subseteq [\tilde{R}]_\emptyset \cap [\tilde{R}^{-1}]_\emptyset^{-1}$ . By coinduction,  $\overline{\tilde{R}_X} \subseteq \overline{(\sim_X)} = \overline{(\approx_X)}$ , so in particular  $R_\emptyset \subseteq (\tilde{R}_\emptyset) \subseteq (\approx_\emptyset) = (\approx)$  using Lemma 62.

Now let us prove the converse (if  $R \subseteq (\approx)$  then  $R^\bullet$  is a non-indexed bisimulation). Let  $R \subseteq (\approx)$  be given, and define the families  $\overline{R_X}$  and  $\overline{\tilde{R}_X}$  as above. Then at each  $X$ , we have  $R_X \subseteq (\approx_X)$ , so  $\tilde{R}_X = (\approx_X) = (\sim_X)$ . Therefore,  $R^\bullet = R_\emptyset \subseteq (\sim_\emptyset) \subseteq$

$[\tilde{R}]_{\varnothing} \cap [\tilde{R}^{-1}]_{\varnothing}^{-1}$ , which means that  $R^{\bullet}$  is a bisimulation.

□

## Chapter 6

### Case Study: Longest Common Subsequence

In this section, we show how erasure and bisimulation work for a more complex example. LCS has a much more sophisticated code generation scheme than `power`, using the monadic memoization technique for eliminating the code duplication problem that is pervasive for staging memoized functions [31].

#### 6.1 The Code

Due to its size, the code for LCS is split up into two figures. Figure 6.1 shows `naive_lcs`, a naïve version that serves as the reference implementation, and an unstaged but memoized version `lcs`. Both of these functions map integers  $i, j$  and arrays  $P, Q$  to the length of the LCS of  $P_0, P_1, \dots, P_{i-1}$  and  $Q_0, Q_1, \dots, Q_{j-1}$ . Note that for simplicity, we compute the length of LCS instead of the actual sequence. The `naive_lcs` runs in exponential time, while `lcs` is a textbook polynomial-time version deploying memoization. The memo table is passed around monadically, using the monad primitives defined in the same figure.

We use a state-continuation monad to hide memo table-passing and to make the code in CPS. Computation in this monad takes a state (the memo table) and a continuation, and calls the continuation with an updated state and return value. Memo tables are functions mapping a key  $k$  and functions  $f, g$  to  $f \ v$  if a value  $v$  is associated with  $k$ , or to  $g \ ()$  if  $k$  is not in the table. The value `empty` is the empty

table, `ext` extends a table with a given key-value pair, and `lookup` looks up the table.\*

Figure 6.2 finally shows a staged, memoized implementation of LCS, which takes the lengths  $i, j$  and unrolls the recursion of `lcs` for that particular pair of lengths. Both `lcs` and `stlcs` are written with open recursion and closed up by memoizing fixed-point combinators `mem` and `memgen`.

## 6.2 Purpose of Monadic Translation

The generator is written in CPS (via the *state-continuation* monad) to solve the code duplication problem that commonly confronts staging of memoized functions. We briefly review the purpose of CPS here using a simpler example, the Fibonacci sequence. See [31] for a thorough treatment.

```
let rec fib n =
  if n <= 1 then n
  else fib (n-1) + fib (n-2)

let rec stfib n = (* unroll recursion by MSP *)
  if n <= 1 then .<n>.
  else .<.(stfib (n-1)) + .~(stfib (n-2))>.
```

The `fib` function computes the  $n$ -th term of the Fibonacci sequence, while `stfib` generates code that results from unrolling the recursion in `fib`. If we memoize these functions, then `fib` reduces from exponential time to linear time, but memoized `stfib` still generates exponential code. The memo table for `fib` accumu-

---

\*This interface is chosen to make the correspondence with  $\lambda^U$  clear. In MetaOCaml, `lookup` can return an option type, but since we left out higher-order constructors from  $\lambda^U$  we encoded the option type here. *Const* covers first-order types like `int option`, but not higher-order types like `(int  $\rightarrow$  int) option` or `(int code) option`.

lates entries like `fib 1`  $\mapsto$  1 (i.e. the cached result for `fib 1` is 1), `fib 2`  $\mapsto$  1, and `fib 3`  $\mapsto$  2, caching an integer for each entry, whereas the table for `stfib` gets entries like `stfib 1`  $\mapsto$  `.<1+0>.`, `stfib 2`  $\mapsto$  `.<(1+0)+0>.`, and `stfib 3`  $\mapsto$  `.<((1+0)+0) + (1+0)>.`, caching progressively larger code. Intuitively, the problem is that whereas `fib`'s memo caches the result of running some code, `stfib`'s memo caches the code itself. For `fib`, copying a value off of the table does not entail replicating and redoing all the computation that went into producing that value, so it does not perform any duplicated work; by contrast, in `stfib`, copying code from the table does result in duplicating all the sub-computations contained therein. Thus, the generated code repeats e.g. `1+0`—the computation for `fib 2`—exponentially many times.

If we could cache only those code of the form `.< $z_n$ >.` where  $z_n \in Var$ , then copying them would not lead to exponential growth. On the one hand, we wish `stfib n` returned `.< $z_n$ >.` for each  $n$  in such a way that the variable  $z_n$  will be bound to  $z_{n-1} + z_{n-2}$  in the final code. On the other hand, we cannot return `.< $z_n$ >.` without a binding for  $z_n$ , while returning `.<let  $z_n = z_{n-1} + z_{n-2}$  in  $z_n$ >.` would defeat the purpose. CPS-translating the generator solves this dilemma. CPS functions do not return but call their callers back, so instead of trying to return `.< $z_n$ >.`, we can call the caller back with `.< $z_n$ >.` as argument, within the (escaped) body of a binding for  $z_n$ :

```
let rec stfib' n k = (* k is the continuaion *)
  if n <= 1 then k .<n>.
  else stfib' (n-1) (fun  $z_{n-1}$   $\rightarrow$ 
    stfib' (n-2) (fun  $z_{n-2}$   $\rightarrow$ 
      .<let  $z_n = .\tilde{z}_{n-1} + .\tilde{z}_{n-2}$  in .~(k .< $z_n$ >.)>..))
```



If `stfib'` is memoized and then invoked with the identity continuation, the memo table is populated with entries of the form `stfib' n ↦ .<zn>.`, while contexts of the form `.<let zn = zn-1+zn-2 in .~•>.` are pushed onto the stack (i.e. the machine stack, not the explicit continuations). When the identity continuation is finally invoked on `.<zn>.`, the `let` in the stack captures that `zn`. In the end, the memoized generator produces a code in A-normal form. For example, `stfib' 4` with memoization returns:

```
.<let z2 = 1 + 0 in
  let z3 = z2 + 1 in
  let z4 = z3 + z2 in
  z4>.
```

CPS translation has the same effect on `genlcs`. The `lcs` function is a textbook example of a memoized algorithm, and staging it would counteract the memoization. To avoid this problem, we put the whole function into CPS (via a monad) and insert a `let` in the memoizing wrapper `memgen`. Whenever `memgen` detects an argument has no cached return value, it calls on `genlcs` to produce some code, and binds it to a new variable `z` to insert into the memo table. This technique significantly improves the quality of the generated code but makes the code generation logic much trickier and poses a challenge for verification.

The purpose of CPS in `stlcs` is to use Swadi et al.'s monadic translation technique for avoiding code duplication [31], which in turn is an adaptation of Bondorf's binding-time improvement [5] to MSP. Naïvely staging a memoized function like `lcs` undoes the effects of memoization and generates an unrolling of `naive_lcs`, duplicating code exponentially many times. CPS secures a sweet spot for `let`-insertion, namely in the memoizing fixed-point combinator `memgen`.

This technique greatly improves the quality of the generated code but makes the code generation logic tricky. For example, perhaps a bit counter-intuitively, `genlcs` generates code inside-out. It generates nested **let**'s that look like

```
let   $z_1 = e_1$  in
let   $z_2 = e_2$  in
...
in  $t$ 
```

where  $e_1$  is a term that computes LCS for smaller  $(i, j)$  than does  $e_2$ . Whereas `naive_lcs` inspects larger  $(i, j)$  before smaller  $(i, j)$ , the generated code above computes smaller  $(i, j)$  before larger  $(i, j)$ . These terms appear in the order that their values would populate the memo table in `lcs`, i.e. the order in which calls to `lcs` return. In general, advanced code generation schemes make describing the generated code's exact shape hard, and monadic memoization is no exception. But as we showed in Chapter 4.2, erasure makes such details irrelevant, letting us get away with rather sketchy characterizations of the generated code.

### 6.3 Correctness Proof

The correctness proof for CBN is just a simplification of CBV, so we will focus on the harder CBV and leave CBN as an exercise. We assume *Const* has unit, booleans, integers, tuples of integers, and arrays thereof with 0-based indices.  $\mathfrak{A}(\subseteq \text{Const})$  stands for the set of all arrays,  $\sigma$  ranges over substitutions  $\text{Var} \xrightarrow{\text{fin}} V^0$ , and  $e \Downarrow^0 \mathbb{Z}$  means  $\exists n \in \mathbb{Z}. e \Downarrow^0 n$ . Inclusion between substitutions  $\sigma' \supseteq \sigma$  is meant in the usual set-theoretic sense ( $\text{dom } \sigma' \supseteq \text{dom } \sigma$  and  $\sigma'|_{\text{dom } \sigma} = \sigma$ ).

Despite the extra baggage that comes with LCS, our overall strategy remains

the same as for **power**: check termination and apply the Erasure Theorem. In this case, we should additionally prove that **naive\_lcs**  $\approx$  **lcs**. The first step is similar to **power** but with additional material needed to account for memoization and CPS. The second step is routine. Let us start with a high-level proof showing where the Erasure Theorem is invoked and what lemmas we need.

**Theorem 64.**  $\lambda_v^U \vdash \text{naive\_lcs} \approx \lambda x.\lambda y.\lambda p.\lambda q.\text{stlcs } x \ y \ p \ q$

*Proof.* By extensionality and Lemma 46, it suffices to prove **naive\_lcs**  $i \ j \ P \ Q \approx_{\uparrow} \text{stlcs } i \ j \ P \ Q$  for every  $i, j, P, Q \in V^0$ .

[If  $i, j \in \mathbb{Z}$ ] Depends on the types and lengths of  $P, Q$ .

[If  $P, Q \in \mathfrak{A}$  and  $i < \text{length}(P) \wedge j < \text{length}(Q)$ ] This is where we need the Erasure Theorem, but note that erasure equates **stlcs** to **lcs**, not to **naive\_lcs**. This difference is immaterial as **naive\_lcs**  $\approx$  **lcs** (Lemma 79). Clearly **naive\_lcs** returns an integer in this case, so **lcs** does as well. Once we show **stlcs**  $i \ j \ P \ Q \Downarrow^0 \mathbb{Z}$  (Lemma 70), then Lemma 33 derived from the Erasure Theorem shows **lcs**  $i \ j \ P \ Q = \text{stlcs } i \ j \ P \ Q$ .

[Else] **naive\_lcs**  $i \ j \ P \ Q \Uparrow^0$  either because of type error ( $P, Q$  are indexed but are not arrays) or index-out-of-bounds. We will show that in this case **stlcs**  $i \ j \ P \ Q$  also diverges (Lemma 75).

[If  $i, j \notin \mathbb{Z}$ ] Both **naive\_lcs**  $i \ j \ P \ Q$  and **stlcs**  $i \ j \ P \ Q$  get stuck. □

We first prove that **stlcs**  $i \ j \ P \ Q \Downarrow^0 \mathbb{Z}$  whenever  $i, j \in \mathbb{Z}$  and  $P, Q$  are arrays with enough elements, which boils down to proving the same for the code generated by **genlcs**. We maintain two invariants, the first being that the memo table should map every key to some  $\langle z \rangle$ , where  $z$  should have an integer value under the substitution that will be in effect when the generated code is run.

**Definition 65.** The set  $G$  of *good memo tables* is the set of all  $T \in E^0$  such that for every  $i, j \in \mathbb{Z}$  and every  $f, g \in V^0$ , either  $\lambda_v^U \vdash \text{lookup } T (i, j) f g = f .\langle z \rangle .$  for some  $z$ , or  $\lambda_v^U \vdash \text{lookup } T (i, j) f g = g ()$ . If for all of the  $z$ 's we have  $z \in \text{dom } \sigma$  and  $\sigma z \in \mathbb{Z}$ , then  $T$  is *covered by*  $\sigma$ , written  $T \in G_\sigma$ .

**Lemma 66.** We have  $\text{empty} \in G_\sigma$  and  $T \in G_\sigma \wedge \sigma z \in \mathbb{Z} \implies \text{ext } T (i, j) .\langle z \rangle . \in G_\sigma$ . Also,  $\text{empty} \in G$  and  $T \in G \implies \text{ext } T (i, j) .\langle z \rangle . \in G$ .

*Proof.* More or less obvious from definitions, but  $\beta_v$ -reducing the application  $\text{ext } T$  requires noting that every  $T \in G_\sigma$  is terminating because  $T (0, 0) (\lambda_{-1}) (\lambda_{-1}) \Downarrow^0 1$ .  $\square$

The other invariant is that the continuation maps every  $.\langle \|e\| \rangle .$  to some  $.\langle \|t\| \rangle .$  where  $\|t\|$  terminates whenever  $\|e\|$  does. But  $e, t$  can contain free variables (the  $z$ 's mentioned above), so termination must be assessed under substitutions. We set  $K_\sigma$  to the set of continuations for which  $\text{FV}(\|t\|) \subseteq \text{dom } \sigma$ ; termination of  $\|e\|, \|t\|$  are assessed under extensions of  $\sigma$  covering  $\text{FV}(\|t\|) \cup \text{FV}(\|e\|)$ .

**Definition 67.** Let the set  $K_\sigma$  of all *good continuations under*  $\sigma$  consist of all  $k \in V^0$  s.t. for any  $e, \sigma' \supseteq \sigma$ , and  $T \in G_{\sigma'}$  with  $\sigma' \|e\| \Downarrow^0 \mathbb{Z}$ , we have  $\exists t. k T .\langle \|e\| \rangle . = .\langle \|t\| \rangle .$  and  $\sigma' \|t\| \Downarrow^0 \mathbb{Z}$ .

Under these invariants, **genlcs** always returns terminating code.

**Lemma 68.** Fix  $\sigma, T \in G_\sigma, (i, j \in \mathbb{Z})$ , and  $(p, q \in \text{Var})$ . If  $\forall \kappa \in K_\sigma. \exists e. \sigma \|e\| \Downarrow^0 \mathbb{Z}$  and

$$\lambda_v^U \vdash \text{genlcs } i j .\langle p \rangle . .\langle q \rangle . T \kappa = .\langle \|e\| \rangle .$$

then  $\forall k \in K_\sigma. \exists e. \sigma \|e\| \Downarrow^0 \mathbb{Z}$  and

$$\lambda_v^U \vdash \text{memgen genlcs } i j .\langle p \rangle . .\langle q \rangle . T k = .\langle \|e\| \rangle .$$

*Proof.* Fix  $k$  and split cases according to whether  $(i, j)$  is found in  $T$ , i.e. whether  $\text{lookup } T (i, j) f g$  invokes  $f$  or  $g$ . If it calls  $f$ , which in this case is  $(\mathbf{fun} \text{ s r} \rightarrow k \text{ s r})$ , then

$$\lambda_v^U \vdash \text{memgen genlcs } i j . \langle p \rangle . \langle q \rangle . T k = k T . \langle z \rangle . ,$$

for some  $z$  s.t.  $\sigma z \in \mathbb{Z}$ , so the conclusion follows from  $k \in K_\sigma$ . Else  $g$  is called, which in this case calls  $\text{genlcs}$ , so

$$\begin{aligned} & \text{memgen genlcs } i j . \langle p \rangle . \langle q \rangle . T k \\ &= \text{genlcs } i j . \langle p \rangle . \langle q \rangle . T k' \end{aligned}$$

where  $k' \stackrel{\text{def}}{=} (\mathbf{fun} \text{ tab r} \rightarrow . \langle \mathbf{let} \dots \rangle .)$ . Hence, it suffices to show  $k' \in K_\sigma$ . Fix  $e$ ,  $\sigma' \supseteq \sigma$  and  $T' \in G_{\sigma'}$ , and assume  $\sigma' \| e \| \Downarrow^0 n \in \mathbb{Z}$ . Letting  $T'' \stackrel{\text{def}}{=} \mathbf{ext} T' (i, j) . \langle z \rangle .$ , we have

$$k' T' . \langle \| e \| \rangle . = . \langle \mathbf{let} \text{ z} = \| e \| \text{ in } . \sim (k T'' . \langle z \rangle .) \rangle .$$

Lemma 66 gives  $T'' \in G_{\sigma'[\mathbf{z} \mapsto n]}$ , while  $\sigma'[\mathbf{z} \mapsto n] \supseteq \sigma$ , so by  $k \in K_\sigma$  the right-hand side equals some  $. \langle \mathbf{let} \text{ z} = \| e \| \text{ in } \| t \| \rangle .$  (which has the form  $. \langle \| e' \| \rangle .$ ) such that  $\sigma'[\mathbf{z} \mapsto n] \| t \| \Downarrow^0 \mathbb{Z}$ . Then

$$\begin{aligned} \sigma'(\mathbf{let} \text{ z} = \| e \| \text{ in } \| t \|) &= (\mathbf{let} \text{ z} = n \text{ in } \sigma' \| t \|) \\ &= \sigma'[\mathbf{z} \mapsto n] \| t \| \Downarrow^0 \mathbb{Z} \end{aligned}$$

noting  $\mathbf{z}$  is fresh for  $\sigma'$  by Barendregt's variable convention [3]. □

**Lemma 69.** Fix  $\sigma, T \in G_\sigma$ ,  $(i, j \in \mathbb{Z})$ , and  $(\mathbf{p}, \mathbf{q} \in \text{Var})$  such that  $\sigma \mathbf{p}, \sigma \mathbf{q} \in \mathfrak{A}$  and  $i < \text{length}(\sigma \mathbf{p}) \wedge j < \text{length}(\sigma \mathbf{q})$ . Then  $\forall k \in K_\sigma. \exists e. \sigma \| e \| \Downarrow^0 \mathbb{Z}$  and

$$\lambda_v^U \vdash \text{genlcs } i j . \langle p \rangle . \langle q \rangle . T k = . \langle \| e \| \rangle .$$

*Proof.* Lexicographic induction on  $(i, j)$ . Fix  $k$ . Now, if  $i < 0$  or  $j < 0$  we have  $\text{genlcs } i j . \langle p \rangle . \langle q \rangle . T k = k T . \langle 0 \rangle .$  and the conclusion follows immediately

from  $k \in K_\sigma$ . If  $i \geq 0 \wedge j \geq 0$ ,

$$\begin{aligned}
& \text{genlcs } i \ j \ .\langle p \rangle . \ .\langle q \rangle . \ T \ k \\
&= \text{bind } (\text{memgen genlcs } (i-1) \ (j-1) \ .\langle p \rangle . \ .\langle q \rangle .) \\
&\quad (\text{fun } n1 \rightarrow \text{bind} \dots) \ T \ k \\
&= \text{memgen genlcs } (i-1) \ (j-1) \ .\langle p \rangle . \ .\langle q \rangle . \ T \ k_1 \tag{*9}
\end{aligned}$$

where  $k_1 \stackrel{\text{def}}{=} (\text{fun } s \ r \rightarrow (\text{fun } n1 \rightarrow \dots) \ r \ s \ k)$ . It suffices to prove  $k_1 \in K_\sigma$ , for then  $(*9) = .\langle \|e\| \rangle .$  for some  $e$  s.t.  $\sigma \|e\| \Downarrow^0 \mathbb{Z}$  by inductive hypothesis and Lemma 68. So fix  $\sigma_1 \supseteq \sigma$ ,  $T_1 \in G_{\sigma_1}$ , and  $e_1$  with  $\sigma_1 \|e_1\| \Downarrow^0 \mathbb{Z}$ , then let us prove  $\exists t. k_1 \ T_1 \ .\langle \|e_1\| \rangle . = .\langle \|t\| \rangle .$  and  $\sigma_1 \|t\| \Downarrow^0 \mathbb{Z}$ . But  $k_1 \ T_1 \ .\langle \|e_1\| \rangle .$  reduces to

$$\begin{aligned}
& \text{bind } (\text{memgen genlcs } (i-1) \ j \ .\langle p \rangle . \ .\langle q \rangle .) \\
&\quad (\text{fun } n2 \rightarrow \ [.\langle \|e_1\| \rangle . / n1] \dots) \ T_1 \ k
\end{aligned}$$

so we see that it suffices to prove that  $k_2 \in K_{\sigma_1}$ , where

$$k_2 \equiv (\text{fun } s \ r \rightarrow (\text{fun } n2 \rightarrow \ [.\langle \|e_1\| \rangle . / n1] \dots) \ r \ s \ k).$$

