STATIC COMPUTATION AND REFLECTION

Ronald Garcia

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
September 2008

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Andrew Lumsdaine, Ph.D.

Daniel P. Friedman, Ph.D.

Amr Sabry, Ph.D.

R. Kent Dybvig, Ph.D.

Charles Livingston, Ph.D.

September 10, 2008

Copyright 2008
Ronald Garcia
All rights reserved

This dissertation is dedicated to Francine Doret-Gordon, Maud Garcia MacArthur, and Julienne Baptiste Derisier: three amazing women whose lives and guidance have enriched my life more than they can ever know.

Acknowledgements

I thank my parents, Francine Doret-Gordon, Marcel Garcia, and Jean-Claude Gordon, for raising me, for loving me, and for supporting me throughout this process.

I thank the members of my committee: Dan Friedman, Amr Sabry, Kent Dybvig, and Chuck Livingston. Each of them has had an invaluable impact on my education, whether it was engaging my curiosity and fueling my passion for my field, teaching me new things and new ways to think and work, helping me tap my potential, or honoring me with their time and attention.

I thank my colleagues in Computer Science, especially the members of the Open Systems Laboratory. Special thanks go to Jeremy and Katie Siek, Jeremiah Willcock, Jaakko Järvi, and Doug Gregor. It's been a pleasure to work with them and learn from them.

I thank the members of the IU Programming Languages Group for friendship, collaborative learning, feedback, and far too many terrible jokes.

I owe immense thanks to Andy Lumsdaine, my advisor and friend, for the privilege of being his student, for listening to me ramble with no end about nothing in particular, for patiently and steadily believing in me when I could not do the same, and for giving me the space and time to grow into the scholar that I am today.

Special thanks go to Stephen Crowley and John Johnson. They pushed on me when I needed it most and taught me a lot about research, philosophy, scholarship, and friendship.

Most importantly, I would like to thank Suzanna Crage. I can't begin to measure the impact you have had on my life or my respect and love for you. I am forever grateful.

Abstract

Most programming languages do not allow programs to inspect their static type information or perform computations on it. C++, however, lets programmers write template metaprograms, which enable programs to encode static information, perform compile-time computations, and make static decisions about run-time behavior. Many C++ libraries and applications use template metaprogramming to build specialized abstraction mechanisms, implement domain-specific safety checks, and improve run-time performance.

Template metaprogramming is an emergent capability of the C++ type system, and the C++ language specification is informal and imprecise. As a result, template metaprogramming often involves heroic programming feats and often leads to code that is difficult to read and maintain. Furthermore, many template-based code generation and optimization techniques rely on particular compiler implementations, rather than language semantics, for performance gains.

Motivated by the capabilities and techniques of C++ template metaprogramming, this thesis documents some common programming patterns, including static computation, type analysis, generative programming, and the encoding of domain-specific static checks. It also documents notable shortcomings to current practice, including limited support for reflection, semantic ambiguity, and other issues that arise from the pioneering nature of template metaprogramming. Finally, this thesis presents the design of a foundational programming language, motivated by the analysis of template metaprogramming, that allows programs to statically inspect type information, perform computations, and generate code. The language is specified as a core calculus and its capabilities are presented in an idealized setting.

Contents

CHAPTER 1

Introduction

This chapter motivates the work presented in this thesis. It begins by explaining the need for programming languages to help programmers manage the inherent complexity of large applications. Discussion then turns to the essential tension between abstraction and performance as well as the need for safety checking of high-level abstractions. Static (or compile-time) metaprogramming can be used to address performance and safety issues that arise in software development.

1. The Challenges of Software Engineering

Computers keep getting faster, smaller, more powerful, and more pervasive, and with these advances, ambitions for their potential uses increase as well. Computing professionals and casual users continually dream of more and more sophisticated applications of computing technology.

Despite our dreams, however, advances in software do not keep pace with hardware advances. The term *software crisis* was coined at the first NATO Software Engineering Conference in 1968 to refer to the detrimental effects that increasing computing power has had on software complexity [**59**]. Forty years later, software developers still face mounting difficulties with building software that meets the needs and wants of users. To make matters worse, the stakes keep getting higher as society becomes more and more reliant upon software and computers; as more of our data is published to the Internet, and only accessible from computers; as the details of our daily lives rely more and more on computers; as our means of communication and interaction become more computerized; as our knowledge gets stored in computers and retrieved by software; and as the scale of problems we attack becomes so large that we must rely on computers over hand or paper calculations.

The heart of the problem is that with increasing sophistication of computer applications comes increasing inherent complexity of software solutions. More requirements must be understood, specified, and related to each other. Larger software architectures must be crafted to address greater functionality and greater interaction between components. And of course much more software must be implemented to address larger domains and more intricate requirements.

Every large software application imposes a substantial amount of complexity that cannot be removed without fundamentally changing what the application can do [**12**]. This inherent complexity must be managed. Software engineering is a human process, so it is programmers who must use the tools at their disposable to decompose complex software projects into implementable, maintainable, verifiable components that combine to produce the right results.

Programming language design as a discipline directly addresses the concerns of mounting software complexity. Language design involves developing features that programmers can use to build well-organized software. Programming languages provide constructs like procedures, classes, and module systems to help programmers break problems down into manageable chunks. These organizational constructs offer programmers the means to abstract and modularize, to systematically package code in ways that help them organize, understand and reuse them. The most powerful abstractions allow programmers to group, parameterize, and name program components.

For example, consider the C programming language. Two of its primary abstractions are functions and structures (or structs). Structures provide a mechanism for *data abstraction*: a C struct packages one or more data elements under a uniform named interface that can be treated as a distinct semantic unit. C functions provide *procedural abstractions*, a means to group a set of statements into a named and parameterized computation that can be reused throughout a program. Functions are a beneficial way to factor out mostly-repetitive computations: the parameters to a function capture any variance between computations.

One less acknowledged abstraction of C is its file structure, namely header files and source files. C's treatment of files is not as integrated with the language as structs and

functions: header files are handled by the C preprocessor (CPP). Nonetheless, C programmers use file structure to build modular software components; it provides a means to package computations as a component, hide local variable and function names from the rest of the program, and publish an abstract interface that can be used by other components of an application.

Object-oriented programming languages like C++ and Java provide classes and objects as primary abstraction mechanisms. A class is a recipe for creating run-time computational objects that maintain their own internal state and publish a common interface of routines and publicly visible variables. Some languages also provide explicit means to define modules, like C++ namespaces and Java packages. These mechanisms provide a more coarse-grained aggregation and abstraction facility than classes, and integrate more closely with their languages than does file structure as it is used in C.

In each of the languages described above, abstraction and aggregation facilities are used to implement *software libraries*, cohesive collections of data structures and routines that address some problem space. Software libraries are one popular and effective means to tackle the inherent complexity of modern software.

Software libraries help developers break a problem into separate components that can be separately developed. Libraries often address a general application area rather than a particular piece of application software. Many software applications are implemented using externally developed libraries to provide some of their needed functionality. In particular, language implementations usually provide a foundation library of some sort that provides the most basic and often-needed components for software development. Furthermore, both free and commercial libraries implement solutions for a variety of problem domains that require substantial and intimate knowledge to address.

Libraries are also used by development groups to organize the internals of large applications. It may not be that the library is ever visible outside of the development group, but it is still vital to managing internal complexity. In fact, the ability to construct an application from a collection of libraries makes it more possible to approach application development as

a modular design and construction problem. The structure of an application's implementation can reflect the structure of the organization that created it, and facilities for modularity can simplify software construction in a modular organization. Libraries themselves can form hierarchies of reusable components: not only do applications benefit from using libraries, but libraries themselves may benefit from being implemented atop other simpler libraries.

Independently developing a piece of program logic that an available library provides has its costs. Since libraries generally provide an opportunity to reuse the artifacts of previous design, implementation, and validation efforts, foregoing this reuse means that a development group must allocate resources to designing, implementing, and validating functionality that might already be available. Beyond initial implementation, there is also the continuing maintenance and improvement cost of such software, a nontrivial recurring cost over the course of a product's lifetime.

However, choosing to use an externally developed library has its own costs. For starters, using any software interface requires knowledge of its invariants and semantics. As such, using a library requires developers to spend time learning how it works. Furthermore, any given software library must make certain design decisions regarding how it approaches a problem space, and some of these decisions will be manifest in the library interface. When the library interface does not correspond with an application's design, an "impedance mismatch" occurs. Limitations in the flexibility of a programming language's abstraction mechanisms force some design decisions to be made in a library that will limit its applicability; such design decisions are better handled at the application level. This loss of control can restrict the convenience and benefit of using a particular software library until the costs outweigh the benefits.

2. Two Pressing Concerns

Challenges remain to, and are possibly even exacerbated by, our current techniques for decomposing software into modular libraries. In this section we consider two pressing issues that hinder component-oriented design, the process of building software by combining existing software artifacts to create new applications. One is the issue of performance and

abstraction penalty: how our organizational techniques make our programs slower and only get worse as the number of abstraction layers increase. We also consider the issue of correctness: what language facilities can developers of large modular software use to provide stronger assurances of a program's behavior as well as the means to detect implementation bugs.

2.1. Abstraction Penalty and Performance. Programming languages expose a fundamental tension between modularity and performance. Language constructs for program organization are meant to help programmers manage the complexity of sizable software projects, but these constructs also affect what machine code is generated for execution. For instance, functions help programmers organize blocks of code into value-parameterized units that may be given names suggestive of their semantics—a significant aid to programmers who must read and understand code later—and applied in as many locations throughout a program as appropriate.

Computers do not benefit from the organizational properties of functions as humans do, but when compilers translate programs with functions, the resulting executable code is still affected by them. Consider what happens when functions and function calls are implemented on a computer. Functions are generally compiled to a machine code representation of a block of parameterized code, and function calls are compiled to branching operations that pass parameters. Passing parameters and branching introduce overhead into the execution process, including memory or register traffic for preparing function arguments as well as pipeline and cache effects due to branching. Furthermore, the branches introduced by function calls can reduce the opportunities to apply local, intraprocedural, optimizations.

Decomposing problems into small-grained functions helps programmers construct maintainable software, but it also introduces function call overhead in the compiled program and hides opportunities for optimization, thereby undermining run-time performance. Optimizing compilers alleviate some of this overhead by inlining, replacing a function call with a specialized copy of the function implementation. However, excessive inlining can also harm performance and naïve inlining can result in nonterminating compilation when faced with

recursive function calls; for this reason compilers must rely on heuristics and analyses to achieve reasonable performance gains from inlining. Furthermore, function inlining requires access to the body of a function, and this requirement can limit modularity. In fact, some programming systems package libraries of functions as dynamically loaded modules. This binary representation of functions makes it much harder for a programming system or run-time to perform function inlining. When performance is critical, the only recourse may be to sacrifice functional abstractions and inline the relevant code by hand. Some programming languages such as C++ provide means for programmers to explicitly implement functions that can be inlined. User-controlled inlining is not a panacea, but it allows programmers some control over the compilation of their code. Where automated techniques fall short, programmer intervention can help.

Object-oriented programming languages use classes, objects, and late binding in order to solve a number of organizational problems. However, in some cases, dynamic binding is used to resolve problems that fundamentally involve static binding. For example, some object-oriented programming patterns like the Template Method pattern [**22**] describe program compositions that represent relationships between classes and objects that need not vary as a program runs. In these cases, dynamic binding is treated as a universal least common denominator for binding problems. While it simplifies a language to use one mechanism for all such problems, this leads directly to a loss of static information that can be used to improve performance.

These kinds of mapping issues can be seen by investigating how compilers translate other abstractions to machine code, including modules and data structures. Modularity interferes with the ability to optimize away abstractions. The performance impact of organizational constructs on run-time performance is called *abstraction penalty*. Various programming systems already use optimization techniques to improve performance while sacrificing some modularity [**37**, **95**, **97**].

Robison [**70**] argues that the economics of compiler development will limit the number, kind, and specificity of readily available optimizations in production compilers for the coming years. To date the provision of a large suite of extensible custom compiler optimizations

has not been viable. There is at the same time a need to provide mechanisms for expert developers to integrate custom optimizations into application components without sacrificing the benefits of high-level abstractions [**10**, **66**, **98**].

Recent changes in hardware evolution also point to the need for mechanisms to alleviate abstraction penalty. For the past three decades, processor designs have followed a trend where the number of transistors per unit area doubles every 18 months. Processor developers have taken advantage of shrinking transistor sizes to increase clock rates and deepen processor pipelines, thereby increasing serial performance of processors. As a result, software applications have enjoyed passive performance improvements over the course of this trajectory. In some cases, performance gains have made previously unaccepted development techniques, such as high-level languages and dynamic components viable. In other cases, developers use the computing power to build more sophisticated applications or applications that operate on greater scales of data than were previously available. In this manner, software developers have been able to immediately take advantage of performance gains introduced by faster processors.

Transistors are still getting smaller, and transistor counts are still growing, but due to difficulties with cooling processors as clock rates increase, processor designers now dedicate increasing transistor counts to adding more on-chip parallelism, primarily by fitting multiple processor cores on a chip. It is assumed by the hardware industry that software developers will find ways to extract parallel performance from applications in order to speed them up. As a side effect, though, the free lunch of automatic serial performance gain is over: it is not yet known how in general to build highly scalable parallel applications.

Even as performance gains increasingly come from parallel execution, sequential performance may matter more than ever. Amdahl's law states that the performance gains of parallel execution are limited by the proportion of a program that can execute in parallel. If a substantial portion of an application must execute serially, then increases in serial performance are needed to get sizable performance improvements. As automatic hardware-provided performance gains lessen, it becomes more cost-effective to allocate resources to

increasing serial performance on the software/compiler side, whereas in the past it might have sufficed to anticipate the speedups that come from upgrading machines.

2.2. Correctness. To build correct software that is partially built from libraries, developers require proper knowledge of how those libraries should be used, and they must have confidence in the correctness of each library's design and implementation. These criteria impose a significant burden on developers, and that burden increases with the scale of the software application under development.

Software construction is a particularly error-prone affair. Developers often introduce errors into software while implementing it, and those bugs must be detected and eliminated. In general, ensuring the correctness of software is an arduous and involved task. No single technique has been shown to effectively address the problem of constructing correct code. Today developer use methods like unit and functional testing, code reviews, formal specification, and heavyweight verification techniques like model checking, to attempt to decrease the number of defects in software. Practitioners argue that it takes a number of these techniques combined to acquire acceptable confidence in the correctness of software [**52**]. Each substantially different approach seems to help.

Software is prone to defects partly because it is so malleable. When constructing large software projects, developers still operate at a fine level of granularity, often working in low-level languages with some of the most basic units of computation. Whereas other fields of engineering have large standard components that are known to interface with each other in straightforward ways, software engineering still relies on custom-crafted parts [**53**]. Library development is one step toward producing larger-scale standard components for software development. However, due to the internal complexity that must be contained by some libraries, the challenge still remains to ensure the internal correctness of libraries themselves, let alone ensuring that they are used correctly.

Another source of software defects can be seen in the representational nature of software. Programs map particular problem domains into computational artifacts, specifically the data and actions of a particular computer. Early programming languages like C and Fortran

focus on providing a reasonable model of the underlying computer architecture so as to simplify this mapping process. In particular, these languages ensure that the operations that the computer runs are legal with respect to the machine's operation. However, as software addresses more complex domains, more help is needed with the process of representing problem domain structures on a computer. Languages need to go beyond checking what constitutes a legal operation on a computer. In particular, language tools are needed to define what set of operations are legal on the artifacts being modeled in the computer and to enforce constraints on the scope of those operations. In this sense, programming languages must attach semantic significance to program data and operations, so that they more closely model the meaning of the problem space, not just its implementation in the computer. For example, a language like C represents boolean values using integral values: The integer 0 represents the value "false" and any other integer represents the value "true". This representation works for most situations, but a real model of booleans does not support arithmetic operations like addition or subtraction. In C, a boolean value can be accidentally passed to a routine that performs arithmetic upon it. If the boolean abstraction were respected by the language, such abuse would be recognized for the semantic error that it is and the programmer would be alerted of the fault.

As individual modules of a software application are constructed, developers need ways to ensure that the code is correct: that the implementation is correct relative to the requirements and that the implementation is correct relative to the specification. Nonetheless, correctness of each component does not imply correctness of an entire application constructed from them: the components must be connected correctly as well.

Validating large-scale software is an extremely complex, if not intractable problem. As such, mechanisms, tools, and techniques for guaranteeing partial correctness can combine to provide relatively strong confidence in the total correctness of a software application and to remove some of the burden for ensuring correctness from the programmers.

Programming languages provide mechanisms to help with attaining both (knowledge and trust). In particular, programming language type systems provide a means for a compiler or interpreter to verify that a library is internally structured and externally used in

a manner that is consistent with some basic correctness criteria. They enforce some basic structural consistency in the library itself, while providing the user some insight into that structure.

Type systems act as filters on programs that are about to be compiled. Many type checkers take as input a program and at least conceptually return a boolean value indicating whether the program type checks. Following this step, a compiler transforms the same program that was passed as input to the type checking phase into machine code. Languages like Fortran, C, and early C++ and ML are examples of languages with such type systems. Programming language type systems have historically been intended to statically check safety properties of programs and to enable some compiler optimizations.

Since typical type systems are tractable, they are necessarily incomplete. While every well-typed program does not exhibit a certain class of errors, there are programs that cannot pass the type checker but are also correct with respect to the same class of errors. In order to provide nontrivial guarantees, a type system must be conservative, only accepting programs that it can prove to be type safe. Limitations in the scope of type systems limit the proving power of the type system. However, when designing a type system, the intention is generally to avoid placing too much burden upon a programmer in order to produce a correct and well-typed program.

Type systems at their most basic enable a compiler to compare the types assigned to identifiers using basic type equality checks. This functionality, along with the ability to define custom types gives programmers some ability to define types that represent semantic properties of their program variables and to check those properties, but the kinds of checks that can be encoded generally have some limitations. As the desire for stronger static checking has grown, type systems have increased in sophistication. Today, some type systems encode enough information that type checking becomes a more involved process. In fact, some type systems support enough mechanism that programmers can encode non-trivial computations within them.

3. Metaprogramming for Library-Centric Software

Recent advances in the design of typed programming languages have introduced developers to an approach to software development that has had an impact on both abstraction penalty and correctness. The C++ type system has been a platform for experimentation with *template metaprogramming*, a family of programming idioms that leverage the power of C++ templates. A number of useful techniques centered around metaprogramming have been developed and used to implement commercial, research, and open source C++ applications and libraries.

Some C++ programs use template metaprogramming to implement high-level abstractions that avoid unacceptable performance penalties. Examples include the Matrix Template Library (MTL) [**77**], Blitz++ Array Library [**94**], and the POOMA parallel partial differential equation solver [**68**]. In other cases, template metaprogramming is used to build expressive high-level abstractions that encode and enforce custom-tailored semantic safety properties. Examples include the Generative Matrix Computation Library [**15**], the Boost Python interface library [**2**], and the Boost Spirit Parsing Library [**17**]. All these libraries leverage static type information to provide flexible, safe, and efficient software components.

Success with building libraries and applications using template metaprogramming techniques has inspired C++ language experts to investigate new techniques to achieve computationally useful results by exploiting the design of the C++ language. Several books have been written about using template metaprogramming for software development [**4**, **6**, **15**]. Furthermore, the C++ language standard has been altered over the years to better support some template metaprogramming idioms. In fact, an upcoming revision of the C++ language will include new language features that were inspired by template metaprogramming techniques [**28**, **29**, **33**].

4. Project: Metaprogramming Revisited

Template metaprogramming was an emergent property of C++ templates, so much of the power of template metaprogramming is not by explicit design: although C++ designers deliberately designed the template system to be flexible, they did not anticipate all the particular applications of templates in modern C++ programs. As a result, template metaprogramming has some severe shortcomings with respect to supporting those uses. Nonetheless, expert C++ programmers resort to template metaprogramming because it helps them build expressive high-level applications in a mostly portable way using a ubiquitous programming language.

Programmers have adopted C++ template metaprogramming because of its practical utility. However, few would argue that template metaprogramming directly expresses an ideal programming model. Template metaprogramming is notoriously difficult and requires conscious abuse of a number of C++ language facilities. Programmers joke about "the determined Real programmer" who can "write Fortran programs in any language," [**65**], yet today C++ programmers regularly implement metaprograms in a language that was not designed for it. Unfortunately, no language has been designed to explicitly support the kinds of functionality that are written as template metaprograms.

However, the collected knowledge of template metaprogramming, more discovered than designed, can be used to inspire such a design. Underlying the cobbled-together collection of template metaprogramming tricks is a core set of traits that fundamentally embody the power of template metaprogramming. These core traits can form the basis for an intentional metaprogramming framework. Indeed this thesis documents such an approach.

This dissertation lays a foundation for addressing the current lack of intentional and explicit support for the style of metaprogramming that is currently performed in C++. It presents an analysis of pre-existing language mechanisms that enable productive metaprogramming. In particular, it studies the capabilities of the C++ programming language. Programmers and researchers have explored the capabilities of C++ templates, and have produced a family of template metaprogramming idioms. This thesis studies the techniques

used to develop software using C++ template metaprograms and their limitations. It looks at some of the properties of C++ templates that programmers use to perform metaprogramming. It discusses some of the important idioms that these properties of templates enable. The result is a selection of core capabilities that together make template metaprogramming a practical tool for building software components.

A principled analysis of how programmers use C++ template metaprogramming reveals that a well-founded programming language model can support the core capabilities that make template metaprogramming useful. Based on this analysis of C++ templates and the fundamental principles underlying template metaprogramming, this thesis formulates and presents a foundational programming language design, or *kernel language*, that directly supports capabilities found in C++ template metaprogramming, particularly the generation and composition of program fragments based on supplied and inferred static semantic information.

The programming language design augments a simple language with support for compile-time (or static) metaprogramming. The metaprogramming aspects of a program execute prior to run-time and provide the programmer with direct means to control the final structure of a program using static semantic information, including types, to steer compilation. Particular emphasis is placed on support for *static computations* and *static reflection* of information, particularly type information and static data. These features are presented in an idealized language that provides enough common language features to motivate the presence of these richer static semantics.

This language captures the static and dynamic properties of compile-time computation, however the resulting language is rather unwieldy. In order to present a language interface that more closely matches a traditional programming language, the design for a *surface language* is presented. It provides expressive extensions to the kernel language for actually writing metaprograms. This language interleaves metaprogramming code and normal programming code in a lexically scoped manner. The phase distinction between the two language levels is made implicit but well-defined. The language cleanly captures nesting of metalanguage and object-language declarations and seamlessly transitions between the

language layers. It also provides the means to define compile-time computations that look like run-time computations, thereby enabling the creation of some pleasant and effective interfaces to metaprogramming-driven functionality. The surface language has a well-defined type system, and its full semantics are defined by type-directed translation to the kernel language. Thus, the kernel language presents a model for a semantically sound but user-friendly programming interface, while the kernel language specifies a full semantics for static metaprogramming.

In order to liberate metaprogramming from the whims of language implementers, it is necessary to capture a particular semantics for reflection, computation, and code generation in a language's specification. In particular, the static semantics, including the type system, of the programming language must pin down its generative behavior. For this reason, both the static and dynamic semantics of the designed language are formalized, building upon traditional sequent-calculus-based static semantics combined with small-step reduction-based operational semantics. The semantics provides static guarantees about which program fragments compose statically and how. This thesis establishes that the kind of static metaprogramming that has evolved in C++ can be given a firm foundation and can be explicitly and intentionally realized.

5. Layout of this Thesis

The remainder of this thesis is organized as follows. Chapter 2 presents some background that is necessary to understand this thesis. In particular it explains metaprogramming with an emphasis on compile-time metaprogramming in a programming language setting.

Chapter 3 analyzes the core capabilities of C++ templates that make it a useful language for doing metaprogramming. The techniques described illustrate some of the desirable properties of a metaprogramming language. These properties motivate the design presented later.

Chapters 4 and 6 present a design for a foundational metaprogramming language. Chapter 4 presents the *kernel language*, which captures the essence of compile-time metaprogramming in a framework that is easy to reason about. It explains how the language supports metaprogramming with types and how its semantics capture a form of compile-time metaprogramming over an object language. The kernel language uses a number of explicit mechanisms to facilitate presenting a full semantics for static metaprogramming. As such, the language is rather cumbersome to use, even compared to C++ template metaprogramming. Chapter 6 presents the *surface language*, which presents a more implicit and concise language interface for metaprogramming. It also presents examples of how the languages can encode static metaprogramming. These show how the surface language, though defined in terms of the kernel language, provides a coherent and complete interface to static metaprogramming capabilities.

The results of this thesis open the door to future work in designing languages for type-manipulating static metaprogramming. As a guide to future work, Chapter 7 presents some discussion as well as directions in which this work can be developed.

CHAPTER 2

Background

This chapter presents background information needed to understand the contributions of this thesis. First, metaprogramming is introduced, starting with a broad conception but eventually focusing on static metaprogramming languages. Because this work particularly addresses metaprogramming for statically typed languages, type systems in general are discussed. Finally, the design space for compile-time metaprogramming is explored, emphasizing the tradeoffs between different design decisions.

1. Metaprogramming

Metaprogramming in its most general sense refers to the idea that programs can treat other programs as data. A metaprogram takes some representation of a program as input and uses it to produce a result, possibly a new program, as output. The concept is simple and general, and appears in many forms throughout software development. Because it is so fundamental, metaprogramming in general is a broad topic that subsumes a number of technologies, motivations, and principles.

The most naïve conception of metaprogramming considers any program that manipulates another program to be a metaprogram. For instance any programming language interpreter or compiler can be viewed as a metaprogram. An interpreter takes a program as input and performs the actions that it encodes. This idea has been codified as a design pattern: represent data as programs in a small language and write a simple interpreter to process those programs [**22**]. A compiler translates a program in one language into an equivalent program in another language. For example, a C++ compiler like GCC processes C++ programs as its input and yields executable machine code as its output.

Besides compilers for general-purpose languages, some specialized language-based tools are considered instances of metaprogramming. Consider, for instance, the YACC parser generator [**35**], a domain-specific language (DSL) that links context-free grammar productions to blocks of annotated C statements. The YACC processor takes a program written in its input language and generates a parser written in C. The parser recognizes the language defined by the grammar and performs the actions dictated by the C code from the input file. YACC provides some sophisticated features that would be difficult for a programmer to implement from scratch, such as facilities for debugging the input grammar. YACC's robustness and flexibility make its use generally preferable to hand-coding parsers.

This thesis focuses specifically on metaprogramming facilities that are integrated into a programming language. A *metaprogramming language* is a programming language that provides mechanisms that are well-suited to manipulating programs. Such a language provides some means to represent programs, usually programs in the language itself or a subset of the language as data. It provides operations to manipulate elements that represent code. Finally, it provides some way to either export or execute the code that it manipulates.

Metaprogramming facilities have long been available in programming languages. An early example of a language with support for metaprogramming is Lisp [**51**]. Beginning with its first implementation, Lisp supported metaprogramming. The syntax of the programming language mirrored the syntax of its list data structures (s-expressions), so Lisp immediately had an internal representation for its programs as data. Also, Lisp provided an `eval` function which could be given a piece of data that represents a Lisp program and would evaluate that program. Thus Lisp immediately provided facilities for constructing and executing programs at run-time.

The Lisp language was extended later with even greater support for metaprogramming. Lisp macros were introduced in 1963 in an MIT AI memo [**30**] that describes a mechanism for adding new special forms to the language that do not obey the standard evaluation rules. Although calls to these macros use the same syntax as normal Lisp functions, their evaluation differs from that of standard Lisp function calls. When the evaluator encounters one of these special forms, the entire expression is passed as an s-expression to a macro

transformer, which performs computations on it and yields a new program fragment to be evaluated.

Lisp introduced both `eval` and macros into the design space for programming languages. Each of these metaprogramming mechanisms has been adopted by other programming languages: many languages (i.e., Tcl [**61**], Perl [**96**], MetaML [**89**], Metaphor [**60**]) support some mechanism akin to `eval`, while many other languages (i.e., Scheme [**41**], Dylan [**67**], JSE [**7**], Maya [**8**], MacroML [**23**], MetaBorg [**11**], JLE [**92**]) provide macros.

When a language supports metaprogramming, it presents two different conceptions of programs: programs as computational processes and programs as data. To distinguish them, we call the part of a metaprogramming language that performs computations on other programs the *metalanguage* because it is the language in which one writes programs "about" other programs. The metalanguage operates on some representation of programs as data, which we call the *object language*. For instance, when using Lisp's `eval` mechanism, the Lisp language plays both the role of the metalanguage and the object language. The same is true for Lisp macros, even though they are fundamentally different from `eval`.

Sheard [**71**] categorizes the space for metaprogramming systems in the following taxonomy:

Generators versus Analyses. Some metaprogramming systems are specifically geared toward generating new object code and combining them in order to build output programs. Other systems analyze existing object code in order to transform it or generate new code or other artifacts. The C preprocessor and the Lisp macro system exhibit these two notions of metaprogramming: program generation and program analysis. The C preprocessor is only capable of program generation, the piecing together of bits of C code. C preprocessor macros take arguments and generate new text in terms of them, but the result of a macro is parametric over its arguments: they simply plug their arguments into the expected location and return new text. Lisp macros, on the other hand, can decompose their arguments, examine their structure and contents, and use that information to determine the output of the macro.

Representation. Different metaprogramming systems use different mechanisms to represent object programs. Some systems, like XVCL [**99**] work at the level of strings. This representation is effective for capturing and manipulating syntactic patterns in programs that have little structural or semantic content associated to them by the programming language. As a result, the design is also language-neutral, though it loses the benefits gained by having a semantic or structural representation of the object language

Like Lisp, some metaprogramming languages encode object language expressions using the same kinds of data structures that are used for other kinds of data (i.e., Template Haskell [**72**]). Other metaprogramming languages, like MetaML and MetaOCaml [**87**, **89**], provide a special representation for code; Scheme macros [**18**] represent programs as syntax objects and provide special operations to build and decompose them. The C preprocessor represents its arguments as strings of tokens. Since preprocessor macros merely combine strings of tokens, without analyzing them structurally, this is an adequate representation.

Automatic versus Manual. Many metaprogramming systems make an explicit distinction between object language programs and metalanguage programs, while others automatically detect phase differences. In particular, partial evaluation [**38**, **39**] performs an automated binding analysis on programs so as to determine which parts should be evaluated early and which parts left for normal run-time. However, partial evaluation is generally seen more as an automated optimization method than as a metaprogramming technique. Metaprogramming languages generally provide an explicit separation between metalanguage expressions and object language expressions so as to capture the intent of the programmer. For instance, the MetaML language is often compared to the output of automatic binding analysis, the first stage of partial evaluation. MetaML requires user-placed annotations because it does not perform automated binding analysis.

Heterogeneous versus homogeneous. Some integrated metaprogramming languages have multiple distinct language layers, where the metalanguage differs significantly from the object language. Other systems have a single language syntax split across multiple stages of evaluation.

For instance, the C preprocessor is a separate language from the rest of C. Historically, it has had a separate implementation (the program CPP), which executes and passes the result along to the real C compiler. On the other hand, Lisp macros are simply a feature of the Lisp language. The body of a macro is a Lisp function that happens to receive an s-expression and return an s-expression. Metaprogramming is implemented using the same facilities as those of the normal language. On the other hand, Scheme's `syntax-rules` [**41**] macro language is distinct from the run-time Scheme language. It is a somewhat constrained pattern matching and manipulation language that does not include run-time Scheme functionality.

Types. Type systems are a common feature for programming languages. Given that a metaprogramming language involves a metalanguage and an object language (possibly being the same language), there are design decisions to be made about whether and which languages of the two are typed, as well as the relationship between those type systems.

For instance, Lisp is an untyped programming language and its macro language is also untyped. As such, Lisp programs use `eval` to execute untyped programs that are constructed at run-time. In contrast, a multi-stage metaprogramming language like MetaML [**89**] is a typed programming language that also supports a run-time evaluation operator (called `run`). MetaML generates and runs typed programs, and the object language programs that it generates are given *code* types in the metalanguage. When run, a program returns a value corresponding to the type of the code. The main idea behind multi-stage type systems is "correctness by construction". The type system guarantees that a program will neither generate nor run code that results in type errors.

Pre-Scheme [**43**] is a statically typed dialect of Scheme that was used to implement the Scheme 48 interpreter [**42**]. Pre-Scheme supports macros in an untyped fashion. As such, Pre-Scheme is an example of a statically typed object language combined with a dynamically typed metalanguage.

MacroML [**23**, **88**] is a macro language for ML dialects. These syntactic abstractions are particularly interesting in that they are fully type checked. Its macros can be proven to expand to type correct object code prior to macro expansion. Such safety imposes a cost: MacroML has limited expressivity compared to other metaprogramming systems such as

Scheme macros. MacroML is an example of a statically typed object language combined with a statically typed metalanguage.

The shift from dynamic metaprogramming to static metaprogramming changes the focus of types in a language design. Since a static metaprogram runs at compile-time, its execution is complete by the time a program has become a full application that is ready for run-time execution. It's not as necessary for the metalanguage to generate type safe code since it is executed prior to run time. No type errors are at risk of persisting into run time code in this model. Whereas a dynamic metaprogramming language might be more concerned about the safety of generated code for the sake of run-time correctness, this is not an issue for static metaprograms. However, as we discuss later, metalanguage type safety affects the modularity of static checking.

Multi-Level versus Two-Level. In the simplest metaprogramming systems, one programming language is used to manipulate programs written in another programming language. In these cases, the metaprogramming system is considered to have two levels. However, if the manipulating language and the manipulated language are the same, then it becomes possible to write programs that manipulate programs that manipulate programs, *ad infinitum*. Such systems are called multi-level, because their metaprograms can have an arbitrary number of levels. A uniform language induces an infinite tower of metalanguages and object languages. For instance, a Lisp program can use s-expressions to implement and `eval` a program that itself calls `eval` on s-expressions. This capability adds substantial expressive power to the system: the ability to write programs that write programs.

In Scheme the results of a macro call are passed back to the macro expander, so it is possible for a Scheme macro call to introduce a new Scheme macro definition, even though the languages do not coincide completely. So the input to the Scheme macro expander is in the language of Scheme including macro definitions and calls and so is the intermediate output, but the final output is pure macro-less Scheme code. Lisp `eval`, on the other hand, operates on normal Lisp expressions, so the same language exists at both the meta-level and the object level.

Static versus Dynamic. Some metaprogramming languages complete all computations on the object language programs before those programs are ever evaluated. Other systems allow the interleaved evaluation of metaprograms and object programs. These differences are sometimes phrased as "compile-time" versus "run-time" metaprogramming, however these notions can be misleading given the existence of batch interpreters and interactive compilers. We distinguish these notions as *static* and *dynamic* metaprogramming.

The distinctive property of a static metaprogramming system is that meta-level evaluation can always be performed completely before the object language program is evaluated. For instance, Scheme macros [**18**, **45**] are fundamentally static. When a Scheme expression is evaluated in an interactive system, the expression first passes through the macro expansion stage, during which macro definitions are evaluated and used to process macro calls. The final output of macro expansion is a Scheme program that has no macro calls. This program is then evaluated by the Scheme run-time evaluator. Macro systems are usually static, but that is not always the case. Some early variants of Lisp macros expanded macro calls on the fly, while interpreting expressions, and allowed programs to modify existing macro definitions. Some of the ways that programmers took advantage of these behaviors could not be done in a macro system that performs all macro expansion prior to interpretation.