Note that we can invoke the inductive hypothesis since  $\sigma_1 \supseteq \sigma$  implies that  $\sigma_1$  satisfies the constraints on  $\sigma_1 p$  and  $\sigma_1 q$ . Fix  $\sigma_2 \supseteq \sigma_1$ ,  $T_2 \in G_{\sigma_2}$ , and  $e_2$  with  $\sigma_2 \|e_2\| \Downarrow^0 \mathbb{Z}$ . Proceeding likewise with the one last call to `memgen genlcs`, we find that it suffices to prove  $k_3 \in K_{\sigma_2}$ , where  $k_3 \equiv (\text{fun } s \ r \rightarrow (\text{fun } n3 \rightarrow \dots) \ r \ s \ k)$ . Fixing  $\sigma_3 \supseteq \sigma_2$ ,  $T_3 \in G_{\sigma_3}$ , and  $e_3$  with  $\sigma_3 \|e_3\| \Downarrow^0 \mathbb{Z}$ ,

$$\begin{aligned}
& k_3 \ T_3 \ .\langle \|e_3\| \rangle . = k \ T_3 \ .\langle \text{if } p.(i) \ q.(j) \ \text{then } \|e_1\| + 1 \\
&\quad \text{else } \max \|e_2\| \ \|e_3\| \rangle .
\end{aligned}$$

As  $\sigma_3 \supseteq \sigma_2 \supseteq \sigma_1$ , from  $\sigma_i \|e_i\| \Downarrow^0 \mathbb{Z}$  we get  $\sigma_3 \|e\| \Downarrow^0 \mathbb{Z}$  (because if  $\sigma_i \|e_i\| \approx n$  for some  $n \in \mathbb{Z}$ , then  $\sigma_3 \|e\| \equiv (\sigma_3|_{\text{Var} \setminus \text{dom } \sigma_i})(\sigma_i \|e_i\|) \approx n$  using Proposition 18). Thus, since  $\sigma_3 p$ ,  $\sigma_3 q$  are arrays of length greater than  $i, j$ , respectively,  $\sigma_3 \|e_i\| \Downarrow^0 \mathbb{Z}$  for  $i = 1, 2$ ,

and  $\sigma_3(\mathbf{if} \ p.(i) \ \dots \|e_3\|) \Downarrow^0 \mathbb{Z}$ . The conclusion then follows from  $k \in K_\sigma$ .  $\square$

**Lemma 70.** If  $i, j \in \mathbb{Z}$ ,  $P, Q \in \mathfrak{A}$ , and  $i < \text{length}(P) \wedge j < \text{length}(Q)$ , then  $\mathbf{stlcs} \ i \ j \ P \ Q \Downarrow^0 \mathbb{Z}$ .

*Proof.* Letting  $\sigma \stackrel{\text{def}}{=} [P, Q/\mathbf{p}, \mathbf{q}]$ , we have  $\mathbf{fun} \ s \ r \rightarrow r \in K_\sigma$  and  $\mathbf{empty} \in G_\sigma$ , so using Lemma 69,

$$\mathbf{stlcs} \ i \ j \ P \ Q = (\mathbf{fun} \ \mathbf{p} \ \mathbf{q} \rightarrow \|e\|) \ P \ Q = \sigma \|e\| \Downarrow^0 \mathbb{Z}.$$

$\square$

The proof that  $\mathbf{stlcs}$  diverges if  $P, Q$  are too short or are not arrays is similar to Lemma 70, only differing in the invariants. This time, the invariant on the continuation holds that the generated code is always divergent for any extension of the current substitution. If  $k \ T \ .\langle \|e\| \rangle. = .\langle \|t\| \rangle.$ , the  $\|e\|$  plays no role in the divergence of  $\|t\|$ . The previously important invariant that every  $\langle z \rangle$  in the table is bound to an integer becomes irrelevant as well. Given these invariants,  $\mathbf{genlcs}$  generates code that diverges, and  $\mathbf{memgen}$  preserves this property.

**Definition 71.** Let the set  $K_\sigma^\uparrow$  of all *erring continuations under  $\sigma$*  consist of all  $k \in V^0$  s.t. for any  $e$  and  $T \in G$ , we have  $\exists t. k \ T \ .\langle \|e\| \rangle. = .\langle \|t\| \rangle.$  and  $\forall \sigma' \supseteq \sigma. \sigma' \|t\| \Uparrow^0$ .

**Lemma 72.** Fix  $\sigma, T \in G_\sigma$ ,  $(i, j \in \mathbb{Z})$ , and  $(\mathbf{p}, \mathbf{q} \in \text{Var})$ . If  $\forall \kappa \in K_\sigma^\uparrow. \exists e. (\forall \sigma' \supseteq \sigma. \sigma' \|e\| \Uparrow^0)$  and

$$\lambda_v^U \vdash \mathbf{genlcs} \ i \ j \ .\langle \mathbf{p} \rangle. \ .\langle \mathbf{q} \rangle. \ T \ \kappa = .\langle \|e\| \rangle.$$

then  $\forall k \in K_\sigma^\uparrow. \exists e. (\forall \sigma' \supseteq \sigma. \sigma' \|e\| \Uparrow^0)$  and

$$\lambda_v^U \vdash \mathbf{memgen} \ \mathbf{genlcs} \ i \ j \ .\langle \mathbf{p} \rangle. \ .\langle \mathbf{q} \rangle. \ T \ k = .\langle \|e\| \rangle.$$

*Proof.* Fix  $k$  and split cases according to whether  $(i, j)$  is found in  $T$ , i.e. whether  $\text{lookup } T (i, j) f g$  invokes  $f$  or  $g$ . If it calls  $f$ , which in this case is  $(\mathbf{fun} \text{ s r} \rightarrow k \text{ s r})$ , then

$$\lambda_v^U \vdash \text{memgen genlcs } i j .\langle p \rangle . .\langle q \rangle . T k = k T .\langle z \rangle .,$$

for some  $z$ , so the conclusion follows from  $k \in K_\sigma^\uparrow$ . Else  $g$  is called, which in this case calls  $\text{genlcs}$  and extends the table, so

$$\begin{aligned} & \text{memgen genlcs } i j .\langle p \rangle . .\langle q \rangle . T k \\ &= \text{genlcs } i j .\langle p \rangle . .\langle q \rangle . T k' \end{aligned}$$

where  $k' \stackrel{\text{def}}{=} (\mathbf{fun} \text{ tab r} \rightarrow .\langle \mathbf{let} \dots \rangle .)$ . Hence, it suffices to show  $k' \in K_\sigma^\uparrow$ . Fix  $e$  and  $T' \in G$ . Letting  $T'' \stackrel{\text{def}}{=} \mathbf{ext} T' (i, j) .\langle z \rangle .$ , we have

$$k' T' .\langle \|e\| \rangle . = .\langle \mathbf{let} \text{ z} = \|e\| \text{ in } .\sim(k T'' .\langle z \rangle .) \rangle .$$

Lemma 66 gives  $T'' \in G$ , so by  $k \in K_\sigma^\uparrow$  the rhs equals some  $.\langle \mathbf{let} \text{ z} = \|e\| \text{ in } \|t\| \rangle .$  (which has the form  $.\langle \|e'\| \rangle .$ ) such that  $\forall \sigma' \supseteq \sigma. \sigma' \|t\| \uparrow^0$ . Then if  $\sigma \|e\| \uparrow^0$  then  $\sigma(\mathbf{let} \text{ z} = \|e\| \text{ in } \|t\|) \uparrow^0$ , and if  $\sigma \|e\| \Downarrow^0 v$  then  $\sigma(\mathbf{let} \text{ z} = \|e\| \text{ in } \|t\|) = \sigma[\text{z} \mapsto v] \|t\| \uparrow^0$ .  $\square$

**Lemma 73.**  $\forall k \in K_\sigma^\uparrow. \exists e. \text{genlcs } i j .\langle p \rangle . .\langle q \rangle . T k = .\langle \|e\| \rangle . \wedge \sigma \|e\| \uparrow^0$  for any fixed  $\sigma, T \in G, (i, j \in \mathbb{Z}), (p, q \in \text{Var})$ .

*Proof.* Lexicographic induction on  $(i, j)$ . Fix  $k$ . If  $i < 0$  or  $j < 0$  then we have  $\text{genlcs } i j .\langle p \rangle . .\langle q \rangle . T k = k T .\langle 0 \rangle .$  and the conclusion follows immediately from  $k \in K_\sigma^\uparrow$ . If  $i \geq 0 \wedge j \geq 0$ ,

$$\begin{aligned} & \text{genlcs } i j .\langle p \rangle . .\langle q \rangle . T k \\ &= \text{bind} (\text{memgen genlcs } (i-1) (j-1) .\langle p \rangle . .\langle q \rangle .) \\ & \quad (\mathbf{fun} \text{ n1} \rightarrow \text{bind} \dots) T k \end{aligned}$$



$$= \text{memgen genlcs } (i-1) (j-1) .\langle p \rangle . .\langle q \rangle . T k_1 \quad (*10)$$

where  $k_1 \stackrel{\text{def}}{=} (\mathbf{fun} \ s \ r \rightarrow (\mathbf{fun} \ n1 \rightarrow \dots) \ r \ s \ k)$ . It suffices to prove  $k_1 \in K_\sigma^\uparrow$ , for then  $(*10) = .\langle \|e\| \rangle .$  for some  $e$  s.t.  $\forall \sigma' \supseteq \sigma. \sigma' \|e\| \uparrow^0$  by inductive hypothesis and Lemma 72. So fix  $T_1 \in G$  and  $e_1$ , then let us prove  $\exists t. k_1 T_1 .\langle \|e_1\| \rangle . = .\langle \|t\| \rangle .$  and  $\forall \sigma' \supseteq \sigma. \sigma' \|t\| \uparrow^0$ . But  $k_1 T_1 .\langle \|e_1\| \rangle .$  reduces to

$$\begin{aligned} & \text{bind } (\text{memgen genlcs } (i-1) j .\langle p \rangle . .\langle q \rangle .) \\ & (\mathbf{fun} \ n2 \rightarrow [.\langle \|e_1\| \rangle . / n1] \dots) T_1 k \end{aligned}$$

so repeating the argument as before, we fix  $e_2, e_3$  and  $T_2, T_3 \in G$ , and we see that it suffices to prove that  $k_3 \in K_\sigma^\uparrow$ , where  $k_3 \equiv (\mathbf{fun} \ s \ r \rightarrow (\mathbf{fun} \ n3 \rightarrow \dots) \ r \ s \ k)$ .

$$\begin{aligned} k_3 T_3 .\langle \|e_3\| \rangle . &= k T_3 .\langle \mathbf{if} \ p.(i) \ q.(j) \ \mathbf{then} \ \|e_1\| + 1 \\ &\quad \mathbf{else} \ \max \|e_2\| \ \|e_3\| \rangle . \end{aligned}$$

so the conclusion follows from  $k \in K_\sigma^\uparrow$ .  $\square$

**Lemma 74.** For any  $\sigma, T \in G, (i, j \in \mathbb{Z}), (p, q \in \text{Var})$ , and  $k \in K_\sigma^\uparrow$ , there exists some  $e$  such that  $\text{memgen genlcs } i \ j .\langle p \rangle . .\langle q \rangle . T k = .\langle \|e\| \rangle . \wedge \sigma \|e\| \uparrow^0$ .

*Proof.* Immediate from Lemmas 72 and 73.  $\square$

**Lemma 75.** If  $i, j \in \mathbb{Z}, P, Q \in V^0$ , and  $i \geq 0, j \geq 0$ , but  $\neg((P, Q \in \mathfrak{A}) \wedge i < \text{length}(P) \wedge j < \text{length}(Q))$ , then  $\text{stlcs } i \ j \ P \ Q \uparrow^0$ .

*Proof.* Let  $\iota \stackrel{\text{def}}{=} (\mathbf{fun} \ s \ r \rightarrow r)$ . Then we have

$$\begin{aligned} & \text{genlcs } i \ j .\langle p \rangle . .\langle q \rangle . \text{empty } \iota \\ &= \text{bind } (\text{memgen genlcs } (i-1) (j-1) .\langle p \rangle . .\langle q \rangle .) (\mathbf{fun} \ n1 \rightarrow \dots) \text{empty } \iota \\ &= \text{memgen genlcs } (i-1) (j-1) .\langle p \rangle . .\langle q \rangle . \text{empty } k \quad (*11) \end{aligned}$$

where  $k \stackrel{\text{def}}{=} (\mathbf{fun} \ s \ r \rightarrow (\mathbf{fun} \ n1 \rightarrow \dots) \ r \ s \ \iota)$ . Then  $k \in K_{[P,Q/p,q]}^\uparrow$  by a similar argument as in Lemma 69. Hence, by Lemma 72 the call to `genlcs` returns  $\cdot \langle \|e\| \rangle$ . s.t.  $[P, Q/p, q] \|e\| \uparrow^0$ , and we have  $\mathbf{stlcs} \ i \ j \ P \ Q = [P, Q/p, q] \|e\| \uparrow^0$ .  $\square$

Now let us prove  $\mathbf{lcs} \approx \mathbf{naive\_lcs}$ .

**Definition 76.** Let the set  $F_{P,Q}$  of *faithful memo tables* for  $P, Q$  consist of all  $T \in E^0$  s.t.  $\forall i, j \in \mathbb{Z}$  and  $\forall f, g \in V^0$ , either  $\exists n \in \mathbb{Z}. \lambda_V^U \vdash \mathbf{naive\_lcs} \ i \ j \ P \ Q = n$  and  $\text{lookup } T \ (i, j) \ f \ g = f \ n$ , or  $\lambda_V^U \vdash \text{lookup } T \ (i, j) \ f \ g = g \ ()$ .

**Lemma 77.** If  $k, P, Q \in V^0$ ,  $T \in F_{P,Q}$ , and  $i, j \in \mathbb{Z}$ , then for some  $T' \in F_{P,Q}$  we have  $\lambda_V^U \vdash \mathbf{mem\_lcs\_rec} \ i \ j \ P \ Q \ T \ k \approx_{\uparrow} k \ T' \ (\mathbf{naive\_lcs} \ i \ j \ P \ Q)$ .

*Proof.* If  $\text{lookup } T \ (i, j) \ f \ g$  invokes  $f$ , then the conclusion is immediate from the definition of faithful tables. Otherwise,

$$\begin{aligned} & \mathbf{mem\_lcs\_rec} \ i \ j \ P \ Q \ T \ k \\ &= \mathbf{lcs\_rec} \ i \ j \ P \ Q \ T \ (\mathbf{fun} \ \text{tab} \ r \rightarrow \dots) \\ &= (\mathbf{fun} \ \text{tab} \ r \rightarrow \dots) \ T'' \ (\mathbf{naive\_lcs} \ i \ j \ P \ Q) \end{aligned} \tag{*12}$$

for some  $T' \in F_{P,Q}$ . If  $(\mathbf{naive\_lcs} \ i \ j \ P \ Q) \uparrow^0$ , then we are done because (\*12)  $\uparrow^0$  and  $k \ T \ (\mathbf{naive\_lcs} \ i \ j \ P \ Q) \uparrow^0$ . Otherwise,  $\exists n_{ij} \in \mathbb{Z}. \mathbf{naive\_lcs} \ i \ j \ P \ Q = n_{ij}$ , and

$$(*12) = k \ (\mathbf{ext} \ T' \ (i, j) \ n_{ij}) \ n_{ij}$$

Then  $n_{ij} = \mathbf{naive\_lcs} \ i \ j \ P \ Q$  by assumption and one can confirm  $\mathbf{ext} \ T' \ (i, j) \ n_{ij} \in F_{P,Q}$  by inspecting definitions.  $\square$

**Lemma 78.** If we have  $k, P, Q \in V^0$ ,  $T \in F_{P,Q}$ , and  $i, j \in \mathbb{Z}$ , then there exists some  $T' \in F_{P,Q}$  such that  $\lambda_V^U \vdash \mathbf{lcs\_rec} \ i \ j \ P \ Q \ T \ k \approx_{\uparrow} k \ T' \ (\mathbf{naive\_lcs} \ i \ j \ P \ Q)$ . If

$k, P, Q \in V^0$ ,  $T \in F_{P,Q}$ , and  $i, j \in \mathbb{Z}$ , then  $\lambda_V^U \vdash \text{lcs\_rec } i \ j \ P \ Q \ T \ k \approx_{\uparrow} k \ T' \ (\text{naive\_lcs } i \ j \ P \ Q)$  for some  $T' \in F_{P,Q}$ .

*Proof.* Lexicographic induction on  $(i, j)$ . If  $i < 0$  or  $j < 0$ , then

$$\text{lcs\_rec } i \ j \ P \ Q \ T \ k = k \ T \ 0 = \text{naive\_lcs } i \ j \ P \ Q.$$

Otherwise,

$$\begin{aligned} & \text{lcs\_rec } i \ j \ P \ Q \ T \ k \\ &= \text{mem lcs\_rec } (i-1) \ (j-1) \ P \ Q \ k_1 \end{aligned} \tag{*13}$$

where  $k_1 \equiv (\mathbf{fun} \ s \ r \rightarrow (\mathbf{fun} \ n1 \rightarrow \dots) \ r \ s \ k)$ . By the inductive hypothesis and Lemma 77,  $\exists T_1 \in F_{P,Q}$  s.t.

$$(*13) \approx k_1 \ T_1 \ (\text{lcs\_naive } (i-1) \ (j-1) \ P \ Q).$$

If  $\text{lcs\_naive } (i-1) \ (j-1) \ P \ Q$  diverges, then so does the right-hand side of this equivalence, and also does  $\text{lcs\_naive } i \ j \ P \ Q$ ; hence  $(*13) \approx_{\uparrow} \text{lcs\_naive } i \ j \ P \ Q$  holds. We can therefore focus on the case where  $\text{lcs\_naive } (i-1) \ (j-1) \ P \ Q$  returns an  $n_{(i-1)(j-1)} \in \mathbb{Z}$ . Proceeding with the two other recursive calls in the same fashion, we find that the only interesting case is when the recursive calls return  $n_{(i-1)j}, n_{i(j-1)} \in \mathbb{Z}$  and  $T_3 \in F_{P,Q}$  such that

$$\begin{aligned} (*13) \approx_{\uparrow} & k \ T_3 \ (\mathbf{if} \ p.(i) = q.(i) \ \mathbf{then} \ n_{(i-1)(j-1)} + 1) \\ & \mathbf{else} \ \max \ n_{(i-1)j} \ n_{i(j-1)}) \end{aligned}$$

but the parenthesized portion is exactly what  $\text{naive\_lcs } i \ j \ P \ Q$  computes.  $\square$

**Lemma 79.**  $\lambda_V^U \vdash \text{lcs} \approx \text{naive\_lcs}$ .

*Proof.* By extensionality and Lemma 46, it suffices to show that  $\text{lcs } i \ j \ P \ Q \approx_{\uparrow} \text{naive\_lcs } i \ j \ P \ Q$  whenever  $i, j, P, Q \in V^0$ . Both sides diverge if  $i, j \notin \mathbb{Z}$ , so assume

$i, j \in \mathbb{Z}$ . By Lemma 78,

$$\exists T \in F_{P,Q}. \text{ lcs } i \ j \ P \ Q \approx_{\uparrow} \iota \ T \ (\text{naive\_lcs } i \ j \ P \ Q) \quad (*14)$$

where  $\iota \stackrel{\text{def}}{=} \mathbf{fun} \ s \ r \rightarrow r$ . If  $\text{naive\_lcs } i \ j \ P \ Q$  diverges, then so does (\*14), and we are done; otherwise, the right-hand side of (\*14) reduces to  $\text{naive\_lcs } i \ j \ P \ Q$ .  $\square$

Let us emphasize how the argument from Definition 65 through Lemma 75, we could ignore many details about the generated code. We did track type information, but we never said what the generated code looks like or what specific values it should compute. In fact, we are blissfully ignorant of the fact that `naive_lcs` has anything to do with `stlcs`. Erasure thus decouples the reasoning about staging from the reasoning about return values.

The proof of  $\text{naive\_lcs} \approx \text{lcs}$  was quite routine. The lack of surprise in this part of the proof is itself noteworthy, because it shows that despite the challenges of open term evaluation (Sections 3.4 and 5), the impact on correctness proofs is very limited.

Having equalities is also an advantage, as it obviates the bookkeeping needed to handle administrative redexes. For example, if (\*11) used small steps instead of equality, the `memgen genlcs ...` must be reduced to a value before `bind` can be contracted, and (\*11) must end with that value instead of `memgen genlcs ...` itself. However, the  $k$  in (\*11) also contains subterms of the form `memgen genlcs ...` which are not reduced to values because they occur under  $\lambda$ . The difference between those terms is often annoying to track and propagate through inductive proofs, and this bookkeeping is something we would rather not spend energy on. Equalities help by keeping these trivial things trivial.

```

let rec naive_lcs i j p q =
  if (i < 0 || j < 0) then 0
  else
    let n1 = naive_lcs (i-1) (j-1) p q in
    let n2 = naive_lcs (i-1) j p q in
    let n3 = naive_lcs i (j-1) p q in
    if p.(i) = q.(j) then n1 + 1
    else max n2 n3

let ext table key v = fun key' f g →
  if key = key' then f v
  else table key' f g
and empty key f g = g ()
and lookup table key f g = table key f g

let ret a = fun s k → k s a
and bind m f = fun s k → m s (fun s r → f r s k)
let eval_m m = m empty (fun s r → r)

let rec lcs_rec i j p q =
  if (i < 0 || j < 0) then ret 0
  else
    bind (mem lcs_rec (i-1) (j-1) p q) (fun n1 →
    bind (mem lcs_rec (i-1) j p q) (fun n2 →
    bind (mem lcs_rec i (j-1) p q) (fun n3 →
    ret (if p.(i) = q.(j)
      then n1 + 1
      else max n2 n3))))))
and mem f i j p q =
  fun tab k →
    lookup tab (i,j)
    (fun r → k tab r)
    (fun _ →
      f i j p q tab (fun tab r →
        let z = r in (k (ext tab (i,j) z) z)))
let lcs i j p q =
  eval_m (lcs_rec i j p q)

```

Figure 6.1 : Unstaged longest common subsequence with helper functions.

```

let rec genlcs i j p q =
  if (i < 0 || j < 0) then ret .<0>.
  else
    bind (memgen genlcs (i-1) (j-1) p q) (fun n1 →
    bind (memgen genlcs (i-1) j      p q) (fun n2 →
    bind (memgen genlcs i      (j-1) p q) (fun n3 →
      ret .<if (.~p).(i) = (.~q).(j)
        then .~n1 + 1
        else max .~n2 .~n3>..)))
and memgen f i j p q = fun tab k →
  lookup tab (i,j)
  (fun r → k tab r)
  (fun _ →
    f i j p q tab (fun tab r →
      .<let z = .~r in
      .~(k (ext tab (i,j) .<z>.) .<z>.)
      >..))
let stlcs i j =
  .!.<fun p q → .~(eval_m (genlcs i j .<p>. .<q>..))>.

```

Figure 6.2 : Staged, memoized longest common subsequence.

## Chapter 7

### Related Works

Taha [32] first discovered  $\lambda^U$ , which showed that functional hygienic MSP admits intensional equalities like  $\beta$ , even under brackets. However, [32] showed the mere existence of the theory and did not explore how to use it for verification, or how to prove extensional equivalences. Moreover, though [32] laid down the operational semantics of both CBV and CBN, it gave an equational theory for only CBN and left the trickier CBV unaddressed.

Yang pioneered the use of an “annotation erasure theorem”, which stated  $e \Downarrow^0 \langle \|t\| \rangle \implies \|t\| \approx \|e\|$  [37]. But there was a catch: the assertion  $\|t\| \approx \|e\|$  was asserted in the unstaged base language, instead of the staged language—translated to our setting, the theorem guaranteed  $\lambda \vdash \|t\| \approx \|e\|$  and not  $\lambda^U \vdash \|t\| \approx \|e\|$ , subject to termination of the generator. In practical terms, this meant that the context of deployment of the staged code could contain no further staging. Code generation must be done offline, and application programs using the generated  $\|t\|$  must be written in a single-stage language, or else no guarantee was made. This interferes with combining analyses of multiple generators and precludes dynamic code generation by `run (. !)`. Yang also worked with operational semantics, and did not explore in depth how equational reasoning interacts with erasure.

This paper can be seen as a confluence of these two lines of research: we complete  $\lambda^U$  by giving a CBV theory with a comprehensive study of its peculiarities, and adapt erasure to produce an equality in the staged language  $\lambda^U$ .

Berger and Tratt [4] devised a Hoare-style program logic for the typed language  $\text{Mini-ML}_e^\square$ . They develop a promising foundation and prove strong properties about it such as relative completeness, but concrete verification tasks considered concern relatively simplistic programs.  $\text{Mini-ML}_e^\square$  also prohibits manipulating open terms, so it does not capture the challenges of reasoning about free variables, which was one of the main challenges to which this thesis faced up. Insights gained from  $\lambda^U$  may help extend such logics to more expressive languages, and our proof techniques will be a good toolbox to lay on top of them.

For proving the correctness of programs in MSP with variable capture, Choi et al. [10] recently proposed an alternative approach with different trade-offs than the erasure approach explored here. They provide an “unstaging” translation of staging annotations into environment-passing code. Their translation is semantics preserving with no proof obligations but leaves an unstaged program that is complicated by environment-passing, whereas our erasure approach leaves a simpler unstaged program at the expense of additional proof obligations. It will be interesting to see how these approaches compare in practice or if they can be usefully combined, but for the moment they seem to fill different niches.



## Chapter 8

### Conclusion and Future Work

This thesis addressed three basic concerns for verifying staged programs. It showed that staging is a non-conservative extension because reasoning under substitutions is unsound in a MSP language, even if we are dealing with unstaged terms. Despite this drawback, untyped functional MSP has a rich set of useful properties. Simple termination conditions guarantee that erasure preserves semantics, which reduces the task of proving the irrelevance of annotations on a program’s semantics to the better studied problem of proving termination. We showed a sound and complete notion of applicative bisimulation for this setting, which allows us to reason under substitution in some cases. In particular, the shocking lack of  $\beta_x$  in  $\lambda_{\mathbf{v}}^U$  is of limited practical relevance as  $C\beta_x$  can be used instead.

These results improve our general understanding of hygienic MSP. We better know the multi-stage  $\lambda$  calculus’ similarities with the plain  $\lambda$  calculus (e.g., completeness of bisimulation), as well as its pathologies and the extent to which they are a problem. The Erasure Theorem gives intuitions on what staging annotations can or cannot do, with which we may educate the novice multi-stage programmer. This understanding has brought us to a level where the proof of a sophisticated generator like LCS is easily within reach. Thus, indeed, MSP can achieve not only genericity with performance, but also correctness, with a reasonable amount of effort.

This work may be extended in several interesting directions. We have specifically identified some open questions about  $\lambda^U$ : which type systems allow reasoning under

substitutions, whether it is conservative over the plain  $\lambda$  calculus for closed terms, and whether the extensionality principle can be strengthened to require equivalence for only closed-term arguments.

Devising a mechanized program logic would also be an excellent goal. Berger and Tratt’s system [4] may be a good starting point, although whether to go with Hoare logic or to recast it in equational style is an interesting design question. A mechanized program logic may let us automate the particularly MSP-specific proof step of showing that erasure preserves semantics. The Erasure Theorem reduces this problem to essentially termination checks, and we can probably capitalize on recent advances in automated termination analysis [16].

Bisimulation is known to work for single-stage imperative languages, though in quite different flavors from applicative bisimulation [23]. Adapting them to MSP would make the emerging imperative hygienic MSP languages [20, 29, 36] susceptible to analysis. The Erasure Theorem does not apply as-is to imperative languages since modifying evaluation strategies can commute the order of effects. Two mechanisms will be key in studying erasure for imperative languages—one for tracking which effects are commuted with which, and one for tracking mutual (in)dependence of effects, perhaps separation logic [28] for the latter. In any case, investigation of imperative hygienic MSP may have to wait until the foundation matures, as noted in the introduction.

Finally, this work focused on functional (input-output) correctness of staged code, but quantifying performance benefits is also an important concern for a staged program. It will be interesting to see how we can quantify the performance of a staged program through formalisms like improvement theory [30].

## Bibliography

- [1] Samson Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley Longman Publishing Co., Inc., 1990.
- [2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [3] Hendrik P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Studies in Logic and The Foundations of Mathematics. North-Holland, 1984.
- [4] Martin Berger and Laurence Tratt. Program logics for homogeneous meta-programming. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 64–81. Springer, 2010.
- [5] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proc. of LFP*, pages 1–10. ACM, 1992.
- [6] Edwin Brady and Kevin Hammond. A verified staged interpreter is a verified compiler. In *Proc. of GPCE*, pages 111–120. ACM, 2006.
- [7] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. of GPCE*, pages 57–76. Springer-Verlag New York, Inc., 2003.
- [8] Jacques Carette and Oleg Kiselyov. Multi-stage programming with functors and monads: Eliminating abstraction overhead from generic code. In *Proc. of GPCE*, pages 256–274. Springer, 2005.
- [9] Jacques Carette, Oleg Kiselyov, and Chung-Chieh Shan. Finally tagless, partially evaluated tagless staged interpreters for simpler typed languages. In *Proc. of APLAS*, pages 222–238. Springer-Verlag, 2007.
- [10] Wontae Choi, Baris Aktemur, Kwangkeun Yi, and Makoto Tatsuta. Static analysis of multi-staged programs via unstaging translation. In *Proc. of POPL*, pages 81–92, New York, NY, USA, 2011. ACM.
- [11] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Sci. Comput. Program.*, pages 25–46, 2006.

- [12] Ronak Kent Dybvig. Writing hygienic macros in scheme with syntax-case. Technical report, Indiana University Computer Science Department, 1992.
- [13] Thomas Forster. *Logic, Induction and Sets*. London Mathematical Society Student Texts. Cambridge University Press, July 2003.
- [14] Andrew D. Gordon. A tutorial on co-induction and functional programming. In *Glasgow functional programming workshop*, pages 78–95. Springer, 1994.
- [15] Andrew D Gordon. Bisimilarity as a theory of functional programming. *Theoretical Computer Science*, pages 5–47, 1999.
- [16] Matthias Heizmann, Neil Jones, and Andreas Podelski. Size-change termination and transition invariants. In *Static Analysis*, pages 22–50. Springer Berlin / Heidelberg, 2011.
- [17] Christoph A. Herrmann and Tobias Langhammer. Combining partial evaluation and staged interpretation in the implementation of domain-specific languages. *Sci. Comput. Program.*, pages 47–65, 2006.
- [18] Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Inf. Comput.*, pages 103–112, 1996.
- [19] Benedetto Intrigila and Richard Statman. The omega rule is  $\Pi_1^1$ -complete in the  $\lambda\beta$ -calculus. In *Proc. of TLCA*, pages 178–193. Springer, 2007.
- [20] Yuki Yoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. Shifting the stage: staging with delimited control. In *Proc. of PEPM*, pages 111–120. ACM, 2008.
- [21] Ik-Soon Kim, Kwangkeun Yi, and Cristiano Calcagno. A polymorphic modal type system for lisp-like multi-staged languages. In *Proc. of POPL*, pages 257–268. ACM, 2006.
- [22] Oleg Kiselyov, Kedar N. Swadi, and Walid Taha. A methodology for generating verified combinatorial circuits. In *Proc. of EMSOFT*, pages 249–258. ACM, 2004.
- [23] Vasileios Koutavas and Mitchell Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. of POPL*, pages 141–152. ACM, 2006.
- [24] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [25] Robert Muller. M-LISP: a representation-independent dialect of LISP with reduction semantics. *ACM Trans. Program. Lang. Syst.*, pages 589–616, 1992.
- [26] Gordon D. Plotkin. The  $\lambda$ -calculus is  $\omega$ -incomplete. *J. Symb. Logic*, pages 313–317, June 1974.

- [27] Gordon D. Plotkin. Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theoretical Computer Science*, pages 125–159, December 1975.
- [28] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, pages 55–74, 2002.
- [29] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proc. of GPCE*, 2010.
- [30] D. Sands. *Improvement theory and its applications*, pages 275–306. Cambridge University Press, 1998.
- [31] Kedar Swadi, Walid Taha, Oleg Kiselyov, and Emir Pašalić. A monadic approach for avoiding code duplication when staging memoized functions. In *Proc. of PEPM*, pages 160–169. ACM, 2006.
- [32] Walid Taha. *Multistage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute, 1999.
- [33] Walid Taha and Michael Florentin Nielsen. Environment classifiers. In *Proc. of POPL*, pages 26–37. ACM, 2003.
- [34] Masako Takahashi. Parallel reductions in lambda-calculus. *Inf. Comput.*, pages 120–127, 1995.
- [35] Takeshi Tsukada and Atsushi Igarashi. A logical foundation for environment classifiers. In *Proc. of TLCA*, pages 341–355. Springer-Verlag, 2009.
- [36] Edwin Westbrook, Mathias Ricken, Jun Inoue, Yilong Yao, Tamer Abdelatif, and Walid Taha. Mint: Java multi-stage programming using weak separability. In *Proc. of PLDI*, 2010.
- [37] Zhe Yang. Reasoning about code-generation in two-level languages. Technical report, BRICS, 2000.
- [38] Yoshihiro Yuse and Atsushi Igarashi. A modal type system for multi-level generating extensions with persistent code. In *Proc. of PPDP*, pages 201–212. ACM, 2006.

## Appendix A

### OCaml

This appendix gives a brief summary of OCaml’s basic constructs and their semantics. Note that the execution model presented here is vigorously simplified from the actual implementation, intended to give just enough background to follow the discussions in this thesis. The reader should see the official OCaml documentations to develop a working understanding of the actual language.

An OCaml program consists of a list of declarations and a single expression. An expression is built from standard arithmetic operations ( $+$ ,  $*$ ,  $<$ ,  $<=$ , etc), conditionals (**if-then-else**), anonymous functions (**fun**  $x \rightarrow e$ ) where  $e$  is any expression, and function application (written as juxtaposition). For example,

```
(fun x  $\rightarrow$  x + x) (1 + 2)
```

is an expression that returns 6—a function that adds two copies of its argument is applied to the sum of 1 and 2. It may help to think of **fun**  $x \rightarrow e$  as being like  $(x \mapsto e)$  in ordinary mathematical notation.

A declaration has one the following three forms:

```
let y = f x
```

```
let g x = x + x
```

```
let rec h x = h x
```

The first line is a declaration of a value, stipulating that  $y$  shall refer to the return value of the function call **f**  $x$ . The second line defines a function  $g$  of one argument.

This function declaration syntax differs from the value declaration syntax on the first line, in that there are more than one names listed between the keyword **let** and the **=**. The third line defines a function **h** that may recurse (i.e. call itself). In this case **h** is non-terminating.

A function declaration like

```
let g x = x + x
```

can always be seen as an abbreviation of a value declaration:

```
let g = fun x → x + x
```

This identification is possible because functions are first-class values in OCaml. Just like integers or booleans are values and therefore can be given names and manipulated, functions can be given names and/or passed as arguments to other functions.

Given declarations and an expression (the main expression), OCaml will evaluate that one expression, drawing definitions from the list of declarations as necessary. Whatever value that results from evaluating that main expression is returned as the result of the whole program. For example, given the declarations

```
let double x = x + x
let rec repeat f n x =
  if n = 0 then x
  else repeat f (n-1) (f x)
```

and the expression

```
repeat double 5 1
```

the result of this program is determined by calling and evaluating the **repeat** functions with the arguments set to **f=double**, **n=5**, and **x=1**. Since **n = 0** is false,

OCaml executes the **else** branch of the **repeat** function's body, which in this case is **repeat double 4 (double 1)**. Thus the result of the whole program is determined by calling **repeat** with **f=double**, **n=4**, and **x=(double 1)**. Continuing in this manner, we can see that the result is obtained by applying **double** five times to **1**, which is **32**.



## Appendix B

### Coinduction

This appendix briefly reviews coinduction as it applies to the definition of applicative bisimilarity. More thorough treatises can be found in [14, 13].

Coinduction is the dual to induction. A coinductive definition finds the greatest fixed point of a set of derivations, whereas an inductive definition finds the least fixed point. Coinduction on a coinductive set  $S$  shows that a certain property implies membership in  $S$ , whereas induction on an inductive set  $S'$  shows that membership in  $S'$  implies a certain property. Construction of the fixed point relies on the Knaster–Tarski Fixed Point Theorem, from which the associated principle of coinduction falls out as a byproduct.

**Definition 80.** A *complete lattice* is a triple  $(\mathcal{L}, \leq, \bigsqcup)$  such that  $(\mathcal{L}, \leq)$  forms a partial order in which every subset  $S \subseteq \mathcal{L}$  has a *least upper bound*  $\bigsqcup S$  in  $\mathcal{L}$ . An *upper bound* for  $S$  is an element  $y \in \mathcal{L}$  such that  $\forall x \in S. x \leq y$ , and the least upper bound for  $S$  is the least such element, i.e.

$$\forall y \in \mathcal{L}. (\forall x \in S. x \leq y) \implies \bigsqcup S \leq y.$$

By abuse of terminology the set  $\mathcal{L}$  by itself may also be called a complete lattice, with  $(\leq)$  and  $\bigsqcup$  to be inferred from context.

**Remark.** This definition forces the existence of greatest lower bounds, in accord with the standard definition of complete lattice. We will only be concerned with upper bounds and maximal fixed points, however.

**Theorem 81** (Knaster–Tarski Fixed Point). Let  $f : \mathcal{L} \rightarrow \mathcal{L}$  be a function from a complete lattice  $\mathcal{L}$  to itself. If  $f$  is monotone—( $x \leq y$ ) implies ( $f x \leq f y$ )—then  $f$  has a greatest fixed point  $z$  which is also the greatest element such that  $z \leq f z$ .