In a dynamic metaprogramming system, evaluation of metalanguage computations may have to occur during evaluation of the object language. For instance, a Lisp program may construct an s-expression representation of a program, `eval` it, and use the results to construct a second program and `eval` that as well. In this context, the part of the Lisp program that constructs other Lisp expressions is the meta-level and the computations that happen within `eval` are object-level. In this example, the meta-level computation cannot be completed until some of the object-level computation is first executed. As such, meta-level and object-level computation interleave. We say that dynamic metaprogramming systems lack a *phase distinction*, meaning that meta-level and object-level computation cannot be completely teased apart and performed sequentially.

The substantial difference between Lisp's two metaprogramming facilities is that while `eval` is a dynamic metaprogramming facility, macros are static. Using `eval`, a Lisp program

can at run-time piece together a list representation of a program and then evaluate it. Macros, on the other hand, allow a Lisp programmer to specify new special forms that can be used within a program and are expanded away at compile-time.

Dynamic and static metaprogramming have tended to serve related but different purposes. Dynamic metaprogramming is used to increase performance as well as to express computations succinctly. Under some circumstances, building and evaluating a dynamic metaprogram at run-time can alleviate interpretive overhead for repetitive tasks. Many computations amount to interpreting some data structure as if it were a program. When a particular interpretation undermines application performance, it is sometimes beneficial to replace the interpretation with an embedded compiler. That is to say, rather than interpret a data structure multiple times, compile the data structure into a program in the language and subsequently evaluate it. The benefits of this process depend on the efficiency of the evaluation mechanism as well as the overhead involved in compiling and using the data. However, this embedded compiler approach can also pay expressiveness dividends: in some cases translating data to a program requires less complexity than implementing an interpreter for the data. In a sense, `eval` gives the programmer an interpreter for free. For instance, the MetaOCaml language is used to implement interpreters for domain-specific languages [**86**]. However, rather than simply interpreting a program, the metaprogram compiles it to a MetaOCaml program and then executes that.

Rather than providing primitives for the development of interpreters, static metaprogramming offers facilities for implementing new abstractions that can be used directly within the program itself. Static metaprogramming allows programmers to grow the language. New abstractions can be provided along with domain-specific optimizations. Such ability is critical to providing libraries of software for domains where performance is critical. They also enable execution of nontrivial computations that are completed prior to application run-time, especially in the case of batch compiled programs. This is generally done to improve performance. However, static metaprogramming tends toward the development of domain-specific embedded languages.

In the following, our interest is more along the lines of Lisp macros than Lisp `eval`: we focus on language support for static metaprogramming. As such, it is important to keep in mind the distinction between the two and their associated uses.

2. Type Systems

A type system is a syntactic mechanism that constrains the structure of programs by associating semantic information with their parts and restricting how those parts may be combined. From a programmer's perspective, a type system is a set of machine-checked annotations on declarations. From a compiler writer's perspective, a type system is a source of static information that can be used to improve performance. From a language theorist's perspective, a type system is a discipline for structuring and reasoning about a language. In short, types play many roles in programming languages.

The typical statically typed programming language requires each variable in a program to have a particular domain of values it can refer to; each declared function must also operate on particular kinds of arguments and return values of a particular kind. This information may be explicitly declared using type annotations or inferred by a type checker. When a statically typed program is compiled, a type checker interprets these annotations and ensures that each variable is only assigned the right kinds of values and that functions are only called with the right kinds of arguments. Many modern programming languages are statically typed.

2.1. What are Type Systems for? Type systems serve different purposes in different programming languages. Nonetheless, they follow a few themes. Pierce [**62**] identifies several roles for type systems in programming languages. Here we discuss four of them.

Detecting Errors. Since type systems keep track of what kinds of arguments can go where, they can detect pieces of code that assign values to invalid locations. Type systems can detect logical data flow errors in programs, regardless of whether those erroneous statements will ever be executed. Thus they can guarantee the absence of a certain class of programming errors from programs.

Many programming errors amount to using the wrong data in the wrong places. Type systems partition the space of legal values that can appear in each program context and check for conformance to this partitioning at compile time. This coarse-grained check can recognize many instances of misused data. Programmers rely on type checkers to automatically detect such program errors before the program is even executed.

Abstraction. Types are used to abstract parts of a program so that they can be treated more modularly. Much research has been done on abstract data types (ADTs) [**49**], a facility for defining new types that hide their internal structure from the rest of a program. Abstract data types publish an interface to their underlying representations, but hide those representations from users. This kind of information hiding limits the dependencies between program modules, making it possible to change the underlying implementation of an ADT so long as the interface invariants are preserved. Abstract data types influenced the design of type systems for object-oriented programming languages [**84**].

Another form of abstraction presented by type systems is *overloading*. Also known as "*ad hoc* polymorphism," overloading refers to the ability to provide multiple definitions of the same identifier which can be disambiguated based on the types of the subexpressions in the context where it is used. Overloading is most commonly associated with functions, but values, like numeric constants, are also overloaded. Many typed languages provide simple arithmetic overloading, where operators like addition and multiplication are defined for both integers and floating-point numbers, say, and the language chooses the right implementation. Often, functions for printing data or serializing structures are also overloaded. Some languages, like C++ and Haskell, allow programmers to define their own overloaded functions. Overloading lets programmers naturally express a conceptually uniform operation that requires different implementations for different kinds of arguments.

Documentation. Many statically typed languages require programmers to annotate some parts of their programs with type information. One classic example of this is the C programming language, where every variable declaration states the type of values that it can take, and every function definition states the types of its arguments as well as its return type. Even languages with type inference require some type declarations. ML data type

declarations specify a set of constructors for each data type and the types of arguments for each of those constructors.

Type declarations impose some burden on developers, but they also act as machine-checked documentation, even when they are not mandatory. In Haskell, for instance, many functions do not require declarations because their types can be inferred. Nonetheless, many Haskell programmers provide type annotations for each top-level function, and those annotations are checked by the compiler against the inferred type of the function. In other languages, programmers must settle for specifying type declarations as comments and manually ensuring their correctness. By annotating the contents of data types and the arguments of functions, types provide a coarse-grained specification of a program's data flow. In this sense, they document program structure.

Efficiency. A well-typed program provides a compiler with information that it can use to improve how it translates the program. Since variables have static types, the compiler can decide locally how to represent those variables in machine code. This is particularly useful when a variable can only hold values of one fixed type. For instance, most modern machine architectures provide special registers for handling floating-point numbers, but they can only be used when the value in question is known to be a floating-point number. Given type information, a compiler can determine, for instance, that an argument to a function is always a floating-point number. The compiler can compile the function such that it receives that floating-point argument in a register rather than in memory. The same sort of operation can be done for small but fixed-size data structures, which can be split across multiple registers when passed as arguments. If the compiler consistently renders these arguments at the interfaces, then calls to functions and the function implementations will be in sync. Some programming languages, like Common Lisp [**81**], provide type annotations solely to increase performance. Such type systems do not facilitate static guarantees about how programs behave.

2.2. Basic Properties of Type Systems. Although the term "type system" does not have a definitive and universally accepted definition, most type systems share a number of common properties.

Types. Every statically typed language has a set of types, which are essentially syntactic entities that represent semantic properties. The most basic languages provide a set of primitive types, like `int` for integers, as well as some types that are formed from other types, such as arrays of `int`s or lists of `int`s and so forth. In the simplest case, all types are denoted by particular symbolic names, and name equality is the only form of type equality. Slightly more complicated languages allow programmers to declare *type aliases*, alternative names for the same type. In these languages, type equality includes not only matching type names, but also types that are explicitly declared as equivalent. Most practical languages also provide structures, tuples, or other means by which programmers can define their own composite types. In addition, some programming languages allow programmers to define *type constructors*, type-level entities that can be applied to types to form other types. In this manner, programmers can implement type-level abstractions that provide parameterized types. When a language supports type constructors, the names of its types have a tree-like structure. The structure of type names generally encode some semantic information about the kinds of values that can have the given type. For instance, the function type $A \to B$ describes a function that takes values of type A and produces values of type B.

Compositionality. In a statically typed language, each expression has a type, and that type is at least in part a function of the types of its immediate subexpressions. For a simple example, consider a conditional expression **if** a **then** b **else** c. In a typical language, the type of this expression is fully determined by the types of its three subexpressions a, b, and c. First, a must have boolean type and the types of b and c must be equal. Given these restrictions, the entire expression has the same type as both b and c. This typing rule captures the intuition that this expression will return either b or c. In general the rules for typing expressions mirror the structure of the expressions themselves.

Although the types of well-typed expressions take on a compositional relationship, the assignment of types to terms is context-dependent. For practical reasons, programming

languages do not require programmers to explicitly specify the type of every subexpression in a program. In general, languages associate types to their primitive constants and require lexically scoped variables to have type annotations. In the case of type inference, only user-defined data types need explicit type annotations. Then the type checker deduces the types of subexpressions based on the variable and data type declarations. So the lexical scope—more generally the context—in which an expression appears partially determines its type.

Guarantees. Modern type systems try to do more than present haphazard mechanisms for constraining the values that variables can take. Programming language theorists have developed semantic techniques that express strong connections between a programming language and its type system. A formally specified language like Standard ML [**56**] has a dynamic semantics—a formalization of how its programs behave at run-time—as well as a static semantics—a definition of the type system. To connect the dynamic semantics and the static semantics, language designers prove a set of theorems that establish dynamic properties of those programs that are well-typed by the static semantics. This idea is captured in the phrase "well-typed programs don't go wrong" (where "wrong" has a precise meaning for each particular language) [**55**]. Two specific properties are usually established: *preservation*, that a program remains consistently well-typed as it executes, and *progress*, that a well-typed program will not fall outside the scope of the dynamic semantics.

2.3. Relevance of Types to Metaprogramming. Program text contains a great deal of static information, some explicitly present in the text and some implicit until uncovered during static semantic analysis. Type systems are one way of codifying static information about a program. Adding sparse type annotations to a program causes every subexpression of a program to have a type, as dictated by the structure of the type system. These types encode useful semantic information that can be exploited by a metaprogramming system. For instance, many C++ libraries use template techniques to access the types of template function arguments and use them to custom generate algorithms (cf. Chapter 3). In an untyped language like Scheme, however, this information must be explicitly

threaded through a program as an extra argument to every function. Threading compromises abstraction barriers and may complicate otherwise simple operations. Types can implicitly carry information across abstraction layers.

The type system of an object language is resolved at compile-time. Likewise, the evaluation of a metaprogram occurs at compile-time. These two operations can be made to interact with each other. In a way, the type system of the object language is a metalinguistic facility that yields semantic information about program subexpressions. In fact, languages like C++ and Haskell support metaprogramming through stylized use of their type systems. We can go so far as to consider the object language types to also be metalanguage data. A metalanguage can thus perform computations on types.

Metaprogramming can be and has been used to support the roles that types play while leveraging the information that they add to programs. Metaprogramming becomes even more powerful if the object language type system becomes more powerful and the metalanguage is extended to take advantage of it.

Detecting Errors. Types generally encode semantic information, but a standard type system cannot necessarily interpret the semantic import of a particular type assignment. Metaprogramming support can allow programmers to interpret type information in ways that take advantage of their semantic content. In this way errors that go beyond matching arguments with functions can be detected using sophisticated routines that execute during compilation.

Abstraction. The same kinds of computations that are used to detect program errors can be used to create and enforce abstractions. As described previously, overloading allows a uniform abstraction to present differing implementations. Using metaprogramming, contextual type information can be used to perform more sophisticated dispatches and can even be used to generate customized program components.

Documentation. The same way that types can be treated as executable documentation, metaprograms can be used to encode program invariants that are useful to a reader and enforceable. One example is the use of metaprogramming in C++ to build concept-checking libraries [**54**, **74**]. Metaprogramming extends the reach of checkable documentation by

making it possible for programmers to add even more terms and notions to the lexicon of their executable documentation and to determine how it is interpreted during compilation.

Efficiency. Metaprogramming can combine domain knowledge with static information to build program components at compile time that alleviate abstraction penalties. C++ programmers have been doing this for years [**93**].

Types can be a great source of information about a program and can be used by metaprograms to implement effective compile-time functionality. This opportunity to access semantic information about a program is motivation for caring about a type system's design when thinking about metaprogramming. It is important to consider how the common traits of type systems—primitive and compound types, compositionality, and context dependence—can be leveraged for metaprogramming.

3. Considerations for Static Metaprogramming with Types

Having discussed metaprogramming and type systems in general, we now turn to the design considerations for a language that supports static metaprogramming and can manipulate object language types from the metalanguage. This section discusses some of the parameters of the space for static metaprogramming and tradeoffs that they entail.

Previously we talked about design decisions regarding static types in metaprogramming languages. This subsection takes a closer look at some of the implications of types in a language with support for static metaprogramming. It focuses particular attention on how the metalanguage and its type system interacts with object language types.

3.1. Referential Transparency. Section 2 discussed some of the common properties of type systems, namely types as syntactic notions, the compositional type structure of expressions, and the context dependency of type assignment. These three properties can be taken into account in the design of a metaprogramming language.

Since types come in both atomic and composite form, it makes sense for a metaprogramming language to be able to examine their structure. In particular, a metaprogramming language that manipulates types could deconstruct composite types into their parts and examine them. Since type equality is the most fundamental property of types and type

aliases, a metaprogramming language benefits from being able to compare types for their equality.

Since type structure is compositional over expressions, it makes sense that a metaprogramming language be able to query the type of a subexpression of a program without knowing the type of an entire program already. However, as discussed before, context information determines the types of variables and constants. To take this into account, a design for a metaprogramming language could extend the notion of typing context to include metalanguage programs as well. That is to say that every object program expression, even ones that are manipulated within a metaprogram, exists within a typing context.

This notion of typing context is closely related to the idea of referentially transparent macros from the Scheme programming language. Any free object language identifiers that are referenced within a Scheme macro definition refer to the bindings for those identifiers that are in scope at the point of definition. Later uses of those macros may be passed other identifiers as arguments, but the identifiers that were present in the definition, even as they appear in the final program, refer to the bindings at the point of definition. In the same sense, a metalanguage that functions we described here is referentially transparent with respect to object language variable types. Object language identifiers that are referenced in definitions of metalanguage computations refer to the bindings and types of variables that are in context at the point of definition. Just as Scheme's hygienic macros [**45**] respect the binding structure of Scheme programs, a metaprogramming language can respect both the binding and typing structure of object language programs.

3.2. Object Language and Metalanguage Types. A programming language with support for metaprogramming has multiple notions of dynamic semantics—the run-time behavior of a program—and multiple notions of static semantics—type checking and other compile-time correctness properties. The metalanguage and the object language each have their own set of semantics . Because of this, some special semantic issues arise in a metaprogramming language design.

Consider the typical statically typed programming language. Its type system prevents some program errors by detecting them at compile time; static metaprogramming is mostly compatible with this behavior. Static metaprogramming executes user-designed metaprograms at compile time, and those metaprograms may signal errors for a variety of reasons. However, these errors occur during execution of the metaprogram, which happens at compile time. These errors happen at the same stage of computation during which object language type checking occurs.

Things get interesting when the metalanguage also has a static type system. A metalanguage type system provides a means to reason about a metaprogram prior to executing it. As such, this type system provides an additional stage of processing to program compilation: first check the metalanguage type system, then execute the metaprogram, then check the object language type system, then compile or interpret the object language program. Just as the traditional type system may guarantee that a certain class of errors won't happen during a program's run-time, a metalanguage type system can guarantee that certain errors won't happen during the metacomputation part of program compilation. This extra level of checking can provide strong guarantees. As mentioned in Section 1, the MacroML programming language provides statically typed macros that are guaranteed to generate only type safe code when they are used. This level of guarantee means that a macro can be deployed with the knowledge that its correct use will never introduce a type error in a program. Errors are caught at the point of macro definition. Contrast this with the PreScheme language. Its macro language is not typed: programs are type checked following macro expansion. As such, a macro may generate ill-typed code at compile-time. Macro-induced type errors are still caught prior to run-time, but not until the macro is used.

Type systems tend to provide some level of modularity. Often, typed programs can be broken up into modules which have interface and implementation, and each module can be compiled independently of the implementation of the other modules. In practice, large programs are constructed by composing separately compiled components into a full application. Each component (or module) is type-checked when it is compiled, and the result is a binary representation of that component, as well as a typed interface that describes

the operations and data descriptions exported by the component. Different modules can be compiled with the expectation that they will be linked with other modules. In that case, a separately compiled component uses only the interface of another component to perform type checking and compilation. Later, the separately compiled components are linked to form an executable. As such, type safety is guaranteed on a component-by-component basis, and the linking process simply ensures that component implementations and interfaces have not drifted out of sync. As long as the interfaces are kept in sync, it can be known that a local module is type-correct prior to linking it with another module.

A static metaprogramming system may not support the same modular type-checking guarantees as the typical typed programming language. Suppose a programming language that supports both implementing and exporting of object language and metalanguage components across module boundaries. Then, just as compiling a module requires access to the type specifications of any interfaces upon which the module relies, the compiler also requires access to metaprograms that are used across modules so it can execute them during compilation. If metaprograms can signal errors during their execution, then some metaprogram errors may not manifest until some module invokes the computation defined by a metaprogram from another module. Some compile-time errors may not be detected until a metaprogram is linked with another module that uses it. Compilation of a second module can signal an error due to a fault in the first module. Just as a type system can induce guarantees on the execution of a normal program, a metalanguage type system can also induce guarantees on the execution of metacomputations.

The unfortunate side-effect of a weak metalanguage type system is that a library containing metaprogramming code could be shipped with a bug which cannot be detected until it is linked with another module that uses it. On the other hand, a strong notion of metalanguage type safety imposes strong restrictions on the expressive power of a metaprogramming language. Just as a traditional type system can reject correct programs, a metalanguage type system can reject correct metaprograms, and this loss in expressiveness must be weighed against the fact that many static metaprogramming errors can be caught prior to application run-time, since metaprograms execute at compile-time. There is a significant

tension between modular safety and expressiveness when it comes to metalanguage type systems.

CHAPTER 3

C++ Template Metaprogramming

C++ is a successful, if somewhat disparaged, example of the benefits of compile-time metaprogramming, and the primary inspiration for this dissertation. This chapter takes a close look at C++ templates and template metaprogramming. It discusses the C++ type system, focusing on how C++ templates have enabled a powerful and broadly useful form of metaprogramming. It gives background on C++ templates, discusses the history of template metaprogramming, and presents an analysis of the core capabilities of C++ templates that enable the use cases of template metaprogramming.

1. C++ Templates

Templates were added to C++ so that programmers could write and use type-safe polymorphic containers [**84**]. In particular, the C++ Standard Committee Libraries Working Group needed templates so that they could provide type-safe containers as part of the language's standard library. Prior to the introduction of templates, C++ container designs sacrificed either generality or type safety. Some software applications of the time included multiple implementations of basic data structures like linked lists and binary trees, each differing in only the type of data it could contain. Such data structures are type-safe, but development and maintenance problems associated with this style of code repetition are well known and best avoided. On the other hand, general-purpose container libraries could not provide specialized data structures for every type of object used in an arbitrary application. Instead, their containers had to discard type information and rely on application developers to manage their types using static or run-time casts. Such data structures are concise and general, but the development and maintenance problems associated with casts are well known and best avoided as well.

Developers quickly realized that they could leverage the C++ preprocessor to develop container libraries that are both type-safe and general. Rather than implement the same container for each type, the container is implemented as a single preprocessor macro that abstracts as a parameter the type of the objects it can contain. Application developers then generate containers for their data types by providing those type names to macro invocations. Unfortunately, preprocessor macros achieve generality and type-safety at the expense of language integration and identifier hygiene. Language integration is sacrificed to the extent that the preprocessor is separate from the C++ language. The preprocessor operates directly on lexical tokens: it does not recognize program constructs, nor does it respect them. During compilation, preprocessor macros are expanded and removed from a program before any semantic analysis occurs. As a result, preprocessor macros cause analysis tools such as debuggers and profilers to provide imprecise information and cause compilers to issue incoherent diagnostics. Identifier hygiene is sacrificed to the extent that the preprocessor does not respect C++ name scoping rules. Preprocessor macros often require manual name mangling to prevent multiple conflicting definitions of class, function, and variable names and to prevent interference between identifiers introduced by an application developer and identifiers generated by the macros. These problems led Stroustrup to propose the adoption and standardization of templates [**84**].

2. The Design of C++ Templates

At the same time that Stroustrup was designing templates for C++, Stepanov and others were investigating a style of programming now called *generic programming* [**58**]. Generic programming is a design technique that seeks to define and implement algorithms in terms of the abstract requirements of the algorithm, rather than in terms of a particular data structure. These abstract requirements codify an interface to the algorithm, and any data structure that implements that interface (i.e., meets the requirements) is valid input to the algorithm. As such, generic algorithms are highly reusable and thus prevent the needless reimplementation of algorithms that can be written once and for all. Musser and Stepanov began their research in the languages Scheme [**44**] and ADA [**57**], but shifted their focus

to C++, culminating in the design and implementation of the Standard Template Library (STL). Stroustrup was impressed by the STL and eventually it was incorporated into the C++ Standard Library [**85**]. The needs of generic programming influenced the design of templates. In particular, Stepanov and others prototyped applications of templates using the C++ preprocessor, and Stroustrup incorporated insights gleaned from those examples into the design. Stepanov's examples were derived from his work on generic programming in ADA [**84**].

A template is a recipe for building similarly structured pieces of code. They are parameterized on types and values. To use a template, it is supplied with a list of arguments that replace the parameter names. Using a template, one can define multiple related pieces of code. C++ has two kinds of templates: *class templates* and *function templates*.

A class templates is a recipe for defining a data type, where some static parameters have been abstracted. For instance, instead of implementing a family of individual linked list data structures:

```
struct int_list {
  int head;
  int_list* tail;
};

struct char_list {
  char head;
  char_list* tail;
};
```

one might define a linked list class template:

```
template <typename T>
struct linked_list {
  T head;
  linked_list<T>* tail;
};
```

The `linked_list` template provides a skeleton for implementing any number of linked list data structures. Now instead of explicitly defining a linked list for `int`s and `char`s, one can use `linked_list<int>` and `linked_list<char>` respectively.

More generally, class templates were conceived as a means to implement type-safe containers, such as resizable arrays, linked lists, hash tables, and associative arrays. Just like any other class, a template can contain member functions and member data.

Similarly, a function template is a recipe for defining functions. Rather than writing a family of functions:

```
int int_head(int_list* lst) {
  return lst->head;
}

char char_head(char_list* lst) {
  return lst->head;
}
```

One can define a function template that covers all cases:

```
template <typename T>
T head(linked_list<T>* lst) {
    return lst->head;
}
```

The C++ template design assumes *instantiation*—the compile-time generation of new code for each unique instance of template arguments. By generating new code for each instance, instantiation ensures that templates can be used to construct statically polymorphic abstractions whose instances run as quickly as their hand-coded equivalents. Other programming languages like Eiffel and Java have analogous template facilities, but do not perform instantiation. Instead, each of their templates results in one unit of compiled code shared by all instances. By using instantiation, templates can use pass-by-copy semantics, where complex objects are constructed on the stack and returned as values, rather than pass-by-reference. Each instantiation of a template is specialized to the needs of its types. As a result, however, templates themselves are not compiled into applications. Rather, the particular instantiations of a template are compiled into run-time code.

Though templates were intended to target a particular issue, their design reached beyond the requirements of polymorphic containers. For starters, templates not only support type parameters but they also support what are called *non-type parameters*. A template may be

parameterized on primitive integral types and enumerations, as well as pointers to values, functions, and member functions. These values may be used within the body of a template as constant run-time values wherever applicable. For instance, one can define a statically sized array abstraction in C++ as follows:

```
template <typename T, int N>
class array {
  T data[N];
  int size() { return N; }
  ...
};
```

The `array` class takes as one of its template parameters an integer that it uses to statically determine the size of the array it stores. Furthermore, the class defines a `size()` member function that immediately returns the size as a compile-time constant.

In addition to top-level class templates, which are used to implement containers, and top-level function templates, which can define operations on all instances of a particular template, the language also supports member function templates and nested template classes. Both of these constructs can be defined within traditional classes or template classes. For example, the following class:

```
class my_container {
  template <typename T> my_container(linked_list<T>* data);
};
```

defines a templated copy constructor that can construct a `my_container` object from any instance of the `linked_list` template. Though not strictly necessary for implementing type safe containers, such templates are useful, and allowing them gives the language a more orthogonal design.

Like traditional functions and member functions, function and member function templates are subject to *overloading*: any unambiguous function call refers to a particular function or function template instantiation among a set of declarations that all have the same name. As part of the support for function overloading, C++ supports *implicit instantiation*: template functions need not be explicitly provided their template arguments when

they are called. The type system deduces the template arguments to a function template based on the types of its run-time arguments. For instance, the functions:

```
template <typename T>
size_t get_size (T&) { return sizeof(T); }
size_t get_size (char&) { return 1; }
```

can both be called without providing any extra annotation:

```
int x;
char y;
get_size(x); // calls the function template
get_size(y); // calls the normal function
```

and the compiler will deduce the template arguments where needed and insert them automatically:

```
int x;
get_size<int>(x); // calls the function template
get_size(y); // calls the normal function
```

Class templates can also be overloaded. Once a class template has been declared, a programmer can define a *specialization*, an implementation for any particular instance of the template. For instance, the C++ Standard Library defines a `vector` template, which is a resizable array:

```
template <class T > class vector { /* ... */ };
```

The `vector` template is specialized for the type `bool`. In C++ the `bool` type is represented with a byte of data. The C++ `vector<bool>` implementation stores booleans using bits. That way an 8-element `vector` of data needs only 1 byte of storage, rather than 8. the following specialization covers the instance of `vector` with `bool` as its type argument:

```
template <> class vector<bool> { /* ... */ };
```

In general, a specialization is used in place of a template instantiation when the arguments to the instance match the arguments declared by the specialization. A developer may use specializations to override a class template with a special-case implementation for a specific set of template arguments. More generally, a programmer may define a *partial*

specialization, a special template with the same name as the original template and a list of arguments that are compatible with the signature of the original template. However, a partial specialization refers to some of its own template parameters within its list of template specialization arguments. Whereas a specialization replaces a specific template instantiation, a partial specialization replaces an entire family of template instantiations. For instance, the `vector` template is also specialized for all pointer types:

```
template <class P> class vector<P*> { };
```

All instances of `vector<P*>` use the same implementation of the vector routines, one defined for the pointer type `void*`. Since all pointers have the same size, one implementation can be used for all the underlying machinery, and the particular specialization simply supplies compile-time casts to ensure type safety. In this manner, the amount of code generated for vectors of pointers is minimized.

Assuming an `Ast` template of one argument, the following partial specialization covers every instance of `Ast` whose argument is also an instance of `Ast`:

```
template <class T> class Ast< Ast<T> > { /* ... */ };
```

In general, if some template instance matches the more specific signature of the partial specialization, that template is instantiated in place of the more general template. C++ defines a set of ordering rules that determine the best matching template specialization. As a result of these overloading and specialization features, templates can be implemented such that the form of the resulting class or function depends on the particular arguments to the class.

2.1. The Birth of Template Metaprogramming. As we can see, templates are powerful. It turns out however, that templates can also perform general computations that affect the resulting run-time code.

In 1994, during the C++ standardization process, Erwin Unrue approached the C++ standards committee with his discovery that C++ templates could be manipulated into performing computation, not merely the generation of classes and functions. He wrote a program that was invalid C++, but the resulting error message included an enumeration of

prime numbers. It turns out however, that programs can do more interesting things with templates than generate mathematically interesting error messages. They can also perform general computations that affect the resulting run-time code. In fact C++ templates are Turing complete (so long as you disregard the phrasing in the standard regarding template instantiation depth; in practice compilers allow you to do just that) [**9**].

C++ templates spawned a genre of programming called *template metaprogramming* [**93**]. In addition to binding types to parameters, C++ templates are capable of performing computations on values and types. This ability has proven useful for many applications.

3. Idioms of Template Metaprogramming

Template metaprogramming has been used to develop high-performance libraries for linear algebra [**25**,**78**,**79**], graph algorithms and data-structures [**47**,**48**,**76**], message passing for parallel programming [**40**], and many other domains. As developers have experimented with template metaprogramming, certain patterns of programming have been discovered and refined. These patterns comprise the basic idioms of template metaprogramming. In this section we discuss those programming patterns and the language features that enable them.

3.1. Static Computation. C++ supports the compile-time evaluation of constant integer expressions and allows integers to be passed as template arguments and subsequently used as constant values in run-time code. C++ metaprograms take advantage of this to perform arbitrary mathematical calculations. They can precompute values at compile time that would otherwise need to be computed at run time. For instance, the program:

```
template <int M, int N>
struct powMN { static const int value = M * powMN<M,N-1>::value; };

template <int M>
struct powMN<M,0> { static const int value = 1; };

int pow_7_5 = powMN<7,5>::value;
```

computes M^N using a common idiom where struct templates are used like recursive functions. Template partial specialization provides a pattern matching facility, and this program uses it to handle the $N = 0$ base case. When compiled, this program assigns the variable `pow_7_5` the value of 7^5, 16807, without performing any run-time computation.

3.2. Code Specialization. C++ templates can also be used to specialize a piece of code with respect to some values known at compile-time. This process is sometimes called *partial evaluation* [**39**]. Rather than simply generate a value, a metaprogram can generate code specialized for the constant data provided at compile time. A variation on the previous example:

```
template <int N>
int powN(int M) { return M * powN<N-1>(M); }

template<>
int powN<0>(int M) { return 1; }

template int powN<5>(int);
```

implements the same operation, using function templates, but only the exponent is statically determined. The instantiated function template takes a base value at run-time. The last line of this example specializes the function for powers of 5, yielding a function that explicitly multiplies its argument five times.

```
template<> int powN<5>(int M) { return M * M * M * M * M; };
```

Bear in mind, however, that this result partly depends on the sophistication of the compiler. The semantics of templates do not force inlining as it is presented above. As a result, a legal implementation could expand the template instantiation to a disastrous cascading implementation:

```
template<> int powN<5>(int M) { return M * powN<4>(M); };
template<> int powN<4>(int M) { return M * powN<3>(M); };
...
```

3.3. Type Structure Analysis. C++ templates can not only perform computations on their value arguments, but they can also analyze types. In particular, template specializations and partial specializations can perform pattern matching against the structure of the names of type arguments, especially the types of instantiated class templates.

Type analysis can be used to manipulate compile-time data structures. For instance, consider the following code, which implements a type-level linked list structure:

```
template <typename H, typename T>
struct type_list { };
struct nil {};
typedef
  type_list<int, type_list<double, type_list<float,nil> > > a_list;
```

The `type_list` template can be used to construct lists of types that can be used as arguments to other templates. In order to make use of the `type_list`, however, a metaprogram must decompose it:

```
template <typename NilType>
struct list_length { static const int value = 0; };

template <typename H, typename T>
struct list_length< type_list<H,T> > {
  static const int value = 1 + list_length<T>::value;
};

int len = list_length<a_list>::value;
```

This metaprogram has the same recursive structure as the numerical exponentiation example, but this computation is driven by the structure of a type rather than the value of a number.

3.3.1. *Static Reflection.* The ability to manipulate C++ types using templates is central to template metaprogramming. For example, it is neither uncommon nor difficult to write a metaprogram that generates one type from another type. Consider a template metaprogram that turns one type into a pointer to that type:

```
template <class T> make_pointer {
  typedef T* value;
};
```

To use this metaprogram, simply provide a type argument and reference the internal type alias: for instance, `make_pointer<int>::value` is equivalent to `int*`. A more complex metaprogram changes a pointer into a base type and leaves non-pointer types unchanged:

```
template <class T> remove_pointer {
  typedef T value;
};

template <class T> remove_pointer<T*> {
  typedef T value;
};
```

This small metaprogram uses partial specialization to implement the case where the argument is some pointer, and the general template to handle the case where the argument is any other type. For instance, both `remove_pointer<int>::value` and `remove_pointer<int*>::value` are equivalent to `int`, and `remove_pointer<int**>::value` is equivalent to `int*`. Using templates to reason about the structure of C++ types is called *static reflection*.

3.4. Type Property Analysis. C++ templates give some structure to the names of types, but C++ classes and structs, being nominally typed at heart, imply more semantic information aside from their name-as-tree. For instance, C++, being an object-oriented language, has notions of base classes and inheritance. Using a template metaprogram, it is possible to query at compile-time whether one type is the base class of another type:

```
template <typename B, typename D> class is_base_of; // declaration

class A { };
class B : public A { };
class C { };

is_base_of<A,B>::value; // compiles to ``true''
is_base_of<C,B>::value; // compiles to ``false''
```

Note that being a base class is not a property that can be seen in the structure of a class's name. B and C look altogether different.

Because of the popularity of these techniques, the next C++ standard will add new metaprograms that require explicit compiler support. One example involves the notion of a virtual destructor. Some C++ classes have virtual destructors because of the object-oriented

aspects of the language. If you have a class that is going to be inherited from, and you wish to correctly "delete" an instance of the subclass through a pointer to the superclass type, then the superclass needs a virtual destructor because that will cause the proper destructor definition to be accessed dynamically. For example, consider the following code

```
class A { virtual ~A() {} };

class B : public A {
  HANDLE h;
  B() { h = acquire_resource(); }
  ~B() { release_resource(h);
};

/* ... */
A* q = new B();
/* ... */
delete q;
```

If class A's destructor were not virtual, then the `delete` call on the last line would fail to properly release whatever resource `B` had acquired.

The next C++ standard will directly support ascertaining whether or not a class has a virtual destructor. This information can be useful particularly within the definition of a template, or a class that uses an object of the given type. As a result it is then possible to write:

```
struct small_struct { int n; };
bool property_query =
  has_virtual_destructor<small_struct>::value;
```

After compilation, the result is as if the programmer had originally written:

```
struct small_struct { int n; };
bool property_query = false;
```

Metaprogramming-based type analysis provides some, but not all, of the capabilities of a general reflection mechanism, as found in the run-time reflection facilities of Java. For example, in Java reflection can be used to iterate over the member functions published by

an object. Unfortunately C++ template metaprogramming does not support this kind of reflection[1].

4. Case Studies

To give a better grasp of the broad capabilities of template metaprogramming, I discuss a few larger examples of how it has been used in production-quality applications and libraries.

4.1. Performance Improvement. The combination of static computation and code generation has been used to implement performance enhancements for production software libraries. The Matrix Template Library (MTL) [**77**], a C++ library of basic linear algebra algorithms and abstractions, uses two sub-libraries to mitigate the abstraction penalty that has plagued previous attempts to build linear algebra libraries with high-level domain-based programming interfaces. Those previous efforts were deemed unsuccessful because the scientists that would use these libraries demand the high performance that is usually provided by the highly-tuned Basic Linear Algebra Subprograms (BLAS). The MTL, with the help of its two sub-libraries, the Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template Library (FAST), provides high level abstractions for matrix computations as well as standard optimizations such as matrix blocking, loop unrolling, and inlining, thereby achieving performance comparable to the BLAS [**78**].