*Proof.* Take  $S \stackrel{\text{def}}{=} \{x \in \mathcal{L} : x \leq f x\}$  and  $z \stackrel{\text{def}}{=} \bigsqcup S$ . Then

$$\begin{array}{ll}
 \forall x \in S. x \leq z & \text{because } z \text{ is an upper bound} \\
 \forall x \in S. f x \leq f z & \text{by monotonicity} \\
 z \leq f z & \text{because } z \text{ is the least upper bound} \quad (1) \\
 z \in S & \text{by definition of } S \\
 f z \leq f (f z) & \text{by (1) and monotonicity} \\
 f z \in S & \text{by definition of } S \\
 z \geq f z & \text{because } z \text{ is an upper bound} \quad (2) \\
 z = f z & \text{by (1)(2)}
 \end{array}$$

Clearly every fixed point of  $f$  and every element  $x \in \mathcal{L}$  such that  $x \leq f x$  are in  $S$ , so  $z$  is the greatest of such elements.  $\square$

The specific complete lattices we need are powerset lattices and product lattices. Both constructions are standard. We omit the straightforward proof that a powerset lattice is a complete lattice.

**Definition 82.** A powerset lattice of a set  $S$  is the complete lattice  $(\wp S, \subseteq, \bigcup)$  where  $\wp S$  denotes the powerset of  $S$ .

**Definition 83.** If  $(\overline{\mathcal{L}_i, \leq_i, \bigsqcup_i})^{i \in I}$  is a family of complete lattices, then its *product* is the triple  $(\prod_{i \in I} \mathcal{L}_i, \leq, \bigsqcup)$  where the ordering operators are defined component-wise:

$$\begin{aligned}
 \overline{x_i}^{i \in I} \leq \overline{y_i}^{i \in I} & \stackrel{\text{def}}{\iff} \forall i \in I. x_i \leq_i y_i \\
 \bigsqcup S & \stackrel{\text{def}}{=} \overline{\bigsqcup_i S_i}^{i \in I}
 \end{aligned}$$

where  $S_i$  is  $\{x_i : \overline{x_j^{j \in I}} \in S\}$ , the set of the  $i$ -th components of all sequences in  $S$ .

**Proposition 84.** A product of complete lattices is always a complete lattice.

*Proof.* The  $(\leq)$  relation clearly inherits reflexivity and transitivity from  $(\leq_i)$ , so  $\prod_i \mathcal{L}_i$  is a partial order. For  $\sqcup$ , let a subset  $S \subseteq \prod_i \mathcal{L}$  be given and set  $\overline{z_i} \stackrel{\text{def}}{=} \sqcup S$ . For an arbitrary  $\overline{x_i} \in S$ , by definition  $\forall i. x_i \leq_i z_i$  so  $\overline{x_i} \leq \overline{z_i}$ . Therefore,  $\overline{z_i}$  bounds  $S$  in  $\prod_i \mathcal{L}_i$ . For any upper bound  $\overline{y_i}$  of  $S$ , for every  $i$ , the  $y_i$  bounds  $S_i$  in  $\mathcal{L}_i$  so  $z_i \leq_i y_i$ . Therefore,  $\overline{z_i} \leq \overline{y_i}$  so  $\overline{z_i}$  is the least upper bound of  $S$  in  $\prod_i \mathcal{L}_i$ .  $\square$

**Notation.** If  $R$  is a binary relation,  $\overline{x_i} R \overline{y_i}$  means  $\forall i. x_i R y_i$ .

A coinductive definition of a set  $S$  in a universe  $U$  is written  $S \stackrel{\text{def}}{=} \nu T. f T$  for some monotonic  $f : \wp U \rightarrow \wp U$ , which defines  $S$  to be the largest solution of the equation  $S = f S$ . The Knaster–Tarski Fixed Point Theorem guarantees the existence of the equation’s solution as well as the associated principle of coinduction:

$$\forall T \subseteq \mathcal{L}. T \subseteq f T \implies T \subseteq S.$$

Thus to show that some property  $\phi$  implies membership in  $S$ , one only needs to show that for some  $T \supseteq \{x \in \mathcal{L} : \phi(x)\}$  it is the case that  $T \subseteq f T$ .

To a first approximation, applicative bisimilarity is just the greatest fixed point of a monotonic function  $[-] : E \times E \rightarrow E \times E$  over the lattice  $\wp(E \times E)$ . An applicative bisimulation is any subset  $R$  of bisimilarity that satisfies  $R \subseteq [R]$ , so Theorem 43 is essentially just a stylized statement of the fact that applicative bisimilarity coincides with observational equivalence. But since  $\lambda^U$  requires a relation indexed by variable sets, the definition of applicative bisimilarity is mutually coinductive. This mutual coinduction is justified in the product lattice  $\prod_{X \in \wp_{\text{fin}} \text{Var}} \wp(E \times E)$ .

## Appendix C

### Proof Details

The main text omits details of proofs that would distract from conveying the core ideas. This appendix fills in those omitted details for nontrivial proofs. The statements of theorems which were stated in the main text are recalled here for the reader's convenience.

**Notation.** BVC stands for Barendregt's variable convention [3], i.e. the assumption that all bound variables are fresh. IH stands for inductive hypothesis.

#### C.1 Substitution

We first prove some miscellaneous facts about substitution.

**Lemma 85.** If  $e \in E^0$  then  $\text{lv } t = \text{lv}([e/x]t)$  (or equivalently,  $t \in E^\ell \iff [e/x]t \in E^\ell$ ).

*Proof.* Straightforward induction on  $t$ . □

**Lemma 86.** For any substitution  $\sigma : \text{Var} \xrightarrow{\text{fin}} E_{\text{cl}}^0$ , we have  $\sigma e \in \text{Arg} \implies e \in \text{dom } \sigma \cup \text{Arg}$ .

*Proof.* Immediate for CBN by Lemma 85. For CBV, perform case analysis on the form of  $e$  and apply Lemma 85. □

**Lemma 87.**  $[e^0/x]a \in \text{Arg}$ .

*Proof.* Follows directly from Lemma 85 in CBN. In CBV,  $a$  is of the form  $\lambda y.t^0$  or  $\langle t^0 \rangle$ , and by BVC  $[e^0/x](\lambda y.t^0) \equiv \lambda y.[e^0/x]t^0$  or  $[e^0/x]\langle t^0 \rangle \equiv \langle [e^0/x]t^0 \rangle$ , respectively. By Lemma 85  $[e^0/x]t^0 \in E^0$  so both of these forms are level-0 values.  $\square$

**Lemma 88.** Let  $\sigma : Var \xrightarrow{\text{fin}} E$  be a substitution and let  $y$  be a variable that is fresh for  $\sigma$ , i.e.  $y \notin \text{dom } \sigma \cup (\bigcup_{x \in \text{dom } \sigma} \text{FV}(\sigma x))$ . Then  $\sigma([e/y]t) \equiv [\sigma e/y](\sigma t)$ .

*Proof.* Induction on  $t$ .  $\square$

## C.2 Proofs for Operational Semantics

This section proves basic properties about  $(\leadsto)$  and  $(\approx)$ .

### C.2.1 Evaluation Contexts Compose

As evaluation contexts compose, in inductive proofs involving small-steps we may assume without loss of generality that SS-CTX is used exactly once. Hence to induct on small-step, we induct on the evaluation context.

**Lemma 89.**  $\mathcal{E}^{\ell, m'}[\mathcal{E}^{m', m}] \in ECtx^{\ell, m}$

*Proof.* Straightforward induction.  $\square$

**Lemma 90.** Any judgment  $e \leadsto_{\ell} t$  has a derivation that uses SS-CTX exactly once.

*Proof.* We can assume SS-CTX is used at least once because  $\bullet \in ECtx^{\ell, \ell}$ . Multiple uses of SS-CTX can be collapsed to one by the preceding lemma.  $\square$

### C.2.2 Determinism, Irreducibility of Values, and Focus

The small-step semantics is a deterministic transition system that halts as soon as a value is reached. Proposition 91 says that a small-step is unique (deterministic).

Proposition 92 states that a value does not step any further. Proposition 93 states that small-step “focuses” on the hole of an evaluation context, never reducing elsewhere until the hole contains a value. These facts have the important consequence that for an expression  $\mathcal{E}[e]$  to terminate,  $e$  must terminate first (Lemma 94), which comes in handy when we want to show that certain terms diverge.

**Proposition 91.**  $(e \rightsquigarrow_{\ell} t_1 \wedge e \rightsquigarrow_{\ell} t_2) \implies t_1 \equiv t_2$ .

*Proof.* Straightforward induction on the evaluation context used in the derivation of  $e \rightsquigarrow_{\ell} t_1$ .  $\square$

**Proposition 92.**  $v^{\ell} \not\equiv \mathcal{E}^{\ell,m}[e^m]$  whenever  $e^m$  is a redex, i.e.  $m = 0$  and  $e^m \equiv (\lambda x.t^0) a$ ,  $m = 0$  and  $e^m \equiv c d$ ,  $m = 0$  and  $e^m \equiv !\langle t^0 \rangle$ , or  $m = 1$  and  $e^m \equiv \sim\langle t^0 \rangle$ .

*Proof.* Straightforward induction on  $\mathcal{E}^{\ell,m}$ .  $\square$

**Proposition 93 (Focus).** An expression  $\mathcal{E}^{\ell,m}[e^m]$  where  $e^m \notin V^m$  small-steps at level  $\ell$  iff  $e^m$  small-steps at level  $m$ .

*Proof.* The “if” direction follows immediately from SS-CTX. For the “only if” direction,  $\mathcal{E}^{\ell,m}[e^m]$  small-steps, so by Lemma 90,  $\mathcal{E}^{\ell,m}[e^m] \equiv \mathcal{E}'[r]$  for some  $\mathcal{E}' \in ECtx^{\ell,n}$  and level- $n$  redex  $r$ . Straightforward induction on  $\mathcal{E}^{\ell,m}$  using the hypothesis  $e^m \notin V^{\ell}$  shows that for some  $\mathcal{E}'' \in ECtx^{m,n}$  we have  $\mathcal{E}^{\ell,m}[\mathcal{E}''] \equiv \mathcal{E}'$ , so  $\mathcal{E}''[r] \equiv e^m$ .  $\square$

**Lemma 94.** If  $e^{\ell} \equiv \mathcal{E}^{\ell,m}[t^m] \rightsquigarrow_{\ell}^n v \in V^{\ell}$ , then  $t^m \rightsquigarrow_m^{n'} u \in V^m$  where  $n' \leq n$ .

*Proof.* By Proposition 93,  $e^{\ell}$  keeps small-stepping to expressions of the form  $\mathcal{E}^{\ell,m}[t]$ , with only the  $t$  changing (i.e. stepping at level  $m$ ), until  $t \equiv u$ . By Proposition 91, these steps must form a prefix of  $e^{\ell} \rightsquigarrow_{\ell}^n v$ , so  $t^m$  does reach a  $u$ , in at most  $n$  steps.  $\square$

**Proof of Lemma 94.**

*Statement.* By Proposition 93,  $e^\ell$  keeps small-stepping to expressions of the form  $\mathcal{E}^{\ell,m}[t]$ , with only the  $t$  changing (i.e. stepping at level  $m$ ), until  $t \equiv u$ . By Proposition 91, these steps must form a prefix of  $e^\ell \rightsquigarrow_\ell^n v$ , so  $t^m$  does reach a  $u$ , in at most  $n$  steps.

### C.2.3 Equivalence of Open- and Closed-Term Observation

As mentioned in chapter 3, open-term observation and closed-term observation coincide in  $\lambda^U$ .

**Definition 95.** Define open observational equivalence ( $\approx^{\text{op}}$ ) just like ( $\approx$ ) but using  $E^0$  in place of  $Prog$ . Formally, let  $e \approx^{\text{op}} t$  iff for every  $C$  such that  $C[e], C[t] \in E^0$ ,  $(\exists u^0. C[e] \Downarrow^0 v \iff \exists v^0. C[t] \Downarrow^0)$  holds and whenever such  $u^0, v^0$  exist they obey  $\forall c. u^0 \equiv c \iff v^0 \equiv c$ .

Let us call  $C$  a *distinguishing context* for  $e$  and  $t$  iff  $C[e], C[t] \in E^0$  but exactly one of these terms terminate at level 0. Thus  $e \approx^{\text{op}} t$  holds iff no distinguishing context exists for  $e$  and  $t$ , whereas  $e \approx t$  holds iff no closing context for  $e$  and  $t$  is distinguishing.

The proof of  $(\approx) = (\approx^{\text{op}})$  in CBV uses the peculiarity of  $\lambda^U$  that it can force evaluation under binders. In CBN, this argument doesn't quite work. Instead we note that we can close up terms without affecting termination by replacing all free variables by divergence. This latter argument works for the plain CBN  $\lambda$  calculus as well but neither in CBV  $\lambda$  nor CBV  $\lambda^U$ .

**Proposition 96.**  $(\approx) = (\approx^{\text{op}})$ .

*Proof.* We prove the CBV case first. Clearly  $\lambda_v^U \vdash e \approx^{\text{op}} t \implies \lambda_v^U \vdash e \approx t$ . For the converse, suppose  $\lambda_v^U \vdash e \not\approx^{\text{op}} t$ , and let  $C$  be a (not necessarily closing) context that distinguishes  $e$  and  $t$ . Let  $\lambda \overline{x}_i$  denote a sequence of  $\lambda$ 's that bind all free variables in  $C[e]$  and  $C[t]$ . Let  $e_1; e_2$  denote sequencing, which checks that  $e_1$  terminates, discarding the return value, and then evaluates  $e_2$ . Sequencing is just syntactic sugar for  $(\lambda_.e_2) e_1$  in CBV. Then the context  $C' \stackrel{\text{def}}{=} \langle \lambda \overline{x}_i. \sim(C; \langle \lambda y. y \rangle) \rangle$  is a closing context that distinguishes  $e$  and  $t$ , so  $\lambda_v^U \vdash e \not\approx t$ .

Now consider CBN. Again, obviously  $\lambda_n^U \vdash e \approx^{\text{op}} t \implies \lambda_n^U \vdash e \approx t$ . Suppose for the converse that  $e \not\approx^{\text{op}} t$ , and let  $C$  be a distinguishing context. We prove in the remainder of this section that a term  $e$  terminates iff  $[\Omega/x]e$  does for a level-0, divergent  $\Omega$  (Lemma 100). Thus, if  $\Omega$  is any closed such term,  $C' \stackrel{\text{def}}{=} (\lambda \overline{x}_i^{i \in I}. C) \overline{\Omega}^{i \in I}$  is a closing, distinguishing context for  $e$  and  $t$ , where  $\lambda \overline{x}_i^{i \in I}$  is again a sequence of binders that bind all relevant variables and  $\lambda \overline{x}_i^{i \in I}. C$  is applied to as many copies of  $\Omega$  as there are  $x_i$ 's.  $\square$

**Lemma 97** (Classification). Suppose  $\sigma : \text{Var} \xrightarrow{\text{fin}} E^0$  is a substitution that maps variables to level-0 expressions. If  $\sigma e \xrightarrow{\ell} s$ , then at least one of the following conditions hold, where any variables bound in the evaluation contexts\* are distinct from  $x$  and fresh for  $\sigma$ .

- (1)  $e \xrightarrow{\ell} t$  for some  $t$ , and  $\forall \sigma' : \text{Var} \xrightarrow{\text{fin}} E_{\text{cl}}^0. \sigma' e \xrightarrow{\ell} \sigma' t$ .
- (2)  $e \equiv \mathcal{E}^{\ell, m}[x]$  and  $\sigma x$  small-steps at level  $m$ .
- (3)  $e \equiv \mathcal{E}^{\ell, 0}[x \ t]$  and  $\sigma x \equiv \lambda y. t^0$  and  $\sigma t \in \text{Arg}$ .
- (4) We are working in CBV, and  $e \equiv \mathcal{E}^{\ell, 0}[(\lambda y. t^0) \ x]$  and  $\sigma x \in V^0$ .

---

\*Unlike in single-stage languages, evaluation contexts can bind variables at level  $> 0$ .



(5)  $e \equiv \mathcal{E}^{\ell,0}[x\ c]$  and  $\sigma x \in \text{Const}$  and  $(\sigma x, c) \in \text{dom } \delta$ .

(6)  $e \equiv \mathcal{E}^{\ell,0}[c\ x]$  and  $\sigma x \in \text{Const}$  and  $(c, \sigma x) \in \text{dom } \delta$ .

(7)  $e \equiv \mathcal{E}^{\ell,0}[!x]$  or  $\mathcal{E}^{\ell,1}[\sim x]$ , and  $\sigma x \equiv \langle t^0 \rangle$ .

*Proof.* Induction on  $e$ .

[If  $e \equiv c$ ] Vacuous:  $\sigma e \equiv c \not\sim_{\ell}$ .

[If  $e \equiv x$ ] Condition (2) holds with  $\mathcal{E}^{\ell,m} \equiv \bullet$ .

[If  $e \equiv e_1\ e_2$ ] Inversion on  $\sigma e \sim_{\ell} s$  generates three cases, two of which are trivial.

[If  $\sigma e_1$  small-steps] Immediate from IH.

[If  $\sigma e_1 \in V^{\ell}$  and  $\sigma e_2$  small-steps] Immediate from IH.

[If  $\sigma e \sim_{\ell} s$  is derived by SS- $\beta$  or SS- $\beta_v$ ] By inversion

$$(i) \ell = 0 \quad (ii) \sigma e_1 \equiv \lambda y. t^0 \quad (iii) \sigma e_2 \in \text{Arg}$$

Case analysis on the form of  $e_1$  generates two cases:  $e_1 \equiv x$  or  $e_1 \equiv \lambda y. e'_1$  for some  $e'_1$  such that  $\sigma \lambda y. e'_1 \equiv \lambda y. t^0$ .

[If  $e_1 \equiv x$ ] Condition (3) holds.

[If  $e_1 \equiv \lambda y. e'_1$ ] Cases analysis on whether  $e_2 \in \text{Arg}$ .

[If  $e_2 \in \text{Arg}$ ] Condition (1) holds: clearly  $e \sim_0 [e_2/y]e'_1$ , and given any  $\sigma'$ ,

$$\sigma' e \equiv (\lambda y. \sigma' e'_1) (\sigma' e_2) \sim_0 [\sigma' e_2/y] (\sigma' e'_1) \equiv \sigma' ([e_2/y] e'_1)$$

where by BVC  $y$  is fresh for  $\sigma'$ , so the first  $(\equiv)$  is immediate and the second  $(\equiv)$  follows by Lemma 88.

[If  $e_2 \notin \text{Arg}$ ] By Lemma 86, it must be the case that  $e_2 \equiv x$  where  $\sigma x \in \text{Arg}$ ,

which means:

[In CBN] Condition (1) holds. We have  $e_1 \ e_2 \equiv (\lambda y.t^0) \ x \rightsquigarrow_0 [x/y]t^0$  and so  $\sigma'e \equiv (\lambda y.\sigma't^0) (\sigma'x) \rightsquigarrow [\sigma'x/y](\sigma't^0) \equiv \sigma'([x/y]t^0)$  where the manipulation of  $\sigma'$  uses BVC with Lemma 88.

[In CBV] Condition (4) holds. From (iii) we have  $\sigma x \in V^0$ , and from  $\sigma(\lambda y.e'_1) \equiv \lambda.t^0 \in E^0$  and Lemma 85 we have  $e'_1 \in E^0$ .

[If  $\sigma e \rightsquigarrow_\ell s$  is derived by SS- $\delta$ ] By inversion  $\sigma e_i \in \text{Const}$  for  $i = 1, 2$ , hence  $e_i \in \text{Const} \cup \text{Var}$ . If  $e_1 \in \text{Var}$  then condition (5) holds, else if  $e_2 \in \text{Var}$  then condition (6) holds, else condition (1) holds.

[If  $e \equiv \lambda y.e'$ ] By BVC  $y$  is fresh for  $\sigma$ , so  $\sigma e \equiv \lambda y.\sigma e'$ . By inversion  $\sigma e'$  small-steps at level  $\ell$  (which is necessarily  $> 0$ ), so IH is applicable. The conclusion is then immediate.

[If  $e \equiv \langle e' \rangle$ ] Immediate from IH.

[If  $e \equiv \sim e'$ ] Inversion generates two cases.

[If  $\sigma e'$  small-steps] Immediate from IH.

[If  $\sigma e \rightsquigarrow_\ell t$  is derived by SS- $E$ ] By inversion,

$$(i) \ \ell = 1 \quad (ii) \ \sigma e' \equiv \langle t \rangle \quad (iii) \ t \in E^0.$$

Case analysis on the form of  $e'$  yields two cases.

[If  $e' \equiv x$ ] Condition (7) holds.

[If  $e' \equiv \langle e'' \rangle$  where  $\sigma e'' \equiv t$ ] Condition (1) holds.

[If  $e \equiv !e'$ ] Similar to the preceding case. □

**Lemma 98.** Let  $\sigma : \text{Var} \xrightarrow{\text{fin}} \{e \in E^0 : e \uparrow^0\}$  be a substitution that substitutes divergent level-0 expressions. Then  $v \in V^\ell \iff \sigma v \in V^\ell$ .

*Proof.* For  $\ell > 0$ , this lemma is a special case of Lemma 85. If  $\ell = 0$  then  $e$  must be  $\lambda x.t^0$  or  $\langle t^0 \rangle$  or  $c$ , so Lemma 85 ensures  $\sigma e \in V^0$ . □

**Lemma 99.** Let  $\sigma, \sigma' : \text{Var} \xrightarrow{\text{fin}} \{t \in E^0 : t \uparrow^0\}$  be substitutions that substitute only divergent level-0 terms. Then  $\sigma e \Downarrow^\ell \iff \sigma' e \Downarrow^\ell$  for any  $\ell, e$ .

*Proof.* By symmetry, proving one direction will suffice. Suppose  $\sigma e \rightsquigarrow_\ell^n v$  for some  $v \in V^\ell$ . We will prove  $\sigma' e \Downarrow^\ell$  by induction on  $n$ . If  $n = 0$ , then by Lemma 98,  $\sigma e \in V^\ell \iff e \in V^0 \iff \sigma' e \in V^\ell$ . If  $n > 0$ , we perform case analysis on the first small-step using Lemma 97. Conditions (3), (5), (6), and (7) are vacuous because they force  $\sigma x \in V^0$ . For the remaining cases:

- (1) The  $\sigma e$  small-steps as  $\sigma e \rightsquigarrow_\ell \sigma t \rightsquigarrow_\ell^{n-1} v$ , and  $\sigma' e \rightsquigarrow_\ell \sigma' t$ . By IH  $\sigma' t \Downarrow^\ell$ , so  $\sigma' e \Downarrow^\ell$ .
- (2) The  $e$  must decompose as  $\mathcal{E}^{\ell, m}[x]$  where  $\sigma x$  small-steps at level  $m$ . By assumption  $\sigma x \in E^0$  so in order for  $\sigma x$  to small-step,  $m = 0$  is necessary. But as  $\sigma e \equiv (\sigma \mathcal{E}^{\ell, m})[\sigma x] \Downarrow^\ell$ , by Lemma 94  $\sigma x \Downarrow^m$  i.e.  $\sigma x \Downarrow^0$ , contrary to assumption. This case is therefore vacuous.
- (4) We must be working with CBV, and  $e$  decomposes as  $\mathcal{E}^{\ell, 0}[(\lambda y. t^0) x]$  where  $\sigma t \equiv \lambda y. t^0$  and  $\sigma x \in V^0$ . By assumption  $\sigma x \notin V^0$ , so this case is vacuous.  $\square$

**Lemma 100.** If  $\Omega$  is a level-0 divergent term, then  $e \Downarrow^\ell \iff [\Omega/x]e \Downarrow^\ell$ .

*Proof.* Take  $\sigma = \emptyset$  and  $\sigma' = [\Omega/x]$  in Lemma 100.  $\square$

### C.3 Proofs for Equational Theory

This section fills in the proof details for confluence and standardization.

#### C.3.1 Confluence

Confluence of the reduction relation was reduced in the main text to confluence of parallel reduction. Parallel reduction is shown to be confluent as follows.

**Lemma 101** (Level Reduction). Suppose  $e \longrightarrow^* t$ . Then  $\text{lv } e \geq \text{lv } t$ , or equivalently,  $\forall \ell. e \in E^\ell \implies t \in E^\ell$ .

*Proof.* Straightforward induction, first on the length of the reduction, then on the (derivation of the) first reduction, using Lemma 85.  $\square$

**Lemma 102.** For any  $v \in V^\ell$ ,  $v \longrightarrow^* e \implies e \in V^\ell$ . Moreover,  $v$  and  $e$  have the same form:  $v \equiv c \iff e \equiv c$ ,  $v \equiv \lambda x.t \iff e \equiv \lambda x.e'$ , and  $v \equiv \langle t \rangle \iff e \equiv \langle e' \rangle$ .

*Proof.* By induction on the length of the reduction  $v \longrightarrow^* e$ , it suffices to prove this assertion for one-step reductions. The case  $\ell > 0$  is just Lemma 101. For  $\ell = 0$ , if  $v$  is  $\lambda x.t$  or  $\langle t \rangle$  for some  $t \in E^0$  then  $e$  is  $\lambda x.t'$  or  $\langle t' \rangle$ , respectively, where  $t \longrightarrow^* t'$ . By Lemma 101  $t' \in E^0$  so  $e \in V^0$ . If  $v \equiv c$  then  $e \equiv c$  because  $c$  is a normal form.  $\square$

**Lemma 103.** If  $a \longrightarrow^* e$  then  $e \in \text{Arg}$ .

*Proof.* Immediate from Lemma 101 for CBN and from Lemma 102 for CBV.  $\square$

**Lemma 104.** If  $e \xrightarrow{n} t$  and  $a \xrightarrow{m} b$  where  $e, t \in E^0$  and  $a, b \in \text{Arg}$ , then we have  $[a/x]e \xrightarrow{n + \#(x,t) \cdot m} [b/x]t$ .

*Proof.* Induction on  $e \xrightarrow{n} t$  with case analysis on the last rule used to derive it. Let  $N \stackrel{\text{def}}{=} n + \#(x, t) \cdot m$ .

[PR-VAR] By inversion,  $e \equiv y \equiv t$  for some  $y$  and  $n = 0$ .

[If  $x \equiv y$ ]  $N = m$ , so  $[a/x]e \equiv a \xrightarrow{N} b \equiv [b/x]t$ .

[If  $x \not\equiv y$ ]  $N = 0$ , so  $[a/x]e \equiv y \xrightarrow{N} y \equiv [b/x]t$ .

[PR-ABS]

(i)  $e \equiv \lambda y.e'$  (ii)  $t \equiv \lambda y.t'$  (iii)  $e' \xrightarrow{n} t'$  by inversion (1)

$\#(x, t) = \#(x, t')$ . because  $x \not\equiv y$  by BVC (2)

$$N = n + \#(x, t') \cdot m \quad \text{by (2), defn of } N \quad (3)$$

$$[a/x]e' \xrightarrow{N} [b/x]t' \quad \text{by IH on (1.iii) with (3)} \quad (4)$$

$$\lambda y.[a/x]e' \xrightarrow{N} \lambda y.[a/x]t' \quad \text{by PR-ABS}$$

Therefore  $[a/x]\lambda y.e' \xrightarrow{N} [a/x]\lambda y.t'$  because  $x \neq y$ .

[PR-APP]

$$\left. \begin{array}{ll} \text{(i)} \ e \equiv e_1 \ e_2 & \text{(ii)} \ t \equiv t_1 \ t_2 \\ \text{(iii)} \ e_i \xrightarrow{n_i} t_i \ (i = 1, 2) & \text{(iv)} \ n = n_1 + n_2 \end{array} \right\} \quad \text{by inversion}$$

$$[a/x]e_i \xrightarrow{n_i + \#(x, t_i) \cdot m} [b/x]t_i \ (i = 1, 2) \quad \text{by IH on (iii)}$$

$$[a/x](e_1 \ e_2) \xrightarrow{N} [b/x](t_1 \ t_2) \quad \text{by PR-APP}$$

where the last step uses the fact that

$$n_1 + \#(x, t_1) \cdot m + n_2 + \#(x, t_2) \cdot m = n_1 + n_2 + \#(x, t_1 \ t_2) \cdot m = N.$$

[PR- $\beta$ ] There exist some  $f \in Arg$  and  $f' \in E$  such that

$$\left. \begin{array}{ll} \text{(i)} \ e \equiv (\lambda y.t') \ f \in E^0 & \text{(ii)} \ t \equiv [f'/y]t' \\ \text{(iii)} \ t \xrightarrow{n_t} t' & \text{(iv)} \ f \xrightarrow{n_f} f' \\ \text{(v)} \ n = n_t + \#(y, t') \cdot n_f + 1 \end{array} \right\} \quad \text{by inversion} \quad (5)$$

$$f' \in Arg \quad \text{by Lemma 103 and (5.iv)} \quad (6)$$

$$[a/x]t \xrightarrow{n_t + \#(x, t') \cdot m} [b/x]t' \quad \text{by IH on (iii)} \quad (7)$$

$$[a/x]f \xrightarrow{n_f + \#(x, f') \cdot m} [b/x]f' \quad \text{by IH on (iv)} \quad (8)$$

Taking

$$M \stackrel{\text{def}}{=} (n_t + \#(x, t') \cdot m) + \#(y, t') \cdot (n_f + \#(x, f') \cdot m) + 1$$

and noting that  $x \not\equiv y$  by BVC,

$$[a/x](\lambda y.t) f \xrightarrow{M} [[b/x]f'/y][b/x]t' \quad \text{by PR-}\beta \text{ on (7)(8) noting (6)}$$

$$[[b/x]f'/y][b/x]t' \equiv [b/x][f'/y]t' \quad \text{because } y \notin \text{FV}(b) \text{ by BVC}$$

$$[a/x](\lambda y.t) f \xrightarrow{M} [b/x][f'/y]t' \quad \text{by the two preceding lines}$$

as required. For the complexity,

$$\begin{aligned} M &= (n_t + \#(x, t') \cdot m) + \#(y, t') \cdot (n_f + \#(x, f') \cdot m) + 1 \\ &= n_t + \#(y, t') \cdot n_f + 1 + (\#(x, t') + \#(y, t')\#(x, f')) \cdot m \\ &= n + \#(x, [f'/y]t') \cdot m \end{aligned}$$

as required.

[Other cases] Trivial. □

**Proof of Lemma 10** (Takahashi's Property).

*Statement.*  $e \twoheadrightarrow t \implies t \twoheadrightarrow e^*$ .

*Proof.* Induction on  $e$  with case analysis on  $e$ .

[If  $e \equiv c$ ]  $c \twoheadrightarrow c \equiv t \equiv c^*$ .

[If  $e \equiv x$ ]  $x \twoheadrightarrow x \equiv t \equiv x^*$ .

[If  $e \equiv \lambda x.e'$ ]

(i)  $t \equiv \lambda x.t'$     (ii)  $e' \twoheadrightarrow t'$  by inversion

$t' \twoheadrightarrow (e')^*$  by IH on (ii)

$\lambda x.t' \twoheadrightarrow \lambda x.(e')^*$  by PR-ABS

and  $\lambda x.(e')^*$  is just  $e^*$ .

[If  $e \equiv c d$  and  $(c, d) \in \text{dom } \delta$ ]  $c d \twoheadrightarrow \delta(c, d) \equiv t \equiv (c d)^*$ .

[If  $e \equiv (\lambda x.e') a$  where  $e' \in E^0$ ]

$$\left. \begin{array}{ll} \text{(i) } t \equiv (\lambda x.t') a' \vee t \equiv [a'/x]t' \\ \text{(ii) } e' \longrightarrow t' \quad \text{(iii) } a \longrightarrow a' \end{array} \right\} \quad \text{by inversion} \quad (1)$$

where the shape of  $t$  depends on whether the last rule used to derive  $e \longrightarrow t$  is PR-APP or PR- $\beta$ .

$$\text{(i) } t' \longrightarrow (e')^* \quad \text{(ii) } a' \longrightarrow a^* \quad \text{by IH on (1.ii)(1.iii)} \quad (2)$$

$$\text{(i) } (e')^* \in E^0 \quad \text{(ii) } a^* \in \text{Arg} \quad \text{using Lemma 103 on (2)} \quad (3)$$

$$t \longrightarrow [a^*/x](e')^* \equiv ((\lambda x.e') a)^* \quad (4)$$

where the  $(\longrightarrow)$  in (4) follows by applying, to (2)(3), the rule PR- $\beta$  if  $t \equiv (\lambda x.t') a'$  or Lemma 104 if  $t \equiv [a'/x]t'$ ; the  $\beta$  expansion under  $\cdot^*$  is justified by the definition of  $\cdot^*$ .

[If  $e \equiv e_1 e_2$  and  $e$  is not a  $\beta$  or  $\delta$  redex]

$$\text{(i) } t \equiv t_1 t_2 \quad \text{(ii) } e_i \longrightarrow t_i \ (i = 1, 2) \quad \text{by inversion}$$

$$t_i \longrightarrow e_i^* \ (i = 1, 2) \quad \text{by IH on (ii)}$$

$$t \longrightarrow e_1^* e_2^* \equiv e^*. \quad \text{by PR-APP and definition of } \cdot^*$$

[If  $e \equiv \sim \langle e' \rangle$ ]

$$\text{(i) } t \equiv t' \vee t \equiv \sim \langle t' \rangle \quad \text{(ii) } e' \longrightarrow t' \quad \text{by inversion}$$

$$t' \longrightarrow (e')^* \quad \text{by IH on (ii)}$$

$$\sim \langle t' \rangle \longrightarrow (e')^* \equiv (\sim \langle e' \rangle)^* \quad \text{by PR-}E \text{ and definition of } \cdot^*$$

[If  $e \equiv ! \langle e' \rangle$  where  $e' \in E^0$ ] Similar to the preceding case.

[If  $e \equiv \langle e' \rangle$  or  $e \equiv \sim e'$  or  $e \equiv ! e'$ , and  $e$  is not a redex] Immediate from IH.  $\square$

### Proof of Proposition 11.

*Statement.*  $(\rightarrow^*)$  is confluent:  $e_1 \leftarrow^n e \rightarrow^k e_2 \implies \exists e'. e_1 \rightarrow^k e' \leftarrow^n e_2$ .

*Proof.* Induction on  $(n, k)$  under lexicographical ordering.

[If  $n = 0$ ]  $e \equiv e_1$ , so take  $e' \stackrel{\text{def}}{=} e_2$ .

[If  $k = 0$ ]  $e \equiv e_2$ , so take  $e' \stackrel{\text{def}}{=} e_1$ .

[If  $n, k > 0$ ]

$$\begin{array}{llll}
\exists e'_1 & e_1 \leftarrow e'_1 \leftarrow^{n-1} e \rightarrow^k e_2 & \text{because } n > 0 & \\
\exists e_3 & e'_1 \rightarrow^k e_3 \leftarrow^{n-1} e_2 & \text{by IH} & (1) \\
\exists e'_3 & e_1 \leftarrow e'_1 \rightarrow^{k-1} e'_3 \rightarrow e_3 & \text{by (1) and } k > 0 & (2) \\
\exists e_4 & e_1 \rightarrow^{k-1} e_4 \leftarrow e'_3 & \text{by IH} & (3) \\
& e_4 \leftarrow e'_3 \rightarrow e_3 & \text{by (2)(3)} & \\
& e_4 \rightarrow (e'_3)^* \leftarrow e_3 & \text{by Takahashi's property} & (4) \\
& e_1 \rightarrow^k (e'_3)^* \leftarrow^n e_2 & \text{by (1)(3)(4)} & \square
\end{array}$$

### C.3.2 Standardization

This section provides proof details of Lemma 17, which is traditionally proved via a “standardization” lemma and we therefore call standardization itself by abuse of terminology. Takahashi’s method obviates the need to define an auxiliary standard reduction, however.

#### Proof of Lemma 14 (Transition).

*Statement.* If  $e \in E^\ell$  and  $v \in V^\ell$  then  $e \xrightarrow{n} v \implies \exists u \in V^\ell. e \rightsquigarrow_\ell^* u \rightarrow v$ .

*Proof.* The conclusion is obvious when  $e \in V^\ell$ , so assume  $e \notin V^\ell$ . Lexicographically induct on  $(n, e)$ , with case analysis on the last rule used to derive the parallel reduc-



tion. Note that before invoking IH on a sub-judgment  $e' \xrightarrow{n} t$  of  $e \xrightarrow{n} v$ , where  $n' \leq n$  and  $e'$  is a subterm of  $e$ , the side conditions  $e' \in E^k$  and  $t \in V^k$  must be checked (where  $k$  is  $\ell$  or  $\ell \pm 1$  depending upon the shape of  $e$ ). If  $k > 0$ , checking the levels of  $e'$  and  $t$  suffice; otherwise, their shapes must be analyzed.

[PR-CONST]  $e \equiv v \equiv c$ , so take  $u \stackrel{\text{def}}{=} c$ .

[PR-VAR] Vacuous:  $x \notin V^\ell$  so  $v \not\equiv x$ .

[PR-ABS]

$$(i) e \equiv \lambda x.t \quad (ii) v \equiv \lambda x.t' \quad (iii) t \xrightarrow{n} t' \quad \text{by inversion} \quad (1)$$

$$(i) \ell > 0 \quad (ii) t \in E^\ell \quad \text{because } \lambda x.t \in E^\ell \setminus V^\ell \quad (2)$$

$$t' \in V^\ell \quad \text{because } \lambda x.t' \in V^\ell \text{ and } \ell > 0 \quad (3)$$

(2.ii) and (3) justify using IH on (1.iii).

$$(i) t \sim_\ell^* u \longrightarrow t' \quad (ii) u \in V^\ell \quad \text{by IH on (1.iii)} \quad (4)$$

$$\lambda x.t \sim_\ell^* \lambda x.u \longrightarrow \lambda x.t' \quad \text{from (4.i)}$$

where  $\lambda x.u \in V^\ell$  because  $u \in V^\ell$  and  $\ell > 0$ .

[PR- $\delta$ ] By inversion  $e \equiv c \ d \longrightarrow \delta(c, d) \equiv v$  where  $(c, d) \in \text{dom } \delta$ . If  $\ell = 0$  take  $u \stackrel{\text{def}}{=} \delta(c, d)$ , otherwise take  $u \stackrel{\text{def}}{=} c \ d$ : then the given constraints are satisfied.

[PR-APP]

$$\left. \begin{array}{ll} (i) e \equiv e_1 \ e_2 & (ii) v \equiv v_1 \ v_2 \\ (iii) e_i \xrightarrow{n_i} v_i \ (i = 1, 2) & (iv) n = n_1 + n_2 \end{array} \right\} \quad \text{by inversion} \quad (5)$$

$$(i) \ell > 0 \quad (ii) v_1, v_2 \in V^\ell \quad \text{from } v \in V^\ell \quad (6)$$

$$e_1, e_2 \in E^\ell \quad \text{from } e \in E^\ell \quad (7)$$

(6.ii) and (7) justify using IH on (5.iii).

$$(i) e_i \rightsquigarrow_\ell^* u_i \longrightarrow v_i \ (i = 1, 2) \quad (ii) u_1, u_2 \in V^\ell \quad \text{by IH on (5.iii)} \quad (8)$$

$$e_1 \ e_2 \rightsquigarrow_\ell^* u_1 \ e_2 \quad \text{by (8.i) and } \bullet e_2 \in ECtx^{\ell, \ell}$$

$$u_1 \ e_2 \rightsquigarrow_\ell^* u_1 \ u_2 \longrightarrow v_1 \ v_2 \quad \text{noting } u_1 \bullet \in ECtx^{\ell, \ell} \text{ from (6.i)}$$

and  $u_1 \ u_2 \in V^\ell$  by (6.i) and (8.ii).