4.2. Type Reflection. The type analysis capabilities of C++ support important applications. For instance, the Boost Python library [**3**] uses type analysis to automate the generation of foreign function interfaces for the Python scripting language [**91**]. For instance, the function:

```
char const* greet() {
 return "hello, world";
}
```

can be reflected into Python using the following code:

[1]Some clever metaprograms have been crafted to simulate querying members of C++ classes, but their capabilities are generally quite limited.

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
    def("greet", greet);
}
```

The above code is valid C++ (taking some advantage of the preprocessor), and succinctly captures the more complicated process of manually specifying a foreign function interface as specified for the C interface. Boost.Python uses sophisticated template metaprograms to compute the signature of the `greet` function and generate the necessary code to link the function into the Python interpreter.

4.3. Policy-Based Design and Generative Programming. Abstract data types hide implementation complexity behind interfaces. They promote program understanding by breaking software into units that can be understood and used without requiring intimate knowledge of the underlying details. There are often many variations and tradeoffs in the implementation of an ADT, with different design choices better suited to different situations. In such situations a library may provide a suite of variations from which the user can choose, but it is costly to maintain many implementations of closely-related data types. One approach to decreasing this cost is to observe that different implementations of a single ADT or related ADTs may have the same or very similar code. To facilitate code maintenance, common functionality can be captured in shared pieces of code. Such factoring limits the amount of repetition in the code base, shrinking the amount of code that must be maintained. Factoring ADT implementations into pieces also makes it easier to develop new variants by reusing the relevant pieces. Each ADT implementation becomes a composition of code units that together suit a particular use case.

4.3.1. *Policy-Based Design.* Consider a design for a configurable list data structure. Sophisticated containers may provide a variety of configuration options. For example, a list class may support several memory management options for its contained elements. The container may copy elements that are placed inside it, hold them by reference without

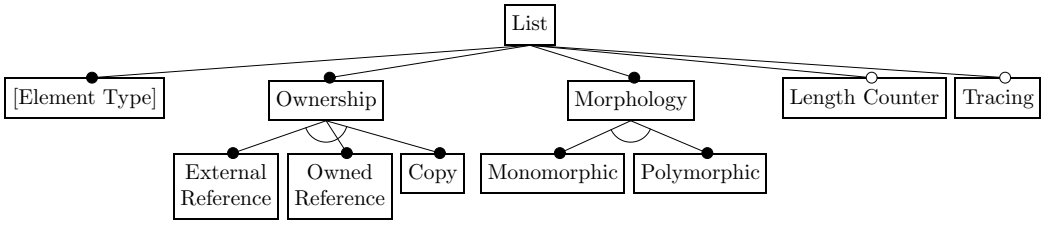

FIGURE 1. Feature Diagram for Lists

taking ownership, or hold them by reference and delete them when the container itself is destroyed. This and several other possible configuration options are presented in Figure 1.

C++ metaprogramming provides sophisticated means to implement this taxonomy. First, C++ template techniques can be used to decompose an abstract data type into categories of functionality and to implement a family of implementations for each category. Templates are used to compose one variation from each category together and construct any particular implementation of the ADT. This decomposition of categories and composition of data-types is called a *policy-based design* [**6**]. In this manner, an exponential number of possible ADT implementations can be provided using a linear number of implementation units.

The following simplified example shows how a policy-based design is implemented:

```
template <typename T, typename TracePolicy>
class policy_list {
  T head_;
  policy_list* tail_;
public:
  policy_list(T& h, policy_list* t) : head_(h), tail_(t) { }

  T& head() { TracePolicy::report_head(head_); return head_; }

  policy_list* tail() { TracePolicy::report_tail(tail_); return tail_; }
};
struct DoNotTrace {
  template <typename T> static void report_head(T&) {}
  template <typename T> static void report_tail(T&) {}
};
```

The code implements a simple linked list template that is parameterized over a `TracePolicy` type. The linked list never creates an object of that type. Rather, `TracePolicy` argument

is a repository for compile-time configuration information. It determines whether and how the list should trace or log its usage. For example, the `DoNotTrace` policy object inhibits the list type from logging its use by providing vacuous implementations of the tracing operations. This basic technique can be used in more sophisticated ways to implement highly configurable data types that are parameterized over multiple interacting policies.

Policy-based designs impose implicit interface requirements on their policy arguments. For example, the `policy_list` can only be instantiated with an argument type that provides a nested `report_head` type and `report_tail` static member function. To do otherwise would yield a compile-time error.

The sophisticated list design in Figure 1 requires more expressive facilities than the simple example above illustrates. For instance, the list type might maintain a count of the number of elements it contains, and provide that information via a `count()` member function. However, when the counting option is not desired, the list type should not provide that member function. Furthermore, the ownership and morphology options are interrelated in a way that is best decomposed into more than two policy options apiece. However, we desire the user interface to this sophisticated list to export the simple interface illustrated in the figure. In some other complex designs, certain combinations of policy choices are invalid. When restrictions apply they should be enforced. Techniques have been developed for producing these results.

4.3.2. *Generative Programming.* Czarnecki and Eisenecker [**15**] introduced a technique for using template metaprogramming to develop flexible C++ libraries based on the general notion of *generative programming*. Following a thorough domain analysis, this technique involves the synthesis of a set of diagrams, like that in Figure 1, that captures the feature space of the domain. Then, using template metaprogramming, the policy-based components that implement the desired feature space—and other metaprogramming-based constructs—are constructed.

To map from the feature space to the pieces that comprise a data type, more template metaprogramming techniques are used to implement a configuration language. It provides the user with a representation of the feature space. To build a type, the user provides a

combination of features to a generator metaprogram, which first checks the validity of the feature combination and then uses them to determine how to glue the pieces together.

The configuration language for the list data type:

```
enum Ownership { ext_ref, own_ref, cp };
enum Morphology { mono, pol };
enum CounterFlag { with_counter, without_counter };
enum TracingFlag { with_tracing, without_tracing };
```

is simply a set of enumerated types: one for each category and one value for each option. The list generator implementation, then, is declared as follows:

```
template < typename ElementType, Ownership ownership,
           Morphology morphology, CounterFlag counterflag,
           TracingFlag tracingflag >
class list_generator;
```

To construct a list data type, supply the list generator with a set of options and reference the nested `type` type:

```
typedef list_generator<double, cp, mono,
          without_counter, with_tracing >::type list_type;
```

The resulting `list_type` holds `double` values, keeps its own copies of its elements, and outputs trace information to the console.

The Matrix Template Library and the Boost Graph Library both use the techniques described here to implement matrix and graph data types respectively. In both cases, algorithms are the primary focus of the libraries, but the generative data types increase their utility immensely by providing many variant structures to which the algorithms apply.

4.4. Domain-Specific Static Checks. Type systems enable compilers to statically guarantee some program properties. As the expressiveness of type systems increases, so too does the ability to provide more information to the type checker and guarantee more complicated invariants. Some type systems are sophisticated enough to allow the programmer to encode domain-specific properties. The ability to embed values in types, called *dependent types* or indexed types, is one mechanism that enables this kind of domain-specific static checking.

Dependent typing differs from other template metaprogramming techniques in that it has less to do with code generation than it does with encoding properties of programs for the purpose of checking them at compile-time. Dependent typing in the more complex cases involves computation, but unlike two-level programming, it does not involve the injection of compile-time values into the run-time.

Consider the following code:

```
template <class T, int N, int M>
Vector<T,N+M>
concat(Vector<T,N>&, Vector<T,M>&);
```

It declares a function template that concatenates two `Vector` objects, one of length `M` and one of length `N`, to yield a `Vector` of length `M+N`. The concatenation operation is templated on an integral value, as is the array class, which encodes its length. This function guarantees at compile-time the length of the resulting `Vector`.

For a more involved example, we turn to physical dimension analysis. Software applications often manipulate numeric values that have primitive physical dimensions of measure such as mass, velocity, momentum, and charge, or complex dimensions composed from those primitive dimensions. For example the Newton, a measure of force, is equivalent to $\mathrm{kg.m/s^2}$, the change in momentum over time. Programming language type systems generally do not account for physical dimensions [**26**]. C++ for instance merely has integral and floating-point numbers. Keeping track of dimensions is generally left to the programmer, but template techniques can be used to express and enforce them.

Primitive physical dimensions, such as mass, time, and length, and aggregate dimensions such as velocity (distance per unit time) can be expressed using compile-time lists of integral values:

```
template <typename T, int length, int mass, int time>
struct quantity {
  T v;
  quantity(T t) : v(t) {}
  // ...
};
```

```
typedef quantity<double, 1, 0, 0> length;
typedef quantity<double, 0, 1, 0> mass;
typedef quantity<double, 0, 0, 1> time;
typedef quantity<double, 1,0,-1> velocity;
typedef quantity<double, 1, 1, -1> momentum;
```

The `quantity` template models values that have physical dimensions. It is parameterized on a value type and a list of compile-time integers, each of which represents a physical dimension. Using type aliases, proper names can be given to the most common dimensions. Arithmetic operations can then enforce dimensional semantics. Multiplying values sums the dimension exponents:

```
template <typename T, int length1, int length2,
          int mass1, int mass2, int time1, int time2>
quantity<T,length1+length2,mass1+mass2,time1+time2>
operator*(quantity<T,length1,mass1,time1>& lhs,
          quantity<T,length2,mass2,time2>& rhs) {
  return
    quantity<T,length1+length2,mass1+mass2,time1+time2>(lhs.v*rhs.v);
}

momentum a_momentum = a_mass*a_velocity;
```

but only values with the same dimensions may be added.

```
template <typename T, int length, int mass, int time>
quantity<T,length,mass,time>
operator+(quantity<T,length,mass,time>& lhs,
          quantity<T,length,mass,time>& rhs) {
  return
    quantity<T,length,mass,time>(lhs.v+rhs.v);
}

momentum a_momentum = a_mass+a_velocity; // type error!
```

More sophisticated metaprogramming techniques can be used to build a more robust and extensible system that also supports numerical conversions between differing unit measures. Combined with operator overloading, it becomes possible to write clear and succinct expressions that use units:

```
  mass value = (5 * kilogram) +
    (7 * pound); // == 8.17514659 * kilogram
```

This can significantly increase both the readability and reliability of software.

5. Shortcomings of C++ Templates for Metaprogramming

Clearly template metaprogramming can be used to perform some staggering feats of programming. However this mechanism is not ideal for the contexts in which programmers use it.

Because template metaprograms use traditional language constructs in nontraditional ways, it is not immediately obvious what is intended by the code that makes up a metaprogram. For instance, keywords that normally signal the definition of a data type may actually indicate the definition of a recursive compile-time function.

Furthermore, although template metaprogramming is used to generate tuned high-performance kernels, this capability is not guaranteed by the language semantics. A conforming C++ compiler is guaranteed to generate static values as computed in the two-level model, however the calls to function templates are not guaranteed to be inlined. If inlining does not occur, then the code will exhibit function call overhead which is often enough to negate the expected benefits of specialization. Because of this, some metaprograms are highly sensitive to the quality of a particular compiler implementation. Sometimes a compiler can perform inlining more effectively than a programmer, but in those cases where a programmer desires and deserves control, that capability should be available.

In fact, C++ template metaprogramming does not support the generation of stand-alone expressions: only functions, classes, unions, member functions, and so on. Programmers simulate generating expressions by making the expression of interest a function template, passing it the necessary arguments (via template and run-time parameters) and tweaking the resulting code until the compiler reliably inlines the function.

Another common complaint about metaprogramming-intensive libraries regards how slowly compilers process them. The compile-time cost of template metaprogramming can seriously undermine the application development process. This cost is partly due to how class template metaprograms abuse the language's facilities for implementing templates. A

great deal of additional and useless bookkeeping must be done by the compiler in addition to performing desired computations.

One prevalent complaint regarding libraries that take advantage of template metaprogramming is that when something goes wrong because of an error in either the implementation or use of the library, the compiler will spew an often large and inscrutable error messages expressed in terms of the language constructs that have been hijacked to implement the metaprograms [**74**]. As a result it is difficult to distinguish user error from implementation error. In the case of user error, the resulting message exposes the implementation details of the library instead of expressing what preconditions the library user has violated.

Despite the power and expressiveness of C++ templates, they are a far cry from the ideal metaprogramming language. Nonetheless, programmers resort to metaprogramming-based solutions when there is no other solution that stays within the confines of the C++ language. The extent to which developers have used C++ templates suggests a great deal about the ingenuity of those developers and their desire for such facilities and expressive power. It clearly says little about how pleasant and accessible C++ template metaprogramming is.

With more sophisticated type systems comes the ability to verify more properties of programs, including domain-specific properties that are specified by programmers. However, the C++ experience has shown that in addition to verifying static properties, direct programmatic access to the static semantics of a programming language can provide software developers with greater expressive power. A type system can enable one part of a program to reason about the static semantics of another part of the same program and use that information to select its own behavior. Static reasoning can guide the generation of type-safe application-specific codes, thereby limiting the quantity of code that must be written by developers while simultaneously increasing the expressive power of software applications and libraries.

CHAPTER 4

A Kernel Language for Metaprogramming

This chapter presents a foundational kernel language for compile-time metaprogramming. To first lay groundwork, a simply-typed object language is presented. This language describes the output of metaprogram execution. Its description introduces the static and dynamic semantic frameworks in which the metaprogramming language is defined.

Following the explication of the object language, the kernel language for metaprogramming is presented. Examples demonstrate how the language constructs operate, and give a sense of how compile-time evaluation proceeds. Finally the formal metatheory of the kernel language is briefly presented. In particular, the kernel language is type safe.

1. A Simply Typed Object Language

Metaprogramming involves computations that treat programs as data. As such, metaprograms require some notion of programs that serve as data for them. This section presents a simple language, called henceforth the *object language*, over which metaprogramming is defined.

The object language is an extension of the the simply-typed lambda calculus [**13**], a small language that is well-suited to many foundational programming language investigations. Its syntax follows:

γ	::=	$\langle$type constant$\rangle$	
x	::=	$\langle$variable$\rangle$	
c	::=	$\langle$value constant$\rangle$	
f	::=	$\langle$function constant$\rangle$	
τ	::=	$\gamma \mid \tau \to \tau$	(type)
e	::=	$x \mid \lambda x : \tau.e \mid e\ e \mid c \mid f$	
	$\mid$	**if** e **then** e **else** e	

The expressions of the object language include variable references, x, first-class functional abstractions, $\lambda x : \tau.e$, and function applications, $e\ e$, as well as conditional expressions, **if** e **then** e **else** e. To more closely model the capabilities of practical languages, the language is parameterized on some set of function constants f and some set of basic constants c upon which those function constants operate. These are the primitives of the programming language. For instance, a language that can perform basic integral numeric operations will have among its basic constants all the integers $(0, 1, -1, 2, -2, \cdots)$ as well as some operations upon those integers like *sub1* for unary subtraction and *zero*? to query whether an integral value is equal to zero. In what follows particular constants are assumed as needed.

The object language is statically typed. Each variable in a program is associated with the type of values that it may hold. Just as in languages like C and Java, a variable may be declared to only be bound to integral values (*int*), or perhaps only to character values (*char*). Since the language supports first-class functions—functions as values—variables can also be given function types. The type $\tau_1 \to \tau_2$ indicates a function that can be applied to values of type τ_1 to yield values of type τ_2. The language assumes some set of basic types γ, which correspond to the types of basic constants c. Two basic constants, **true** and **false** are assumed and they are associated with the **bool** type for boolean values.

The object language singles out a subset of the expressions as values v.

$$v \quad ::= \quad c \mid f \mid \lambda x : \tau.e$$

Intuitively, values are the expected results of evaluating programs or subprograms.

This language is small, but it can simulate many standard language constructs. For instance, to simulate the **let** variable binding construct

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

use the expression

$$(\lambda x : \tau.e_2)\ e_1$$

where τ is the type of e_1.

Presenting variable binding as a derived form keeps the foundational core of the object language small. Derived forms enable non-essential but pragmatically useful language constructs to be understood in terms of more basic constructs, and also suggest an implementation strategy.

1.1. Dynamic Semantics. The dynamic semantics of the object language is formalized using the reduction semantics approach [**20**]. In reduction semantics, the operational behavior of language constructs is expressed using term-rewrite rules over distinguished subexpressions of a program. The object language has four basic rewrites, called its *notions of reduction*:

$\boxed{r \longrightarrow e}$

$(\lambda x : \tau.e)\ v$	$\longrightarrow$	$e[v/x]$	
$f\,c$	$\longrightarrow$	$\delta(f, c)$	$(f, c) \in DOM(\delta)$
if true then e_1 **else** e_2	$\longrightarrow$	e_1	
if false then e_1 **else** e_2	$\longrightarrow$	e_2	

Each reduction rule transforms a *redex*, the expression on the left hand side of the arrow, into a *contractum*, the result of a reduction. Two of the object-language reduction rules handle application expressions where both the operator and operand are values, $v_1\ v_2$. The first rule is the call-by-value β_v rule of the lambda calculus [**64**]: when a function abstraction is applied to a value, the value is substituted throughout the function's body for its bound variable. The notation $e[v/x]$ denotes the process of replacing every free instance of the variable x in e with the value v. The second reduction rule handles the constant parameters of the language. An auxiliary partial function $\delta(f, c)$ defines the result of applying a function constant f to a basic constant c. For example, if the object language is augmented with natural numbers and a successor function *add1*, then $\delta(\mathit{add1}, n) = n+1$, where n is a natural number. The δ function need not be defined for every combination of constants; for instance, in a language that also has character constants, $\delta(\mathit{zero?}, \texttt{'x'})$ need not be defined. The last two notions of reduction address conditional expressions and rely on the two assumed boolean constants **true** and **false**.

The notions of reduction express how a subexpression might be evaluated, but they do not imply what order expressions might be evaluated in. To establish an order of evaluation, the reduction semantics uses evaluation contexts.

$$E \quad ::= \quad \Box \mid E[\Box\ t] \mid E[v\ \Box] \mid E[\textbf{if}\ \Box\ \textbf{then}\ e_1\ \textbf{else}\ e_2]$$

An evaluation context represents a program with a single hole in it, represented by the symbol $\Box$. An evaluation context E can be turned into an object-language expression by plugging an expression e into its hole, generally expressed with the notation $E[e]$. For instance, the context $\Box$ is simply a hole, so plugging a term into it yields the term itself:

$$\Box[e] = e.$$

Evaluation contexts are constructed by progressively filling in the hole of a context with a new expression that has a hole in it. For example,

$$(\Box[add1\ \Box])[\Box\ 7] = add1\ (\Box\ 7)$$

The definition of evaluation contexts limits the set of legal "subexpressions with holes" that can extend a context.

Evaluation contexts are used specifically to define program evaluation steps.

$$\frac{r \rightarrow e}{E[r] \longmapsto E[e]}$$

The one-step evaluation relation $e \longmapsto e'$ specifies how and where each single step of reduction happens in a program. It decomposes a program into a context and a redex, performs a notion of reduction on the redex, and plugs the resulting contractum into the context. Multi-step evaluation, $\longmapsto\!\!\!\!\!\rightarrow$, is defined as the reflexive-transitive closure of single-step evaluation

$$\frac{}{e \longmapsto\!\!\!\!\!\rightarrow e} \qquad \frac{e_1 \longmapsto\!\!\!\!\!\rightarrow e_2 \qquad e_2 \longmapsto e_3}{e_1 \longmapsto\!\!\!\!\!\rightarrow e_3}$$

This, in turn is used to define an evaluator, a partial function from programs (closed expressions) to values.

DEFINITION 1. *If e is a program, then*

$$eval(e) = \begin{cases} c & \text{if } e \longmapsto\!\!\!\!\!\rightarrow\ c \\ f & \text{if } e \longmapsto\!\!\!\!\!\rightarrow\ f \\ \textbf{closure} & \text{if } e \longmapsto \lambda x : \tau.e \end{cases}$$

Evaluation contexts impose an order of evaluation on expressions. They restrict which subexpressions of a program can be operated on. For instance, notice that two kinds of contexts, $E[\Box\ t]$ and $E[v\ \Box]$ have to do with applications. The first kind allows a context to always decompose down into the operator position of an application. The second kind, on the other hand, only allows the operand position of an application to be visited if the operator position is a value. These two contexts together force evaluation to proceed from left to right when faced with an application expression. Also, the evaluation context $E[\textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2]$ establishes that the predicate position of a conditional is evaluated prior to either the consequent or alternative.

For example, take the following program.

$$(\lambda x : \textbf{int}.\mathit{add1}\ 7)\ \textbf{if}\ \mathit{zero}?\ 1\ \textbf{then}\ 6 + 7\ \textbf{else}\ 5$$

This program has three subexpressions that are redexes and could thus be reduced based on the notions of reduction: *add1* 7, *zero*? 1, and $6 + 7$. However, the evaluation contexts impose a strict ordering on which subexpressions to reduce.

$$(\lambda x : \textbf{int}.\mathit{add1}\ 7)\ \textbf{if}\ \mathit{zero}?\ 1\ \textbf{then}\ 6 + 7\ \textbf{else}\ 5 \longmapsto$$

$$(\lambda x : \textbf{int}.\mathit{add1}\ 7)\ \textbf{if false then}\ 6 + 7\ \textbf{else}\ 5 \longmapsto$$

$$(\lambda x : \textbf{int}.\mathit{add1}\ 7)\ 5 \longmapsto \mathit{add1}\ 7 \longmapsto 8$$

As a result of this ordering, the redex $6 + 7$ is never reduced; by reducing the conditional expression early, this redex is discarded.

In order to guarantee that the evaluator defined by the semantics is a function, it is sufficient to show that one-step reduction is deterministic, and therefore implementable on a computer. The property that guarantees this determinism is called *unique decomposition*.

LEMMA 1 (Unique Decomposition). *If e is a closed program, then one and only one of the following is true:*

(1) *e is a value (v).*
(2) *There is a unique evaluation context E and a unique value application $v_1\ v_2$ such that $e = E[v_1\ v_2]$.*
(3) *There is a unique evaluation context E and a unique conditional expression $\textbf{\textit{if}}\ v\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2$ such that $e = E[\textbf{\textit{if}}\ v\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2]$.*

The unique decomposition lemma proves that given any program, there is at most one possible redex that can be reduced using one-step reduction. If the term is a value, then there are no such redexes and evaluation is complete (according to our definition of *eval*) If the program decomposes to a redex r, then one-step reduction applies. Finally, a program might only decompose to terms that are not redexes. One example is the term $c_1\ \lambda x : \tau.e$. No application redex has a basic constant as its operator. When such a situation arises, evaluation has become *stuck*: one-step reduction can make no further progress.

Consider the following example programs of the object language and the result of evaluating them:

$$\begin{array}{lcl} 7 & \longmapsto\!\!\!\!\rightarrow & 7 \\ \textit{add1}\ (\textit{add1}\ 3) & \longmapsto\!\!\!\!\rightarrow & 5 \\ \textbf{if}\ \textit{zero}?\ 5\ \textbf{then}\ 4\ \textbf{else}\ 9 & \longmapsto\!\!\!\!\rightarrow & 9 \end{array}$$

The expression **if** *zero*? 5 **then** 4 **else** 9 checks if 5 is equal to zero . Upon determining that it is not, evaluation skips over the *consequent expression* 4 and evaluates the *alternate expression*, producing the value 9.

The derived **let** form declares variables and binds values to them. For instance, consider the following expression and its result:

$$\begin{array}{l} \textbf{let}\ x = 9 \\ \textbf{in} \qquad\qquad \longmapsto\!\!\!\!\rightarrow\ 8 \\ \quad \textit{sub1}\ x \end{array}$$

Typing Environment

ε ::= $x : \tau$ (environment bindings)

Γ ::= $\overline{\varepsilon_i}$ (typing environment)

$\boxed{\Gamma \vdash e : \tau}$ Well-typed expression

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\mathit{type}(c) = \gamma}{\Gamma \vdash c : \gamma} \qquad \frac{\mathit{type}(f) = \gamma_1 \to \gamma_2}{\Gamma \vdash f : \gamma_1 \to \gamma_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau}$$

FIGURE 1. Type System for the Object Language

It binds the value 9 to the variable x within the scope of its body, which is marked off by **in** and delimited by indentation-sensitive syntax as is done in Python or Haskell. The value of x is used to compute *sub1* x and yield the value 8.

Recall that the language has first-class functions, introduced as λ abstractions. Each abstracted variable is annotated with a type, which indicates the type of the argument to the function. Consider the following examples:

$$\mathbf{let}\ f = (\lambda x : \mathbf{int}.\mathit{double}\ x)\ \mathbf{in}\ f\ 5 \qquad \longmapsto\!\!\!\!\to \quad 10$$

$$\mathbf{let}\ f = (\lambda g : \mathbf{int} \to \mathbf{int}.g\ 7)\ \mathbf{in}\ f\ (\lambda x : \mathbf{int}.\mathit{double}\ x) \qquad \longmapsto\!\!\!\!\to \quad 14$$

Here, $(\lambda x : \mathbf{int}.\mathit{double}\ x)$ is the function that takes any value of type **int** and doubles it. Functions are first-class in that they can be passed as arguments to other functions or defined and used immediately. Functions can be named by **let**-binding them to variables. The function $(\lambda g : \mathbf{int} \to \mathbf{int}.g\ 7)$ accepts a function that transforms **int**s into **int**s, and returns the result of applying that function to 7.

1.2. Static Semantics. The static type system for the object language is defined as a three-place typing relation, $\Gamma \vdash e : \tau$. The relation is inductively defined using the set of type rule schema defined in Figure 1.

The type system for this language is standard [**62**]. It is parameterized over the set of basic types γ, as well as the functional and basic constants, f and c. For example, the object language may include the ground type **int** in order to classify integral values. To

connect the constants to their types, a typing function *type* maps each basic constant to a primitive type and each function constant to a function type. The *type* function assigns the **bool** type to the constants **true** and **false**.

The typing relation $\Gamma \vdash e : \tau$ asserts that in the typing environment Γ, the expression e has the type τ. A well-typed program is a closed expression that can be typed in the empty context, i.e., $\vdash e : \tau$.

To account for the function, value, and type constant parameters of the language, a restriction is imposed on the definition of the delta function δ of the dynamic semantics and the *type* function of the static semantics: If $\Gamma \vdash f\ c : \tau$, then $\delta(f, c)$ must be defined and $\mathit{type}(\delta(f, c)) = \tau$.

To connect these static semantics to the object language's dynamic semantics, *type safety* properties must be established. Type safety implies that all well-typed programs have certain desirable properties with respect to their types and how they are evaluated by the reduction semantics. In short, the well-typing relationship imposes certain guarantees on the dynamic semantics of programs. Type safety boils down to two properties: *progress* and *preservation*.

THEOREM 1 (Progress). *If e is a program and $\vdash e : \tau$ then either e is a value or there is some e' such that $e \longmapsto e'$.*

With respect to the one-step reduction relation, a program can be stopped at the final value, stuck at an invalid application, or able to take a reduction step. The progress theorem proves that well-typed programs are never stuck.

THEOREM 2 (Preservation). *If e is a program and $\vdash e : \tau$, and $e \longmapsto e'$, then $\vdash e' : \tau$.*

One-step reduction transforms one program into another program. The trace of a program's evaluation is then a progression of programs. The preservation theorem proves that one-step reduction never transforms a well-typed program into an ill-typed program. When combined with the progress theorem, this means that evaluation can never get stuck if

it starts with a well-typed program: it only terminates when the program has been reduced to a value.

1.3. Extending the Object Language. As discussed earlier, **let** binding can be simulated in the kernel language rather than added explicitly to it. In fact, it is possible to extend the kernel language syntax with this construct and mechanically transform programs from such an extended language down to the original kernel language.

Consider the object language extended with **let** binding

$$e ::= \ldots \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

Corresponding to the change above, the type system is also extended.

$$\frac{\Gamma \vdash e_1 : t_2 \qquad \Gamma, x : \tau_1 \vdash e_2 : t_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2}$$

Now, instead of extending the dynamic semantics of the language with new rules for the behavior of these constructs, a translation is defined from the extended language to the core language. Recall that the translation from a **let** binding to the core language introduces additional type information. Because types play a role in the translation, it must use type information. For this reason, the translation is defined over typing derivations, rather than directly defining it upon the terms of the language.

A *translation* system augments a type system with an extra term that represents the output of the translation process. In the case of the object language, $\Gamma \vdash e : \tau$ becomes $\Gamma \vdash e \rightsquigarrow e' : \tau$. The translation rules for the core language terms extend the original type rules in a straightforward manner: they define translation in terms of the translation of subterms. For instance, consider the translation rule for applications.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_1}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_2}$$

The translation of an application $e_1\ e_2$ is simply the application of the translations of its immediate subterms. The rule for **let**, on the other hand, performs the nontrivial

transformation described earlier.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \rightsquigarrow (\lambda x : \tau_1 . e_2')\ e_1' : \tau_2}$$

Vital to the soundness of this translation is the property that this translation preserves typability. That is to say, if e is a well-typed term of the extended language that translates to e', then e' is a well-typed term in the core kernel language. It is thus necessary to prove separately that translation always produces well-typed terms.

THEOREM 3. *If* $\Gamma \vdash e \rightsquigarrow e' : \tau$, *then* $\Gamma \vdash e' : \tau$.

Using this translation strategy, the dynamic semantics of **let** expressions is fully determined by how they are translated to more basic core language expressions.

2. The Kernel Metaprogramming Language

Having fully specified the object language, this section defines a language for compile-time metaprogramming over it. By design, the language is syntactically a pure superset of the object language. All object-language programs are legal input to the metaprogramming language: they correspond to programs that perform no metacomputations.

The metaprogramming language has a stratified syntax. Instead of simply one kind of expression, the language is split into two language levels that are defined in terms each other: the *code language* and the *metalanguage*. The expressions of the code language are as follows:

x	::=	$\langle$variable$\rangle$	
c	::=	$\langle$basic constant$\rangle$	
f	::=	$\langle$function constant$\rangle$	
e	::=	$x \mid \lambda x : e^s . e \mid e\ e$	(code language)
	$\mid$	$c \mid f \mid$ **if** e **then** e **else** e	
	$\mid$	$\sim e^s$	

Code-language expressions represent programs-as-data in the metaprogramming language, extending the object language with support for metaprogramming. The code language differs in two ways from the object language. First, the set of expressions is augmented

with a form $\sim e^s$. Second, rather than annotating each abstraction variable with some type τ, the variables are annotated with a term e^s. However, as is explained below, the terms e^s include the types τ, so an abstraction $\lambda x : \tau.e$ from the object language is a legal term of the code language. In fact, all expressions in the simple object language are valid code-language expressions. Just as the object language distinguishes a subset of expressions as values, the metaprogramming language distinguishes a subset of code-language expressions as *pure code*.

τ	::=	$\gamma \mid \tau \to \tau$	(ground type)
e^o	::=	$x \mid \lambda x : \tau.e^o \mid e^o\ e^o \mid c \mid f \mid \textbf{if}\ e^o\ \textbf{then}\ e^o\ \textbf{else}\ e^o$	(pure code)

Pure code is the subset of the code language that corresponds exactly to the terms of the object language. This is by design: the final result of evaluating a metaprogram is a program in the object language.

Terms e^s are expressions of the metalanguage.

α	::=	⟨type variable⟩	
γ	::=	⟨type constant⟩	
c^s	::=	$\gamma \mid \gamma^? \mid \to^? \mid \textbf{dom} \mid \textbf{cod} \mid \textbf{typeof}$	(metalanguage constants)
e^s	::=	$x \mid \alpha \mid \lambda x : \tau^s.e^s \mid e^s\ e^s \mid \Lambda\alpha.e^s \mid e^s[\tau^\alpha]$	(metalanguage)
	|	$c \mid f \mid \textbf{if}\ e^s\ \textbf{then}\ e^s\ \textbf{else}\ e^s$	
	|	$c^s \mid \prec e \succ \mid \%e^s \mid e^s \to e^s \mid e^s =_\tau e^s$	

Metalanguage expressions represent portions of a program that encode static computations. Whereas the code language represents programs as data, the metalanguage represents programs that operate on programs. The metaprogramming language is a two-level language, featuring computations that execute across two different stages, the metaprogram at compile-time and the object program at run-time.

The metalanguage and the code language have much in common. Both languages include among their terms the variables x, the basic constants c, and the function constants f; each includes a form of function abstraction, $\lambda x : \tau^s.e^s$ for the metalanguage, and $\lambda x : e^s.e$

for the code language; and each has a form of function application $e^s\ e^s$ and conditionals **if** e^s **then** e^s **else** e^s.

Each of the languages explicitly refers to the other. The metalanguage expression $\prec e \succ$ includes a subexpression from the code language, while the code-language expression $\sim e^s$ includes a metalanguage expression as a subexpression. These alone suffice to show that the two language definitions are mutually recursive.

As with the code language, a subset of metalanguage expressions is distinguished and called the metalanguage *values*.

c^{s^-}	::=	$\gamma^? \mid \rightarrow^? \mid \mathbf{dom} \mid \mathbf{cod} \mid \mathbf{typeof}$	(non-type metaconstant)
v^s	::=	$c \mid f \mid c^{s^-} \mid \prec e^o \succ \mid \tau \mid \lambda x : \tau^s . e^s \mid \Lambda \alpha . e^s$	(metavalue)

The basic and function constants are values of the metalanguage. Also, any type τ of the code language is a value of the metalanguage. This hints at the metalanguage's ability to perform computations over types. In fact, base types γ are constants of the metalanguage, just like the basic constants c. Furthermore, the function type constructor $\rightarrow$ is an expression of the metalanguage, $e^s \rightarrow e^s$. This explains why the code language annotates bound variables with metalanguage expressions: these expressions can denote code-language types. Thus the code language can use metalanguage computations to determine the types of its variables. Code expressions $\prec e^o \succ$, where the code subexpression is pure object-language code, are also values of the metalanguage. This property highlights that metaprograms can produce object-language code as values.

The code-language expression $\sim e^s$ provides the ability for the code language to explicitly escape to the metalanguage, compared to the implicit escape that occurs in typing contexts. The metaprogramming language is intended to be able to manipulate programs as data, and here is one place that this capability is highlighted. The code-language expression $\sim e^s$ is called a *splice*. The code language escapes to the metalanguage, which performs a computation and returns a *code object*, $\prec e^o \succ$. The metalanguage expression $\prec e \succ$ performs the analogous role of the code-language $\sim e^s$ expression: it allows the metalanguage to drop

back into the code language. In particular, such an expression is a code object constructor. The result of a code object constructor is a piece of object-language code. When such an object is the result of a code-language splice expression $\sim e^s$, the object-language code is spliced inline into the code-language expression. This behavior is analogous to Lisp's `unquote` operation. These operations are described in more detail below.

The metalanguage includes all of the basic constants and function constants of the code language, but it also includes some of its own. Besides the primitive code-language types γ, the language also includes a set of function constants that operate on types. For each primitive type γ, the language includes a predicate $\gamma^?$, which determines whether or not its type argument is equivalent to the type γ. The language also offers operators specifically geared toward function types. The operator $\rightarrow^?$ operates on a type and returns a boolean indicating whether the type given is a function type $\tau_1 \rightarrow \tau_2$. The **dom**(or *domain*) and **cod** (or *codomain*) operators decompose function types into their constituent parts, extracting the domain or codomain, respectively, of a function type, and acting as identity on any non-function type. The language also provides a type equality operator $\tau_1 =_\tau \tau_2$ which given two types indicates whether they are equivalent.

The **typeof** operator provides a means to access the type of a piece of code. This operator brings into play observations about type systems from Chapter 2.

Section 1 presents the static type system for object-language programs. Since metalanguage code objects are terms in this language, the **typeof** operation appeals to its type system to compute the type of expressions. The type system of the simple object language is expressed as a ternary relation $\Gamma \vdash e : \tau$, meaning that in a particular type environment Γ, the term e has the type τ. On the one hand, the type of an expression is partly determined by the types of its subexpressions; but on the other hand, the type environment Γ used to type each subexpression of a term is partly determined by the larger expression in which it is embedded. Specifically, a lambda abstraction $\lambda x : \tau.e$ is assigned a type based on the type of its body e when typed in a context that maps the variable x to the type τ. Thus, typing is partly compositional, partly context-dependent.

The type system itself induces the compositional type structure of expressions. However, the source of typing context is not as obvious. In particular, given a call to **typeof** in a metaprogram, a reasonable type environment Γ is needed to compute the type. The source of the proper type environment lies in the structure of metacomputations. Since a metaprogram is also an expression of the code language, all metalanguage computations are reached through escapes from the code language, either as a splice expression or as a type-annotation computation. Some of those escapes may occur in the body e of a code-language expression $\lambda x : e^s.e$. If the metalanguage expression that annotates the variable above is well-formed, then it evaluates to a code-language type τ. Thus, as far as the code language is concerned, the term e exists in the lexical context of the binding $x : \tau$. Since free code-language variables are bound by lambda abstractions, the metalanguage uses those bindings to determine the typing context used by a **typeof**. As formalized below, those context bindings exist within the evaluation context, or run-time stack, of metaprogram execution. Therefore the **typeof** operator conceptually performs stack inspection [**14**] to determine the relevant type environment.

The basic and functional constants of the code language are constants of the metalanguage. This suggests opportunities for the metalanguage to interact with the code language. In particular, the metalanguage provides the expression form $\%e^s$, which is referred to as *cross-stage persistence* [**89**]. The % operator, when applied to a basic constant, yields a piece of code in the code language that contains the constant. This operator is the means by which metalanguage constants can be embedded into the run-time language. Thus, for instance, a metaprogram can perform a sophisticated computation at compile-time and embed the result of that computation into the output program. This is an instance of pure static computation. Cross-stage persistence can only be applied to basic constants, however. Metalanguage abstractions cannot be persisted because they can contain computations that do not have a run-time analog. For instance, the run-time language has no values of type **code** or **type**. As such, persisting metalanguage expressions that manipulate these entities would have no meaning in the final output program.

The metaprogramming language has three different variable abstraction forms: code-language term variable abstraction, $e = \lambda x : e^s.e$, metalanguage term variable abstraction, $e^s = \lambda x : \tau^s.e^s$, and metalanguage type variable abstraction, $e^s = \Lambda\alpha.e^s$. As is standard, the language formalization implies a variable convention. The free variables in any expression are assumed to be different from any bound variables that appear in the same context. Furthermore, the variables bound by any abstraction are subject to alpha-conversion as fits the context. Since the code language and the metalanguage use the same term variables, and both kinds of abstraction can contain each other, all bound variables from either form are assumed to differ. In the metaprogramming language, programs are identified with the closed expressions of the code language, meaning that no term or type variables are free.

2.1. Dynamic Semantics. To formalize the discussion above, this subsection presents a reduction semantics for the metaprogramming language. Figure 2 presents the evaluation contexts for the language. Just as the syntax for the metaprogramming language is defined as two mutually recursive categories, the same is true of the evaluation contexts. The evaluation contexts split into two groups: the metalanguage contexts E^s and the code contexts E. Since all programs of the metaprogramming language are terms of the code language, each evaluation context represents a term of the code language that has a hole in it. A metalanguage context E^s is equivalent to a term of the code language that has a hole in a location that expects a term of the metalanguage. A code context E is a term of the code language with a hole in a location that expects a term of the code language. The base case of all evaluation contexts is a code context hole $E = \Box$. Some code contexts E are formed by extending metalanguage contexts E^s and vice-versa. For instance, a code context E whose hole is plugged with the splice form $\sim\Box$ becomes a metalanguage context $E[\sim\Box]$. Similarly, a metalanguage context E^s whose hole is filled with the code constructor form $\prec\Box\succ$ becomes a code context $E^s[\prec\Box\succ]$.

As is standard practice for reduction semantics, the metaprogramming language defines a set of redexes, which are distinguished terms of the metalanguage e^s, and notions of

$$
\begin{array}{lcl}
E^s & ::= & E^s[\Box\ e^s] \\
 & | & E^s[v^s\ \Box] \\
 & | & E^s[\textbf{if}\ \Box\ \textbf{then}\ e^s\ \textbf{else}\ e^s] \\
 & | & E[\lambda x : \Box.e] \\
 & | & E[\sim\Box] \\
 & | & E^s[\%\Box] \\
 & | & E^s[\Box \rightarrow e^s] \\
 & | & E^s[\tau \rightarrow \Box] \\
 & | & E^s[\Box =_\tau e^s] \\
 & | & E^s[v^s =_\tau \Box] \\
 & | & E^s[\Box[\tau^\alpha]] \\
E & ::= & \Box \\
 & | & E[\Box\ e] \\
 & | & E[e^o\ \Box] \\
 & | & E[\lambda x : \tau.\Box] \\
 & | & E[\textbf{if}\ \Box\ \textbf{then}\ e\ \textbf{else}\ e] \\
 & | & E[\textbf{if}\ e^o\ \textbf{then}\ \Box\ \textbf{else}\ e] \\
 & | & E[\textbf{if}\ e^o\ \textbf{then}\ e^o\ \textbf{else}\ \Box] \\
 & | & E^s[\prec\Box\succ]
\end{array}
$$

FIGURE 2. Kernel Language Evaluation Contexts

reduction which represent computational steps of the metalanguage. Redexes are denoted by the metavariable r.

In combination with metalanguage contexts, each notion of reduction captures a step of metaprogram execution.

$$\text{(meta-eval)}\frac{e_1^s \rightarrow_{E^s} e_2^s}{E^s[e_1^s] \longmapsto E^s[e_2^s]}$$

Thus, if a program can be decomposed into a metalanguage context and a metalanguage term that corresponds to one of the metalanguage redexes, then the program can be reduced one step using the corresponding notion of reduction.

The metaprogramming language includes a function application reduction.

$$(\lambda x : \tau^s.e^s)\ v^s \rightarrow_{E^s} e^s\ [v^s/x]^s$$

Its behavior is equivalent to function application in the dynamic semantics of the object language. Function application is defined in terms of capture-avoiding substitution, as is standard, but substitution in this language must take into account the hygiene of both

metalanguage variables and code-language variables. For instance, consider the following reduction.

$$(\lambda f : \mathbf{code}.\ \prec \lambda x : \mathbf{int}.x * \sim f \succ)\ \ \prec x + 7 \succ \longmapsto \prec \lambda z : \mathbf{int}.z * \sim \prec x + 7 \succ \succ$$

The code-language variable x is free in the code object $\prec x + 7 \succ$, but the same variable is bound in the code expression $\prec \lambda x : \mathbf{int}.x * \sim f \succ$. In order to prevent the inadvertent capture of x by the code-language abstraction, substitution must rename the bound variable in the code expression. This means that during substitution, variable renaming occurs in both the metalanguage and the code language. During metaprogram evaluation, substitution never applies to code variables. Nonetheless, code variables must sometimes be renamed.

The polymorphic type application rule is expressed in terms of a type substitution operation.

$$(\Lambda\alpha.e^s)[\tau^\alpha] \rightarrow_{E^s} e^s\ \{\tau^\alpha/\alpha\}^s$$

This substitution differs from the analogous mechanism from the polymorphic lambda calculus in that type variables also appear as term expressions in the metalanguage. Take, for instance the following trace.

$$(\Lambda\alpha.\lambda f : \alpha \to \mathbf{bool}.\lambda c : \alpha.$$
$$\quad \mathbf{if}\ (f\ c)\ \mathbf{then}\ \prec \lambda t : \alpha.\lambda f : \alpha.t \succ\ \mathbf{else}\ \prec \lambda t : \alpha.\lambda f : \alpha.f \succ)[\mathbf{int}]\ \mathit{zero}?\ 5 \longmapsto$$

$$(\lambda f : \mathbf{int} \to \mathbf{bool}.\lambda c : \mathbf{int}.$$
$$\quad \mathbf{if}\ (f\ c)\ \mathbf{then}\ \prec \lambda t : \mathbf{int}.\lambda f : \mathbf{int}.t \succ\ \mathbf{else}\ \prec \lambda t : \mathbf{int}.\lambda f : \mathbf{int}.f \succ)\ \mathit{zero}?\ 5 \longmapsto\!\!\!\!\to$$

$$\mathbf{if}\ (\mathit{zero}?\ 5)\ \mathbf{then}\ \prec \lambda t : \mathbf{int}.\lambda f : \mathbf{int}.t \succ\ \mathbf{else}\ \prec \lambda t : \mathbf{int}.\lambda f : \mathbf{int}.f \succ \longmapsto$$

$$\prec \lambda t : \mathbf{int}.\lambda f : \mathbf{int}.f \succ$$

The type **int** is substituted into the polymorphic function, and as a result the type annotations of the metalanguage variables f and c become ground types. However, the type variable α also appears in the type annotations of the code objects $\prec \lambda t : \alpha.\lambda f : \alpha. \ldots \succ$. In such contexts, type annotations are metalanguage expressions. As such metalanguage

type variables play two roles, as parameters to metalanguage types and as parameters to metalanguage terms.

The cross-stage persistence (CSP) rule encodes the semantics described earlier: CSP coerces constant values into their equivalent code representations.

$$\%c \rightarrow_{E^s} \prec c \succ$$

For example, consider the following trace.

$$(\lambda n : \mathbf{int} \prec 7 * \sim\%(n+4) \succ)\ 6 \longmapsto \prec 7 * \sim\%(6+4) \succ \longmapsto$$

$$\prec 7 * \sim\%10 \succ \longmapsto \prec 7 * \sim\prec 10 \succ \succ$$

After 6 is substituted for n in the function, reduction focuses on the body of the CSP operation, reducing the sum $6 + 4$ to 10. The CSP operation then becomes the current redex, and it coerces the metalanguage value 10 to the piece of code $\prec 10 \succ$.

The reduction steps for conditional expressions are standard, as is the delta rule for applying functional constants.

$$\begin{array}{lll}
\mathbf{if\ true\ then}\ e_1^s\ \mathbf{else}\ e_2^s & \rightarrow_{E^s} & e_1^s \\
\mathbf{if\ false\ then}\ e_1^s\ \mathbf{else}\ e_2^s & \rightarrow_{E^s} & e_2^s \\
f\ c & \rightarrow_{E^s} & \delta(f,c) \qquad (f,c) \in DOM(\delta)
\end{array}$$

The $\rightarrow$ operator is not associated with a particular notion of reduction because it plays the role of a data constructor in the language. Once its two arguments have been evaluated, the arrow operator returns a type as a value. For example, the expression $\mathbf{int} \rightarrow (\mathbf{if\ true\ then\ bool\ else\ int})$ evaluates to the object-language type $\mathbf{int} \rightarrow \mathbf{bool}$.

The rules for $\gamma^?$ and $\rightarrow^?$ query the identity or structure, respectively, of type values.

$$\begin{array}{lll}
\gamma^?\ \gamma & \rightarrow_{E^s} & \mathbf{true} \\
\gamma^?\ \tau & \rightarrow_{E^s} & \mathbf{false} \qquad \tau \not\equiv \gamma \\
\rightarrow^?\ (\tau_1 \rightarrow \tau_2) & \rightarrow_{E^s} & \mathbf{true} \\
\rightarrow^?\ \tau & \rightarrow_{E^s} & \mathbf{false} \qquad \tau \not\equiv \tau_1 \rightarrow \tau_2
\end{array}$$

The **dom** and **cod**operations access the domain and codomain, respectively, of a function type.

$$\begin{array}{lll} \mathbf{dom}(\tau_1 \to \tau_2) & \to_{E^s} & \tau_1 \\ \mathbf{dom}(\tau) & \to_{E^s} & \tau \qquad \tau \not\equiv \tau_1 \to \tau_2 \\ \mathbf{cod}(\tau_1 \to \tau_2) & \to_{E^s} & \tau_2 \\ \mathbf{cod}(\tau) & \to_{E^s} & \tau \qquad \tau \not\equiv \tau_1 \to \tau_2 \end{array}$$

If either is applied to a type that is not a function, the operation acts as identity. The language defines these operations in this manner for simplicity and accessibility. A more full-fledged metaprogramming language would replace these operations with a pattern-matching and dispatch facility.

Finally, the type equality operation $=_\tau$ performs a structural comparison between any two types.

$$\begin{array}{lll} \tau =_\tau \tau & \to_{E^s} & \mathbf{true} \\ \tau_1 =_\tau \tau_2 & \to_{E^s} & \mathbf{false} \qquad \tau_1 \not\equiv \tau_2 \end{array}$$

In addition to the notions of reduction of the metalanguage, the metaprogramming language defines one reduction step for the code language. Since only one reduction rule applies to terms of the code language, it is presented as part of the corresponding program reduction step, though it could be considered a notion of reduction for the code language.

$$(\text{splice})\frac{}{E[\sim\prec e^o \succ] \longmapsto E[e^o]}$$

If a program can be decomposed into a code context and a splice operation applied to a piece of pure object-language code, then the pure code is spliced into place. For an example of splicing in action, consider the following program trace.

$$\begin{array}{c} \sim((\lambda f : \mathbf{code} \to \mathbf{code}.\ \prec\sim(f\ \prec 2\succ) * \sim(f\ \prec 3\succ)\succ)\ (\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)) \longmapsto \\ \sim\prec\sim((\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)\ \prec 2\succ) * \sim((\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)\ \prec 3\succ)\succ \longmapsto \\ \sim\prec\sim\prec 5+\ \sim\prec 2\succ\succ\ * \sim((\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)\ \prec 3\succ)\succ)\ \longmapsto \\ \sim\prec\sim\prec 5+2\succ\ * \sim((\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)\ \prec 3\succ)\succ)\ \longmapsto \\ \sim\prec(5+2) * \sim((\lambda x : \mathbf{code}.\ \prec 5+\ \sim x\succ)\ \prec 3\succ)\succ)\ \longmapsto\!\!\!\!\to \\ \sim\prec(5+2)\ *(5+3)\succ \end{array}$$

The first step of reduction substitutes a function for the variable f, and the second step substitutes the code $\prec 2 \succ$ for the variable x, but the next two steps focus on splices of pure code and each splices a pure code object into its surrounding code object.

Finally, the metaprogramming language specifies the **typeof** operation as two context-sensitive rules.

$$\text{(typeof)}\frac{\Gamma^s(E^s) \vdash e^o : \tau}{E^s[\textbf{typeof } \prec e^o \succ] \longmapsto E^s[\tau]} \qquad \text{(notype)}\frac{\forall \tau . \Gamma^s(E^s) \nvdash e^o : \tau}{E^s[\textbf{typeof } \prec e^o \succ] \longmapsto \bot}$$

As described earlier, the **typeof** operation is specified in terms of the type system for the object language. If the code argument to the operation is typeable in terms of the object-language type system, then that type is the result of this operation. If the code argument cannot be typed, then execution of the metaprogram aborts by transitioning to $\bot$.

The definition of **typeof** relies on two properties of the object-language type system. First, in the simply-typed calculus, every typeable expression has one and only one type. Therefore, the mapping from typeable expressions and contexts to types is a function, meaning that the result of **typeof** is well-defined and single-valued. Second, the type system of the simply-typed calculus is decidable, which means that given a term and a typing context, it is always possible to determine whether or not a term can be typed in the object-language type system.

Furthermore, unlike every other reduction step of the metaprogramming language, the **typeof** operation is in part a function of the current evaluation context E^s. The **typeof** operation relies on stack inspection to determine the proper type environment to use for typing its argument.

$$\boxed{\Gamma^*(E^*) : E^* \to \overline{x_i : \tau_i}}$$

$$\begin{array}{lcl} \Gamma(E[\lambda x : \tau . \square]) & = & \Gamma(E), x : \tau \\ \Gamma^*(E^*[\ldots]) & = & \Gamma^*(E^*) \end{array}$$

The functions $\Gamma^s(E^s)$ and $\Gamma(E)$ traverse the evaluation context, collecting variable-type annotation pairs from code-language function abstractions. The definition above uses the notation Γ^* to abbreviate the trivial cases, which do not directly contribute bindings to the

type environment. The dynamic context of a metaprogram corresponds to the lexical context of the code-language expression under construction. For this reason, the variable-type annotations embedded in the current evaluation context correspond to the type environment that the object-language type system would use to deduce the type of an expression at the same position in the program.

Consider a simple use of the **typeof** operator to compute a type annotation for a lambda abstraction:

$$\lambda x : \mathbf{int}.(\lambda y : (\mathbf{typeof} \quad \prec \mathit{zero}?\ x \succ).x * 4)\ \mathbf{false}$$

The **typeof** operator uses the object-language type system to type the expression $\mathit{zero}?\ x$ in a context where the variable x has the type **int**, resulting in the **bool** type. The result of metacomputation is as follows.

$$\lambda x : \mathbf{int}.(\lambda y : \mathbf{bool}.x * 4)\ \mathbf{false}$$

One notable property of the metaprogramming language is that even though metaprograms can only construct syntactically well-formed object-language expressions, they can still construct ill-typed object code. Because of this, the **typeof** operation can fail, and the behavior of the language is to abort evaluation, as though an uncaught exception were thrown. Such behavior is not ideal for a practical programming language, but it suffices for theoretical purposes. A more complete language definition could guarantee production of a complete program by allowing **typeof** to optionally produce a type. One simple approach to this introduces an optional type to the language, like the `Maybe` type of Haskell.

Since a metaprogram might never call **typeof** during its computation, the final result of metaprogram execution may be a well-formed but ill-typed object-language program. A complete language infrastructure with support for metaprogramming must still type-check the output of metaprogram execution in order to guarantee that the resulting object-language program is well-typed. Nonetheless, the metaprogramming language design enables metaprograms to take advantage of intermediate type information gleaned from

subexpressions using the **typeof** operator. By combining metaprogram execution with full type checking, all type errors are still guaranteed to be caught prior to run-time.

2.2. The Type System. The metalanguage provides constructs to manipulate the types of the code language/object language, but the metalanguage itself is also statically typed. Thus, the metaprogramming language distinguishes a set of metalanguage types.

$$\tau^s \quad ::= \quad \alpha \mid \gamma \mid \textbf{code} \mid \textbf{type} \mid \tau^s \to \tau^s \mid \forall\alpha.\tau^s$$

The set of types τ^s subsumes the set of code-language types τ. Thus every function from the language of pure code is syntactically a legal metalanguage function.

Just as in the code language, the variables abstracted by lambda expressions have type annotations, $\lambda x : \tau^s.e^s$. The types of the metalanguage extend the types of the code language: every code-language type is a valid metalanguage type.

The metalanguage adds two particular types to the language in order to address its role as a metaprogramming language. The **type** type is the type of all pieces of metalanguage data that represent types. For instance, the **int** type, when used as data in the metalanguage, is given the type **type**. The type **type** plays the same role as the *kind* $*$ from formal type theory. In type theory, a kind is the type of a type. Analogously, the type of each code-language type is **type**.

The metalanguage also has a type **code**, which is the type of constructed pieces of object code. Recall that code is created in the metalanguage using expressions of the form $\prec e \succ$. This operation drops computation into the code language, which may escape back into the metalanguage. The result of such an expression, when its body has completed evaluation, is a piece of well-formed object-language code. That code has type **code**. Notice that all code has a single type. This is in stark contrast to a multi-stage metaprogramming language like MetaML [**89**], where the type of a code-forming expression is also annotated with its eventual type. In this model, the object-language type of a code language is not statically known. This makes the type system weaker, but it also raises restrictions on the capabilities of the metaprogramming language. Since the metalanguage type system does not need to

prove the type of the code-language expressions it constructs, metaprograms have greater flexibility in the kinds of code objects they can construct.

As is standard, the type system of the metaprogramming language is defined using type environments Γ.

$$\begin{array}{lcll} \varepsilon & ::= & x : \tau^s \mid x : \mathbf{dyn} \mid \alpha : * & \text{(environment bindings)} \\ \Gamma & ::= & \overline{\varepsilon_i} & \text{(environment)} \end{array}$$

Type environments for the metaprogramming language contain three kinds of type bindings. One kind of binding associates a variable with a metalanguage type τ^s. This is the binding used for metalanguage variables. Another kind of binding marks a term variable as **dyn**, meaning it belongs to the code language. The final binding establishes that a type variable α is active in the current type environment. In practice this annotation ensures that the variable convention is preserved.

The language definition imposes well-formedness conditions upon type environments. A type environment can only contain one binding for any term or type variable, so a type environment can be treated as a partial function on variables. Also, since metalanguage types can contain references to type variables, those type variables must be accounted for. Therefore, a type environment is well-formed only if every type variable α referenced in a binding of the form $x : \tau^s$ appears in an earlier binding $\alpha : *$ in the environment.

$\boxed{\Gamma \ \mathbf{wf}}$

$$\frac{}{\emptyset \ \mathbf{wf}} \qquad \frac{\Gamma \ \mathbf{wf} \quad \alpha \notin FV(\Gamma)}{\Gamma, \alpha : * \ \mathbf{wf}} \qquad \frac{\Gamma \ \mathbf{wf} \quad x \notin FV(\Gamma)}{\Gamma, x : \mathbf{dyn} \ \mathbf{wf}}$$

$$\frac{\Gamma \ \mathbf{wf} \quad x \notin FV(\Gamma) \quad FV(\tau^s) \subseteq FV(\Gamma)}{\Gamma, x : \tau^s \ \mathbf{wf}}$$

From here on, well-formedness of type environments is assumed.

The type system is defined as two mutually recursive relations, one that establishes metalanguage terms as well-typed ($\Gamma \vdash e^s : \tau^s$), and one that establishes code-language terms as well-formed ($\Gamma \vdash e \ \mathbf{wf}$). As is suggested by the form of these two relations, the code-language expressions do not directly have types associated with them: a code-language

expression is simply considered well-formed or not. On the other hand, expressions in the metalanguage have types.

The rules for well-formed variable references in the code language and well-typed variable references in the metalanguage are similar.

$$\frac{(x : \tau^s) \in \Gamma}{\Gamma \vdash x : \tau^s} \qquad \frac{(x : \mathbf{dyn}) \in \Gamma}{\Gamma \vdash x\ \mathbf{wf}}$$

A metalanguage term variable is well-typed if the type environment associates it with a type τ^s. Similarly, a code-language term is well typed if the type environment marks it as dynamic, $x : \mathbf{dyn}$. Due to the well-formedness requirements of type environments, bindings are unique, so any particular term variable can only be associated with the metalanguage or the code language. This property ensures that a bound variable in a well-typed term can only be used in contexts that correspond to its binding context: code-language-bound variables can only be referenced in code-language subterms and metalanguage-bound variables can only be referenced in metalanguage subterms.

The relationship between the types of the metalanguage and the well-formedness criteria of the code language can be seen in the expressions that sit at the intersection between the two languages. For instance, take the splice rule.

$$\frac{\Gamma \vdash e^s : \mathbf{code}}{\Gamma \vdash {\sim} e^s\ \mathbf{wf}}$$

A splice is an expression of the code language, but it has an expression of the metalanguage as a subexpression. In order for a splice expression to be well-formed, the metalanguage expression must have type **code**, meaning that its evaluation will result in a piece of object-language code that can be spliced into place.

Consider also the code-language lambda expression rule.

$$\frac{\Gamma \vdash e^s : \mathbf{type} \qquad \Gamma, x : \mathbf{dyn} \vdash e\ \mathbf{wf}}{\Gamma \vdash \lambda x : e^s . e\ \mathbf{wf}}$$

The variable annotation position of a code-language abstraction is also an expression of the metalanguage, and it must have type **type**, meaning that the subexpression must yield an object-language type. The metalanguage can produce both object-language types

and object-language code. However the metalanguage is more versatile. Its type system includes typings for all basic and functional constants, meaning that the language can express general computations: all code-language expressions can be expressed and executed in the metalanguage.

The body of a code-language abstraction must be deemed well-formed in an environment extended with its abstracted variable x tagged as **dyn**. By well-formedness of type environments, the variable x cannot already be bound in the current type environment, but by the variable convention the binding of x can be renamed implicitly if it conflicts with any variable in the current type environment. Thus, any reference to x in the body of e refers to this particular binding.

Now, consider how the metalanguage type rules interact with the code language, particularly in the code-forming expression form.

$$\frac{\Gamma \vdash e \ \mathbf{wf}}{\Gamma \vdash \prec e \succ : \mathbf{code}}$$

This expression of the metalanguage contains a code-language subexpression. In order for a code expression to be well-typed, in which case it will have type **code**, the code-language subexpression must be well-formed: any nested splice expressions must have type **code**, and any type annotations must have type **type**.

Now consider the cross-stage persistence form of the metalanguage.

$$\frac{\Gamma \vdash e^s : \gamma}{\Gamma \vdash \%e^s : \mathbf{code}}$$

This expression has a metalanguage subexpression, but that expression is restricted to yield a value of a primitive type. Intuitively, this means that the result of evaluating that expression must be a basic constant c, which is both a legal metalanguage expression and code-language expression.

In the code language, basic constants and function constants are always well-formed.

$$\frac{}{\Gamma \vdash c \ \mathbf{wf}} \qquad \frac{}{\Gamma \vdash f \ \mathbf{wf}}$$

Recall, however, that the meta-language can construct ill-typed expressions in the object language. Thus the type system ensures that the combination of code-language expressions fits the syntactic structure of the code language.

On the other hand, basic constants and function constants are typed when they appear in the metalanguage.

$$\frac{\mathit{type}(f) = \gamma_1 \to \gamma_2}{\Gamma \vdash f : \gamma_1 \to \gamma_2} \qquad \frac{\mathit{type}(c) = \gamma}{\Gamma \vdash c : \gamma}$$

By assumption, the types assigned to basic constants and function constants are the same as the types that are assigned to them by the type system of the code language.

code-language application expressions and conditional expressions are well-formed so long as their immediate subexpressions are well-formed.

$$\frac{\Gamma \vdash e_1 \ \mathbf{wf} \quad \Gamma \vdash e_2 \ \mathbf{wf}}{\Gamma \vdash e_1 \ e_2 \ \mathbf{wf}} \qquad \frac{\Gamma \vdash e_1 \ \mathbf{wf} \quad \Gamma \vdash e_2 \ \mathbf{wf} \quad \Gamma \vdash e_3 \ \mathbf{wf}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \ \mathbf{wf}}$$

In contrast, metalanguage application and conditional expressions are typed in the standard simply-typed fashion.

$$\frac{\Gamma \vdash e_1^s : \tau_1^s \to \tau_2^s \quad \Gamma \vdash e_2^s : \tau_1^s}{\Gamma \vdash e_1^s \ e_2^s : \tau_2^s} \qquad \frac{\Gamma \vdash e_1^s : \mathbf{bool} \quad \Gamma \vdash e_2^s : \tau^s \quad \Gamma \vdash e_3^s : \tau^s}{\Gamma \vdash \mathbf{if}\ e_1^s\ \mathbf{then}\ e_2^s\ \mathbf{else}\ e_3^s : \tau^s}$$

Similarly, the type rules for metalanguage lambda abstractions are standard.

$$\frac{\Gamma, x : \tau_1^s \vdash e^s : \tau_2^s}{\Gamma \vdash \lambda x : \tau_1^s . e^s : \tau_1^s \to \tau_2^s}$$

The metalanguage introduces a number of constants specifically for metaprogramming. Each of these constants is given an appropriate type by the type system. In fact, just as the *type* function establishes the types of basic and function constants, a similar function could be used to define the types of these operators and any other metalanguage-specific operators and constants. Here, however, their type rules are explicitly presented because they are fundamental to the uses of the metaprogramming language. Their type assignments

are straightforward.

$$\frac{}{\Gamma \vdash \mathbf{dom} : \mathbf{type} \to \mathbf{type}} \qquad \frac{}{\Gamma \vdash \mathbf{cod} : \mathbf{type} \to \mathbf{type}} \qquad \frac{}{\Gamma \vdash \gamma^{?} : \mathbf{type} \to \mathbf{bool}}$$

$$\frac{}{\Gamma \vdash \to^{?} : \mathbf{type} \to \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{typeof} \ : \mathbf{code} \to \mathbf{type}}$$

The type equality operation $=_\tau$ can be considered a special case of the language constants that uses infix notation and takes two arguments.

$$\frac{\Gamma \vdash e_1^s : \mathbf{type} \qquad \Gamma \vdash e_2^s : \mathbf{type}}{\Gamma \vdash e_1^s =_\tau e_2^s : \mathbf{bool}}$$

Metalanguage type abstractions are typed in a manner similar to the polymorphic lambda calculus [**27**, **69**].

$$\frac{\Gamma, \alpha : * \vdash e^s : \tau^s}{\Gamma \vdash \Lambda\alpha.e^s : \forall\alpha.\tau^s}$$

As with term abstractions, A type abstraction is well typed if its body is well-typed in a context where the abstracted variable is added to the current environment. Rather than specifying a type for the type variable, it is specified to have kind $*$, meaning that it is the type of a term. In this language, the kind marker adds no extra information, but in an extended language, type variables could have a variety of kinds, as in the calculus F^ω [**27**].

Type applications are also well-typed along standard lines.

$$\frac{\Gamma \vdash e_1^s : \forall\alpha.\tau^s \qquad \Gamma \vdash \tau^\alpha : \mathbf{type}}{\Gamma \vdash e_1^s[\tau^\alpha] : \tau^s \left\{\tau^\alpha/\alpha\right\}^\tau}$$

The operator position of a type application must be a computation that results in a value of polymorphic type. The operand position of a type application is restricted so that it does not have the computational power of the metalanguage: only code-language types with type variables are allowed. Nonetheless, these operands must meet well-typing criteria. In particular, the operand must be a valid type according to the current type environment. In practice this means that every type variable that occurs in the operand must be in the domain of the current type environment. This property helps ensure that the type system respects the variable convention.

The type system has three rules that relate to metalanguage terms denoting code-language types.

$$\frac{(\alpha : *) \in \Gamma}{\Gamma \vdash \alpha : \mathbf{type}} \qquad \frac{}{\Gamma \vdash \gamma : \mathbf{type}} \qquad \frac{\Gamma \vdash e_1^s : \mathbf{type} \quad \Gamma \vdash e_2^s : \mathbf{type}}{\Gamma \vdash e_1^s \to e_2^s : \mathbf{type}}$$

All primitive types γ, when used as expressions, have type **type**. Also, the function type constructor $\to$ takes as its arguments two well-typed metalanguage expressions of type **type**, and the result itself is of type **type**. Finally, any type variable that is in the domain of the type environment is a valid expression of type **type**. In a sense, the type rules for γ and $\to$ together replace the syntactic (BNF) definition of an object-language type, replacing syntactic set membership with computational type membership.

2.3. Reflective Parametric Polymorphism. The design of parametric polymorphism in the metalanguage illustrates an interesting aspect of the language design. The traditional model of parametric polymorphism is System F [**27**]. That language is essentially the simply typed lambda calculus augmented with a type abstraction and type application operation. These operations make it possible to express generalized programs that are not tied to a particular family of ground types but would otherwise require multiple implementations of the same code with different types in order to function, leading to redundant code. In System F, a type variable can represent any type, and a type application can take any type of the language, including parametric types.

The type system of the metaprogram introduces similar type application and abstraction operators to the metalanguage. However, significant restrictions are placed on them. In particular, the operand of a type application can only be a variable-parameterized type of the object language. Types like **code**, **type**, and even quantified types $\forall\alpha.\tau^s$ cannot be arguments to a type application.

This form of parametric polymorphism, in its limited scope, adds expressiveness to the metaprogramming language. In System F, types only appear as type annotations on variables, as arguments to type applications, and as abstracted type variables. In the metaprogramming language, however, type variables can also appear in term contexts. In

fact, as the syntax of the metalanguage dictates, a type variable α can appear anywhere that a term variable x can appear, so long as that context expects a value of type **type**. Type variables extend the expressiveness of the language because they can also appear in the types of metalanguage terms τ^s, and can appear in the operand position of type applications, two positions where term variables x are not legal. A program can be abstracted over code-language types and abstracted types can, in turn, be used both as data in the metalanguage and as type information.

One downside of this hybrid polymorphism is that as specified it does not cooperate with cross-stage persistence. For instance, the program

$$(\Lambda\alpha.\lambda x : \alpha.\%x)[\mathbf{int}]\ 5$$

Is not well typed, because x does not have a type γ in the expression $\%x$, even though the dynamic semantics of the language would produce the desired answer. The metaprogramming language type system can be altered to support parameterized cross-stage persistence, however the resulting type system further limits type parameterization. First, replace the type schemes τ^a with primitive type schemes.

$$\begin{array}{lll} \gamma^\alpha & ::= & \gamma \mid \alpha \\ e^s & ::= & \ldots \mid e^s[\gamma^\alpha] \end{array}$$

The type rule for cross-stage persistence could then be altered accordingly.

$$\frac{e^s : \gamma^\alpha}{\%e^s : \mathbf{code}}$$

As a result of these changes, type parameterization only applies to primitive types, but cross-stage persistence can be used in type-parametric contexts.

$$(\Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.\%(f\ (f\ x)))[\mathbf{int}]\ \mathit{add1}\ 4$$

In this example, a type parameterized expression applies some function twice to some value of primitive type before coercing the result into code.

While the above approach limits the expressiveness of hybrid type parameterization in order to extend cross-stage persistence to type-parameterized values, a more expansive approach could extend the language and type system so that some functional values can also be persisted into the code language (cf. Chapter 7).

2.4. Metatheory. Given the formal definition of the metaprogramming language, it is now possible to reason precisely about its behavior, in particular to ensure that the language specification does in fact define a consistent language with well-behaved functionality.

THEOREM 4 (Unique Decomposition). *If* $\Gamma \vdash e$ ***wf****, and* e *is not proper code, then* e *can be uniquely decomposed into one and only one of* $E^s[x]$, $E^s[\alpha]$, $E^s[r]$, *or* $E[\sim\prec e^o \succ]$.

This theorem implies a term e can only be subject to reduction $e \longmapsto e'$, in one way. Any given code-language term can only be decomposed into a context and a redex at most one way. Another lemma guarantees that a closed code-language term cannot be decomposed into a metalanguage context E^s and a term or type variable. Thus, a well-typed program e is either proper code, which is the final result of running a metaprogram, or subject to a step of reduction.

THEOREM 5 (Progress).

(1) *If* e *is closed,* $\Gamma \vdash e$ ***wf****, then* e *is proper code or there is some* e' *such that* $e \longmapsto e'$ *or* $e \longmapsto \bot$.

THEOREM 6 (Preservation). *If* e *is closed,* $\Gamma \vdash e$ ***wf****, and* $e \longmapsto e'$, *then* $\Gamma \vdash e'$ ***wf****.*

These two theorems validate the relationship between the static and dynamic semantics of the metaprogramming language. Bear in mind that a type safe program can still fail to terminate with an answer if an application of **typeof** cannot produce a type for its code-language argument. However, this is by design: the language allows **typeof** to fail.

2.5. Examples. Now that the entire metaprogramming language has been introduced, consider some example programs that utilize its capabilities. The first example demonstrates

how the language can stage programs so that parts of them can be pre-computed prior to traditional run-time.

$$\sim((\lambda ctSum : \mathbf{int}.\ \prec(\lambda rtSum : \mathbf{int}.rtSum - \sim\%ctSum)\ (5+3)\succ)\ (5+3))$$

This program escapes to the metalanguage in order to compute the value of $5+3$. That value is bound, via function application, to *ctSum*, and the function body creates a piece of code that persists the value of *ctSum* into a larger expression.

The result of metacomputation is the following object-language program.

$$(\lambda rtSum : \mathbf{int}.rtSum - 8)\ (5+3)$$

The computation for *rtSum*'s value is left intact, but *ctSum* has been computed and its value substituted in its place.

This next example also performs a computation at compile-time and embeds the result in run-time code.

$$\begin{array}{l}\sim((\lambda pow : \mathbf{int} \to \mathbf{int} \to \mathbf{int}.\%(pow\ 5\ 7))\\ \quad(\mathbf{fix}\ (\lambda p : \mathbf{int} \to \mathbf{int} \to \mathbf{int}.\\ \qquad\lambda m : \mathbf{int}.\lambda n : \mathbf{int}.\\ \qquad\quad \mathbf{if}\ zero?\ n\ \mathbf{then}\ 1\ \mathbf{else}\ m * (p\ m\ (sub1\ n)))))\end{array}$$

This example assumes support for recursion via a fixpoint operator **fix**. A recursive exponentiation function is defined and passed to a function under the name *pow*. There, the function is used to compute the value of 5^7, which is injected into a code object. The final result of evaluating this program is 78125, which is the result of splicing the code object into the otherwise empty program.

A variant of the previous example implements an exponentiation function generator, rather than immediately performing the operation.

$$\begin{array}{l}\sim((\lambda pow : \mathbf{code} \to \mathbf{int} \to \mathbf{code}.\ \prec\lambda m : \mathbf{int}.\ \sim(pow\ \prec m\succ\ 7)\succ)\\ \quad(\mathbf{fix}\ (\lambda p : \mathbf{code} \to \mathbf{int} \to \mathbf{code}.\\ \qquad\lambda m : \mathbf{code}.\lambda n : \mathbf{int}.\\ \qquad\quad \mathbf{if}\ zero?\ n\ \mathbf{then}\ \ \prec 1\succ\ \ \mathbf{else}\ \ \prec\sim m *\ \sim(p\ m\ (sub1\ n))\succ)))\end{array}$$

The next example defines a function that takes a type and returns the number of curried arguments it takes, zero if the type is not a function:

$$\sim((\lambda numArgs : \mathbf{type} \to \mathbf{code}.\%(numArgs\ (\mathbf{int} \to \mathbf{bool} \to \mathbf{int})))$$
$$(\mathbf{fix}\ (\lambda n : \mathbf{type} \to \mathbf{code}.$$
$$\lambda t : \mathbf{type}.$$
$$\mathbf{if}\ (\to^?\ t)\ \mathbf{then}\ 1 + n\ (\mathbf{cod}\ t)\ \mathbf{else}\ 0)))$$

Finally, the following example takes two type definitions and computes a new type that could represent the result of adding values of the two types:

$$\sim((\lambda promote : \mathbf{type} \to \mathbf{type} \to \mathbf{code}.$$
$$\prec(\lambda x : promote\ \mathbf{int}\ \mathbf{float}.x * x)\ (55 + 7.7)\succ)$$
$$(\lambda x : \mathbf{type}.\ \lambda y : \mathbf{type}.$$
$$\mathbf{if}\ x =_\tau y\ \mathbf{then}\ x$$
$$\mathbf{else\ if}\ (\mathbf{int}^? x)\ \&\&\ (\mathbf{float}^? y)\ \mathbf{then\ float}$$
$$\mathbf{else}\ \ldots))$$

The result of evaluating this program is an object-language program with a definite type annotation on its argument.

$$(\lambda x : \mathbf{float}.x * x)\ (55 + 7.7)$$

CHAPTER 5

Modelling the Kernel Language

This chapter describes an implementation of the kernel language developed using the PLT Redex [**50**] system for directly implementing reduction semantics.

1. Motivation

To gain intuitions about the kernel language, as well as to provide a tool for learning about the style of metaprogramming that it provides, implementing and observing programs in action is key. Furthermore, an implementation provides a means to experiment actively with the language design. This section describes an implementation of the full kernel language using the PLT Redex modeling tool. PLT Redex is a library of the Dr. Scheme [**21**] programming system. It provides tools for directly implementing, debugging, and executing reduction semantics. The full implementation is presented in Appendix C

2. Syntax

The syntax of a reduction semantics is represented using the `language` form. For instance, the definition of the kernel metalanguage begins as follows.