[PR- $\beta$ ]

$$\left. \begin{array}{ll} (i) e \equiv (\lambda x.t) a & (ii) v \equiv [a'/x]t' \\ (iii) t \xrightarrow{n_1} t' & (iv) a \xrightarrow{n_2} a' \\ (v) n = n_1 + \#(x, t') \cdot n_2 + 1 \end{array} \right\} \quad \text{by inversion} \quad (9)$$

$$(i) \ell = 0 \quad (ii) t \in E^0 \quad \text{because } e \in E^0 \setminus V^\ell \quad (10)$$

$$a' \in Arg \quad \text{by Lemma 103 and (9.iv)} \quad (11)$$

$$[a/x]t \xrightarrow{n-1} [a'/x]t' \quad \text{by Lemma 104 and (11)} \quad (12)$$

$$[a/x]t \in E^0 \quad \text{by Lemma 85 and (10.ii)} \quad (13)$$

IH can be invoked on (12) because (13) and  $[a'/x]t' \equiv v \in V^0$ .

$$(i) [a/x]t \rightsquigarrow_0^* u \longrightarrow [a'/x]t' \quad (ii) u \in V^0 \quad \text{by IH} \quad (14)$$

$$e \rightsquigarrow_0 [a/x]t \rightsquigarrow_0^* u \longrightarrow [a'/x]t' \quad \text{by SS-}\beta$$

[PR-BRK, PR-ESC, or PR-RUN] All of these cases are similar. PR-BRK is worked out here as an example.

$$(i) e \equiv \langle e' \rangle \quad (ii) v \equiv \langle v' \rangle \quad (iii) e' \xrightarrow{n} v' \quad \text{by inversion} \quad (15)$$

$$e' \in E^{\ell+1} \quad \text{because } \langle e' \rangle \in E^\ell \quad (16)$$

$$v' \in V^{\ell+1} \quad \text{because } \langle v' \rangle \in V^\ell \quad (17)$$

(16)(17) justify using IH on (15.iii).

$$\begin{array}{lll}
 \text{(i) } e' \underset{\ell+1}{\rightsquigarrow}^* u' \longrightarrow v' & \text{(ii) } u' \in V^{\ell+1} & \text{by IH on (15.iii)} \\
 \langle e' \rangle \underset{\ell}{\rightsquigarrow}^* \langle u' \rangle \longrightarrow \langle v' \rangle & & \text{from (18.i)}
 \end{array} \tag{18}$$

where  $\langle u' \rangle \in V^\ell$  by (18.ii).

[PR- $E$ ]

$$\text{(i) } e \equiv \sim \langle e' \rangle \quad \text{(ii) } e' \xrightarrow[n-1]{} v \quad \text{by inversion} \tag{19}$$

$$\text{(i) } \ell > 0 \quad \text{(ii) } e' \in E^\ell \quad \text{because } \sim \langle e' \rangle \in E^\ell \tag{20}$$

Invoking IH on (19.ii) is justified by (20.ii) and the assumption that  $v \in V^\ell$ .

$$\begin{array}{lll}
 \text{(i) } e' \underset{\ell}{\rightsquigarrow}^* u \longrightarrow v & \text{(ii) } u \in V^\ell & \text{by IH on (19.ii)} \\
 \sim \langle e' \rangle \underset{\ell}{\rightsquigarrow}^* \sim \langle u \rangle & & \text{from (21.i)}
 \end{array} \tag{21}$$

Then split cases on  $\ell$ .

[If  $\ell = 1$ ]

$$\begin{array}{ll}
 \sim \langle u \rangle \underset{1}{\rightsquigarrow} u & \text{by } E_V \text{ using (21.ii)} \\
 \sim \langle e' \rangle \underset{1}{\rightsquigarrow}^* \sim \langle u \rangle \underset{1}{\rightsquigarrow} u \longrightarrow v & \text{immediately}
 \end{array}$$

[If  $\ell > 1$ ]

$$\sim \langle e' \rangle \underset{1}{\rightsquigarrow}^* \sim \langle u \rangle \longrightarrow v \quad \text{by (21.i)}$$

where

$$\langle u \rangle \in E^{\ell-2} = V^{\ell-1} \quad \text{by (21.ii) and } \ell > 1$$

$$\sim \langle u \rangle \in V^\ell \quad \text{immediately}$$

[PR-R]

$$(i) e \equiv !\langle e' \rangle \quad (ii) e' \xrightarrow[n-1]{} v \quad (iii) e' \in E^0 \quad \text{by inversion} \quad (22)$$

$$\ell = 0 \quad \text{because } !\langle e' \rangle \notin V^\ell \quad (23)$$

while (22.iii)

IH on (22.ii) is justified by (22.iii) and the assumption  $v \in V^\ell$ .

$$(i) e' \rightsquigarrow_0^* u \longrightarrow v \quad (ii) u \in V^0 \quad \text{by IH on (22.ii)} \quad (24)$$

$$!\langle e' \rangle \rightsquigarrow_0 e' \quad \text{by } R_V \text{ using (22.iii)}$$

$$!\langle e' \rangle \rightsquigarrow_0 e' \rightsquigarrow_0^* u \longrightarrow v \quad \text{immediately} \quad \square$$

**Proof of Lemma 15** (Permutation).

*Statement.* If  $e, t, s \in E^\ell$  then  $e \xrightarrow{n} t \rightsquigarrow_\ell s \implies \exists t' \in E^\ell. e \rightsquigarrow_\ell^+ t' \longrightarrow s$ .

*Proof.* Induction on  $n$  with case analysis on the last rule used to derive  $e \xrightarrow{n} t$ . In all cases but PR- $\beta$ , the complexity  $n$  obviously diminishes in the IH, so we omit this check in other cases. In fact, except for PR- $\beta$  we omit the complexity annotation altogether.

[PR-CONST] Vacuous:  $e \equiv c \equiv t$  so  $t \not\rightsquigarrow_\ell$ .

[PR-VAR] Vacuous:  $e \equiv x \equiv t$  so  $t \not\rightsquigarrow_\ell$ .

[PR-ABS]

$$(i) e \equiv \lambda x. e' \quad (ii) t \equiv \lambda x. t' \quad (iii) e' \longrightarrow t' \quad \text{by inversion} \quad (1)$$

$$(i) \ell > 0 \quad (ii) s \equiv \lambda x. s' \quad (iii) t' \rightsquigarrow_\ell s' \quad \text{by inversion on } \lambda x. t' \rightsquigarrow_\ell s \quad (2)$$

$$e' \longrightarrow t' \rightsquigarrow_\ell s' \quad \text{by (1.iii)(2.iii)}$$

$$e' \rightsquigarrow_\ell^+ t'' \longrightarrow s' \quad \text{by IH} \quad (3)$$

$$\lambda x.e' \rightsquigarrow_{\ell}^+ \lambda x.t'' \longrightarrow \lambda x.s' \quad \text{from (3) and (2.i)}$$

[PR-APP]

$$(i) e \equiv e_1 e_2 \quad (ii) t \equiv t_1 t_2 \quad (iii) e_i \longrightarrow t_i \ (i = 1, 2) \quad \text{by inversion} \quad (4)$$

Inversion on  $t_1 t_2 \rightsquigarrow_{\ell} s$  generates four cases.

[If  $t_1 t_2 \rightsquigarrow_{\ell} s$  is derived by SS- $\delta$ ]

$$\left. \begin{array}{ll} (i) \ell = 0 & (ii) t_i \equiv c_i \ (i = 1, 2) \\ (iii) (c_1, c_2) \in \text{dom } \delta & (iv) s \equiv \delta(c_1, c_2) \end{array} \right\} \quad \text{by inversion} \quad (5)$$

$$\exists v_i. e_i \rightsquigarrow_{\ell}^* v_i \longrightarrow c_i \quad \text{by Transition on} \quad (6)$$

(4.iii)(5.ii)

$$v_i \equiv c_i \quad \text{by Lemma 102 and (6)}$$

$$e_1 e_2 \rightsquigarrow_0^* c_1 e_2 \rightsquigarrow_0^* c_1 c_2 \rightsquigarrow_0 \delta(c_1, c_2) \equiv s \quad \text{noting } (\bullet e_2), (c_1 \bullet) \in ECtx^{0,0}$$

$$e_1 e_2 \rightsquigarrow_0^+ \delta(c_1, c_2) \longrightarrow s \quad \text{since } (\longrightarrow) \text{ is reflexive}$$

[If  $t_1 t_2 \rightsquigarrow_{\ell} s$  is derived by SS- $\beta$ ]

$$\left. \begin{array}{ll} (i) \ell = 0 & (ii) t_1 \equiv \lambda x.t_3 \\ (iii) t_2 \in Arg & (iv) s \equiv [t_2/x]t_3 \end{array} \right\} \quad \text{by inversion} \quad (7)$$

Noting that  $t_1 \in V^0$ ,

$$(i) e_1 \rightsquigarrow_0^* v_1 \longrightarrow t_1 \quad (ii) v_1 \in V^0 \quad \text{by Transition on (7.iii)} \quad (8)$$

$$v_1 \equiv \lambda x.e_3 \quad \text{by Lemma 102 with (7.ii)(8.i)} \quad (9)$$

$$e_3 \longrightarrow t_3 \quad \text{by inversion on (8.i) using (7.ii)(9)} \quad (10)$$

Now split cases by evaluation strategy.

[CBV] Observing that  $t_2 \in Arg = V^0$ ,

$$(i) e_2 \rightsquigarrow_0^* v_2 \longrightarrow t_2 \quad (ii) v_2 \in V^0 \quad \text{by Transition on (7.iii)} \quad (11)$$

$$e_1 e_2 \rightsquigarrow_0^* (\lambda x.e_3) v_2 \rightsquigarrow_0 [v_2/x]e_3 \quad \text{by (11.i)(9) and SS-}\beta \quad (12)$$

$$[v_2/x]e_3 \longrightarrow [t_2/x]t_3 \quad \text{by Lemma 104 using (10)(11.i)} \quad (13)$$

$$e_1 e_2 \rightsquigarrow_0^+ [v_2/x]e_3 \longrightarrow s \quad \text{by (12)(13)(7.iv)}$$

[CBN] Observing that  $t_2 \in Arg = E^0$ ,

$$e_1 e_2 \rightsquigarrow_0^* (\lambda x.e_3) t_2 \rightsquigarrow_0 [t_2/x]e_3 \quad \text{by (11.i)(9) and SS-}\beta \quad (14)$$

$$[t_2/x]e_3 \longrightarrow [t_2/x]t_3 \quad \text{by Lemma 104 using (10)} \quad (15)$$

$$e_1 e_2 \rightsquigarrow_0^+ [t_2/x]e_3 \longrightarrow s \quad \text{by (14)(15)(7.iv)}$$

[If  $t_1 \in V^\ell$  but  $t_1 t_2$  is not a  $\beta$  redex]

$$(t_1 \bullet) \in ECtx^{\ell,\ell} \quad \text{because } t_1 \in V^\ell \quad (16)$$

$$(i) s \equiv t_1 s_2 \quad (ii) t_2 \rightsquigarrow_\ell s_2 \quad \text{by Proposition 93 and (16)} \quad (17)$$

$$e_2 \longrightarrow t_2 \rightsquigarrow_\ell s_2 \quad \text{by (4.iii)(17)}$$

$$(i) e_2 \rightsquigarrow_\ell^+ s'_2 \longrightarrow s_2 \quad (ii) s'_2 \in E^\ell \quad \text{by IH} \quad (18)$$

$$e_1 e_2 \rightsquigarrow_\ell^+ e_1 s'_2 \longrightarrow t_1 s_2 \quad \text{from (18.i)(4.iii)}$$

[If  $t_1 \notin V^\ell$ ]

$$(\bullet t_2) \in ECtx^{\ell,\ell} \quad \text{clearly} \quad (19)$$

$$(i) s \equiv s_1 t_2 \quad (ii) t_1 \rightsquigarrow_\ell s_1 \quad \text{by Proposition 93 and (19)}$$

$$(iii) e_1 \rightsquigarrow_\ell^+ s'_1 \longrightarrow s_1 \quad (iv) s'_1 \in E^\ell \quad \text{by IH} \quad (20)$$

$$e_1 e_2 \rightsquigarrow_\ell^+ s'_1 e_2 \longrightarrow s_1 t_2 \quad \text{from (20.i)(4.iii)}$$

[PR- $\delta$ ] Vacuous: by inversion  $t \in V^0$  so  $t$  cannot step.

[PR- $\beta$ ] This is the only case where it's nontrivial to check that the complexity diminishes.

$$\left. \begin{array}{ll} \text{(i)} \ e \equiv (\lambda x.e') \ a & \text{(ii)} \ t \equiv [a'/x]e'' \\ \text{(iii)} \ e' \xrightarrow{n_1} e'' & \text{(iv)} \ a \xrightarrow{n_2} a' \\ \text{(v)} \ e' \in E^0 & \\ \text{(vi)} \ n = n_1 + \#(x, e'') \cdot n_2 + 1 & \end{array} \right\} \text{by inversion} \quad (21)$$

$$a' \in \text{Arg} \quad \text{by Lemma 103 and (21.iv)} \quad (22)$$

$$[a/x]e' \xrightarrow{n_1 + \#(x, e'') \cdot n_2} [a'/x]e'' \quad \text{by Lemma 104 and} \quad (23)$$

(21.iii)(21.iv)(22)

Observe that the complexity is indeed smaller. Noting (21.ii),

$$\text{(i)} \ [a/x]e' \xrightarrow{\ell}^+ t' \longrightarrow s \quad \text{(ii)} \ t' \in E^\ell \quad \text{by IH on (23) and } t \xrightarrow{\ell} s \quad (24)$$

Now, to connect (24.i) to  $e$ :

$$e \xrightarrow{0} [a/x]e' \quad \text{by SS-}\beta_v \quad (25)$$

$$[a/x]e' \in E^0 \quad \text{by (25)} \quad (26)$$

$$\ell = 0 \quad \text{because (24.i)(26)} \quad (27)$$

$$e \xrightarrow{0} [a/x]e' \xrightarrow{0}^+ t' \longrightarrow s \quad \text{by (24.i)(25)(27)}$$

[PR-Esc]

$$\text{(i)} \ e \equiv \sim e' \quad \text{(ii)} \ t \equiv \sim t' \quad \text{(iii)} \ e' \longrightarrow t' \quad \text{by inversion} \quad (28)$$

$$\text{(i)} \ \ell > 0 \quad \text{(ii)} \ t', e' \in E^{\ell-1} \quad \text{because } \sim e', \sim t' \in E^\ell \quad (29)$$

Inversion on  $\sim t' \xrightarrow{\ell} s$  generates two cases.

[If  $\sim t' \rightsquigarrow_{\ell} s$  is derived by SS- $E_V$ ]

$$(i) \ell = 1 \quad (ii) t' \equiv \langle s \rangle \quad (iii) s \in E^0 \quad \text{by inversion} \quad (30)$$

$$t' \in V^0 \quad \text{by (30.ii)(30.iii)} \quad (31)$$

$$(i) e' \rightsquigarrow_0^* v \twoheadrightarrow t' \quad (ii) v \in V^0 \quad \text{by Transition,} \quad (32)$$

using (28.iii)(31)

$$(i) v \equiv \langle s' \rangle \quad (ii) s' \in E^0 \quad \text{by Lemma 102,} \quad (33)$$

using (30.ii)(30.iii)(31)(32)

$$\sim \langle s' \rangle \rightsquigarrow_1 s' \quad \text{by SS-}E_V \quad (33)$$

$$s' \twoheadrightarrow s \quad \text{by inversion on (32.i)} \quad (34)$$

$$\sim \langle e' \rangle \rightsquigarrow_1^* \sim \langle s' \rangle \rightsquigarrow_1 s' \twoheadrightarrow s \quad \text{by (32.i)(34)(33)}$$

[If  $t'$  small-steps]

$$(i) s \equiv \sim s' \quad (ii) t' \rightsquigarrow_{\ell-1} s' \quad \text{by inversion} \quad (35)$$

$$e' \twoheadrightarrow t' \rightsquigarrow_{\ell-1} s' \quad \text{by (28.iii)(35.ii)}$$

$$e' \rightsquigarrow_{\ell-1}^+ t' \twoheadrightarrow s' \quad \text{by IH}$$

$$\sim e' \rightsquigarrow_{\ell}^+ \sim t' \twoheadrightarrow \sim s' \quad \text{immediately}$$

[PR- $E$ ]

$$(i) e \equiv \sim \langle e' \rangle \quad (ii) e' \twoheadrightarrow t \quad \text{by inversion} \quad (36)$$

$$(i) \ell > 0 \quad (ii) e' \in E^{\ell} \quad \text{because } \sim \langle e' \rangle \in E^{\ell} \quad (37)$$

$$e' \twoheadrightarrow t \rightsquigarrow_{\ell} s \quad \text{by (36.ii) and assumption}$$

$$(i) e' \rightsquigarrow_{\ell}^+ t' \twoheadrightarrow s \quad (ii) t' \in E^{\ell} \quad \text{by IH, justified by (37.ii)} \quad (38)$$

$$\sim \langle e' \rangle \rightsquigarrow_{\ell}^+ \sim \langle t' \rangle \twoheadrightarrow s \quad \text{using SS-CTX and PR-}E$$

where (38.ii) guarantees  $\sim \langle t' \rangle \in E^{\ell}$ .



[PR-RUN]

$$(i) e \equiv !e' \quad (ii) t \equiv !t' \quad (iii) e' \twoheadrightarrow t' \quad (iv) e', t' \in E^\ell \quad \text{by inversion} \quad (39)$$

Inversion on  $!t' \rightsquigarrow_\ell s$  generates two cases.[If  $!t' \rightsquigarrow_\ell s$  is derived by SS- $R_V$ ]

$$(i) \ell = 0 \quad (ii) t' \equiv \langle s \rangle \quad (iii) s \in E^0 \quad \text{by inversion} \quad (40)$$

$$(i) e' \rightsquigarrow_0^* v \twoheadrightarrow \langle s \rangle \quad (ii) v \in V^0 \quad \text{by Transition, using} \quad (41)$$

(39.iii)(40.ii)(40.iii)

$$(i) v \equiv \langle s' \rangle \quad (ii) s' \in E^0 \quad \text{by Lemma 102} \quad (42)$$

$$s' \twoheadrightarrow s \quad \text{by inversion on (41.i),} \quad (43)$$

using (42)

$$!\langle e' \rangle \rightsquigarrow_0^* !\langle s' \rangle \rightsquigarrow_0 s' \twoheadrightarrow s \quad \text{from (41.i)(42.i)(43)}$$

[If  $t'$  small-steps]

$$(i) s \equiv !s' \quad (ii) t' \rightsquigarrow_\ell s' \quad (iii) s' \in E^\ell \quad \text{by inversion} \quad (44)$$

$$e' \twoheadrightarrow t' \rightsquigarrow_\ell s' \quad \text{by (39.iii)(44.ii)} \quad (45)$$

$$(i) e' \rightsquigarrow_\ell t'' \twoheadrightarrow s' \quad (ii) t'' \in E^\ell \quad \text{by IH}$$

$$!e' \rightsquigarrow_\ell !t'' \twoheadrightarrow !s' \quad \text{immediately}$$

[PR- $R$ ]

$$(i) e \equiv !\langle e' \rangle \quad (ii) e' \twoheadrightarrow t \quad (iii) e', t \in E^0 \quad \text{by inversion} \quad (46)$$

$$e' \twoheadrightarrow t \rightsquigarrow_\ell s \quad \text{by (46.ii) and premise}$$

$$\ell = 0 \quad \text{because (46.iii) but } t \rightsquigarrow_\ell s$$

$$(i) e' \rightsquigarrow_0^+ t' \twoheadrightarrow s \quad (ii) t' \in E^\ell \quad \text{by IH}$$

$$!\langle e' \rangle \rightsquigarrow_0 e' \rightsquigarrow_0^+ t' \twoheadrightarrow s \quad \text{immediately} \quad \square$$

## C.4 Proofs for Generalized Axioms

This section fills in the details of proofs that showed the unsoundness of some equations in Chapter 3.3.

### Proof of Proposition 20.

*Statement.*  $\forall C. \exists L(C) \in \mathbb{N}. \text{lv } e \geq L(C) \implies \text{lv } C[e] = \text{lv } e + \Delta C.$

*Proof.* Induction on  $C$ .

[If  $C \equiv \bullet$ ] Take  $L(\bullet) \stackrel{\text{def}}{=} 0$ ; then  $\text{lv } C[e] = \text{lv } e + L(\bullet).$

For the remaining cases, I will take the existence of  $L(C')$  for granted, where  $C'$  names the immediate subcontext of  $C$ . This assumption is justified by IH. In each case,  $\text{lv } e \geq L(C)$  is implicitly assumed once  $L(C)$  is defined.

[If  $C \equiv \lambda x. C'$  or  $! C'$ ] Take  $L(C) \stackrel{\text{def}}{=} L(C')$ . Then  $\text{lv } C[e] = \text{lv } C'[e] = \text{lv } e + L(C') = \text{lv } e + L(C).$

[If  $C \equiv t \ C'$  or  $C' \ t$ ] Take  $L(C) \stackrel{\text{def}}{=} \max(L(C'), \text{lv } t - \Delta C')$ . Then  $\text{lv } e + \Delta C' \geq L(C) + \Delta C' \geq \text{lv } t - \Delta C' + \Delta C' = \text{lv } t$ , so  $\text{lv } C[e] = \max(\text{lv } t, \text{lv } e + \Delta C') = \text{lv } e + \Delta C' = \text{lv } e + \Delta C$ . Note that taking the maximum with  $L(C')$  is necessary to justify IH.

[If  $C \equiv \langle C' \rangle$ ] Take  $L(C) \stackrel{\text{def}}{=} \max(L(C'), 1 - \Delta C')$ . Then  $\text{lv } e + \Delta C' - 1 \geq 1 - \Delta C' + \Delta C' - 1 = 0$ , so  $\text{lv } C[t] = \max(\text{lv } C'[t] - 1, 0) = \text{lv } e + \Delta C' - 1 = \text{lv } e + \Delta \langle C' \rangle$ .

Note that taking the maximum with  $L(C')$  is necessary to justify IH.

[If  $C \equiv \sim C'$ ] Take  $L(C) \stackrel{\text{def}}{=} L(C')$ . Then  $\text{lv } C[e] = \text{lv } C'[t] + 1 = \text{lv } e + \Delta C' + 1 = \text{lv } e + \Delta(\sim C').$  □

**Lemma 105.**  $\text{lv } e \leq \text{size}(e).$

*Proof.* Straightforward induction on  $e$ . □

**Proof of Lemma 22** (Context Domination).

*Statement.*  $\ell > \text{size}(C) \implies C \in ECtx^{\ell,m}$ .

*Proof.* Induction on  $C$ . There is a precondition  $\ell' > \text{size}(C')$  for applying IH to the subcontext  $C'$  to obtain  $C' \in ECtx^{\ell',m}$ . This precondition holds because IH is invoked with  $\text{size}(C') \leq \text{size}(C) - 1$  and  $\ell - 1 \leq \ell'$  in each case.

[If  $C \equiv \bullet$ ] Clearly  $C \in ECtx^{\ell,\ell}$ , and  $m \stackrel{\text{def}}{=} \ell$  satisfies  $\Delta \bullet(m) = \ell$ .

[If  $C \equiv C' e$ ]

$$C' \in ECtx^{\ell,m} \quad \text{by IH} \quad (1)$$

$$\text{lv } e \leq \text{size}(C) < \ell \quad \text{using Lemma 105} \quad (2)$$

$$e \in E^\ell \quad \text{immediately} \quad (3)$$

$$C' e \in ECtx^{\ell,m} \quad \text{by (1)(3)}$$

[If  $C \equiv e C'$ ]

$$C' \in ECtx^{\ell,m} \quad \text{by IH} \quad (4)$$

$$\text{lv } e \leq \text{size}(C) < \ell \quad \text{using Lemma 105} \quad (5)$$

$$e \in E^{\ell-1} = V^\ell \quad \text{immediately} \quad (6)$$

$$C' e \in ECtx^{\ell,m} \quad \text{by (4)(6)}$$

[If  $C \equiv \langle C' \rangle$ ] IH gives  $C' \in ECtx^{\ell-1,m}$ , so  $\langle C' \rangle \in ECtx^{\ell,m}$ .

[If  $C \equiv \sim C'$ ] IH gives  $C' \in ECtx^{\ell+1,m}$ , so  $\sim C' \in ECtx^{\ell,m}$ .

[If  $C \equiv !C'$ ] IH gives  $C' \in ECtx^{\ell,m}$ , so  $!C' \in ECtx^{\ell,m}$ . □

## C.5 Proofs for Extensionality

The proof of Proposition 44 (extensionality) relies on commutation of substitution, concluding  $[a/x]\sigma e \approx [a/x]\sigma t$  from  $\forall b. [b/x]e \approx [b/x]t$ . This inference is justified in

this section.

**Lemma 106.** Given a simultaneous substitution  $\sigma : Var \xrightarrow{\text{fin}} Arg \stackrel{\text{def}}{=} [\overline{a_i}/\overline{x_i}]$  and a pair of expressions  $e$  and  $t$ , there exists a sequential substitution  $\sigma' \stackrel{\text{def}}{=} [\overline{c_j}/\overline{z_j}]^j [\overline{b_i}/\overline{x_i}]^i$  such that  $\sigma'e = \sigma e$  and  $\sigma't = \sigma t$ . Furthermore, the order in which the variables  $\overline{x_i}$  are substituted for is arbitrary.

*Proof.* Observe that if either  $\forall i, j. x_i \notin \text{FV}(a_j)$  or  $\forall i. a_i \in Var$  then a simultaneous substitution  $[\overline{a_i}/\overline{x_i}]$  can be made sequential as  $[\overline{a_i}/\overline{x_i}]$ . Furthermore, in the former case the individual substitutions  $[a_i/x_i]$  commute with each other, so their order is arbitrary. Thus using fresh variables  $\overline{z_i}$ ,

$$[\overline{a_i}/\overline{x_i}] \equiv [\overline{x_i}/\overline{z_i}] \left[ \overline{[\overline{z_j^j}/\overline{x_j^j}] a_i^i / \overline{x_i}} \right] \equiv [\overline{x_i}/\overline{z_i}]^i \left[ \overline{[\overline{z_j^j}/\overline{x_j^j}] a_i / z_i} \right]^i.$$

But, unfortunately,  $x_i \notin Arg$  in CBV, so the substitution  $[\overline{x_i}/\overline{z_i}]$  does not have the signature  $Var \xrightarrow{\text{fin}} Arg$ . By giving up syntactic equality between  $\sigma$  and  $\sigma'$ , the new substitution can be made to have the required signature. Choose an arbitrary  $v \in V_{\text{cl}}^0$  and substitute a stuck expression  $z_i v$  instead of just  $z_i$  for  $x_i$ , then substitute  $\lambda_{..}x_i$  for  $z_i$  to resolve this stuck application and contract it to  $x_i$  by  $\beta$  substitution. We get:

$$\sigma' \equiv [\overline{\lambda_{..}x_i}/\overline{z_i}] \left[ \overline{[\overline{z_j} v^j / \overline{x_j^j}] a_i^i / \overline{x_i}} \right] \equiv [\overline{\lambda_{..}x_i}/\overline{z_i}]^i \left[ \overline{[\overline{z_j} v / \overline{x_j}]^j a_i / x_i} \right]^i$$

Note that  $z_j v \notin Arg$  is not a problem: this lemma only asserts  $\forall i. [\overline{z_j v / x_j}]^j a_i \in Arg$ , which follows from Lemma 87. Then for each  $i$ ,

$$\sigma' x_i \equiv \left[ \overline{\lambda_{..}x_j^j / \overline{z_j^j}} \right] \left[ \overline{z_j v^j / \overline{x_j^j}} \right] a_i \equiv \left[ \overline{(\lambda_{..}x_j) v^j / \overline{x_j^j}} \right] a_i = [\overline{x_j^j} / \overline{x_j^j}] a_i \equiv a_i \equiv \sigma x_i.$$

I omit the trivial induction argument that this equality extends to  $\sigma'e = \sigma e$  and  $\sigma't = \sigma t$ . □

**Remark.** The only reason we refer to a pair of expressions instead of one expression in Lemma 106 is because we need fresh variables  $\bar{z}_i$ . If each  $z_i$  is requested to be fresh for only  $\sigma$  and  $e$ , then it might fail to be fresh for  $t$ .

**Lemma 107.**  $\forall a. [a/x]e \approx [a/x]t \implies \forall \sigma. \forall a. [a/x]\sigma e \approx [a/x]\sigma t$ .

*Proof.* Without loss of generality,  $\sigma \equiv [\bar{b}_i^{i \in I} / \bar{x}_i^{i \in I}]$  where  $I = \{1, 2, \dots, \# \text{dom } \sigma\}$ . If  $x \in \text{dom } \sigma$ , then  $[a/x]\sigma \equiv [\overline{[a/x]b_i^{i \in I}} / \bar{x}_i^{i \in I}]$  and  $\exists i. x \equiv x_i$ ; if not, then  $[a/x]\sigma \equiv [\bar{b}_i^{i \in (\{0\} \cup I)} / \bar{x}_i^{i \in (\{0\} \cup I)}]$  where  $b_0 \stackrel{\text{def}}{=} a$  and  $x_0 \stackrel{\text{def}}{=} x$ . Either way,  $[a/x]\sigma$  is equal to a single parallel substitution of the form  $\sigma' \stackrel{\text{def}}{=} [\bar{b}_i / \bar{x}_i]$  such that  $\exists i. x_i \equiv x$ .

By Lemma 106, there exists a sequential substitution  $\overline{[a_i/x_i]}[a'/x]$  such that  $\sigma'e = \overline{[a_i/x_i]}[a'/x]e$  and  $\sigma't = \overline{[a_i/x_i]}[a'/x]t$ . Note that the lemma explicitly states that we can require  $x$  to be substituted first. Then

$$[a/x]\sigma e \equiv \sigma'e = \overline{[a_i/x_i]}[a/x]e \approx \overline{[a_i/x_i]}[a'/x]t = \sigma't \equiv [a/x]\sigma t. \quad \square$$

## C.6 Proofs for Soundness and Completeness of Applicative Bisimulation

This section provides proof details pertaining to the soundness and completeness of indexed applicative bisimulation. The main text gives a high-level explanation of the entire proof, so this section will fill in just the missing pieces without repeating the explanations.

### Proof of Proposition 51.

*Statement.* Define applicative mutual similarity as  $\overline{(\sim'_X)} \stackrel{\text{def}}{=} (\overline{\lesssim_X}) \cap (\overline{\gtrsim_X})$ . Then  $\overline{(\sim_X)} = \overline{\sim'_X}$ .

*Proof.* We show double containment.

[For  $(\sim_X) \subseteq (\sim'_X)$ ] Obviously  $\forall X. (\sim_X) = [\sim]_X \cap [\sim^{-1}]_X^{-1} \subseteq [\sim]_X$ , so  $\overline{(\sim_X)} \subseteq \overline{(\lesssim_X)}$ . By the way,  $\forall X. (\sim_X^{-1}) = [\sim]_X^{-1} \cap [\sim^{-1}]_X = [\sim^{-1}]_X [(\sim^{-1})^{-1}]_X^{-1}$ , so by coinduction  $\overline{(\sim_X^{-1})} \subseteq \overline{(\sim_X)}$ . It follows that  $\overline{(\sim_X^{-1})} \subseteq \overline{(\lesssim_X)}$ , i.e.  $\overline{(\sim_X)} \subseteq \overline{(\gtrsim_X)}$ . Therefore,  $\overline{(\sim_X)} \subseteq \overline{(\gtrsim_X)} \cap \overline{(\lesssim_X)} = \overline{(\sim'_X)}$ .

[For  $(\sim'_X) \subseteq (\sim_X)$ ] Suppose  $e \sim'_X t$  and let  $\ell = \max(\text{lv } e, \text{lv } t)$ . Then we have  $e \lesssim_X t$ , so whenever  $\sigma e \Downarrow^\ell u$  for some  $\sigma, u$  we have  $\sigma t \Downarrow^0 v$  and  $u \{\overline{\lesssim_X}\}^\ell v$ . Furthermore we have  $t \lesssim_X e$ , so  $\sigma t \Downarrow^0 v$  implies  $\sigma e \Downarrow^0 u'$  and  $v \{\overline{\lesssim_X}\}^\ell u'$ . Evaluation is deterministic, so  $u \equiv u'$ —therefore,  $u \{\overline{\sim'_X}\}^\ell v$ . Arguing similarly, we can show  $\exists v. \sigma t \Downarrow^\ell v \implies \exists u. \sigma e \Downarrow^\ell u \wedge u \{\overline{\sim'_X}\}^\ell v$ . Hence  $\overline{(\sim'_X)} \subseteq [\sim']_X \cap [(\sim')^{-1}]_X^{-1}$ ; then by coinduction,  $\overline{(\sim'_X)} \subseteq \overline{(\sim_X)}$ .  $\square$

**Lemma 108.** If  $\overline{R_X}$  is a family of relations, then

- (i)  $e [R]_X t \implies \forall \sigma : X | \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e [R]_{X \setminus \text{dom } \sigma} \sigma t$  and
- (ii)  $\forall Y \subseteq X. ((\forall \sigma : Y | \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e [R]_{X \setminus \text{dom } \sigma} \sigma t) \implies e [R]_X t)$ .

*Proof.*

- (i) Suppose  $e [R]_X t$  and let  $\sigma : \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  be given. Then for any  $\sigma' : (X \setminus \text{dom } \sigma) | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$ , the composition of the substitutions satisfies  $\sigma' \sigma : X | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$ , where  $(\sigma' \sigma) e \stackrel{\text{def}}{=} \sigma'(\sigma e)$ . Thus by assumption  $\sigma'(\sigma e) \Downarrow^\ell v \implies (\sigma'(\sigma t)) \Downarrow^\ell u \wedge v \{\overline{R_Y}\}^\ell u$ . Therefore,  $(\sigma e) R_{X \setminus \text{dom } \sigma}(\sigma t)$ .
- (ii) Suppose  $\forall \sigma : Y | \text{Var} \xrightarrow{\text{fin}} \text{Arg}. (\sigma e) [R]_{X \setminus \text{dom } \sigma}(\sigma t)$  for some  $Y$  and let any  $\sigma' : X | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  be given. By the assumption  $Y \subseteq X \subseteq \text{dom } \sigma'$  and Lemma 106,  $\sigma'$  can be decomposed as  $\sigma' \equiv \sigma'' \sigma$  for some  $\sigma : Y | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  and  $\sigma'' : X \setminus \text{dom } \sigma | \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  such that  $\sigma''(\sigma e) = \sigma' e$  and  $\sigma''(\sigma t) = \sigma' t$ . Then by assumption  $(\sigma e) [R]_{X \setminus \text{dom } \sigma}(\sigma t)$ , so  $\sigma''(\sigma e) \Downarrow^\ell v \implies (\sigma''(\sigma t)) \Downarrow^\ell u \wedge v \{R\}^\ell u$ .

The statement about  $(\lesssim_\emptyset)$  follows by taking  $R = (\lesssim)$  and  $Y = X$ .  $\square$

**Proof of Lemma 54.**

*Statement.*  $e \lesssim_X t \iff \forall \sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}. \sigma e \lesssim_\emptyset \sigma t$ .

*Proof.* Immediate from Lemma 108.  $\square$

**Lemma 109.**  $e \widehat{\lesssim}_X t \wedge a \widehat{\lesssim}_{X \setminus \{x\}} b \implies [a/x]e \widehat{\lesssim}_{X \setminus \{x\}} [b/x]t$ .

*Proof.* Induction on  $e$  with case analysis on  $e$ .

[If  $e \equiv x$ ]

$$\begin{array}{ll} x \lesssim_X t & \text{by inversion} \\ [b/x]x \lesssim_{X \setminus \{x\}} [b/x]t & \text{by Lemma 54} \end{array} \quad (1)$$

$$[a/x]x \widehat{\lesssim}_{X \setminus \{x\}} [b/x]x \quad \text{because } a \widehat{\lesssim}_{X \setminus \{x\}} b \text{ by assumption} \quad (2)$$

$$[a/x]x \widehat{\lesssim}_{X \setminus \{x\}} [b/x]t \quad \text{by Proposition 59 (iii) and (1)(2)}$$

[If  $e \equiv \tau \overline{e_i} \neq \lambda x.e^0$  (note: includes the case  $e \equiv c$ )]

$$(i) \forall i. e_i \widehat{\lesssim}_X s_i \quad (ii) \tau \overline{s_i} \lesssim_X t \quad \text{by assumption} \quad (3)$$

$$\forall i. [a/x]e_i \widehat{\lesssim}_{X \setminus \{x\}} [b/x]s_i \quad \text{by IH on (3.i)} \quad (4)$$

$$\left. \begin{array}{l} \tau \overline{[a/x]e_i} \equiv [a/x](\tau \overline{e_i}) \\ \tau \overline{[b/x]s_i} \equiv [b/x](\tau \overline{s_i}) \end{array} \right\} \quad \text{since } \tau \neq x \text{ and } \tau \neq \lambda x.\bullet \quad (5)$$

$$[a/x](\tau \overline{e_i}) \widehat{\lesssim}_{X \setminus \{x\}} [b/x](\tau \overline{s_i}) \quad \text{by (4)(5) and Proposition 59 (ii)} \quad (6)$$

$$[b/x](\tau \overline{s_i}) \lesssim_{X \setminus \{x\}} [b/x]t \quad \text{by (3.ii) using Lemma 54} \quad (7)$$

$$[a/x](\tau \overline{e_i}) \widehat{\lesssim}_{X \setminus \{x\}} [b/x]t \quad \text{by (6)(7) using Proposition 59 (iii)}$$

[If  $e \equiv \lambda y.e_1$  where  $e_1 \in E^0$ ]

$$(i) e_1 \widehat{\lesssim}_Y s_1 \quad (ii) \lambda y.s_1 \lesssim_X t \quad (iii) Y \setminus \{y\} = X \quad \text{by inversion} \quad (8)$$

$$[a/x]e_1 \widehat{\lesssim}_{Y \setminus \{x\}} [b/x]s_1 \quad \text{by IH on (8.i)} \quad (9)$$

$$\left. \begin{array}{l} \lambda y.[a/x]e_1 \equiv [a/x]\lambda y.e_1 \\ \lambda y.[b/x]s_1 \equiv [b/x]\lambda y.s_1 \end{array} \right\} \quad \text{using BVC} \quad (10)$$

$$[a/x]\lambda y.e_1 \in E^0 \quad \text{by Lemma 85 and } e_1 \in E^0 \quad (11)$$

$$[a/x]\lambda y.e_1 \widehat{\lesssim}_{Y \setminus \{x,y\}} [b/x]\lambda y.s_1 \quad \text{by (9)(10)(11)}$$

$$[a/x]\lambda y.e_1 \widehat{\lesssim}_{X \setminus \{x\}} [b/x]\lambda y.s_1 \quad \text{using (8.ii)} \quad (12)$$

$$[b/x]\lambda y.s_1 \lesssim_{X \setminus \{x\}} [b/x]t \quad \text{by (8.ii) and Lemma 54} \quad (13)$$

$$[a/x]\lambda y.e_1 \lesssim_{X \setminus \{x\}} [b/x]t \quad \text{by (12)(13) and Lemma 54}$$

□

**Lemma 110.**  $v \{\widehat{\lesssim}_X\}^\ell w \{\lesssim_X\}^\ell u \implies v \{\widehat{\lesssim}_X\}^\ell u.$

*Proof.* Easily confirmed by inspecting Definition 50 using Proposition 59 (iii). □

**Lemma 111.** If  $u, v \in V^\ell$ , then  $u \{\widehat{\lesssim}_X\}^\ell v \implies u \widehat{\lesssim}_\emptyset v.$

*Proof.* Straightforward case analysis on  $\ell$ , then on the form of  $u$ , using reflexivity and context respecting property of  $\widehat{\lesssim}_X$ . □

### Proof of Lemma 60.

*Statement.*  $e \widehat{\lesssim}_X t \implies e [\lesssim_X]t$

*Proof.* Let  $\ell = \max(\text{lv } e, \text{lv } t)$  and  $e \equiv \tau \overline{e_i}$ . Fix a  $\sigma$  and assume  $\sigma e \rightsquigarrow_\ell^n v$ . Inversion gives some  $s \equiv \tau \overline{s_i}$  and  $Y$  where

$$(i) \ \forall i. e_i \widehat{\lesssim}_Y s_i \quad (ii) \ s \lesssim_X t \quad (1)$$



and either  $Y \setminus \{x\} = X$  if  $e \equiv \lambda x.e^0$  for some  $e^0$  or  $Y = X$  otherwise. We will show  $\sigma t \Downarrow^\ell u \wedge v \{\widehat{\lesssim}_X\}^\ell u$  by lexicographic induction on  $(n, e)$  with case analysis on the form of  $e$ . But before delving into the main induction, we note two simplifications.