```
(define meta-k
  (language
   (g int bool)
   (x variable-not-otherwise-mentioned)
   (a variable-not-otherwise-mentioned)
   (c number #t #f)
   (f add1 zero? not sub1)
   (bop + < - *)

   ...
   ...))
```

The above specification is comparable to the specification given on Page 65. The `language` form consists of a list of productions. Each production begins with a nonterminal symbol and enumerates all constructions that are instances of that nonterminal. In the partial definition above, the nonterminal `g` is equivalent to γ in the formal specification of the kernel language, and it represents all primitive types. This particular model includes integral (`int`) and boolean (`bool`) types. The nonterminal `c` uses the special keyword `number` as well as the literal Scheme boolean values `#t` and `#f` to include the integrals and booleans in the language. The nonterminal `f` includes several common unary operators, as well as several binary arithmetic and boolean operators. The binary operators are a straightforward extension of the language design.

In addition to the basic constants and primitive operators, the language also defines metalanguage-specific operators.

```
(g? int? bool?)
(cs- g? ->? dom cod typeof)
(cs g cs-)
```

The nonterminal `g?` corresponds to the primitive type predicates $\gamma^?$ from the kernel language specification. The other metalanguage operators directly correspond to the specification.

Since PLT Redex represents language syntax using s-expressions, prefix notation is the most natural way to specify the structure of programs. As such, the kernel language is given a prefix-based notation.

```
;; e - code language
(e x (lam (x e^s) e) (e e) c f (if e e e) (splice e^s)
   (bop e e))

;; e^o - pure code
(e^o x (lam (x t) e^o) (e^o e^o) c f (if e^o e^o e^o)
     (bop e^o e^o))
```

The code language uses essentially Lisp notation. A function abstraction of the form $\lambda x : e^s.e$ is rendered `(lam (x e^s) e)`. The notation for conditionals drops the **then** and **else** keywords. The splice operation $\sim e^s$ is rendered `(splice e^s)`, and binary operations are expressed using prefix notation.

Superscripts are replaced with carets; for instance, the object language symbol e^o is rendered `e^o`.

The meta language is represented as follows, and corresponds to the specification on Page 66.

```
;; e^s - metalanguage
(e^s x (lam (x t^s) e^s) (e^s e^s) (fix (x t^s) e^s)
     (tlam a e^s)
     (tapp e^s t^a)
     c f
     (if e^s e^s e^s)
     cs
     (code e)
     (csp e^s)
     (-> e^s e^s)
     (=t e^s e^s)
     (bop e^s e^s))

;; v^s - metalanguage values
(v^s c f cs- (code e^o) t (lam (x t^s) e^s) (tlam a e^s))
(v^s+ v^s x (fix (x t^s) e^s))
```

Expressions $\prec e \succ$, which create object language code, are rendered `(code e)`. Type abstractions $\Lambda\alpha.e^s$ are rendered `(tlam a e^s)` and type applications $e^s[\tau^\alpha]$ are rendered `(tapp e^s t^a)`.

To enable writing interesting recursive programs, the model adds a recursive fixpoint expression `(fix (x t^s) e^s)` to the metalanguage. This syntax is equivalent to **fix** $(\lambda x : \tau^s.e^s)$.

The symbol τ for types is replaced with the character `t`. Thus, the various type formation rules are represented as follows.

```
(t g (-> t t))
(t^a a g (-> t^a t^a))
(t^s a g code type (-> t^s t^s) (forall (a) t^s))
```

Finally, since evaluation can abort in the case of a failed call to **typeof** , the language specifies an equivalent to $\bot$.

```
;; abort expression
(bottom bottom)
```

The kernel language was specified in Chapter 4 using inside-out contexts. Its contexts are built by progressively filling in the hole of a context with expressions that have holes. Since PLT Redex does not support this style of context definition, the evaluation contexts had to be modified from the specification in Figure 2 for implementation.

```
(E^s hole (E^s e^s) (v^s E^s) (code E)
     (if E^s e^s e^s) (csp E^s) (-> E^s e^s) (-> t E^s)
     (=t E^s e^s) (=t v^s E^s) (tapp E^s t^a)
     (bop E^s e^s) (bop v^s E^s))

(E (hole spl) (E e) (e^o E) (lam (x E^s) e) (lam (x t) E)
   (splice E^s) (if E e e) (if e^o E e) (if e^o e^o E)
   (bop E e) (bop e^o E))
```

In this model, a context `E^s` represents a metalanguage term e^s with a hole in it and a context `E` represents a code-language term e with a hole in it. In contrast, the kernel language specification presents only contexts that represent code-language terms with holes. This PLT Redex model's evaluation contexts achieves the same results as the specification by differentiating holes that are plugged with code-language terms from holes that are plugged with meta language holes. In particular, named holes `(hole spl)` are filled with code-language expressions, whereas unnamed holes `hole` are filled with metalanguage expressions. The reduction rules utilize these two kinds of holes to express which expressions they are applicable to.

In PLT Redex, each notion of reduction $\rightarrow_{E^s}$ from the metalanguage becomes its own context-sensitive reduction step because the system does not directly support defining a generalized reduction rule like (meta-eval) from Page 2.1. Consider a typical metalanguage reduction rule, particularly the addition rule.

```
(--> (in-hole E_1 (+ number_1 number_2))
     (in-hole E_1 ,(+ (term number_1) (term number_2)))
     "+")
```

The reduction rules are specified as a list including the arrow `-->`, a context-sensitive pattern to match, the result, a title string, and in some cases a side condition for the rule. In this syntax, the pattern `(in-hole E_1 (+ number_1 number_2))` is equivalent to the specification

$E[n_1 + n_2]$. This reduction rule, like all the other metalanguage reduction rules, relies on an unnamed hole in its pattern matching. In contrast, the splice rule utilizes the named hole to ensure that the expression is detected in a context that is plugged with a code-language term.

```
(--> (in-named-hole spl E_1 (splice (code e^o_1)))
     (in-hole E_1 e^o_1)
     "splice")
```

Since the named hole is part of the E contexts, it clearly indicates a hole waiting for a code-language expression.

The rule for function application takes advantage of a substitution function `mesubst`.

```
(--> (in-hole E_1 ((lam (x_1 t^s_1) e^s_1) v^s_1))
     (in-hole E_1 (mesubst (x_1 v^s_1 e^s_1)))
     "lam")
```

The function `mesubst` directly implements the specification of $e^s\,[e^s/x]^s$, capture-avoiding substitution of metalanguage expressions for metalanguage variables in metalanguage terms. Its implementation is mutually recursive with the `mcsubst` function $e\,[e^s/x]^s$, which is its code-language counterpart. The formal specification of both functions can be found in Appendix C.

The reduction rule for `fix` substitutes itself for any instance of its bound variable within its body.

```
(--> (in-hole E_1 (fix (x_1 t^s_1) e^s_1))
     (in-hole E_1 (mesubst (x_1 (fix (x_1 t^s_1) e^s_1) e^s_1)))
     "fix")
```

The `typeof` operator is implemented using a function `compute-type-of`, which inspects the current evaluation context to produce a typing context.

```
(--> (in-hole E_1 (typeof (code e^o_1)))
     (in-hole E_1 (compute-type-of (E_1 (typeof (code e^o_1)))))
     "typeof"
     (side-condition
      (test-match meta-k t_1
        (term (compute-type-of (E_1 (typeof (code e^o_1))))))))
```

```
(--> (in-hole E_1 (typeof (code e^o_1)))
     bottom
     "notype"
     (side-condition
      (eq? (term (compute-type-of (E_1 (typeof (code e^o_1)))))
           (term bottom))))
```

If the code expression is not well-typed, the `compute-type-of` function returns `bottom`, which then becomes the result of the entire program evaluation.

3. Examples

To evaluate a program, the redex model implements a function `k-red`, which given a kernel metaprogram evaluates it to completion. To illustrate this in action, consider some example programs from Chapter 4, each juxtaposed with its equivalent Redex program.

Consider first the program from Page 86, which statically computes an exponent and persists the result into the final program. Evaluation of this program produces a constant value, such that no computation is performed at run-time.

```
> (k-red
 '(splice ((lam (pow (-> int (-> int int)))
                (csp ((pow 5) 7)))
           (fix (p (-> int (-> int int)))
                (lam (m int)
                     (lam (n int)
                          (if (zero? n)
                              1
                              (* m ((p m) (sub1 n))))))))))
78125
```

The next example, from Page 86, uses a statically determined exponent to custom-generate an exponentiation function. Evaluating this metaprogram produces a custom-tailored implementation of exponentiation.

```
> (k-red
 '(splice ((lam (pow (-> code (-> int code)))
                (code (lam (m int) (splice ((pow (code m)) 7)))))
           (fix (p (-> code (-> int code)))
                (lam (m code)
                     (lam (n int)
                          (if (zero? n)
```

```
                              (code 1)
                              (code (* (splice m)
                                       (splice ((p m) (sub1 n))))))))))))
(lam (m4 int) (* m4 (* m4 (* m4 (* m4 (* m4 (* m4 (* m4 1))))))))
```

The final example, from Page 86, produces the number of curried arguments to a particular function type.