Firstly, we may assume  $s \in E^\ell$  without loss of generality, by thinking of the induction scheme to be  $(n, e, m)$  rather than  $(n, e)$ , where  $m \stackrel{\text{def}}{=} \text{lv } s$ . Whenever  $m > \ell$  we have  $\sigma e \Downarrow^m \sigma e$ ,  $\sigma t \Downarrow^m \sigma t$ , and  $\sigma e_i \Downarrow^{m'} \sigma e_i$ , all terminating in zero steps, and where

$$m' = \begin{cases} m + 1 & \text{if } \tau \equiv \langle \bullet \rangle \\ m - 1 & \text{if } \tau \equiv \sim \bullet \\ m + 1 & \text{else} \end{cases}$$

Then invoking IH on  $e_i \widehat{\lesssim}_Y s_i$  gives  $\sigma s_i \Downarrow^{m'} w_i$  and  $\sigma e_i \{\widehat{\lesssim}_X\}^{m'} w_i$ , from which Lemma 111 gives  $\sigma e_i \widehat{\lesssim}_X w_i$ . Given that  $m > 0$ , the only way in which  $\forall i. w_i \in V^{m'}$  can fail to imply  $\tau \overline{w_i} \in V^m$  is to have  $m = 1$ , and  $\tau \equiv \sim \bullet$ , and  $w_1 \equiv \langle s^0 \rangle$  for some  $s^0$ . But then  $m > \ell \geq \text{lv}(\sim e_1) \geq 1$ , so this does not happen. Hence  $\forall i. \sigma e_i \widehat{\lesssim}_Y w_i$ ,  $\tau \overline{w_i} \lesssim_X \sigma t$ , and  $\tau \overline{w_i} \in V^m = E^{m-1}$ , so we have effectively lowered  $m$ —hence the conclusion follows by IH.

Secondly, now having the assumption  $s \in E^\ell$ , all that we need to establish is  $\sigma s \Downarrow^\ell w$  for some  $w$  such that  $v \{\widehat{\lesssim}_X\}^\ell w$ . For then  $s \lesssim_X t$ , or more to the point  $s [\lesssim]_X t$ , gives  $\exists u. \sigma t \Downarrow^\ell u$  and  $w \{\widehat{\lesssim}_X\}^\ell u$ . Then by Lemma 110 it follows that  $v \{\widehat{\lesssim}_X\}^\ell u$ .

To summarize, all we have to do is to fix  $\sigma : X \mid \text{Var} \xrightarrow{\text{fin}} \text{Arg}$  and  $s \equiv \tau \overline{s_i}$ , assume  $s \in E^\ell$ , (1), and  $\sigma e \rightsquigarrow_\ell^n v$  where  $\ell = \max(\text{lv } e, \text{lv } t)$ , then to show by lexicographic induction on  $(n, e)$  that  $\sigma s \Downarrow^\ell w$  and  $u \{\widehat{\lesssim}_X\}^\ell w$ . Note that the only case where  $Y$  may not equal  $X$  is when  $e$  has the form  $\lambda x.e^0$ .

[If  $e \equiv x$ ]  $x \equiv \tau \equiv e \equiv s$ , so the conclusion follows by noting that  $\{\widehat{\lesssim}_X\}^\ell$  is reflexive since  $\widehat{\lesssim}_X$  is.

[If  $e \equiv c$ ] Same as preceding case.

[If  $e \equiv e_1 e_2$ ] Noting that  $\bullet \sigma e_2 \in Ectx^{\ell, \ell}$  regardless of the value of  $\ell$  or evaluation strategy, we have

$$\left. \begin{array}{ll} \text{(i)} \sigma e_1 \rightsquigarrow_{\ell}^{n_1} v_1 & \text{(ii)} v_1 \in V^{\ell} \\ \text{(iii)} v_1 (\sigma e_2) \rightsquigarrow_{\ell}^{n-n_1} v & \text{(iv)} n_1 \leq n \end{array} \right\} \text{ by Lemma 94} \quad (2)$$

$$\text{(i)} \sigma s_1 \Downarrow^{\ell} w_1 \quad \text{(ii)} v_1 \{\widehat{\sim}_X\}^{\ell} w_1 \quad \text{by IH on (2.i) using (2.iv)} \quad (3)$$

The rest of this case's proof depends on  $\ell$ , the shape of  $v_1$ , and the evaluation strategy.

[If  $\ell = 0$ ,  $v_1 \equiv \lambda x.e'_1$ , and we are in CBN]

$$\text{(i)} w_1 \equiv \lambda x.s'_1 \quad \text{(ii)} e'_1 \widehat{\sim}_{\{x\}} s'_1 \quad \text{by Lemma 94} \quad (4)$$

$$\sigma e_2 \widehat{\sim}_{\emptyset} \sigma s_2 \quad \text{from (1.i) by Lemma 109} \quad (5)$$

$$[\sigma e_2/x]e'_1 \widehat{\sim}_{\emptyset} [\sigma s_2/x]s'_1 \quad \text{from (4.ii)(5) by Lemma 109} \quad (6)$$

$$[\sigma e_2/x]e'_1 \rightsquigarrow_{\ell}^{n-n_1-1} v \quad \text{from (4.iii) and } v_1 \equiv \lambda x.e'_1 \quad (7)$$

The conclusion follows from IH on (6), justified by (7).

[Else] It must be the case that either  $\ell > 0$ ,  $v_1 \equiv \lambda x.e'_1$  where  $e'_1 \in E^0$ , or  $v_1 \equiv c$  for some  $c$ , so we have  $v_1 \bullet \in Ectx^{\ell, \ell}$ . Therefore,

$$\left. \begin{array}{ll} \text{(i)} \sigma e_2 \rightsquigarrow_{\ell}^{n_2} v_2 & \text{(ii)} v_2 \in V^{\ell} \\ \text{(iii)} v_1 v_2 \rightsquigarrow_{\ell}^{(n-n_1-n_2)} v & \text{(iv)} n_2 \leq n \end{array} \right\} \text{ from (2.iii) by Lemma 94} \quad (8)$$

$$\text{(i)} \sigma s_2 \Downarrow^{\ell} w_2 \quad \text{(ii)} v_2 \{\widehat{\sim}_X\}^{\ell} w_2 \quad \text{by IH on (8.i) using (8.iv)} \quad (9)$$

[If  $\ell > 0$ ] We have  $v_1 v_2, w_1 w_2 \in V^{\ell}$ , and:

$$v_i \widehat{\sim}_{\emptyset} w_i \quad (i = 1, 2) \quad \text{from (3.ii)(9.ii) since } \ell > 0 \quad (10)$$

$$\begin{aligned}
v_1 \ v_2 &\widehat{\lesssim}_{\emptyset} w_1 \ w_2 && \text{by context-respecting property} \\
v_1 \ v_2 &\{\widehat{\lesssim}_X\}^\ell w_1 \ w_2 && \text{because } \ell > 0
\end{aligned}$$

[If  $\ell = 0$ ,  $v_1 \equiv \lambda x.e'_1$ , and we are in CBV]

$$(i) \ w_1 \equiv \lambda x.s'_1 \quad (ii) \ e'_1 \widehat{\lesssim}_{\{x\}} s'_1 \quad \text{by Lemma 94} \quad (11)$$

$$v_2 \widehat{\lesssim}_{\emptyset} w_2 \quad \text{from (9.ii) by Lemma 111} \quad (12)$$

$$[v_2/x]e'_1 \widehat{\lesssim}_{\emptyset} [w_2/x]s'_1 \quad \text{from (16.ii)(12) by Lemma 109} \quad (13)$$

$$[v_2/x]e'_1 \rightsquigarrow_\ell^{n-n_1-n_2-1} v \quad \text{from (16.iii) and } v_1 \equiv \lambda x.e'_1 \quad (14)$$

The conclusion follows from IH on (13), justified by (14).

[If  $\ell = 0$ ,  $v_1 \ v_2 \equiv c \ d$ , and  $(c, d) \in \text{dom } \delta$ ]

$$w_1 \ w_2 \equiv c \ d \quad \text{by (3.ii)(9.ii)} \quad (15)$$

$$\left. \begin{aligned}
\sigma e \rightsquigarrow_0^* v_1 \ v_2 \rightsquigarrow_0^* \delta(c, d) \\
\sigma s \rightsquigarrow_0^* w_1 \ w_2 \rightsquigarrow_0^* \delta(c, d)
\end{aligned} \right\} \text{by (3.ii)(9.ii)} \quad (16)$$

This concludes the case when  $e$  is an application.

[If  $e \equiv \lambda x.e_1$ ]

[If  $\ell = 0$ ] This is the only case where  $Y \setminus \{x\} = X$  and not necessarily  $Y = X$ .

We have  $\sigma e, \sigma s \in V^0$ , so these expressions terminate to themselves and we are to show  $\sigma e \{\widehat{\lesssim}_X\}^0 \sigma s$ .

$$\sigma e_1 \widehat{\lesssim}_{Y \setminus X} \sigma s_1 \quad \text{by (1.i) and Lemma 109} \quad (17)$$

$$Y \setminus X = \{x\} \vee Y \setminus X = \emptyset \quad \text{from } Y \setminus \{x\} = X \quad (18)$$

$$\sigma e_1 \widehat{\lesssim}_{\{x\}} \sigma s_1 \quad \text{from (17)(18) monotonicity of } (\widehat{\lesssim}_X)$$

$$\lambda x.\sigma e_1 \{\widehat{\lesssim}_X\}^\ell \sigma \lambda x.s_1 \quad \text{by definition}$$

$$\sigma e \equiv \sigma \lambda x.e_1 \{\widehat{\lesssim}_X\}^\ell \sigma \lambda x.s_1 \equiv \sigma s \quad \text{by BVC}$$

[If  $\ell > 0$ ] By BVC,  $\sigma e \equiv \lambda x. \sigma e_1$  and  $\sigma s \equiv \lambda x. \sigma s_1$ . Thus, noting that  $\lambda x. \bullet \in$

$$ECtx^{\ell, \ell},$$

$$(i) \sigma e_1 \xrightarrow{\ell}^n v' \quad (ii) v \equiv \lambda x. v' \quad (iii) v \in V^\ell \quad \text{using Lemma 94} \quad (19)$$

$$(i) \sigma s_1 \Downarrow^\ell w' \quad (ii) v' \{\widehat{\sim}_X\}^\ell w' \quad (iii) w' \in V^\ell \quad \text{by IH on (19.i)} \quad (20)$$

$$v' \widehat{\sim}_\emptyset w' \quad \text{from (20.ii) and } \ell > 0$$

$$\lambda x. v' \widehat{\sim}_\emptyset \lambda x. w' \quad \text{by Proposition 59 (ii)}$$

$$\lambda x. v' \{\widehat{\sim}_X\}^\ell \lambda x. w' \quad \text{immediately}$$

[If  $e \equiv \langle e_1 \rangle$ ] Noting that  $\langle \bullet \rangle \in ECtx^{\ell, \ell+1}$ ,

$$(i) \sigma e_1 \xrightarrow{\ell+1}^{n'} v' \quad (ii) v' \in V^{\ell+1} \quad (iii) n' \leq n \quad \text{by Lemma 94} \quad (21)$$

$$(i) \sigma s' \Downarrow^{\ell+1} w' \quad (ii) v' \{\widehat{\sim}_X\}^{\ell+1} w' \quad (iii) w' \in V^{\ell+1} \quad \text{by IH on (21.i)} \quad (22)$$

$$v' \widehat{\sim}_\emptyset w' \quad \text{by (22.ii)} \quad (23)$$

$$\langle v' \rangle \widehat{\sim}_\emptyset \langle w' \rangle \quad \text{by Proposition 59 (ii)} \quad (24)$$

$$\langle v' \rangle \{\widehat{\sim}_X\}^\ell \langle w' \rangle \quad \text{by (23) if } \ell = 0; \text{ or}$$

$$\text{by (24) if } \ell > 0$$

[If  $e \equiv \sim e_1$ ]

$$\ell > 0 \quad \text{because } \sim e_1 \in E^\ell \quad (25)$$

$$\sim \bullet \in ECtx^{\ell, \ell-1} \quad \text{by (25)} \quad (26)$$

$$(i) \sigma e_1 \xrightarrow{\ell-1}^{n'} v' \quad (ii) v' \in V^{\ell-1} \quad (iii) n' \leq n \quad \text{by Lemma 94 using (26)} \quad (27)$$

$$\left. \begin{array}{l} (i) \sigma s_1 \Downarrow^{\ell-1} w' \quad (ii) v' \{\widehat{\sim}_X\}^{\ell-1} w' \\ (iii) w' \in V^{\ell-1} \end{array} \right\} \quad \text{by IH on (27.i) using (27.iii)} \quad (28)$$

Split cases on  $\ell$ .

[If  $\ell = 1$ ] Clearly  $\sim v' \notin E^0 = V^1$ , so  $\sim v'$  must small-step at level 1.

$$(i) v' \equiv \langle e'_1 \rangle \quad (ii) e'_1 \in E^0 = V^1 \quad \text{by inversion, using (27.ii)} \quad (29)$$

$$(i) w' \equiv \langle s'_1 \rangle \quad (ii) s'_1 \in E^0 = V^1 \quad (iii) e'_1 \widehat{\lesssim}_{\emptyset} s'_1 \quad \text{by (28.ii)(29.i)} \quad (30)$$

$$\sim s_1 \rightsquigarrow^* \sim \langle s'_1 \rangle \Downarrow^1 s'_1 \quad \text{from (28.i)(30.i)}$$

$$e'_1 \{\widehat{\lesssim}_X\}^1 s'_1 \quad \text{from (30.iii)}$$

[If  $\ell > 1$ ]

$$v', w' \in V^{\ell-1} = E^{\ell-2} \quad \text{because } \ell > 1$$

$$\sim v', \sim w' \in V^{\ell-1} = E^{\ell-2} \quad \text{immediately}$$

$$v' \widehat{\lesssim}_{\emptyset} w' \quad \text{by (28.ii) and } \ell - 1 > 0$$

$$\sim v' \widehat{\lesssim}_{\emptyset} \sim w' \quad \text{by Proposition 59 (ii)}$$

$$\sim v' \{\widehat{\lesssim}_X\}^{\ell} \sim w' \quad \text{immediately}$$

[If  $e \equiv !e_1$ ] Noting that  $!\bullet \in ECtx^{\ell, \ell}$ ,

$$(i) \sigma e_1 \rightsquigarrow_{\ell}^{n'} v' \quad (ii) v \in V^{\ell} \quad (iii) n' \leq n \quad \text{by Lemma 94} \quad (31)$$

$$(i) \sigma s_1 \Downarrow^{\ell} w' \quad (ii) v' \{\widehat{\lesssim}_X\}^{\ell} w' \quad (iii) w' \in V^{\ell} \quad \text{by IH on (31.i)} \quad (32)$$

Split cases on  $\ell$ .

[If  $\ell = 0$ ] Since  $!v' \in V^0$ , the  $!v'$  must small-step.

$$\left. \begin{array}{ll} (i) v' \equiv \langle e'_1 \rangle & (ii) e'_1 \rightsquigarrow_0^{n''} v \\ (iii) e'_1 \in E^0 & (iv) n'' \leq n - n' - 1 \end{array} \right\} \quad \text{by inversion} \quad (33)$$

$$(i) w' \equiv \langle s'_1 \rangle \quad (ii) e'_1 \widehat{\lesssim}_{\emptyset} s'_1 \quad (iii) s'_1 \in E^0 \quad \text{by (32.ii)(32.iii)(33.ii)} \quad (34)$$

$$(i) s'_1 \Downarrow^0 w \quad (ii) v \{\widehat{\lesssim}_X\}^0 w \quad \text{by IH on (33.ii) using (33.iv)} \quad (35)$$

$$\sigma(!\langle s_1 \rangle) \rightsquigarrow_0^* !\langle s'_1 \rangle \rightsquigarrow_0 s'_1 \Downarrow^0 w \quad \text{by (32) and (36.i)} \quad (36)$$

[If  $\ell > 0$ ]

$$!v', !w' \in V^{\ell} \quad \text{by (31.ii)(32.iii)}$$

$$v' \widehat{\sim}_{\varnothing} w'$$

by (32.ii)

$$!v' \widehat{\sim}_{\varnothing} !w'$$

by Proposition 59 (ii)

$$!v' \{\overline{\widehat{\sim}_X}\}^{\ell} !w'$$

immediately

□

## Appendix D

### Summary of Notations

The following figures summarize the mathematical notations used in this document. Table D.1 lists the syntax for relations and annotations. Entries are sorted in order of appearance in the main text.

at level $\ell$ (constraint)	$e^\ell, v^\ell$ , etc.	finite iteration	$R^n$
hole	$\bullet$	parallel reduction	$e \xrightarrow{n} t$
term size	$\text{size}(e)$	erasure	$\ e\ $
context size	$\text{size}(C)$	CBN variant	$R_{\mathbf{n}}$
syntactic equality	$e \equiv t$	CBV variant	$R_{\mathbf{v}}$
primitive reduction	$e \xrightarrow{m}^{\text{prim}} t$	careful reduction	$e \longrightarrow_{\mathbf{v}\Downarrow} t$
small-step	$e \rightsquigarrow_{\ell} t$	careful equality	$e =_{\mathbf{v}\Downarrow} t$
observational equivalence	$e \approx t$	observational order	$e \lesssim t$
termination	$e \Downarrow^\ell$	applicative simulation	$e \lesssim_X t$
divergence	$e \Uparrow^\ell$	applicative bisimulation	$e \sim_X t$
reduction	$e \longrightarrow t$	precongruence candidate	$e \hat{\lesssim}_X t$
provable equality	$e = t$		
free variables	$FV(e)$		
reflexive-transitive closure	$R^*$		
transitive closure	$R^+$		

Table D.1 : Summary of notations.