```
> (k-red
 '(splice ((lam (numargs (-> type code))
                (csp (numargs (-> int (-> bool int)))))
           (fix (n (-> type code))
                (lam (t type)
                     (if (->? t)
                         (add1 (n (cod t)))
                         0))))))
2
```

CHAPTER 6

Extending the Metaprogramming Language

The metaprogramming language of Chapter 4 provides functionality for expressing static metaprograms, and precisely specifies both its static and dynamic semantics, but its constructs can be inconvenient for some common operations. In particular, some metaprogramming idioms used in languages like C++ are awkward to express in the kernel language.

In this chapter, a *surface* language for metaprogramming is defined. The surface language provides more expressive language features on top of the kernel metaprogramming language. It introduces new constructs and simple coercions that make some programming idioms more convenient and give the metaprogramming language a more stylized presentation.

The semantics of the surface language is formalized as a static type system that induces a type-directed translation to the kernel language, the same technique used in Chapter 4 to add **let** binding to the object language. Just as the **let** translation makes necessary use of type information, the surface language uses the information acquired from its type checking rules to determine how programs translate to the kernel language. As such, the meaning of programs is type-dependent. To establish soundness of the surface language, type-directed transformation is proven to preserve well-typing.

1. Motivation for a Surface Metaprogramming Language

The metaprogramming language from the last chapter supports many of the metaprogramming capabilities discussed earlier. It supports performing general computations prior to run-time and the use of those computations in programs; it supports the programmatic construction and combination of programs and subexpressions, in essence the ability to treat programs as data; it supports performing computations over the types of programs,

deconstructing them into parts, comparing them for equality, as well as deducing and using the types of programs and subexpressions.

However since the kernel metaprogramming language captures the essence of a model of metaprogramming, its features exhibit a minimalist, though comprehensive, model for metaprogramming that is short on usability conveniences.

For instance, quoting and unquoting is quite natural for a run-time metaprogramming language, but it is not as well-suited to a static metaprogramming language. A run-time metaprogramming language, like Lisp with its `eval` mechanism, is generally constructed under the assumption that the meta-level, which manipulates programs as data, is the language level at which most programs are written. Adding `eval` to the language makes the standard interface to the language into a metalanguage. This feature is very powerful, but most Lisp programs do not make extensive use of it. Building programs as data and then running them is a well-supported but special-case use of Lisp. Thus run-time metaprogramming capabilities augment the existing run-time language.

In contrast, a static metaprogramming language augments the language that it extends such that the existing language becomes an object language, not a metalanguage. As such, the language for programs as data is the level at which most program logic is written. This change of priorities suggests that a programming language with support for static metaprogramming might be best organized if the metaprogramming features are integrated in a way that localizes metaprogramming and gives preference to the code language. Consider the design of C++. Its template language extends the previously existing run-time language while preserving the general structure of programs. Templates are a compile-time mechanism that builds on the existing run-time language. Extending an existing language with minimally intrusive features seems to be a wise design philosophy for static metaprogramming features.

This chapter introduces extensions to the kernel metaprogramming language that give preference to the code language in the style described above. Treating the kernel language as a lower-level target language, the *surface language* introduces constructs that complement the basic metaprogramming model with more expressive features. These constructs utilize

type information to simplify the presentation of metaprograms, resulting in a less invasive interface to metaprogramming functionality.

2. Syntax

The surface language has a similar structure to the kernel language. Its definition is also split into a code language and a metalanguage, each of which is defined in terms of the other. The syntax of the surface code language is close in design to the kernel code language.

$$
\begin{array}{lcll}
x & ::= & \langle \text{variable} \rangle & \\
c & ::= & \langle \text{value constant} \rangle & \\
f & ::= & \langle \text{function constant} \rangle & \\
e & ::= & x \mid \lambda x : e^s.e \mid e\ e & \text{(code language)} \\
 & \mid & \textbf{let } x = e \textbf{ in } e & \\
 & \mid & \textbf{let meta } x = e^s \textbf{ in } e & \\
 & \mid & \textbf{if } e \textbf{ then } e \textbf{ else } e & \\
 & \mid & c \mid f & \\
 & \mid & e^s[e^s] &
\end{array}
$$

In fact, the surface code language is almost a pure extension. It includes the same variables, function abstractions, applications, conditionals, and constants from the kernel language. It adds three new forms: two forms of **let** binding and a new form of application, $e^s[e^s]$. The surface language omits the splice form $\sim e^s$, but as shown below, its absence causes no loss of expressive power.

The syntax of the metalanguage is also similar to its kernel language counterpart.

$$
\begin{array}{lcll}
\alpha & ::= & \langle \text{type variable} \rangle & \\
\gamma & ::= & \langle \text{type constant} \rangle & \\
c^s & ::= & \gamma \mid \gamma^? \mid \rightarrow^? \mid \textbf{dom} \mid \textbf{cod} \mid \textbf{typeof} & \text{(metalanguage constants)} \\
e^s & ::= & x \mid \alpha \mid \lambda x : \tau^s.e^s \mid e^s\ e^s & \text{(metalanguage)} \\
 & \mid & \textbf{let } x = e^s \textbf{ in } e^s & \\
 & \mid & \textbf{if } e^s \textbf{ then } e^s \textbf{ else } e^s & \\
 & \mid & c \mid f \mid c^s & \\
 & \mid & e^s =_\tau e^s \mid \prec e \succ \mid e^s \rightarrow e^s & \\
 & \mid & \textbf{fgen } [\overline{x_i}](x : \textbf{code } \tau^x).e^s & \\
 & \mid & \textbf{fgen } [\overline{\alpha_i}](x : \textbf{meta } \tau^\alpha).e^s &
\end{array}
$$

The surface metalanguage also adds a form of **let** binding, as well as two new forms of functional abstraction, marked with the **fgen** keyword. Absent from this language is the syntax for type abstraction $\Lambda\alpha.e^s$ and type application $e^s[\tau^\alpha]$. These features are omitted for simplicity, but as discussed below, much of their functionality is still present in this language.

As with the kernel language, the surface code language and metalanguage are defined by mutual induction: the code language e contains the metalanguage e^s and vice-versa. In particular, the syntax for code-language expressions e includes the metalanguage e^s in terms with the form $e^s[e^s]$ and **let meta** $x = e^s$ **in** e as well as in the type annotations of function abstractions $\lambda x : e^s.e$; the syntax for metalanguage expressions e^s includes code-language terms e as the body of code construction expressions $\prec e \succ$.

3. Translational Semantics

In Chapter 4, the semantics of the kernel language are expressed in terms of a reduction semantics that transforms metaprograms into pure object programs and a type system that guarantees that well-typed programs will not get stuck. The surface language builds upon this language definition. In particular, its features are defined in terms of how they correspond to terms of the kernel language. This chapter presents the surface language semantics as a type-directed translation from surface language expressions to kernel language expressions. The technique was introduced in the last chapter, where object-language **let** expressions are defined by translation into direct function applications. The surface language translation system is embodied in two mutually recursively defined relations, $\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks}$ and $\Gamma \vdash e \rightsquigarrow e^k$. The first relation indicates that a surface metalanguage term e^s is well-typed, having the surface language type τ^s, and translates to the kernel metalanguage term e^{ks}; the second relation indicates that a surface code-language term e is well-formed and translates to the kernel code-language term e^k. A type-directed translation system of this sort doubles as a type system: discarding the final term of each translation relation yields a standard type system.

Throughout this chapter, relations, terms, and types of the kernel language are differentiated from the same components of the surface language by adding a superscript k. For instance, the well-typedness relation of the kernel metalanguage is written $\Gamma^k \vdash^k e^{ks} : \tau^{ks}$.

3.1. Direct Translation Cases. The basic and function constants of the surface metalanguage and code language have the same meanings as they do in the kernel language.

$$\frac{\mathit{type}(c) = \gamma}{\Gamma \vdash c : \gamma \rightsquigarrow c} \qquad \frac{\mathit{type}(f) = \gamma_1 \to \gamma_2}{\Gamma \vdash f : \gamma_1 \to \gamma_2 \rightsquigarrow f}$$

The translation rules for constants use the same *type* function as the kernel language.

The metalanguage-specific constants also have trivial translations and the same type assignments as in the kernel language.

$$\frac{}{\Gamma \vdash \gamma : \mathbf{type} \rightsquigarrow \gamma} \qquad \frac{}{\Gamma \vdash \mathbf{dom} : (\mathbf{type} \to \mathbf{type}) \rightsquigarrow \mathbf{dom}}$$

$$\frac{}{\Gamma \vdash \mathbf{typeof} : (\mathbf{code} \to \mathbf{type}) \rightsquigarrow \mathbf{typeof}} \qquad \frac{}{\Gamma \vdash \mathbf{cod} : (\mathbf{type} \to \mathbf{type}) \rightsquigarrow \mathbf{cod}}$$

$$\frac{}{\Gamma \vdash \to^? : (\mathbf{type} \to \mathbf{bool}) \rightsquigarrow \to^?} \qquad \frac{}{\Gamma \vdash \gamma^? : (\mathbf{type} \to \mathbf{bool}) \rightsquigarrow \gamma^?}$$

Metalanguage term and type variable references are typed using the same rules as in the kernel language. Similarly, code-language references to variables marked **dyn** in the type context have equivalent type rules as the kernel language.

$$\frac{x : \tau^s \in \Gamma}{\Gamma \vdash x : \tau^s \rightsquigarrow x} \qquad \frac{\alpha : * \in \Gamma}{\Gamma \vdash \alpha : \mathbf{type} \rightsquigarrow \alpha} \qquad \frac{x : \mathbf{dyn} \in \Gamma}{\Gamma \vdash x \rightsquigarrow x}$$

The translation rules given in Figure 1 are defined to be simple congruences. Each of these type rule has the same structure as the analogous rule for the kernel language and the surface language term that it addresses translates to a term fully defined by the translations of its subterms. For instance, the rule for meta-language **if** requires that the predicate position have **bool** type and that both the consequence and alternative have the same type. The translation of such a term is a kernel language **if** term where each of the three subexpressions is replaced with its translation. Surface language translation rules are compositional in this manner when the surface language expression has the same meaning as the corresponding syntax in the kernel language.

$$\frac{\Gamma \vdash e_1^s : (\tau_1^s \to \tau_2^s) \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \tau_1^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s\ e_2^s : \tau_2^s \rightsquigarrow e_1^{ks}\ e_2^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{type} \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \mathbf{type} \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s \to e_2^s : \mathbf{type} \rightsquigarrow e_1^{ks} \to e_2^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{type} \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \mathbf{type} \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s =_\tau e_2^s : \mathbf{bool} \rightsquigarrow e_1^{ks} =_\tau e_2^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{bool} \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks} \qquad \Gamma \vdash e_3^s : \tau^s \rightsquigarrow e_3^{ks}}{\Gamma \vdash \mathbf{if}\ e_1^s\ \mathbf{then}\ e_2^s\ \mathbf{else}\ e_3^s : \tau^s \rightsquigarrow \mathbf{if}\ e_1^{ks}\ \mathbf{then}\ e_2^{ks}\ \mathbf{else}\ e_3^{ks}}$$

$$\frac{\Gamma \vdash e \rightsquigarrow e^k}{\Gamma \vdash \prec e \succ : \mathbf{code} \rightsquigarrow \prec e^k \succ} \qquad \frac{\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks} \qquad \Gamma, x : \tau_1^s \vdash e^s : \tau_2^s \rightsquigarrow e^{ks}}{\Gamma \vdash \lambda x : \tau_1^s . e^s : \tau_1^s \to \tau_2^s \rightsquigarrow \lambda x : \tau_1^{ks} . e^{ks}}$$

$$\frac{\Gamma \vdash e^s : \mathbf{type} \rightsquigarrow e^{ks} \qquad \Gamma, x : \mathbf{dyn} \vdash e \rightsquigarrow e^k}{\Gamma \vdash \lambda x : e^s . e \rightsquigarrow \lambda x : e^{ks} . e^k} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma \vdash e_2 \rightsquigarrow e_2^k}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1^k\ e_2^k}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma \vdash e_2 \rightsquigarrow e_2^k \qquad \Gamma \vdash e_3 \rightsquigarrow e_3^k}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \mathbf{if}\ e_1^k\ \mathbf{then}\ e_2^k\ \mathbf{else}\ e_3^k}$$

FIGURE 1. Congruent Translation Rules

3.2. Code-Language Metavariable Binding. Metaprograms often define metalanguage expressions, bind them to variables, and use them throughout a computation. In C++ this is done by defining templates; in Scheme it is done by defining or lexically binding a macro. The kernel language does not have an explicit mechanism for binding a meta-level variable to a meta-level value, but it provides enough expressiveness to achieve that effect. Kernel metalanguage and code-language variable bindings obey a lexical scope discipline. The type system does not allow metalanguage variables to be referenced in the code language, and vice versa, but a code-language expression can refer to any code-language variable abstraction that encloses it, even if that abstraction escapes to the metalanguage between the point of variable binding and the point of variable reference. For instance, the expression:

$$\lambda x : \mathbf{int}.\ \sim((\lambda y : \mathbf{int} \prec x \succ)\ 7)$$

refers to the code-language bound variable x inside a metalanguage function that contains a code object expression; This is perfectly legal. Similarly, metalanguage expressions can refer to any metalanguage-bound variables, regardless of intervening splices and code expressions.

A metalanguage variable can be bound to a value for the scope of a code-language expression using a lambda application. Recall that in the object language, a **let** binding form can be simulated using an immediate application of a typed function to an expression. In the metaprogramming language, a similar idiom occurs. However this idiom binds a variable of the metalanguage so that it can be used later in the scope of a code-language expression. A number of examples at the end of Chapter 4 have the following general structure:

$$\sim((\lambda x : \mathbf{int}.\ \prec \ldots \succ)\ e^s$$

This code-language construction performs a disciplined procedure: it escapes to the metalanguage, evaluates the value of e^s, binds it to the metalanguage variable x, and reverts to the code language to evaluate the expression $\prec \ldots \succ$. This code structure resembles the **let** simulation, except that it operates across multiple language levels. The variable x is bound in the metalanguage, and is in scope for the code-language expression represented by the ellipses, $\prec \ldots \succ$.

The surface language captures this common idiom with a language form: its code language adds a new form of declaration: **let meta**.

$$e ::= \ldots \mid \mathbf{let\ meta}\ x = e^s\ \mathbf{in}\ e$$

The **let meta** form enables a metalanguage identifier to be defined from within the code language and sets the scope of that definition as the nested code-language expression. Just as **let** enables the declaration of run-time variables bound to run-time expressions, **let meta** enables the declaration of metalanguage variables bound to metalanguage expressions. As with the **let** expression, the body of a **let meta** expression is the scope of the variable it binds. The **let meta** expression is a code-language expression and its body, the scope of the variable binding, is a code-language expression. As such, this form is a code-language mechanism for defining metalanguage variables. It binds lexically scoped metalanguage

variables to compile-time expressions. The example code idiom above can be rewritten simply using this new form.

$$\textbf{let meta } x = e^s \textbf{ in } \ldots$$

The **let meta** syntax implicitly switches from the code language to the metalanguage and back. The translation rule for **let meta** captures the kernel language idiom directly.

$$\frac{\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks} \qquad \Gamma, x : \tau^s \vdash e \rightsquigarrow e^k \qquad \Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}}{\Gamma \vdash \textbf{let meta } x = e^s \textbf{ in } e \rightsquigarrow \sim((\lambda x : \tau^{ks}.\ \prec e^k \succ)\ e^{ks})}$$

The structure of this rule matches the rule for **let** from the object language, although the body of the form uses the code-language translation rule.

3.3. Implicit Phase Distinction. The kernel language provides the ability to explicitly embed metalanguage values into code-language objects using the CSP operator $\%e^s$, as well as the ability to integrate those code objects into larger code-language expressions using the splice operator, $\sim e^s$. While this machinery clearly distinguishes phases of computation and the operations that connect the metalanguage and code language, it often seems to impose a verbosity that clutters metaprograms.

For instance, consider how the kernel language treats variable references. The kernel language scoping rules determine whether a variable was bound in the metalanguage or the code language. However, even though the two language levels share the same variables, introducing the result of a metalanguage computation to the code language requires an explicit escape to the metalanguage. If a kernel metalanguage variable is bound to a code-language code object, then the only sensible role it can play in the code language is to be the object of a splice operation. Nonetheless, the kernel language requires that the variable be spliced explicitly, as in the following metalanguage function.

$$e^s = (\lambda x : \textbf{code}.\ \prec 4 + \sim x >)$$

In the kernel language, it would be a type error to refer to the variable x directly in the code language because it is a variable of the metalanguage.

Bear in mind that this restriction on the kernel language is justified. The kernel language dynamic semantics use a pair of capture-avoiding substitution functions $e[v^s/x]$ and $e^s[v^s/x]^s$ to substitute a metalanguage value for free metalanguage references to the variable x in the body of a code-language term e or a metalanguage term e^s respectively. The two functions are defined by mutual recursion because each language contains subterms belonging to the other language. The code-language substitution function leaves code-language variable references unchanged ($x[v^s/x] = x$) while the metalanguage substitution function performs substitution ($x[v^s/x]^s = v^s$). This specification guarantees that the substitution operation is well-defined: substitution into a code-language term always yields a code-language term and substitution into a metalanguage term always yields a metalanguage term. A substitution function that operates uniformly across both language levels could substitute metalanguage-specific values—for instance the **dom** function constant—into code-language expressions, yielding results that do not even syntactically conform to the structure of metaprograms.

Because the kernel language is defined by type-directed translation, it is possible to account for type information in its semantics. In particular, some references to metalanguage variables from the code language can be given well-defined meanings. The code language treats references to variables of type **code** in its code language as implicit splice operations.

$$\frac{x : \mathbf{code} \in \Gamma}{\Gamma \vdash x \rightsquigarrow \sim x}$$

Furthermore, since values of primitive type can be persisted into the kernel code language by using the CSP operator, variables of primitive type are implicitly persisted and spliced into code-language contexts.

$$\frac{x : \gamma \in \Gamma}{\Gamma \vdash x \rightsquigarrow \sim \%x}$$

This implicit phase distinction, automatically introducing metalanguage computations into the code language, is used in several places in the surface language. To present it uniformly, a subset of surface language types is distinguished as the spliceable types.

$$\boxed{\tau^{\sim} \quad ::= \quad \gamma \mid \mathbf{code}}$$

A translation-time function $[\![\tau^\sim]\!]$ determines how an expression of a particular type must be "escaped" in the kernel language in order to make the phase distinction explicit.

$$[\![\tau^\sim]\!] = \begin{cases} \sim & \text{if } \tau^\sim = \textbf{code} \\ \sim\% & \text{if } \tau^\sim = \gamma \end{cases}$$

Using these definitions, the rules for variable references can be combined into a single rule.

$$\frac{x : \tau^\sim \in \Gamma}{\Gamma \vdash x \rightsquigarrow [\![\tau^\sim]\!]x}$$

This rule does not clash with the standard code variable reference rule since $x : \tau^\sim$ does not overlap with x : **dyn**. This rule does not allow references to metalanguage variables that do not have spliceable types. Thus, the meaning of code-language variable references is type-dependent in an intuitive way.

One side-effect of support for implicit phase distinction is that the kernel language splice operator $\sim e^s$ is not needed in the surface language. Any use of $\sim e^s$ can be replaced with **let meta** $x = e^s$ **in** x.

3.4. Direct Metalanguage Function Calls. Implicit phase distinction combined with the **let meta** binding form enables programs to escape to the metalanguage to perform computations and to embed the results of those computations into the code language. However, these forms currently give preference to embedding values that have already been computed. In order to embed the results of function calls, they must currently be given names and subsequently referenced. Since the metalanguage and code language share the same set of variable references, no syntactic change is necessary to the language in order to add metalanguage variable references to the code language. With function applications, this is not the case. Metalanguage anonymous functions extend the language with new types, expressions, and forms, that are not compatible with the syntax. In order to embed metalanguage function calls into the code language, Syntactic support is needed.

By providing a different syntax for direct metalanguage function applications, it becomes straightforward to write expressions that use metalanguage functions from the code

language. The surface language provides the syntax $e^s[e^s]$ for applying metalanguage functions. Since this syntax differs from that of a normal function application, the two metalanguage subexpressions can be recognized as an application of a metalanguage function to a metalanguage argument and transform it in the obvious manner.

$$\frac{\Gamma \vdash e_1^s : \tau_1^s \to \tau^\sim \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \tau_1^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s[e_2^s] \rightsquigarrow [\![\tau^\sim]\!](e_1^{ks}\ e_2^{ks})}$$

To ensure compatibility with the implicit phase distinction rules of the surface language, the operator must return a spliceable type. This guarantees that the operation has well-defined semantics in the kernel language.

3.5. Functional Generators. For software engineering purposes, it sometimes makes sense to build abstractions that do not telegraph their underlying implementation. For instance, many Lisp macros have interfaces that look like normal function calls, even as they perform syntactic manipulations in their implementations. The surface language extensions introduced so far go a long way towards building a seamless interface between metalanguage and code language. Nonetheless, metalanguage function calls are always obvious because of their special syntax. However, by taking advantage of type information, the surface language can introduce a new kind of expression that looks like a normal function call but hides metalanguage operations.

The surface language introduces *functional generators*, modeled after C++ function templates, as another form of static abstraction. They are specifically meant to provide unobtrusive interfaces to metalanguage code from the code language. In particular, the syntax for applying a functional generator to an argument is the same as the standard object-language syntax for function application, with the restriction that the operator position is always a metalanguage variable reference of a particular type. Distinguishing a functional generator call from a traditional function application requires contextual information from the surface language. In a sense, functional generator calls resemble macro calls from languages like Lisp and Scheme.

In short, functional generators ascertain the types of their arguments and make them available to the metalanguage expression that defines their bodies. Because of their syntactic connection to standard code-language applications, they can hide metaprogramming machinery behind an interface that looks like a normal function call.

Functional generators are functional abstractions in that they define parameterized expressions that expect arguments and yield values that depend on them. There are two kinds of functional generators, and each kind treats its arguments differently. Although the two variants broadly perform the same tasks, their semantics differ in ways that are forced by the rest of the language semantics.

The two kinds of functional generators are differentiated by their syntax, especially the qualifying annotation associated with their arguments. If the parameter of a functional generator is marked **meta**, then the generator expects a static value of the object language, computable at compile-time, meaning it must be a constant or the result of a metalanguage expression.

$$e^s ::= \ldots \mid \textbf{fgen}\ [\overline{\alpha_i}](x : \textbf{meta}\ \tau^\alpha).e^s$$

If a parameter of a functional generator is marked **code**, then the argument must be an expression in the code language. That piece of code will be passed as an argument to the generator body during metacomputation.

$$e^s ::= \ldots \mid \textbf{fgen}\ [\overline{x_i}](x : \textbf{code}\ \tau^x).e^s$$

The two kinds of functional generators are called *metagenerators* and *code generators*, respectively.

Both kinds of functional generator can manipulate type information gleaned from their arguments. Each functional generator abstracts a set of either term variables $[\overline{x_i}]$ for code generators, or type variables $[\overline{\alpha_i}]$ for metagenerators. The argument position of a code generator or a metagenerator is annotated with a type pattern τ^x or a type schema τ^α respectively. A type pattern is an object-language type that is parameterized with metalanguage term variables; a type schema is an object-language type that is parameterized

with metalanguage type variables. The pattern and qualifier annotate the generator's parameter variable. The body of a functional generator is a metalanguage expression that can use its type parameters as well as its argument parameter. In this manner, functional generators reflect on the types of the arguments to which they are applied. This behavior is analogous to how function templates perform type deduction in C++.

The scope of metagenerator and code generator type parameters differ from each other. Since a metagenerator's type is parameterized over type variables α, then just as in the kernel language, those variables can be used as types in metalanguage type annotations as well as in metalanguage expressions. On the other hand, the type parameters of code generators are metalanguage term variables x, so they can only appear in contexts where metalanguage expressions of type **type** are valid. They cannot be used in the type annotations of metalanguage expressions.

Corresponding to the code generators and metagenerators are generator type.

$$\begin{array}{rcl} \tau^s & ::= & \ldots \mid [\overline{x_i}](\mathbf{code}\ \tau^x) \Rightarrow \tau^\sim \\ & \mid & [\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \Rightarrow \tau^\sim \end{array}$$

These types provide information used to translate applications of functional generators into kernel language terms. Thus, the type system differentiates functional generators from normal metalanguage functions. Functional generators can only return spliceable types $\tau^\sim$ because they can only be used from the code language, so their results must then be compatible with the implicit phase distinction. They are specifically designed to implement interfaces between the code language and the metalanguage.

The translation rule for a code generator expression shows that code generators encode substantial logic.

$$\frac{\overline{x_i} \vdash \tau^x\ \mathbf{wf} \qquad \overline{e_i^{ks} = x_i\, cpat(\tau^x, x_\tau)} \qquad x_\tau \notin FV(e^s) \qquad \Gamma, \overline{x_i : \mathbf{type}}, x : \mathbf{code} \vdash e^s : \tau^\sim \rightsquigarrow e^{ks}}{\begin{array}{l} \Gamma \vdash \mathbf{fgen}\ [\overline{x_i}](x : \mathbf{code}\ \tau^x).e^s : [\overline{x_i}](\mathbf{code}\ \tau^x) \to \tau^\sim \\ \quad \rightsquigarrow \lambda x : \mathbf{code}. \\ \qquad (\lambda x_\tau : \mathbf{type}. \\ \qquad\ \overline{((\lambda x_i : \mathbf{type}.\ e^{ks}\ )\ e_i^{ks})}) \\ \qquad \mathbf{typeof}\ x) \end{array}}$$

For a code generator expression to be well-typed and translatable to the kernel language, its type parameter τ^x must be well-formed relative to the set of parameter variables x_i: each variable must appear once and only once in the type parameter. The translation relies on a function *cpat*, which generates for each parameter variable x_i a metalanguage expression e_i^{ks} that corresponds to querying a metalanguage type value x_τ for the type corresponding to its position in τ^x. For instance, given the type parameter $x_1 \to (x_2 \to x_3)$, the corresponding expressions are $e_1^{ks} = \textbf{dom}\ x_\tau$, $e_2^{ks} = \textbf{dom}\ (\textbf{cod}\ x_\tau)$ and $e^{ks_3} = \textbf{cod}\ (\textbf{cod}\ x_\tau)$ (a more complete definition checks for arrow types and signals errors as needed). Furthermore, the body of the generator must be well-typed in a typing context that binds each variable x_i to type **type** and binds the variable x to type **code**. The body e^s must translate to some kernel language expression e^{ks}.

The entire code generator translates to the kernel language as follows. The resulting expression is a function that takes an argument x of type **code**, and immediately ascertains its type using the **typeof** operation. The type of x is bound to a variable x_τ, which is used to decompose the type of x in accordance with the pattern τ^x. This is done by binding each variable x_i to the results of the expression e_i^{ks} that accesses the relevant portion of the type x_τ. Finally, in the context of the variable x and the variables x_i, the translated body of the generator e^{ks} is evaluated.

By comparison, the translation rule for a metagenerator definition is simple.

$$\frac{\overline{\alpha_i} \vdash \tau^\alpha\ \textbf{wf} \qquad \Gamma, \overline{\alpha_i : *}, x : \tau^\alpha \vdash e^s : \tau^\sim \rightsquigarrow e^{ks}}{\begin{array}{l}\Gamma \vdash \textbf{fgen}\ [\overline{\alpha_i}](x : \textbf{meta}\ \tau^\alpha).e^s : [\overline{\alpha_i}](\textbf{meta}\ \tau^\alpha) \to \tau^\sim \\ \quad \rightsquigarrow \overline{\Lambda\alpha_i}.\lambda x : \tau^\alpha.e^{ks}\end{array}}$$

The translation binds the type variable parameters α_i using the kernel type abstraction form $\Lambda\alpha.e^s$, and then abstracts the generator argument x as having the given type pattern type. As it turns out, the heavy lifting for metagenerators happens when they are used.

Functional generators are expressions of the metalanguage, but they are used exclusively in the code language. When a functional generator is bound to a metalanguage variable, that variable can be used as the operator of an application expression in the code language.

This functionality is analogous to how macros in Lisp are bound to identifier names and recognized when they are used as the operator of an expression. The surface code language can readily detect when an application in the code language is a functional generator call, rather than a traditional code-language application expression: the operator position of the application must be a variable and the variable must have a functional generator type. Were this expression a normal application, the variable would be bound to the **dyn** marker for code-language-bound expressions. If the operator position is not a variable reference, then the application cannot be a functional generator call.

The translation rule for code generator applications is very simple.

$$\frac{\Gamma \vdash x : [\overline{x_i}](\mathbf{code}\ \tau^x) \to \tau^\sim \rightsquigarrow x \qquad \Gamma \vdash e \rightsquigarrow e^k}{\Gamma \vdash x\ e \rightsquigarrow [\![\tau^\sim]\!](x \prec e^k \succ)}$$

As discussed above, an application whose operator is a variable with code generator type is a code generator call. A code generator application is well formed if the argument to the generator is a well-formed code-language expression. The application translates into an expression that escapes to the metalanguage and passes the argument, transformed into a code expression, to the generator. The translation of the code generator's definition computes the argument's type and decomposes it as needed, so the translation of the application is simple. This simplicity is not possible for the metagenerators because the kernel language does not support decomposing metalanguage types.

The translation rule for applications of metagenerators to arguments is more involved than for code generators.

$$\frac{\begin{array}{c}\Gamma \vdash x : [\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \to \tau^\sim \rightsquigarrow x \\ \Gamma \vdash e^o : \tau^{\alpha'} \rightsquigarrow e^{ks} \qquad \overline{\tau_i^\alpha = \alpha_i mpat(\tau^\alpha, \tau^{\alpha'})}\end{array}}{\Gamma \vdash x\ e^o \rightsquigarrow [\![\tau^\sim]\!](x[\overline{\tau_i^\alpha}]\ e^{ks})}$$

The type rules for these functional generators require the argument to the generator to fit the syntax of the object language e^o, but be well-typed in the metalanguage. Every object-language expression is a sensible expression of both the code language and the metalanguage. Metagenerators utilize this property to embed metalanguage expressions directly into the code language. Every code-language term e^o is also a metalanguage term, and the type

system of the metalanguage subsumes the type system of the object language, so that the static semantics of this argument are standard.

The above translation rule interprets the argument to a metagenerator as a metalanguage expression rather than as a code-language expression. The rule implicitly elevates its argument to the syntax of the metalanguage and induces well-typed translation. When the argument to the generator is a constant expression, the behavior is as straightforward as possible. However, in the case of a complex expression, all variables referenced in the metagenerator's argument must be bound in the metalanguage. These semantics do not cause any ambiguity because any references to code-language variables results in ill-typed code.

The metagenerator application rule uses an external function *mpat* to decompose the deduced type $\tau^{\alpha'}$ of the argument e^o according to the pattern specified by τ^{α}, yielding a list of type expressions τ_i^{α}. In the translation, these type expressions are applied to the variable representing the functional generator, followed by the translation of the argument. In short, the translation of a metagenerator uses the type information gleaned from type checking to determine the type arguments to the generator. Then the result of evaluating the actual argument as a metalanguage expression is passed as the final argument.

Although metagenerators utilize the parametric polymorphism facilities of the kernel language in their implementation, they do not capture all possible uses of that feature. However, metagenerators conveniently present an expressive analogue to code generators, and give the surface language the benefits ascribed to implicit instantiation in other languages that support genericity. It captures a usage pattern that relates closely to how templates are used in C++.

This description points out a substantial difference between code generators and metagenerators. A code generator translates into metaprogramming code that when executed deduces the type of its argument, destructs that type according to a pattern, and makes that information available to a computation. In contrast, metagenerator applications destruct the type of their argument during the type-directed translation process. Whereas a type

mismatch with a code expression will get caught during execution of a metaprogram, the analogous error in a metagenerator application is detected before metaprogram execution.

The rules for functional generators perform the automatic splicing and CSP based on their return type, so the results of a functional generator call are embedded in the surrounding code-language expression. Furthermore, functional generators give a flavor of first-class macros because they can be passed as values to functions in the metalanguage that subsequently use them.

Functional generators are in essence type-directed abstractions. This is especially the case with the code generators, which take a code argument and decompose the type. The same code as generated by the translation, though tedious, could be written in the kernel language. On the other hand, the type decomposition performed in the application of metagenerators is related to implicit instantiation in languages with support for generics, whose addition is more than cosmetic [**24**] With metagenerators, though there is some concern about scope confusion. If the argument to a metagenerator is simple, like an integer or some such, then all is clear. When it becomes a function, then since the expression must be from the metalanguage an implicit language level phase shift occurs. All variables referenced in the argument to a metagenerator must be metalanguage variables. Because of the phase distinction, however, it is not possible for an expression to change meaning because it goes from being the argument to a typical code expression to being the argument of a metagenerator. The worst that can happen is that a well-phased program becomes ill-phased when an argument to a metagenerator references code-language abstractions.

Chapter 4 discusses the implications of altering the set of types that can be represented by type variables α in the kernel language. Restricting type variables to primitive types enables parametric CSP operations in polymorphic functions. If the parametric part of the kernel type system is restricted to only allow primitive types, then the pattern matching system for metagenerators must be similarly restricted in order to match the kernel language model, but this would enable the bodies of metagenerators to persist values of parametric type.

3.6. Alternative Binding Translation. For convenience, the metalanguage can also be augmented with a traditional **let** form that allows the definition of metalanguage identifiers when operating at the meta level.

$$\frac{\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks} \qquad \Gamma \vdash e_1^s : \tau_1^s \rightsquigarrow e_1^{ks} \qquad \Gamma, x : \tau_1^s \vdash e_2^s : \tau_2^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash \textbf{let } x = e_1^s \textbf{ in } e_2^s : \tau_2^s \rightsquigarrow (\lambda x : \tau_1^{ks}.e_2^{ks})\ e_1^{ks}}$$

This translation exactly matches the structure of the definition of **let** for the object language from the last chapter. It relies on being able to assign a type the bound value e_1^s so that the variable x can be given a type annotation, since it is bound by a function abstraction in the kernel language that specifies the type of x. The type system determines the type that the surface language assigns to e_1^s, but that type must be re-interpreted in the context of the kernel language type system. To do this, the typing rule appeals to a relation $\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks}$, which transforms surface language types to kernel language types. This relation is discussed further in Section 4.

The surface code language also has a **let** form, which serves the same purpose as the form by the same name defined for the object language in Chapter 4. However, since the kernel language does not have such a binding form, the surface language provides it as an additional extension.

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma, x : \textbf{dyn} \vdash e_2 \rightsquigarrow e_2^k}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 \rightsquigarrow (\lambda x : \textbf{typeof } \prec e_1^k \succ .e_2^k)\ e_1^k}$$

Although the metalanguage **let** uses the standard technique, the code-language binding form takes advantage of the capabilities of the metaprogramming language to define **let**. The surface language type system does not compute the types of code-language expressions, but the kernel metalanguage can dynamically ascertain types using the **typeof** form. For this reason, surface code-language **let** forms translate into immediate application of a function, as is typical for this translation, but the type annotation on the variable is a metalanguage expression that computes the type of the bound expression. If the expression e_1^k is ill-typed in the object-language type system, then an error is flagged during execution of the kernel language metaprogram.

4. Metatheory

Whereas the static semantics of the kernel language merely isolates a set of well-behaved programs, The translation system for the surface language defines both its static and dynamic semantics. The well-formed surface language programs are identified with the set of closed code-language expressions e that can be translated in the empty type environment $\vdash e \rightsquigarrow e^k$; the dynamic semantics of those programs is identified with the dynamic semantics of their translations e^k into the kernel language.

The type system of the kernel language is linked to its reduction semantics by a proof of type safety, which establishes guarantees for the behavior of well-typed programs. Because the surface language static semantics links it to the kernel language dynamic semantics, a similar statement of guarantees can be proven.

As with the kernel language type system, the surface language translation system uses type environments and imposes well-formedness conditions on them. However, the well-formedness relation on type environments, like the type system, doubles as a translation relation from surface language type environments to kernel language type environments. The translation is trivial for term variables that are marked dynamic $x : \mathbf{dyn}$ and for type variable bindings $\alpha : *$. Translating metalanguage term variables, however, is more involved.

$$\frac{\Gamma \rightsquigarrow \Gamma^k \qquad \Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}}{\Gamma, x : \tau^s \rightsquigarrow \Gamma^k, x : \tau^{ks}}$$

The translation for metalanguage term variable bindings appeals to a type translation relation $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$ to determine the kernel language equivalent of each surface language type assignment. Most types from the surface language have the same representation in the kernel language as in the surface language, but functional generator types must be translated to types that correspond to the kernel language expressions that implement functional

generators.

$$\frac{\overline{x_i} \vdash \tau^x \ \mathbf{wf} \qquad \Gamma \vdash \tau^{\sim} \rightsquigarrow \tau^{ks}}{\Gamma \vdash [\overline{x_i}](\mathbf{code}\ \tau^x) \Rightarrow \tau^{\sim} \rightsquigarrow \mathbf{code} \to \tau^{ks}}$$

$$\frac{\overline{\alpha_i} \vdash \tau^{\alpha} \ \mathbf{wf} \qquad \Gamma, \overline{\alpha_i : *} \vdash \tau^{\sim} \rightsquigarrow \tau^{ks}}{\Gamma \vdash [\overline{\alpha_i}](\mathbf{meta}\ \tau^{\alpha}) \Rightarrow \tau^{\sim} \rightsquigarrow \overline{\forall \alpha_i}.\tau^{\alpha} \to \tau^{ks}}$$

Code generator types translate into function types that accept code objects and return spliceable values. Metagenerator types, on the other hand, translate into polymorphic types that accept multiple parametric type arguments and a value type and return a spliceable value. Each functional generator type must be well-formed. The expressions $\overline{x_i} \vdash \tau^x$ **wf** and $\overline{\alpha_i} \vdash \tau^{\alpha}$ **wf** guarantee that the type pattern or type schema associated with a functional generator uses each abstracted variable once and only once. Furthermore, the return types of generators must be well-formed.

The relations that translate surface language types, and in turn surface language environments, into kernel language artifacts facilitate a proof that the translation rules given above define a sound metaprogramming language.

THEOREM 7 (Preservation of Well-Typing across Translation).

(1) *If* $\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks}$, $\Gamma \rightsquigarrow \Gamma^k$, *and* $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$, *then* $\Gamma^k \vdash^k e^{ks} : \tau^{ks}$;

(2) *If* $\Gamma \vdash e \rightsquigarrow e^k$, *and* $\Gamma \rightsquigarrow \Gamma^k$, $\Gamma^k \vdash^k e^k$ ***wf***.

The preservation theorem in short says that the translation system of the surface language associates well-formed surface language programs with well-formed kernel language programs. As such, the execution of well-formed surface programs have the same guarantees as those for well-formed kernel language programs.

The semantics of the surface language makes essential use of the kernel language semantics for its implementation. The surface language's dynamic semantics is dependent on its type system. For example, the translation of a functional metagenerator application in the code language differs substantially from the translation of a traditional code-language

application, which means that their dynamic semantics are quite different. That translation depends critically on the types that are assigned to terms by the surface language type system. In fact, an ill-typed program in the surface language has no semantics whatsoever, because only well-typed surface language programs have dynamic semantics, as given by their kernel language translations.

On the other hand, the kernel language dynamic semantics are independent of its type system. The behavior of both typed and untyped kernel language programs is well defined, even if some of those programs are not well-behaved. As such, the translation process from surface to kernel language eradicates the type-dependencies of the surface language forms, lowering programs into a language whose dynamic semantics are independent of its type system, but whose type system provides guarantees about dynamic behavior.

5. Examples

Chapter 4 presents some example programs that utilize the capabilities of the kernel metaprogramming language. Comparing those examples to their surface language equivalents shows how the features of the surface language help support metaprogramming.

To illustrate how the surface language presents a comprehensible model of computation even without viewing programs through the lens of translation to the kernel language informal dynamic semantics are presented for some programs in terms of the surface language syntax (retaining the code-language **let** form for clarity).

The kernel language program

$$\sim((\lambda ctSum : \mathbf{int}.\ \prec(\lambda rtSum : \mathbf{int}.rtSum - \sim\%ctSum)\ (5+3)\succ)\ (5+3))$$

which demonstrates variable binding and computation at both language levels is rendered in the surface language as follows.

$$\begin{array}{c}\mathbf{let\ meta}\ ctSum = 5+3\ \mathbf{in}\\ \mathbf{let}\ rtSum = 5+3\ \mathbf{in}\\ rtSum - ctSum\end{array}$$

This program presents two bindings, one of *ctSum* and one of *rtSum* to the same right-hand-side expressions, and in the context of both computes $rtSum - ctSum$.

Rather than switching from code language to metalanguage and back, the surface language version of this program uses the **let meta** binding form to introduce metalanguage variables. Because *ctSum* is bound using **let meta**, it is a metalanguage variable, whereas *rtSum* is a code-language value because it is bound using **let**. This means that the actual value of *ctSum* can be used in computations during compilation. The example also relies on type-directed implicit phase distinction to introduce the results of metaprogram computations into code-language expressions.

Conceptually, evaluation of this metaprogram proceeds as follows. First, the right-hand side of *ctSum* is evaluated:

$$\begin{array}{c}\textbf{let meta } ctSum = 8 \textbf{ in} \\ \quad \textbf{let } rtSum = 5 + 3 \textbf{ in} \\ \quad\quad rtSum - ctSum\end{array}$$

Then its value is substituted wherever it is referenced

$$\begin{array}{c}\textbf{let meta } ctSum = 8 \textbf{ in} \\ \quad \textbf{let } rtSum = 5 + 3 \textbf{ in} \\ \quad\quad rtSum - 8\end{array}$$

Once computation completes, the **let meta** binding is no longer needed, so the residual program is left behind.

$$\begin{array}{c}\textbf{let } rtSum = 5 + 3 \textbf{ in} \\ \quad rtSum - 8\end{array}$$

These computation steps occur in the kernel language, but a programmer may reason in terms of the surface language syntax.

A variant of the above kernel language program binds a piece of code to a metalanguage variable and then uses a splice alone rather than a splice and CSP combination to introduce the value into the code language.

$$(\lambda rtSum : \textbf{int}.\ {\sim}((\lambda ctSumCode : \textbf{code}.\ {\prec} rtSum {-}\ {\sim} ctSumCode {\succ})\ \ {\prec} 5 + rtSum {\succ}))\ (5 + 3)$$

This program renders in the surface language in a similar manner as the last example.

$$\begin{array}{l}\textbf{let } \mathit{rtSum} = 5 + 3 \textbf{ in} \\ \quad \textbf{let meta } \mathit{ctSumCode} = \prec 5 + \mathit{rtSum} \succ \textbf{ in} \\ \quad\quad \mathit{rtSum} - \mathit{ctSumCode}\end{array}$$

In this case, the two variable bindings are reversed, *ctSumCode* being bound in the scope of *rtSum*. This is needed because *ctSumCode* is bound to a code value that references *rtSum*. The surface language is lexically scoped, just like the kernel language, so variable references are always statically resolved. The body of these expressions is the same as in the previous example. In particular, the reference to *ctSumCode* does not require splicing: type directed translation inserts the needed splice.

Conceptually, this program evaluates to an object-language term that substitutes the code object bound to *ctSumCode* where that variable is referenced.

$$\begin{array}{l}\textbf{let } \mathit{rtSum} = 5 + 3 \textbf{ in} \\ \quad \mathit{rtSum} - (5 + \mathit{rtSum})\end{array}$$

The next variant uses a direct call to a metalanguage function from the object language.

$$\begin{array}{l}\textbf{let } \mathit{rtSum} = 5 + 3 \textbf{ in} \\ \quad \textbf{let meta } \mathit{ctSumGen} = \lambda x : \textbf{int}. \prec x + \mathit{rtSum} \succ \textbf{ in} \\ \quad\quad \mathit{rtSum} - \mathit{ctSumGen}[5]\end{array}$$

In this program, the metalanguage variable *ctSumGen* is bound to a function that accepts a static integer value and returns a piece of code. In the body of the example, the generator is applied to the metalanguage constant 5, and yields the same code as was assigned to *ctSumCode* in the previous example. Neither the reference to x nor the call to *ctSumGen* require escaping.

The next two examples use functional generators in programs similar to the above.

$$\begin{array}{l}\textbf{let } \mathit{rtSum} = 5 + 3 \textbf{ in} \\ \quad \textbf{let meta } \mathit{ctSumA} = \\ \quad\quad\quad \textbf{fgen } [t](m : \textbf{meta } t). \prec m + \mathit{rtSum} \succ \textbf{ in} \\ \quad\quad \mathit{rtSum} - (\mathit{ctSumA}\ (2 + 3))\end{array}$$

This example binds the metalanguage variable *ctSumA* to a metagenerator that takes an integral argument and returns a piece of code. The body of the expression applies *ctSumA* to an expression $2 + 3$, and since the operator is a metagenerator, the operand is treated as an expression of the metalanguage, evaluating it to 5. The argument is then passed to the body of the metagenerator, yielding the code object $\prec 2 + 3 \succ$, which is spliced into place to yield the same result as the previous example.

A variant on the above example uses a code generator instead of a metagenerator.

$$\begin{array}{l}\textbf{let } rtSum = 5 + 3 \textbf{ in} \\ \quad \textbf{let meta } ctSumB = \\ \qquad \textbf{fgen } [t](n : \textbf{code } t).\ \prec 5 + n \succ \ \textbf{ in} \\ \quad rtSum - (ctSumB\ rtSum)\end{array}$$

In this program, the code generator *ctSumB* takes a code argument and produces an analogous piece of code. In this case, though, the body is applied to *rtSum*, which is a code-language variable. Since *ctSumB* is a code generator, it is passed its operand as a code object $\prec rtSum \succ$. The code is spliced into the larger object which is returned to the site of the code generator call and spliced into place.

As in the last chapter, recursive metaprogramming functions can be constructed if a construct for well-typed recursion is added to the language. In this case, a recursive variant of **let meta** is assumed.

The next example implements a compile-time exponentiation function:

$$\begin{array}{l}\textbf{letrec meta } pow : \textbf{int} \to \textbf{int} \to \textbf{int} = \\ \quad \lambda n : \textbf{int}.\lambda m : \textbf{int}. \\ \qquad \textbf{if } zero?\ n \\ \qquad\ \textbf{then } 1 \\ \qquad\ \textbf{else } m * pow\ (sub1\ n)\ m) \textbf{ in} \\ \quad \textbf{let meta } pow5 = pow\ 5 \textbf{ in} \\ \qquad pow5[7]\end{array}$$

A variant of the previous example generates exponentiation expressions.

```
letrec meta pow : int → code → code =
  λn : int.λm : code.
    if zero? n
     then ≺1≻
     else ≺m ∗ (pow (sub1 n))[m]≻ in
  let meta pow5 = pow 5 in
    λm : int.pow5[≺m≻]
```

The next example, also derived from the last chapter, defines a metalanguage function that takes a type and returns the number of curried arguments it takes, zero if the type is not a function:

```
letrec meta numArgs : type → int =
     λt : type.
       if (→? t)
        then 1 + (numArgs (cod t))
        else 0 in
   numArgs[int → bool → int]
```

Finally, the last example recreates the kernel language program that takes two type definitions and computes a new type that could represent the result of adding values of the two types:

```
let meta promote =
     λx : type.λy : type.
       if x =τ y then x
        else if int? x and float? y then float
        else ... in
  (λx : (promote int (typeof 7.7)). ...)(55 + 7.7)
```

CHAPTER 7

Discussion

The following chapter concludes this dissertation. It discusses what has been done and its implications, suggests some directions in which this work can be advanced, and concludes.

1. Metaprogramming as a Language Laboratory

Some of the metaprogramming techniques and mechanisms described in Chapter 3 have contributed substantially to the construction of C++ programs, as evidenced by the number of libraries and applications that utilize them. However, a number of those libraries provide C++ with partial implementations of facilities that other languages provide natively and more completely, features like tuples [**32**], first-class functions [**34**], and named function parameters [**1**]. But as discussed in Chapter 1, uses of these techniques have led to advances in the design of the language, including novel but effective features.

One goal of this work is to explore the capabilities of undecidable metaprogramming and to build a system in which to investigate new static abstractions, then later find ways to validate programs that use them before they are deployed and executed. Many advances in type systems have come from trying to build type systems that encapsulate useful functionality available in untyped or dynamically checked languages. This work extends that tradition by building an expressive language in which programmers can experiment with new constructs and principles. Static metaprogramming can be seen as a means to deploy immediately useful abstractions as well as a language laboratory for discovering fundamental abstractions that can be added to a traditional type system or programming language.

2. Future Work

The work presented in the previous chapters lays the groundwork for future investigations into the potential scope, safety, and expressive power for static metaprogramming. It points toward some next steps as well as toward some broader questions.

2.1. Broader Support for Cross-Stage Persistence. Static metaprograms derive great expressive power from the ability to embed the results of metalanguage computations into object-language programs. This technique is used heavily in C++ template metaprogramming to construct programs that are specialized to compile-time invariants.

The language designs presented in Chapters 4 and 6 support the persisting of object-language basic constants into object-language programs. This capability, however, does not capture the full potential for cross-stage persistence.

In a multi-stage programming language like MetaML, any value whatsoever can be persisted across metalanguage stages, including especially first-class functions. In a homogeneous metaprogramming language this capability is semantically coherent because any artifact of an early computation stage has a representation in a later stage computation.

The static metaprogramming language presented here, however, is heterogeneous, and some computations from the metalanguage have no analogue in the object language, the output of metaprogramming. For instance, the metalanguage can manipulate code and types, which are values that do not exist in the run-time language. In the syntax of the kernel metaprogramming language, the programs $\sim \%\mathbf{int}$ and $\sim \% \prec 7 \succ$ are nonsense because the code language does not contain a run-time representation of types as values or of code as values. However, the programs $\sim \%add1$ and $\sim \%\lambda x : int.x$ can be given quite sensible semantics. Heterogeneous programming languages must take greater care with support for persistence [**19**].

In the semantics of the kernel language, basic constants are the only elements that can be persisted because the current type system is not capable of differentiating the nonsense expressions above from the sensible expressions. For instance, the expression

$$\lambda x : int.\mathbf{if}\ \mathbf{int}^{?}\ \mathbf{bool}\ \mathbf{then}\ x\ \mathbf{else}\ 0$$

has the same type as $\lambda x : int.x$ in the metalanguage type system, but only the latter can be sensibly persisted into the code language. On the other hand, values with primitive types γ are always basic constants, making them clearly safe for persistence. This limitation to basic constants is analogous to what is currently possible with C++ template metaprogramming. Only a few kinds of types can be persisted into the run-time code. Not even floating point numbers can be persisted in that language, even though they could be considered a form of basic constant.

In order to semantically support cross-stage persistence more broadly from values of the metalanguage to expressions of the code language, the metalanguage type system must be further refined. In particular, the type system must be able to differentiate values that contain metalanguage-only expressions from values that also belong to the object language. Metalanguage-only values can only be mistaken for object-language values if they are contained in function abstractions. For this reason, it seems likely that a new form of function typing, perhaps assisted by a different form of function abstraction, can help the metalanguage type system more precisely type function abstractions.

One approach to this problem involves splitting the current metalanguage function abstraction expression into two different varieties, a *pure* abstraction $\lambda x : \tau.e^o$, which can only contain forms and types from the object language, and an *impure* abstraction $\lambda^\star x : \tau^s.e^s$, which supports all metalanguage expressions. The type system could then vary how the two expressions are typed. The pure abstraction could be typed in terms of the object-language type system $\vdash^o$ to ensure that it is safe for persistence.

$$\frac{\Gamma, x : \tau_1 \vdash^o e^o : \tau_2}{\Gamma \vdash \lambda x : \tau_1.e^o : \tau_1 \rightarrow \tau_2}$$

Syntactically, the object language e^o is the "safe subset" of both languages, capturing the set of persistable expressions. However, in order to ensure that no subexpressions of e^o use variables with metalanguage-only types like **code** or **type**, it is necessary to type the expression using the object-language type system.

To differentiate pure from impure functions, a new function type $\tau^s \Rightarrow \tau^s$ could be added to the language for impure metalanguage functions. Then the typing rule for impure

functions would be identical to the current function type rule.

$$\frac{\Gamma, x : \tau_1^s \vdash e^s : \tau_2^s}{\Gamma \vdash \lambda^\star x : \tau_1^s . e^s : \tau_1^s \Rightarrow \tau_2^s}$$

The typing rule for cross-stage persistence could then be updated to support all pure types.

$$\frac{\Gamma \vdash e^s : \tau}{\Gamma \vdash \%e^s : \mathbf{code}}$$

Ideally, pure metalanguage functions would be usable anywhere that impure metafunctions are expected. To support this, some safe form of coercion or subtyping between pure function types and impure function types is necessary.

$$\frac{\Gamma \vdash e^s : \tau \to \tau}{\Gamma \vdash e^s : \tau \Rightarrow \tau}$$

The design of such a coercion scheme requires great care to guarantee that an impure function can never be coerced to a pure function.

Besides a safe typing model, the implementation model for persisting pure functions into code-language code needs to be investigated. While constants are generally seen as straightforward to persist into a language, persisting functions that may have resulted from partial applications is not as obvious.

If the metalanguage is extended to support persistence of all terms of object-language type, then the parametric polymorphic functions of the metalanguage can be extended to support persistence of any value with type τ^α, since those types instantiate to pure object-language types. The following function then becomes legal:

$$\Lambda\alpha.\lambda x : \alpha \to \alpha.\lambda y : \alpha.\ \prec\!\sim\!\%(x\ y)\!\succ$$

This function parametrically takes a function x, a function y, and persists the result of applying x to y. Even though the exact type of this result is unknown, it is guaranteed to be a persistable value since any instantiation of α must be an object-language type τ.

2.2. Expressive Types in the Code Language. The metaprogramming language explicitly supports the programmatic manipulation of the object-language type system. However, the set of object-language types is limited compared to what is available in most

functional programming languages. In particular, the type system of the object language is simply typed and explicitly supports one type constructor for building arrow types. Programming languages like Standard ML and Objective Caml feature type systems that support parametric polymorphism, type constructors, and the definition of new types.

The framework presented in this thesis seems compatible with these more expressive type systems, and adding support for them to a metalanguage is likely to involve more engineering than innovation.

Two aspects of such an undertaking are worth discussing. A language that combines type constructors and parametric polymorphism can have type-level variables that either represent pure types or also type constructors. To support more complex type variables, the type system of the metalanguage must extend the definition of well-formed types to ensure that the types assigned to variables are always properly constructed, meaning of kind $*$, and that the types of types reflect the richer structure added to the type level.

Furthermore, an object language with parametric polymorphism has types that bind variables. The metalanguage mechanisms for analyzing such types must take variable binding into account. Recent work on nominal logic [**63**] and languages with support for programming with binders [**46**, **73**] provide possible models for defining the dynamic semantics of metaprograms that manipulate polymorphic types.

Finally, some languages feature types that exhibit richer relations than merely structure and equivalence. In particular, some types in object oriented languages have subtype relationships between each other. Other languages feature type systems that have qualified types [**29**, **36**], which associate complex predicates to types, and singleton types [**82**], which are types that represent particular run-time values. These extensions to the relationships between types offer new opportunities for increasing the expressiveness of metaprogramming.

2.3. Metaprogramming over Declarations. Unlike the typical production programming language, the object language has no constructs for adding declarations. In

particular, most programming languages provide a means to declare new types, type constructors, algebraic data types for functional programming languages, and classes for object-oriented languages. Support for more sophisticated code-language types involves being able to reason about types once they have been declared, but a metaprogramming language could also support computations that generate new type declarations. Programming paradigms like generative programming require the ability to construct new declarations while executing metaprograms. Given the popularity and significance of these techniques, the design space for metaprogramming over type declarations and the like requires investigation.

To support metaprogramming over declarations, the metalanguage type system must be augmented to support types that refer to declarations. Mechanisms for declaration metaprogramming are likely to vary substantially with the kind of declaration being constructed. It is likely that the relationship between well-typed expressions of the metalanguage and syntactically well-formed expressions of the object language will apply to these extensions.

2.4. Practical Implementation. This thesis presents a primarily theoretical perspective on static metaprogramming. The analysis of the mechanisms of C++ template metaprogramming provide enough understanding of the techniques, idioms, and core capabilities of metaprogramming to inspire the presented design. Furthermore the examples presented above demonstrate how this language design captures many of those idioms in a more principled, intentional, and modern approach. The formal specification of the languages provides a precise definition of their semantics, and metatheoretic results verify that the concepts presented are coherent, consistent, and sound.

In order to further elaborate the design toward practical ends, this language design would be improved by gaining experience developing software libraries and applications that take advantage of them. In order to do so, an implementation geared toward these ends is proposed. In particular, a metaprogramming extension to the $\mathcal{G}$ generic programming language is proposed [**80**], starting with its theoretical underpinnings, the calculus F^G [**75**]. The $\mathcal{G}$ research programming language has been used to demonstrate language support for generic programming and has been used to develop substantial case studies in library

design. Adding support for static metaprogramming extensions to $\mathcal{G}$ will introduce even greater support for library-centric software design to the language.

Several issues in implementing the metaprogramming language deserve consideration. First, the semantics of the language require that hygiene be preserved for code-language expressions. The same techniques used to implement hygienic macros [**45**] and multi-stage metaprogramming languages [**90**] can be applied to preserve hygiene in static metaprograms.

Previous work on stack inspection for security [**5**, **14**] has produced state-passing implementations of stack-based access privilege checks. The same implementation model applies to the **typeof** type reflection operator. While general mechanical techniques have been developed for deriving abstract machines from reduction semantics [**16**], more care is required to produce a correct corresponding compositional interpreter.

Cross-stage persistence as presented in previous chapters can be implemented simply, so long as every run-time value of the object language can be given a textual representation. However extension of persistence to higher-order values, as proposed in Section 2.1, presents more implementation challenges. The ability to persist functions that are used in the metalanguage into the code language implies that the metaprogram run-time system must maintain a representation of functions that can be executed and serialized into code objects. This suggests that the closures representing persistable functions must carry an intentional representation that is parameterized on the values over which their environments are closed. Functions that cannot be persisted need not carry such a representation, and in fact any operation that coerces persistable functions to non-persistable functions may discard this extra information. Developing an effective and reasonably space- and time-efficient implementation model for metaprogramming with higher-order persistence is a problem worthy of consideration.

2.5. Metaprogramming for Languages with Implicit Typing. The metaprogramming language takes advantage of manifest type annotations in the code language in order to perform some of its computations. In particular, the **typeof** operation collects type annotations from the metaprogram evaluation stack in order to compute the

type of subexpressions. However, production functional programming languages tend to have inference-based type systems. Rather than mandating that all variable abstractions be annotated, these languages perform type checking by using the structure of program expressions to construct a set of equations that constrain the types of variables and then solving them. It is not obvious that a type system based entirely on inference and featuring no type annotations would be suited to this metaprogramming model. Nonetheless, many languages that do not support full type inference still support some form of local inference, possibly limited to the bodies of top-level function definitions.

Since the type information in the context of an expression may be incomplete at any point during metacomputation, waiting for more type variable constraints to impose definitive form on a type. It may be possible to perform some metacomputations using partial information, but the need for more complete type information could lead to "deadlock" on computations that cannot complete. This situation seems to call for a more demand-driven evaluation order, possibly based on data-flow or logic programming.

Though the prevalence of nontrivial type annotations in mainstream programming languages provide great opportunity for leveraging type information at the meta-level, the extent to which type deduction and type inference systems at the object-language level are compatible with static metaprogramming may yield fruitful insights into the nature of type structure and the scope of the presented design.

2.6. Intensional Code Analysis. One notable feature of macro systems is support for code analysis, the ability to decompose programs as data into their bare components. Unrestrained manipulation of data that represents programs provides the greatest level of flexibility, but this flexibility can undermine expressiveness if it is not provided in a manner that is compatible with higher-level reasoning principles. Just as hygienic macros support reasoning about programs while respecting variable structure, so should static metaprogramming support reasoning about programs while respecting type structure. Hygienic macros have advanced to provide the benefits of hygiene in a framework that also supports the expressiveness of unrestrained computation [**18**]. A model for metaprogramming that

supports code analysis benefits from achieving an analogous balance between expressive power and structure-preserving reasoning principles.

2.7. Stronger Type Guarantees. As mentioned earlier, a metaprogram may signal an error during static computation, even if it is well-typed. This design raises modularity concerns. For instance, a buggy library of metaprogramming routines can generate ill-typed code under certain conditions. These errors may not be detected until an application developer uses the library in an application. The same issue arises with template metaprograms in C++. Efforts have been made to build systems that provide *meta type safety guarantees* [**31**], the property that a metaprogram can be statically checked prior to compile-time computation and be guaranteed to never produce compile-time type-related errors. Such guarantees often impose strong constraints on the computational power of the metaprogramming system. In future work on static metaprogramming language design, the relationship between power and safety deserves further investigation as well as efforts to develop metaprogramming type systems that strengthen metacomputation guarantees without unreasonably compromising expressive power.

3. Conclusion

Christopher Strachey said it best: "A programming language is a rather large body of new and somewhat arbitrary mathematical notation introduced in the hope of making the problem of controlling computing machines somewhat simpler [**83**]." In the 40 years since Strachey offered this definition, massive effort has been applied to the problem of controlling computing machines. Many control and organizational language constructs have been proposed and explored in an ongoing attempt to improve the software development process. This dissertation investigates a model of static metaprogramming that supports the treatment of code as data and the acquisition and manipulation of type information. Its approach was especially inspired by those who mined the depths of an untamed language's capabilities to uncover new principles for program organization and abstraction. By examining those principles and placing them on a firm foundation, this undertaking furthers the hopes ascribed to programming languages for the last four decades.

Bibliography

[1] Dave Abrahams. The Boost Parameters library user manual. http://boost.org/libs/parameter/.

[2] Dave Abrahams. The Boost Python user manual. http://boost.org/libs/python/.

[3] David Abrahams and Ralf W. Grosse-Kunstleve. Building hybrid systems with Boost.Python. *C/C++ Users Journal*, 21, July 2003.

[4] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[5] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005.

[6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[7] Jonathan Bachrach and Keith Playford. The Java syntactic extender (JSE). In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 31–42, New York, NY, USA, 2001. ACM Press.

[8] Jason Baker and Wilson C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 270–281, New York, NY, USA, 2002. ACM Press.

[9] Martin Böhme and Bodo Manthey. The computational power of compiling C++. *Bulletin of the EATCS*, 81:264–270, 2003.

[10] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008. Special issue on experimental software and toolkits.

[11] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM Press.

[12] Frederick P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, anniversary edition, 1995.

[13] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[14] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, 2004.

[15] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[16] Olivier Danvy and Lasse R. Nielsen. Refocusing in reduction semantics. Technical Report RS-04-26, BRICS, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, November 2004.

[17] Joel de Guzman. The Boost Spirit user manual. http://boost.org/libs/spirit/.

[18] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, dec 1992.

[19] Jason Eckhardt, Roumen Kaiabachev, Emir Pasalic, Kedar N. Swadi, and Walid Taha. Implicitly heterogeneous multi-stage programming. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *Lecture Notes in Computer Science*, pages 275–292. Springer, 2005.

[20] Matthias Felleisen and Robert Hieb. A revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[21] Robert Bruce Findler, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Drscheme: A pedagogic programming environment for scheme. In Hugh Glaser, Pieter H. Hartel, and Herbert Kuchen, editors, *Programming Languages: Implementations, Logics, and Programs, 9th International Symposium, PLILP'97, Including a Special Trach on Declarative Programming Languages in Education, Southampton, UK, September 3-5, 1997, Proceedings*, volume 1292 of *Lecture Notes in Computer Science*, pages 369–388. Springer, 1997.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[23] Steven E. Ganz, Amr Sabry, and Walid Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 74–85, New York, NY, USA, 2001. ACM Press.

[24] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2):145–205, March 2007.

[25] Ronald Garcia and Andrew Lumsdaine. MultiArray: a C++ library for generic programming with arrays. *Software—Practice and Experience*, 35(2):159–188, 2005.

[26] Narain H. Gehani. Units of measure as a data attribute. *Computer Languages*, 2(3):93–111, 1977.

[27] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures de l'Arithmétique d'Ordre Supérieur*. Thèse de doctorat d'état, Université Paris VII, Paris, France, 1972.

[28] Douglas Gregor and Jaakko Järvi. Variadic templates for C++. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1101–1108, New York, NY, USA, 2007. ACM Press.

[29] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 291–310, New York, NY, USA, 2006. ACM Press.

[30] Timothy P. Hart. MACRO definitions for LISP. AI Memo AIM-57, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, MIT Artificial Intelligence Project—RLE and MIT Computation Center, Cambridge, Massachusetts, October 1963.

[31] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 399–424. Springer, 2007.

[32] Jaakko Järvi. Tuple types and multiple return values. *C/C++ Users Journal*, 19:24–35, August 2001.

[33] Jaakko Järvi and John Freeman. Lambda expressions for C++0x. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, New York, NY, USA, 2008. ACM Press. To appear.

[34] Jaakko Järvi, Gary Powell, and Andrew Lumsdaine. The Lambda Library: unnamed functions in C++. *Software—Practice and Experience*, 33:259–291, 2003.

[35] S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[36] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[37] Mark P. Jones. Dictionary-free overloading by partial evaluation. *Lisp and Symbolic Computation.*, 8(3):229–248, 1995.

[38] Neil D. Jones. An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[39] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[40] Prabhanjan Kambadur, Douglas Gregor, Andrew Lumsdaine, and Amey Dharurkar. Modernizing the C++ interface to MPI. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, LNCS, pages 266–274, Bonn, Germany, September 2006. Springer.

[41] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, September 1998.

[42] Richard Kelsey and Jonathan Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1994.

[43] Richard A. Kelsey. Pre-Scheme: A Scheme dialect for systems programming, December 05 1997.

[44] Aaron Kershenbaum, David Musser, and Alexander Stepanov. Higher order imperative programming. Technical Report 88-10, Rensselaer Polytechnic Institute, April 1988.

[45] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In Richard P. Gabriel, editor, *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 151–181, Cambridge, MA, August 1986. ACM Press.

[46] M. R. Lakin and A. M. Pitts. A metalanguage for structural operational semantics. In M. Morazán, editor, *Trends in Functional Programming Volume 8*, pages 19–35. Intellect, 2008.

[47] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. Generic graph algorithms for sparse matrix ordering. In *ISCOPE'99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[48] Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 399–414, New York, NY, USA, 1999. ACM Press.

[49] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[50] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311. Springer, 2004.

[51] John McCarthy. History of LISP. *History of Programming Languages I*, pages 173–185, 1981.

[52] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, 1993. ISBN: 1–55615–484–4.

[53] M. Douglas McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Proceedings of Software Engineering Concepts and Techniques, 1 968 NATO Conference on Software Engineering*, pages 138–155, January 1969. `http://www.cs.dartmouth.edu/~doug/components.txt`.

[54] Brian McNamara and Yannis Smaragdakis. Static interfaces in C++. In *First Workshop on C++ Template Programming*, October 2000.

[55] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[56] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[57] David R. Musser and Alexander A. Stepanov. A library of generic algorithms in Ada. In *SIGAda '87: Proceedings of the 1987 annual ACM SIGAda international conference on Ada*, pages 216–225, New York, NY, USA, 1987. ACM.

[58] David R. Musser and Alexander A. Stepanov. Generic programming. In *ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, pages 13–25, London, UK, 1989. Springer-Verlag.

[59] Peter Naur and Brian Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. Scientific Affairs Division, NATO, Brussels 39 Belgium, 1969.

[60] Gregory Neverov and Paul Roe. Metaphor: A multi-stage, object-oriented programming language. In Gabor Karsai and Eelco Visser, editors, *GPCE*, volume 3286 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2004.

[61] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[62] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[63] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[64] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[65] Ed Post. Real programmers don't use Pascal. *Datamation*, 29(7), July 1983. Available from *http://www.pbm.com/~lindahl/real.programmers.html*.

[66] Dan Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.

[67] Apple Computer Eastern Research and Technology. *Dylan: An object oriented dynamic language*, June 94.

[68] John Reynders. The POOMA framework – A templated class library for parallel scientific computing. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing (8th PPSC'97)*, Minneapolis, Minnesota, USA, March 1997. SIAM (Philadelphia).

[69] John C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *LNCS*, pages 408–425, Berlin, 1974. Springer-Verlag.

[70] Arch D. Robison. Impact of economics on compiler optimization. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 1–10, New York, NY, USA, 2001. ACM.

[71] Tim Sheard. A taxonomy of meta-programming systems. http://web/cecs.pdf.edu/ sheard/staged.

[72] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In Manuel M. T. Chakravarty, editor, *ACM SIGPLAN Haskell Workshop 02*, pages 1–16. ACM Press, October 2002.

[73] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[74] Jeremy Siek and Andrew Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *First Workshop on C++ Template Programming*, October 2000.

[75] Jeremy Siek and Andrew Lumsdaine. Essential language support for generic programming. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, pages 73–84, New York, NY, USA, June 2005. ACM Press.

[76] Jeremy G. Siek, Lee-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[77] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A unifying framework for numerical linear algebra. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.

[78] Jeremy G. Siek and Andrew Lumsdaine. A rational approach to portable high performance: The basic linear algebra instruction set (BLAIS) and the fixed algorithm size template (FAST) library. In *Parallel Object Oriented Scientific Computing*. ECOOP, 1998.

[79] Jeremy G. Siek and Andrew Lumsdaine. *Advances in Software Tools for Scientific Computing*, chapter A Modern Framework for Portable High Performance Numerical Linear Algebra. Springer, 2000.

[80] Jeremy G. Siek and Andrew Lumsdaine. Language requirements for large-scale generic libraries. In Robert Glück and Michael R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference, GPCE 2005, Tallinn, Estonia, September 29 - October 1, 2005, Proceedings*, volume 3676 of *Lecture Notes in Computer Science*, pages 405–421. Springer, 2005.

[81] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

[82] Christopher A. Stone. *Singleton Kinds and Singleton Types*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, August 2000.

[83] C. Strachey. Towards a formal semantics. In T. B. Steel, editor, *Formal Language Description Languages for Computer Programming*, pages 198–220. North-Holland, 1966.

[84] Bjarne Stroustrup. *Design and Evolution of C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[85] Bjarne Stroustrup. Evolving a language in and for the real world: C++ 1991-2006. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 4–1–4–59, New York, NY, USA, 2007. ACM.

[86] Kedar Swadi, editor. *Second MetaOCaml Workshop (GPCE05)*, September 2005.

[87] Walid Taha, Cristiano Calcagno, Xavier Leroy, Ed Pizzi, Emir Pasalic, Jason Lee Eckhardt, Roumen Kaiabachev, and Oleg Kiselyov. MetaOCaml: A compiled, type-safe multi-stage programming language. Available online from `http://www.metaocaml.org/`, 2004.

[88] Walid Taha and Patricia Johann. Staged notational definitions. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 97–116, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[89] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, NY, USA, 1997. ACM Press.

[90] Walid Mohamed Taha. *Multistage programming: its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999. Supervisor-Tim Sheard.

[91] Guido van Rossum. *Python Reference Manual*. Stichting Mathematisch Centrum, Amsterdam, 1996.

[92] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for Java. In *European Conference on Object Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer Verlag, July 2007. To Appear.

[93] Todd Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[94] Todd L. Veldhuizen. Arrays in Blitz++. In Denis Caromel, R. R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments, Second International Symposium, ISCOPE 98, Santa Fe, NM, USA, December 8-11, 1998, Proceedings*, volume 1505 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998.

[95] Oscar Waddell and R. Kent Dybig. Fast and effective procedure inlining. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 35–52, London, UK, 1997. Springer-Verlag.

[96] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, 1992.

[97] Stephen Weeks. Whole-program compilation in MLton. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 1–1, New York, NY, USA, 2006. ACM Press.

[98] Jeremiah J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software*. PhD thesis, Indiana University, January 2008.

[99] Hongyu Zhang and Stan Jarzabek. An XVCL approach to handling variants: A KWIC product line example. In *APSEC '03: Proceedings of the Tenth Asia-Pacific Software Engineering Conference Software Engineering Conference*, page 116, Washington, DC, USA, 2003. IEEE Computer Society.

APPENDIX A

Kernel Language Metatheory and Proofs

Symbol Classes

γ ::= ⟨type constant⟩
x ::= ⟨variable⟩
α ::= ⟨type variable⟩
c ::= ⟨basic constant⟩
f ::= ⟨function constant⟩
c^{s^-} ::= $\gamma^?$ $|{\rightarrow}^?|$ **dom** | **cod** | **typeof** (non-type meta constant)
c^s ::= γ | c^{s^-} (meta constant)

Terms

pgm ::= e (program)

e ::= x | $\lambda x : e^s.e$ | $e\ e$ (code language)
| c | f | **if** e **then** e **else** e
| $\sim e^s$
e^o ::= x | $\lambda x : \tau.e^o$ | $e^o\ e^o$ (pure code)
| c | f | **if** e^o **then** e^o **else** e^o

e^s ::= x | α | $\lambda x : \tau^s.e^s$ | $e^s\ e^s$ | $\Lambda\alpha.e^s$ | $e^s[\tau^\alpha]$ (metalanguage)
| c | f | **if** e^s **then** e^s **else** e^s
| c^s
| $\prec e \succ$
| $\%_0 e^s$
| $e^s \rightarrow e^s$
| $e^s =_\tau e^s$
v^s ::= c | f | c^{s^-} | $\prec e^o \succ$ | τ | $\lambda x : \tau^s.e^s$ | $\Lambda\alpha.e^s$ (meta value)

Types

τ ::= γ | $\tau \rightarrow \tau$ (ground type)
τ^α ::= α | γ | $\tau^\alpha \rightarrow \tau^\alpha$ (type scheme)

Meta Types

τ^s ::= α
| γ
| **code**
| **type**
| $\tau^s \rightarrow \tau^s$
| $\forall\alpha.\tau^s$

ε ::= $x : \tau^s$ | x : **dyn** | $\alpha : *$ (environment bindings)
Γ ::= $\overline{\varepsilon_i}$ (environment)

FIGURE 1. Kernel Language Syntax

$\boxed{\Gamma \textbf{ wf}}$

$$\frac{}{\emptyset \textbf{ wf}} \qquad \frac{\Gamma \textbf{ wf} \quad \alpha \notin FV(\Gamma)}{\Gamma, \alpha : * \textbf{ wf}} \qquad \frac{\Gamma \textbf{ wf} \quad x \notin FV(\Gamma)}{\Gamma, x : \textbf{dyn wf}}$$

$$\frac{\Gamma \textbf{ wf} \quad x \notin FV(\Gamma) \quad FV(\tau^s) \subseteq FV(\Gamma)}{\Gamma, x : \tau^s \textbf{ wf}}$$

FIGURE 2. Well-formed Environment

$\boxed{\Gamma \vdash e^s : \tau^s}$

$$\frac{(x : \tau^s) \in \Gamma}{\Gamma \vdash x : \tau^s} \qquad \frac{(\alpha : *) \in \Gamma}{\Gamma \vdash \alpha : \textbf{type}} \qquad \frac{}{\Gamma \vdash \textbf{dom} : \textbf{type} \to \textbf{type}}$$

$$\frac{}{\Gamma \vdash \textbf{cod} : \textbf{type} \to \textbf{type}} \qquad \frac{}{\Gamma \vdash \gamma^? : \textbf{type} \to \textbf{bool}} \qquad \frac{}{\Gamma \vdash \to^? : \textbf{type} \to \textbf{bool}}$$

$$\frac{}{\Gamma \vdash \textbf{typeof } : \textbf{code} \to \textbf{type}} \qquad \frac{\mathit{type}(f) = \gamma_1 \to \gamma_2}{\Gamma \vdash f : \gamma_1 \to \gamma_2} \qquad \frac{\mathit{type}(c) = \gamma}{\Gamma \vdash c : \gamma}$$

$$\frac{\Gamma \vdash e_1^s : \textbf{bool} \quad \Gamma \vdash e_2^s : \tau^s \quad \Gamma \vdash e_3^s : \tau^s}{\Gamma \vdash \textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s : \tau^s} \qquad \frac{}{\Gamma \vdash \gamma : \textbf{type}} \qquad \frac{\Gamma \vdash e \textbf{ wf}}{\Gamma \vdash \prec e \succ : \textbf{code}}$$

$$\frac{\Gamma \vdash e_1^s : \textbf{type} \quad \Gamma \vdash e_2^s : \textbf{type}}{\Gamma \vdash e_1^s \to e_2^s : \textbf{type}} \qquad \frac{\Gamma \vdash e_1^s : \textbf{type} \quad \Gamma \vdash e_2^s : \textbf{type}}{\Gamma \vdash e_1^s =_\tau e_2^s : \textbf{bool}}$$

$$\frac{\Gamma \vdash e^s : \gamma}{\Gamma \vdash \%e^s : \textbf{code}} \qquad \frac{\Gamma \vdash e_1^s : \tau_1^s \to \tau_2^s \quad \Gamma \vdash e_2^s : \tau_1^s}{\Gamma \vdash e_1^s \; e_2^s : \tau_2^s}$$

$$\frac{\Gamma, x : \tau_1^s \vdash e^s : \tau_2^s \quad x \notin FV(\Gamma)}{\Gamma \vdash \lambda x : \tau_1^s . e^s : \tau_1^s \to \tau_2^s} \qquad \frac{\Gamma, \alpha : * \vdash e^s : \tau^s \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash \Lambda \alpha . e^s : \forall \alpha . \tau^s}$$

$$\frac{\Gamma \vdash e_1^s : \forall \alpha . \tau^s \quad \Gamma \vdash \tau^\alpha : \textbf{type}}{\Gamma \vdash e_1^s [\tau^\alpha] : \tau^s \left\{ \tau^\alpha / \alpha \right\}^\tau}$$

$\boxed{\Gamma \vdash e \textbf{ wf}}$

$$\frac{(x : \textbf{dyn}) \in \Gamma}{\Gamma \vdash x \textbf{ wf}} \qquad \frac{}{\Gamma \vdash c \textbf{ wf}} \qquad \frac{}{\Gamma \vdash f \textbf{ wf}}$$

$$\frac{\Gamma \vdash e^s : \textbf{type} \quad \Gamma, x : \textbf{dyn} \vdash e \textbf{ wf} \quad x \notin FV(\Gamma)}{\Gamma \vdash \lambda x : e^s . e \textbf{ wf}} \qquad \frac{\Gamma \vdash e_1 \textbf{ wf} \quad \Gamma \vdash e_2 \textbf{ wf}}{\Gamma \vdash e_1 \; e_2 \textbf{ wf}}$$

$$\frac{\Gamma \vdash e_1 \textbf{ wf} \quad \Gamma \vdash e_2 \textbf{ wf} \quad \Gamma \vdash e_3 \textbf{ wf}}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ wf}} \qquad \frac{\Gamma \vdash e^s : \textbf{code}}{\Gamma \vdash \sim e^s \textbf{ wf}}$$

FIGURE 3. Kernel Language Static Semantics

Evaluation Contexts

$$
\begin{array}{lcl}
E^s & ::= & E^s[\Box\ e^s] \\
 & | & E^s[v^s\ \Box] \\
 & | & E^s[\textbf{if } \Box \textbf{ then } e^s \textbf{ else } e^s] \\
 & | & E[\lambda x : \Box.e] \\
 & | & E[\sim\Box] \\
 & | & E^s[\%\Box] \\
 & | & E^s[\Box \rightarrow e^s] \\
 & | & E^s[\tau \rightarrow \Box] \\
 & | & E^s[\Box =_\tau e^s] \\
 & | & E^s[v^s =_\tau \Box] \\
 & | & E^s[\Box[\tau^\alpha]] \\
E & ::= & \Box \\
 & | & E[\Box\ e] \\
 & | & E[e^o\ \Box] \\
 & | & E[\lambda x : \tau.\Box] \\
 & | & E[\textbf{if } \Box \textbf{ then } e \textbf{ else } e] \\
 & | & E[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e] \\
 & | & E[\textbf{if } e^o \textbf{ then } e^o \textbf{ else } \Box] \\
 & | & E^s[\prec\Box\succ] \\
 & & \\
E^* & ::= & E \mid E^s
\end{array}
$$

FIGURE 4. Evaluation Contexts

$\boxed{r \rightarrow_{E^s} e^s}$ **Notions of Reduction**

$(\lambda x : \tau^s.e^s)\ v^s$	$\rightarrow_{E^s}$	$e^s\,[v^s/x]^s$		(term application)
$(\Lambda\alpha.e^s)[\tau^\alpha]$	$\rightarrow_{E^s}$	$e^s\,\{\tau^\alpha/\alpha\}^s$		(type application)
$\%c$	$\rightarrow_{E^s}$	$\prec c \succ$		(cross-stage persistence)
if true then e_1^s **else** e_2^s	$\rightarrow_{E^s}$	e_1^s		
if false then e_1^s **else** e_2^s	$\rightarrow_{E^s}$	e_2^s		
$f\ c$	$\rightarrow_{E^s}$	$\delta(f,c)$	$(f,c) \in DOM(\delta)$	(delta rules)
$\gamma^?\ \gamma$	$\rightarrow_{E^s}$	**true**		(base type match)
$\gamma^?\ \tau$	$\rightarrow_{E^s}$	**false**	$\tau \not\equiv \gamma$	(base type mismatch)
$\rightarrow^?\ (\tau_1 \rightarrow \tau_2)$	$\rightarrow_{E^s}$	**true**		(arrow type match)
$\rightarrow^?\ \tau$	$\rightarrow_{E^s}$	**false**	$\tau \not\equiv \tau_1 \rightarrow \tau_2$	(arrow type mismatch)
$\mathbf{dom}(\tau_1 \rightarrow \tau_2)$	$\rightarrow_{E^s}$	τ_1		(arrow type domain)
$\mathbf{dom}(\tau)$	$\rightarrow_{E^s}$	τ	$\tau \not\equiv \tau_1 \rightarrow \tau_2$	(non-arrow type domain)
$\mathbf{cod}(\tau_1 \rightarrow \tau_2)$	$\rightarrow_{E^s}$	τ_2		(arrow type codomain)
$\mathbf{cod}(\tau)$	$\rightarrow_{E^s}$	τ	$\tau \not\equiv \tau_1 \rightarrow \tau_2$	(non-arrow type codomain)
$\tau =_\tau \tau$	$\rightarrow_{E^s}$	**true**		(type equality)
$\tau_1 =_\tau \tau_2$	$\rightarrow_{E^s}$	**false**	$\tau_1 \not\equiv \tau_2$	(type inequality)

$^\ddagger \Gamma \vdash f\ c : \gamma \Rightarrow (f,c) \in DOM(\delta) \wedge \Gamma \vdash \delta(f,c) : \gamma.$

$\boxed{e \longmapsto e \cup \{\ \bot\ \}}$ **Context-Sensitive Reduction**

$$\text{(meta-eval)}\ \frac{e_1^s \rightarrow_{E^s} e_2^s}{E^s[e_1^s] \longmapsto E^s[e_2^s]} \qquad \text{(splice)}\ \frac{}{E[\sim\prec e^o \succ] \longmapsto E[e^o]}$$

$$\text{(typeof)}\ \frac{\Gamma^s(E^s) \vdash e^o : \tau}{E^s[\mathbf{typeof}\ \prec e^o \succ] \longmapsto E^s[\tau]} \qquad \text{(notype)}\ \frac{\forall \tau.\Gamma^s(E^s) \not\vdash e^o : \tau}{E^s[\mathbf{typeof}\ \prec e^o \succ] \longmapsto \bot}$$

FIGURE 5. Reduction Rules

$\boxed{\Gamma^s(E^s) : E^s \to \overline{x_i : \tau_i}}$

$$
\begin{array}{lcl}
\Gamma^s(E^s[\Box\ e^s]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[v^s\ \Box]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\textbf{if } \Box \textbf{ then } e^s \textbf{ else } e^s]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\%\Box]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\Box \to e^s]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\tau \to \Box]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\Box =_\tau e^s]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[v^s =_\tau \Box]) & = & \Gamma^s(E^s) \\
\Gamma^s(E^s[\Box[\tau^\alpha]]) & = & \Gamma^s(E^s) \\
\Gamma^s(E[\lambda x : \Box.e]) & = & \Gamma(E) \\
\Gamma^s(E[\sim\Box]) & = & \Gamma(E)
\end{array}
$$

$\boxed{\Gamma(E) : E \to \overline{x_i : \tau_i}}$

$$
\begin{array}{lcl}
\Gamma(\Box) & = & \emptyset \\
\Gamma(E[\Box\ e]) & = & \Gamma(E) \\
\Gamma(E[e^o\ \Box]) & = & \Gamma(E) \\
\Gamma(E[\lambda x : \tau.\Box]) & = & \Gamma(E), x : \tau \\
\Gamma(E[\textbf{if } \Box \textbf{ then } e \textbf{ else } e]) & = & \Gamma(E) \\
\Gamma(E[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e]) & = & \Gamma(E) \\
\Gamma(E[\textbf{if } e^o \textbf{ then } e^o \textbf{ else } \Box]) & = & \Gamma(E) \\
\Gamma(E^s[\prec\Box\succ] & & \\
\Gamma(E^s[\prec\Box\succ]) & = & \Gamma^s(E^s)
\end{array}
$$

FIGURE 6. Type Context Extraction

$\boxed{(\cdot)\,[e^s/x]^s : e^s \to e^s}$

$$
\begin{array}{lcl}
x\,[e_0^s/x]^s & = & e_0^s \\
x_1\,[e_0^s/x_2]^s & = & x_1 \qquad x_1 \not\equiv x_2 \\
\alpha\,[e_0^s/x]^s & = & \alpha \\
(\lambda x : \tau^s.e^s)\,[e_0^s/x]^s & = & \lambda x : \tau^s.e^s \\
(\lambda x_1 : \tau^s.e^s)\,[e_0^s/x_2]^s & = & \lambda x_3 : \tau^s.e^s\,[x_3/x_1]^s\,[e_0^s/x_2]^s \qquad x_1 \not\equiv x_2 \\
 & & \quad x_3 = \begin{cases} x_1 & x_1 \notin FV^s(e_0^s) \text{ or } x_2 \notin FV^s(e^s) \\ x_f & \text{for } x_f \notin FV^s(e_0^s) \cup FV^s(e^s) \end{cases} \\
(e_1^s\ e_2^s)\,[e_0^s/x]^s & = & e_1^s\,[e_0^s/x]^s\ e_2^s\,[e_0^s/x]^s \\
(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s)\,[e_0^s/x]^s & = & \textbf{if } e_1^s\,[e_0^s/x]^s \textbf{ then } e_2^s\,[e_0^s/x]^s \textbf{ else } e_3^s\,[e_0^s/x]^s \\
(\Lambda\alpha.e^s)\,[e_0^s/x]^s & = & \Lambda\alpha.e^s\,[e_0^s/x]^s \\
(e^s[\tau^\alpha])\,[e_0^s/x]^s & = & (e^s\,[e_0^s/x]^s)[\tau^\alpha] \\
c\,[e_0^s/x]^s & = & c \\
f\,[e_0^s/x]^s & = & f \\
c^s\,[e_0^s/x]^s & = & c^s \\
(e_1^s =_\tau e_2^s)\,[e_0^s/x]^s & = & e_1^s\,[e_0^s/x]^s =_\tau e_2^s\,[e_0^s/x]^s \\
\prec e \succ\,[e_0^s/x]^s & = & \prec e\,[e_0^s/x] \succ \\
(\%_0 e^s)\,[e_0^s/x]^s & = & \%_0 e^s\,[e_0^s/x]^s \\
(e_1^s \to e_2^s)\,[e_0^s/x]^s & = & e_1^s\,[e_0^s/x]^s \to e_2^s\,[e_0^s/x]^s
\end{array}
$$

$\boxed{(\cdot)\,[e_0^s/x] : e \to e}$

$$
\begin{array}{lcl}
x_1\,[e_0^s/x_2] & = & x_1 \\
(\lambda x_1 : e^s.e)\,[e_0^s/x_2] & = & \lambda x_3 : e^s\,[e_0^s/x_2]^s\,.e|_{x_1}^{x_3}\,[e_0^s/x_2] \\
 & & \quad x_3 = \begin{cases} x_1 & x_1 \notin FV^s(e_0^s) \text{ or } x_2 \notin FV(e) \\ x_f & \text{for } x_f \notin FV^s(e_0^s) \cup FV(e) \end{cases} \\
(e_1\ e_2)\,[e_0^s/x] & = & e_1\,[e_0^s/x]\ e_2\,[e_0^s/x] \\
c\,[e_0^s/x] & = & c \\
f\,[e_0^s/x] & = & f \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)\,[e_0^s/x] & = & \textbf{if } e_1\,[e_0^s/x] \textbf{ then } e_2\,[e_0^s/x] \textbf{ else } e_3\,[e_0^s/x] \\
(\sim e^s)\,[e_0^s/x] & = & \sim e^s\,[e_0^s/x]^s
\end{array}
$$

FIGURE 7. Term Substitution

$\boxed{(\cdot)\{\tau^\alpha/\alpha\}^s : e^s \to e^s}$

$$
\begin{array}{lcl}
x\{\tau^\alpha/\alpha\}^s & = & x \\
\alpha\{\tau^\alpha/\alpha\}^s & = & \tau^\alpha \\
\alpha_1\{\tau^\alpha/\alpha_2\}^\tau & = & \alpha_1 \qquad \alpha_1 \not\equiv \alpha_2 \\
(\lambda x : \tau^s.e^s)\{\tau^\alpha/\alpha\}^s & = & \lambda x : \tau^s\{\tau^\alpha/\alpha\}^\tau.e^s\{\tau^\alpha/\alpha\}^s \\
(e_1^s\ e_2^s)\{\tau^\alpha/\alpha\}^s & = & e_1^s\{\tau^\alpha/\alpha\}^s\ e_2^s\{\tau^\alpha/\alpha\}^s \\
(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s)\{\tau^\alpha/\alpha\}^s & = & \textbf{if } e_1^s\{\tau^\alpha/\alpha\}^s \textbf{ then } e_2^s\{\tau^\alpha/\alpha\}^s \textbf{ else } e_3^s\{\tau^\alpha/\alpha\}^s \\
(\Lambda\alpha.e^s)\{\tau^\alpha/\alpha\}^s & = & \Lambda\alpha.e^s \\
(\Lambda\alpha_1.e^s)\{\tau^\alpha/\alpha_2\}^s & = & (\Lambda\alpha_3.e^s\{\alpha_3/\alpha_1\}^s\{\tau^\alpha/\alpha_2\}^s) \qquad \alpha_1 \not\equiv \alpha_2 \\
 & & \alpha_3 = \begin{cases} \alpha_1 & \alpha_1 \notin FV^s(\tau^\alpha) \text{ or } \alpha_2 \notin FV^s(e^s) \\ \alpha_f & \text{for } \alpha_f \notin FV^s(\tau^\alpha) \cup FV^s(e^s) \end{cases} \\
(e^s[\tau_1^\alpha])\{\tau_2^\alpha/\alpha\}^s & = & e^s\{\tau_2^\alpha/\alpha\}^s[\tau_1^\alpha\{\tau_2^\alpha/\alpha\}^\tau] \\
c\{\tau^\alpha/\alpha\}^s & = & c \\
f\{\tau^\alpha/\alpha\}^s & = & f \\
c^s\{\tau^\alpha/\alpha\}^s & = & c^s \\
(e_1^s =_\tau e_2^s)\{\tau^\alpha/\alpha\}^s & = & e_1^s\{\tau^\alpha/\alpha\}^s =_\tau e_2^s\{\tau^\alpha/\alpha\}^s \\
\prec e \succ\{\tau^\alpha/\alpha\}^s & = & \prec e\{\tau^\alpha/\alpha\}\succ \\
(\%_0 e^s)\{\tau^\alpha/\alpha\}^s & = & \%_0 e^s\{\tau^\alpha/\alpha\}^s \\
(e_1^s \to e_2^s)\{\tau^\alpha/\alpha\}^s & = & (e_1^s\{\tau^\alpha/\alpha\}^s \to e_2^s\{\tau^\alpha/\alpha\}^s)
\end{array}
$$

$\boxed{(\cdot)\{\tau^\alpha/\alpha\} : e \to e}$

$$
\begin{array}{lcl}
x\{\tau^\alpha/\alpha\} & = & x \\
(\lambda x_1 : e^s.e)\{\tau^\alpha/\alpha\} & = & (\lambda x_1 : e^s\{\tau^\alpha/\alpha\}^s.e\{\tau^\alpha/\alpha\}) \\
(e_1\ e_2)\{\tau^\alpha/\alpha\} & = & e_1\{\tau^\alpha/\alpha\}\ e_2\{\tau^\alpha/\alpha\} \\
c\{\tau^\alpha/\alpha\} & = & c \\
f\{\tau^\alpha/\alpha\} & = & f \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)\{\tau^\alpha/\alpha\} & = & \textbf{if } e_1\{\tau^\alpha/\alpha\} \textbf{ then } e_2\{\tau^\alpha/\alpha\} \textbf{ else } e_3\{\tau^\alpha/\alpha\} \\
(\sim e^s)\{\tau^\alpha/\alpha\} & = & \sim e^s\{\tau^\alpha/\alpha\}^s
\end{array}
$$

$\boxed{(\cdot)\{\tau^\alpha/\alpha\}^\tau : \tau^s \to \tau^s}$

$$
\begin{array}{lcl}
\alpha\{\tau^\alpha/\alpha\}^\tau & = & \tau^\alpha \\
\alpha_1\{\tau^\alpha/\alpha_2\}^\tau & = & \alpha_1 \qquad \alpha_1 \not\equiv \alpha_2 \\
\gamma\{\tau^\alpha/\alpha\}^\tau & = & \gamma \\
\textbf{code}\{\tau^\alpha/\alpha\}^\tau & = & \textbf{code} \\
\textbf{type}\{\tau^\alpha/\alpha\}^\tau & = & \textbf{type} \\
(\tau_1^s \to \tau_2^s)\{\tau^\alpha/\alpha\}^\tau & = & \tau_1^s\{\tau^\alpha/\alpha\}^\tau \to \tau_2^s\{\tau^\alpha/\alpha\}^\tau \\
(\forall\alpha.\tau^s)\{\tau^\alpha/\alpha\}^\tau & = & (\forall\alpha.\tau^s) \\
(\forall\alpha_1.\tau^s)\{\tau^\alpha/\alpha_2\}^\tau & = & (\forall\alpha_3.\tau^s\{\alpha_3/\alpha_1\}^\tau\{\tau^\alpha/\alpha_2\}^\tau) \qquad \alpha_1 \not\equiv \alpha_2 \\
 & & \alpha_3 = \begin{cases} \alpha_1 & \alpha_1 \notin FV^s(\tau^\alpha) \text{ or } \alpha_2 \notin FV^s(\tau^s) \\ \alpha_f & \alpha_f \notin FV^s(\tau^\alpha) \cup FV^s(\tau^s) \end{cases}
\end{array}
$$

FIGURE 8. Type Schema Substitution

$\boxed{(\cdot)|_{x_1}^{x_2\,s} : e^s \to e^s}$

$$
\begin{array}{lcl}
x_3|_{x_1}^{x_2\,s} & = & x_3 \\
\alpha|_{x_1}^{x_2\,s} & = & \alpha \\
(\lambda x_3 : \tau^s.e^s)|_{x_1}^{x_2\,s} & = & \lambda x_4 : \tau^s.e^s\,[x_4/x_3]^s\,|_{x_1}^{x_2\,s} \\
 & & x_4 = \begin{cases} x_3 & x_3 \not\equiv x_2 \text{ or } x_1 \notin FV^s(e^s) \\ x_f & \text{for } x_f \notin \{\, x_2 \,\} \cup FV^s(e^s) \end{cases} \\
(e_1^s\ e_2^s)|_{x_1}^{x_2\,s} & = & e_1^s|_{x_1}^{x_2\,s}\ e_2^s|_{x_1}^{x_2\,s} \\
(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s)|_{x_1}^{x_2\,s} & = & \textbf{if } e_1^s|_{x_1}^{x_2\,s} \textbf{ then } e_2^s|_{x_1}^{x_2\,s} \textbf{ else } e_3^s|_{x_1}^{x_2\,s} \\
(\Lambda\alpha.e^s)|_{x_1}^{x_2\,s} & = & \Lambda\alpha.e^s|_{x_1}^{x_2\,s} \\
(e^s[\tau^\alpha])|_{x_1}^{x_2\,s} & = & e^s|_{x_1}^{x_2\,s}[\tau^\alpha] \\
c|_{x_1}^{x_2\,s} & = & c \\
f|_{x_1}^{x_2\,s} & = & f \\
c^s|_{x_1}^{x_2\,s} & = & c^s \\
(e_1^s =_\tau e_2^s)|_{x_1}^{x_2\,s} & = & e_1^s|_{x_1}^{x_2\,s} =_\tau e_2^s|_{x_1}^{x_2\,s} \\
\prec e \succ |_{x_1}^{x_2\,s} & = & \prec e|_{x_1}^{x_2} \succ \\
(\%_0 e^s)|_{x_1}^{x_2\,s} & = & \%_0 e^s|_{x_1}^{x_2\,s} \\
(e_1^s \to e_2^s)|_{x_1}^{x_2\,s} & = & e_1^s|_{x_1}^{x_2\,s} \to e_2^s|_{x_1}^{x_2\,s}
\end{array}
$$

$\boxed{(\cdot)|_{x_1}^{x_2} : e \to e}$

$$
\begin{array}{lcl}
x_1|_{x_1}^{x_2} & = & x_2 \\
x_3|_{x_1}^{x_2} & = & x_3 \qquad x_3 \not\equiv x_1 \\
(\lambda x_1 : e^s.e)|_{x_1}^{x_2} & = & \lambda x_1 : e^s|_{x_1}^{x_2\,s}.e \\
(\lambda x_3 : e^s.e)|_{x_1}^{x_2} & = & \lambda x_4 : e^s|_{x_1}^{x_2\,s}.e|_{x_3}^{x_4}|_{x_1}^{x_2} \qquad x_3 \not\equiv x_1 \\
 & & x_4 = \begin{cases} x_3 & x_3 \not\equiv x_2 \text{ or } x_1 \notin FV(e) \\ x_f & \text{for } x_f \notin \{\, x_2 \,\} \cup FV(e) \end{cases} \\
(e_1\ e_2)|_{x_1}^{x_2} & = & e_1|_{x_1}^{x_2}\ e_2|_{x_1}^{x_2} \\
c|_{x_1}^{x_2} & = & c \\
f|_{x_1}^{x_2} & = & f \\
(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)|_{x_1}^{x_2} & = & \textbf{if } e_1|_{x_1}^{x_2} \textbf{ then } e_2|_{x_1}^{x_2} \textbf{ else } e_3|_{x_1}^{x_2} \\
(\sim e^s)|_{x_1}^{x_2} & = & \sim e^s|_{x_1}^{x_2\,s}
\end{array}
$$

FIGURE 9. Code Variable Renaming

$\boxed{FMV^s(e^s) : e^s \to X}$

$$
\begin{array}{lcl}
FMV^s(x) & = & \{\, x \,\} \\
FMV^s(\alpha) & = & \emptyset \\
FMV^s(c) & = & \emptyset \\
FMV^s(f) & = & \emptyset \\
FMV^s(c^s) & = & \emptyset \\
FMV^s(\lambda x : \tau^s.e^s) & = & FMV^s(e^s) - \{\, x \,\} \\
FMV^s(e_1^s\ e_2^s) & = & FMV^s(e_1^s) \cup FMV^s(e_2^s) \\
FMV^s(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s) & = & FMV^s(e_1^s) \cup FMV^s(e_2^s) \cup FMV^s(e_3^s) \\
FMV^s(\Lambda\alpha.e^s) & = & FMV^s(e^s) \\
FMV^s(e^s[\tau^\alpha]) & = & FMV^s(e^s) \\
FMV^s(\prec e \succ) & = & FMV(e) \\
FMV^s(\%e^s) & = & FMV^s(e^s) \\
FMV^s(e_1^s \to e_2^s) & = & FMV^s(e_1^s) \cup FMV^s(e_2^s) \\
FMV^s(e_1^s = e_2^s) & = & FMV^s(e_1^s) \cup FMV^s(e_2^s)
\end{array}
$$

$\boxed{FMV(e) : e \to X}$

$$
\begin{array}{lcl}
FMV(x) & = & \emptyset \\
FMV(c) & = & \emptyset \\
FMV(f) & = & \emptyset \\
FMV(\lambda x : e^s.e) & = & FMV^s(e^s) \cup FMV(e) \\
FMV(e_1\ e_2) & = & FMV(e_1) \cup FMV(e_2) \\
FMV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & FMV(e_1) \cup FMV(e_2) \cup FMV(e_3) \\
FMV(\sim e^s) & = & FMV^s(e^s)
\end{array}
$$

FIGURE 10. Free Meta Variables

$\boxed{FCV^s(e^s) : e^s \to X}$

$$
\begin{array}{lcl}
FCV^s(x) & = & \emptyset \\
FCV^s(\alpha) & = & \emptyset \\
FCV^s(c) & = & \emptyset \\
FCV^s(f) & = & \emptyset \\
FCV^s(c^s) & = & \emptyset \\
FCV^s(\lambda x : \tau^s.e^s) & = & FCV^s(e^s) \\
FCV^s(e_1^s\ e_2^s) & = & FCV^s(e_1^s) \cup FCV^s(e_2^s) \\
FCV^s(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s) & = & FCV^s(e_1^s) \cup FCV^s(e_2^s) \cup FCV^s(e_3^s) \\
FCV^s(\Lambda\alpha.e^s) & = & FCV^s(e^s) \\
FCV^s(e^s[\tau^\alpha]) & = & FCV^s(e^s) \\
FCV^s(\prec e \succ) & = & FCV(e) \\
FCV^s(\%e^s) & = & FCV^s(e^s) \\
FCV^s(e_1^s \to e_2^s) & = & FCV^s(e_1^s) \cup FCV^s(e_2^s) \\
FCV^s(e_1^s = e_2^s) & = & FCV^s(e_1^s) \cup FCV^s(e_2^s)
\end{array}
$$

$\boxed{FCV(e) : e \to X}$

$$
\begin{array}{lcl}
FCV^s(x) & = & \{\, x \,\} \\
FCV^s(c) & = & \emptyset \\
FCV^s(f) & = & \emptyset \\
FCV^s(\lambda x : e^s.e) & = & FCV^s(e^s) \cup (FCV(e) - \{\, x \,\}) \\
FCV(e_1\ e_2) & = & FCV(e_1) \cup FCV(e_2) \\
FCV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & FCV(e_1) \cup FCV(e_2) \cup FCV(e_3) \\
FCV(\sim e^s) & = & FCV^s(e^s)
\end{array}
$$

FIGURE 11. Free Code Variables

$\boxed{FTV^s(e^s) : e^s \to X}$

$$
\begin{array}{lcl}
FTV^s(x) & = & \emptyset \\
FTV^s(\alpha) & = & \{\, \alpha \,\} \\
FTV^s(c) & = & \emptyset \\
FTV^s(f) & = & \emptyset \\
FTV^s(c^s) & = & \emptyset \\
FTV^s(\lambda x : \tau^s.e^s) & = & FTV^s(\tau^s) \cup FTV^s(e^s) \\
FTV^s(e_1^s\ e_2^s) & = & FTV^s(e_1^s) \cup FTV^s(e_2^s) \\
FTV^s(\textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s) & = & FTV^s(e_1^s) \cup FTV^s(e_2^s) \cup FTV^s(e_3^s) \\
FTV^s(\Lambda\alpha.e^s) & = & FTV^s(e^s) - \{\, \alpha \,\} \\
FTV^s(e^s[\tau^\alpha]) & = & FTV^s(e^s) \cup FTV^s(\tau^\alpha) \\
FTV^s(\prec e \succ) & = & FTV(e) \\
FTV^s(\%e^s) & = & FTV^s(e^s) \\
FTV^s(e_1^s \to e_2^s) & = & FTV^s(e_1^s) \cup FTV^s(e_2^s) \\
FTV^s(e_1^s = e_2^s) & = & FTV^s(e_1^s) \cup FTV^s(e_2^s)
\end{array}
$$

$\boxed{FTV(e) : e \to X}$

$$
\begin{array}{lcl}
FTV^s(x) & = & \emptyset \\
FTV^s(c) & = & \emptyset \\
FTV^s(f) & = & \emptyset \\
FTV^s(\lambda x : e^s.e) & = & FTV^s(e^s) \cup FTV(e) \\
FTV(e_1\ e_2) & = & FTV(e_1) \cup FTV(e_2) \\
FTV(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & FTV(e_1) \cup FTV(e_2) \cup FTV(e_3) \\
FTV(\sim e^s) & = & FTV^s(e^s)
\end{array}
$$

FIGURE 12. Free Type Variables

$\boxed{FV(e) : e \to X \cup \mathcal{A}}$
$\boxed{FV^s(e^s) : e^s \to X \cup \mathcal{A}}$

$$
\begin{array}{lcl}
FV(e) & = & FCV(e) \cup FMV(e) \cup FTV(e) \\
FV^s(e^s) & = & FCV^s(e^s) \cup FMV^s(e^s) \cup FTV^s(e^s)
\end{array}
$$

FIGURE 13. Free Variables

$\boxed{E^* \leq E^*}$ Context Prefix

$$\frac{E_1^* = E_2^*}{E_1^* \leq E_2^*} \qquad \frac{E_1^* < E_2^*}{E_1^* \leq E_2^*} \qquad \frac{E_1^* < E_2^* \quad E_2^* <_1 E_3^*}{E_1^* < E_3^*} \qquad \frac{E_1^* <_1 E_2^*}{E_1^* < E_2^*}$$

$$\frac{}{E^s <_1 E^s[\Box\ e^s]} \qquad \frac{}{E^s <_1 E^s[v^s\ \Box]} \qquad \frac{}{E <_1 E[\lambda x : \Box.e]}$$

$$\frac{}{E^s < E^s[\textbf{if } \Box \textbf{ then } e^s \textbf{ else } e^s]} \qquad \frac{}{E <_1 E[\sim\Box]} \qquad \frac{}{E^s <_1 E^s[\%\Box]}$$

$$\frac{}{E^s <_1 E^s[\Box \to e^s]} \qquad \frac{}{E^s <_1 E^s[\tau \to \Box]} \qquad \frac{}{E^s <_1 E^s[\Box = e^s]}$$

$$\frac{}{E^s <_1 E^s[v^s = \Box]} \qquad \frac{}{E^s <_1 E^s[\Box[\tau^\alpha]]} \qquad \frac{}{E <_1 E[\Box\ e]} \qquad \frac{}{E <_1 E[e^o\ \Box]}$$

$$\frac{}{E <_1 E[\lambda x : \tau.\Box]} \qquad \frac{}{E <_1 E[\textbf{if } \Box \textbf{ then } e \textbf{ else } e]}$$

$$\frac{}{E <_1 E[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e]} \qquad \frac{}{E <_1 E[\textbf{if } e^o \textbf{ then } e^o \textbf{ else } \Box]}$$

$$\frac{}{E^s <_1 E^s[\prec\Box\succ]}$$

FIGURE 14. Context Prefix Relations

LEMMA 2 (Inversion of Typing and Well-Formedness).

(1) *If* $\Gamma \vdash x : \tau^s$, *then* $(x : \tau^s) \in \Gamma$.
(2) *If* $\Gamma \vdash \alpha : \tau^s$, *then* $\tau^s = \textbf{\textit{type}}$ *and* $(\alpha : *) \in \Gamma$.
(3) *If* $\Gamma \vdash \textbf{\textit{dom}} : \tau^s$, *then* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{type}}$.
(4) *If* $\Gamma \vdash \textbf{\textit{cod}} : \tau^s$, *then* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{type}}$.
(5) *If* $\Gamma \vdash \gamma^? : \tau^s$, *then* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{bool}}$.
(6) *If* $\Gamma \vdash \to^? : \tau^s$, *then* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{bool}}$.
(7) *If* $\Gamma \vdash \textbf{\textit{typeof}}\ : \tau^s$, *then* $\tau^s = \textbf{\textit{code}} \to \textbf{\textit{type}}$.
(8) *If* $\Gamma \vdash c : \tau^s$, *then* $\tau^s = \gamma$ *for some* γ.
(9) *If* $\Gamma \vdash f : \tau^s$, *then* $\tau^s = \gamma_1 \to \gamma_2$ *for some* γ_1 *and* γ_2.
(10) *If* $\Gamma \vdash \textbf{\textit{if}}\ e_1^s\ \textbf{\textit{then}}\ e_2^s\ \textbf{\textit{else}}\ e_3^s : \tau^s$ *then* $\Gamma \vdash e_1^s : \textbf{\textit{bool}}$, $\Gamma \vdash e_2^s : \tau^s$, *and* $\Gamma \vdash e_3^s : \tau^s$.
(11) *If* $\Gamma \vdash \gamma : \tau^s$, *then* $\tau^s = \textbf{\textit{type}}$.
(12) *If* $\Gamma \vdash \prec e \succ : \tau^s$, *then* $\tau^s = \textbf{\textit{code}}$ *and* $\Gamma \vdash e\ \textbf{\textit{wf}}$.
(13) *If* $\Gamma \vdash e_1^s \to e_2^s : \tau^s$, *then* $\tau^s = \textbf{\textit{type}}$ *and* $\Gamma \vdash e_1^s : \textbf{\textit{type}}$ *and* $\Gamma \vdash e_2^s : \textbf{\textit{type}}$.
(14) *If* $\Gamma \vdash e_1^s =_\tau e_2^s : \tau^s$, *then* $\tau^s = \textbf{\textit{bool}}$ *and* $\Gamma \vdash e_1^s : \textbf{\textit{type}}$ *and* $\Gamma \vdash e_2^s : \textbf{\textit{type}}$.
(15) *If* $\Gamma \vdash \%e^s : \tau^s$ *then* $\tau^s = \textbf{\textit{code}}$ *and* $\Gamma \vdash e^s : \gamma$.
(16) *If* $\Gamma \vdash e_1^s\ e_2^s : \tau^s$, *then there is a type* $\tau^{s'}$ *such that* $\Gamma \vdash e_1^s : \tau^{s'} \to \tau^s$ *and* $\Gamma \vdash e_2^s : \tau^{s'}$.
(17) *If* $\Gamma \vdash \lambda x : \tau_1^s . e^s : \tau^s$, *then there is a type* $\tau^{s'}$ *such that* $\Gamma, x : \tau_1^s \vdash e^s : \tau^{s'}$ *and* $\tau^s = \tau_1^s \to \tau^{s'}$; $x \notin FV(\Gamma)$.
(18) *If* $\Gamma \vdash \Lambda\alpha . e^s : \tau^s$, *then for some* $\tau^{s'}$, $\tau^s = \forall\alpha . \tau^{s'}$ *and* $\Gamma, \alpha : * \vdash e^s : \tau^{s'}$; $\alpha \notin FV(\Gamma)$.
(19) *If* $\Gamma \vdash e_1^s[\tau^\alpha] : \tau^s$ *Then there is a type* $\forall\alpha . \tau^{s'}$ *such that* $\Gamma \vdash e_1^s : \forall\alpha . \tau^{s'}$ *and* $\tau^s = \tau^{s'}\{\tau^\alpha/\alpha\}^\tau$.
(20) *If* $\Gamma \vdash x\ \textbf{\textit{wf}}$, *then* $(x : \textbf{\textit{dyn}}) \in \Gamma$.
(21) *If* $\Gamma \vdash \lambda x : e^s . e\ \textbf{\textit{wf}}$, *then* $\Gamma \vdash e^s : \textbf{\textit{type}}$ *and* $\Gamma, x : \textbf{\textit{dyn}} \vdash e\ \textbf{\textit{wf}}$; $x \notin FV(\Gamma)$.
(22) *If* $\Gamma \vdash e_1\ e_2\ \textbf{\textit{wf}}$, *then* $\Gamma \vdash e_1\ \textbf{\textit{wf}}$ *and* $\Gamma \vdash e_2\ \textbf{\textit{wf}}$.
(23) *If* $\Gamma \vdash \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3\ \textbf{\textit{wf}}$ *then* $\Gamma \vdash e_1\ \textbf{\textit{wf}}$, $\Gamma \vdash e_2\ \textbf{\textit{wf}}$, *and* $\Gamma \vdash e_3\ \textbf{\textit{wf}}$.
(24) *If* $\Gamma \vdash \sim e^s\ \textbf{\textit{wf}}$, *then* $\Gamma \vdash e^s : \textbf{\textit{code}}$.

PROOF. By cases on derivations of $\Gamma \vdash e^s : \tau^s$ and $\Gamma \vdash e\ \mathbf{wf}$. □

LEMMA 3 (Context Typing).

(1) *If* $\Gamma \vdash E[e]\ \textbf{\textit{wf}}$, *then* $\Gamma' \vdash e\ \textbf{\textit{wf}}$ *for some* Γ'.
(2) *If* $\Gamma \vdash E^s[e^s]\ \textbf{\textit{wf}}$, *then* $\Gamma' \vdash e^s : \tau^s$ *for some* Γ'.

PROOF. By mutual induction on the structure of E and E^s.

CASE ($E = \Box$). $\Gamma \vdash \Box[e]\ \ \mathbf{wf} = \Gamma \vdash e\ \ \mathbf{wf}$.

CASE ($E = E'[\Box\ e']$).

$$\Gamma \vdash (E'[\Box\ e'])[e]\ \mathbf{wf} = \Gamma \vdash E'[e\ e']\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash e\ e'\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e\ \mathbf{wf}.$$

CASE ($E = E'[e^o\ \Box]$).

$$\Gamma \vdash (E'[e^o\ \Box])[e]\ \mathbf{wf} = \Gamma \vdash E'[e^o\ e]\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash e^o\ e\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e\ \mathbf{wf}.$$

CASE ($E = E'[\lambda x : \tau . \Box]$).

$$\Gamma \vdash (E'[\lambda x : \tau . \Box])[e]\ \mathbf{wf} = \Gamma \vdash E'[\lambda x : \tau . e]\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$

$$\Gamma' \vdash \lambda x : \tau . e\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow} \Gamma', x : \mathbf{dyn} \vdash e\ \mathbf{wf}.$$

Case $(E = E'[\textbf{if } \Box \textbf{ then } e \textbf{ else } e])$.

$$\Gamma \vdash (E'[\textbf{if } \Box \textbf{ then } e_1 \textbf{ else } e_2])[e] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash \textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2 \textbf{ wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e \textbf{ wf}.$$

Case $(E = E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e])$.

$$\Gamma \vdash (E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e'])[e] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e^o \textbf{ then } e \textbf{ else } e'] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash \textbf{if } e^o \textbf{ then } e \textbf{ else } e' \textbf{ wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e \textbf{ wf}.$$

Case $(E = E'[\textbf{if } e^o \textbf{ then } e^o \textbf{ else } \Box])$.

$$\Gamma \vdash (E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])[e] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash \textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e \textbf{ wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e \textbf{ wf}.$$

Case $(E = E^{s'}[\prec \Box \succ])$.

$$\Gamma \vdash (E^{s'}[\prec \Box \succ])[e] \textbf{ wf} = \Gamma \vdash E^{s'}[\prec e \succ] \textbf{ wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash \prec e \succ : \tau^s \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e \textbf{ wf}.$$

Case $(E^s = E^{s'}[\Box\ e^{s'}])$.

$$\Gamma \vdash (E^{s'}[\Box\ e^{s'}])[e^s] \textbf{ wf} = \Gamma \vdash E^{s'}[e^s\ e^{s'}] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash e^s\ e^{s'} : \tau^s \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \tau^{s'} \to \tau^s.$$

Case $(E^s = E^{s'}[v^s\ \Box])$.

$$\Gamma \vdash (E^{s'}[v^s\ \Box])[e^s] \textbf{ wf} = \Gamma \vdash E^{s'}[v^s\ e^s] \textbf{ wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash v^s\ e^s : \tau^s \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \tau^{s'}.$$

Case $(E^s = E'[\lambda x : \Box.e'])$.

$$\Gamma \vdash (E'[\lambda x : \Box.e'])[e^s] \textbf{ wf} = \Gamma \vdash E'[\lambda x : e^s.e'] \textbf{ wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash \lambda x : e^s.e' \textbf{ wf} \overset{lm\ 2}{\Rightarrow}$$
$$\Gamma' \vdash e^s : \textbf{type}.$$

Case $(E^s = E^{s'}[\textbf{if } \Box \textbf{ then } e_1^s \textbf{ else } e_2^s])$.

$$\Gamma \vdash E^{s'}[\textbf{if } \Box \textbf{ then } e_1^s \textbf{ else } e_2^s][e^s] \textbf{ wf} = \Gamma \vdash E^{s'}[\textbf{if } e^s \textbf{ then } e_1^s \textbf{ else } e_2^s] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash \textbf{if } e^s \textbf{ then } e_1^s \textbf{ else } e_2^s : \tau^s \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \textbf{bool}.$$

Case $(E^s = E'[\sim \Box])$.

$$\Gamma \vdash (E'[\sim \Box])[e^s] \textbf{ wf} = \Gamma \vdash E'[\sim e^s] \textbf{ wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash \sim e^s \textbf{ wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \textbf{code}.$$

Case $(E^s = E^{s'}[\%\Box])$.

$$\Gamma \vdash (E^{s'}[\%\Box])[e^s] \textbf{ wf} = \Gamma \vdash E^{s'}[\%e^s] \textbf{ wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash \%e^s \textbf{ wf} \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \gamma.$$

Case $(E^s = E^{s'}[\Box \to e^s])$.

$$\Gamma \vdash (E^{s'}[\Box \to e^{s'}])[e^s] \textbf{ wf} = \Gamma \vdash E^{s'}[e^s \to e^{s'}] \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash e^s \to e^{s'} : \tau^s \overset{lm\ 2}{\Rightarrow} \Gamma' \vdash e^s : \textbf{type}.$$

Case ($E^s = E^{s'}[\tau \to \square]$).

$$\Gamma \vdash (E^{s'}[\tau \to \square])[e^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[\tau \to e^s]\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash \tau \to e^s : \tau^s \overset{lm}{\Rightarrow}{}^2\ \Gamma' \vdash e^s : \mathbf{type}.$$

Case ($E^s = E^{s'}[\square =_\tau e^s]$).

$$\Gamma \vdash (E^{s'}[\square =_\tau e^{s'}])[e^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[e^s =_\tau e^{s'}]\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash e^s =_\tau e^{s'} : \tau^s \overset{lm}{\Rightarrow}{}^2\ \Gamma' \vdash e^s : \mathbf{type}.$$

Case ($E^s = E^{s'}[v^s =_\tau \square]$).

$$\Gamma \vdash (E^{s'}[v^s =_\tau \square])[e^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[v^s =_\tau e^s]\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma' \vdash v^s =_\tau e^s : \tau^s \overset{lm}{\Rightarrow}{}^2\ \Gamma' \vdash e^s : \mathbf{type}.$$

Case ($E^s = E^{s'}[\square[\tau^\alpha]]$).

$$\Gamma \vdash (E^{s'}[\square[\tau^\alpha]])[e^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[e^s[\tau^\alpha]]\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma' \vdash e^s[\tau^\alpha] : \tau^s \overset{lm}{\Rightarrow}{}^2$$
$$\Gamma' \vdash e^s : \forall\alpha.\tau^{s'}.$$

□

Lemma 4 (Replacement).

(1) *If* $\Gamma \vdash E[e_1]$ ***wf****, and* $\Gamma' \vdash e_1$ ***wf*** *implies* $\Gamma' \vdash e_2$ ***wf*** *for all* Γ'*, then* $\Gamma \vdash E[e_2]$ ***wf****.*
(2) *If* $\Gamma \vdash E^s[e_1^s]$ ***wf****, and* $\Gamma' \vdash e_1^s$ ***wf*** *implies* $\Gamma' \vdash e_2^s$ ***wf*** *for all* Γ'*, then* $\Gamma \vdash E^s[e_2^s]$ ***wf****.*

Proof. By mutual induction on the structure of E and E^s.

Case ($E = \square$). $\Gamma \vdash \square[e_1]\ \mathbf{wf} = \Gamma \vdash e_1\ \mathbf{wf}$ and by assumption, $\Gamma \vdash e_2\ \mathbf{wf}$.

Case ($E = E'[\square\ e]$).

$$\Gamma \vdash (E'[\square\ e])[e_1]\ \mathbf{wf} = \Gamma \vdash E'[e_1\ e]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^3\ \Gamma' \vdash e_1\ e\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^2\ \Gamma' \vdash e_1\ \mathbf{wf};\ \Gamma' \vdash e\ \mathbf{wf}$$
$$\overset{hyp}{\Rightarrow} \Gamma' \vdash e_2\ \mathbf{wf} \Rightarrow \Gamma' \vdash e_2\ e\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma \vdash E'[e_2\ e]\ \mathbf{wf} = \Gamma \vdash (E'[\square\ e])[e_2]\ \mathbf{wf}.$$

Case ($E = E'[e^o\ \square]$).

$$\Gamma \vdash (E'[e^o\ \square])[e_1]\ \mathbf{wf} = \Gamma \vdash E'[e^o\ e_1]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^3\ \Gamma' \vdash e^o\ e_1\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^2\ \Gamma' \vdash e^o\ \mathbf{wf};\ \Gamma' \vdash e_1\ \mathbf{wf}$$
$$\overset{hyp}{\Rightarrow} \Gamma' \vdash e_2\ \mathbf{wf} \Rightarrow \Gamma' \vdash e^o\ e_2\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma \vdash E'[e^o\ e_2]\ \mathbf{wf} = \Gamma \vdash (E'[e^o\ \square])[e_2]\ \mathbf{wf}$$

Case ($E = E'[\lambda x : \tau.\square]$).

$$\Gamma \vdash (E'[\lambda x : \tau.\square])[e_1]\ \mathbf{wf} = \Gamma \vdash E'[\lambda x : \tau.e_1]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^3$$
$$\Gamma' \vdash \lambda x : \tau.e_1\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^2 \Gamma', x : \mathbf{dyn} \vdash e_1\ \mathbf{wf} \overset{hyp}{\Rightarrow} \Gamma', x : \mathbf{dyn} \vdash e_2\ \mathbf{wf} \Rightarrow$$
$$\Gamma' \vdash \lambda x : \tau.e_2\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma \vdash E'[\lambda x : \tau.e_2]\ \mathbf{wf} = \Gamma \vdash (E'[\lambda x : \tau.\square])[e_2]\ \mathbf{wf}$$

CASE $(E = E'[\textbf{if } \Box \textbf{ then } e_3 \textbf{ else } e_4])$.

$$\Gamma \vdash (E'[\textbf{if } \Box \textbf{ then } e_3 \textbf{ else } e_4])[e_1] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e_1 \textbf{ then } e_3 \textbf{ else } e_4] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3}$$
$$\Gamma' \vdash \textbf{if } e_1 \textbf{ then } e_3 \textbf{ else } e_4 \textbf{ wf} \overset{lm}{\Rightarrow}{}^{2} \Gamma' \vdash e_1 \textbf{ wf};\ \Gamma' \vdash e_3 \textbf{ wf};\ \Gamma' \vdash e_4 \textbf{ wf} \overset{hyp}{\Rightarrow}$$
$$\Gamma' \vdash e_2 \textbf{ wf} \Rightarrow \Gamma' \vdash \textbf{if } e_2 \textbf{ then } e_3 \textbf{ else } e_4 \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash E'[\textbf{if } e_2 \textbf{ then } e_3 \textbf{ else } e_4] \textbf{ wf} = \Gamma \vdash (E'[\textbf{if } \Box \textbf{ then } e_3 \textbf{ else } e_4])[e_2] \textbf{ wf}$$

CASE $(E = E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e])$.

$$\Gamma \vdash (E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e_3])[e_1] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e^o \textbf{ then } e_1 \textbf{ else } e_3] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3}$$
$$\Gamma' \vdash \textbf{if } e^o \textbf{ then } e_1 \textbf{ else } e_3 \textbf{ wf} \overset{lm}{\Rightarrow}{}^{2} \Gamma' \vdash e^o \textbf{ wf};\ \Gamma' \vdash e_1 \textbf{ wf};\ \Gamma' \vdash e_3 \textbf{ wf} \overset{hyp}{\Rightarrow}$$
$$\Gamma' \vdash e_2 \textbf{ wf} \Rightarrow \Gamma' \vdash \textbf{if } e^o \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash E'[\textbf{if } e^o \textbf{ then } e_2 \textbf{ else } e_3] \textbf{ wf} = \Gamma \vdash (E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e_3])[e_2] \textbf{ wf}$$

CASE $(E = E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])$.

$$\Gamma \vdash (E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])[e_1] \textbf{ wf} = \Gamma \vdash E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e_1] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3}$$
$$\Gamma' \vdash \textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e_1 \textbf{ wf} \overset{lm}{\Rightarrow}{}^{2} \Gamma' \vdash e_1^o \textbf{ wf};\ \Gamma' \vdash e_2^o \textbf{ wf};\ \Gamma' \vdash e_1 \textbf{ wf} \overset{hyp}{\Rightarrow}$$
$$\Gamma' \vdash e_2 \textbf{ wf} \Rightarrow \Gamma' \vdash \textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e_2 \textbf{ wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e_2] \textbf{ wf} = \Gamma \vdash (E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])[e_2] \textbf{ wf}$$

CASE $(E = E^{s'}[\prec \Box \succ])$.

$$\Gamma \vdash (E^{s'}[\prec \Box \succ])[e_1] \textbf{ wf} = \Gamma \vdash E^{s'}[\prec e_1 \succ] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3} \Gamma' \vdash \prec e_1 \succ : \textbf{code} \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash e_1 \textbf{ wf} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2 \textbf{ wf} \Rightarrow$$
$$\Gamma' \vdash \prec e_2 \succ : \textbf{code} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[\prec e_2 \succ] \textbf{ wf} = \Gamma \vdash (E^{s'}[\prec \Box \succ])[e_2] \textbf{ wf}$$

CASE $(E^s = E^{s'}[\Box\ e^s])$.

$$\Gamma \vdash (E^{s'}[\Box\ e^s])[e_1^s] \textbf{ wf} = \Gamma \vdash E^{s'}[e_1^s\ e^s] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3} \Gamma' \vdash e_1^s\ e^s : \tau^s \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash e_1^s : \tau^{s'} \to \tau^s;\ \Gamma' \vdash e^s : \tau^{s'} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \tau^{s'} \to \tau^s \Rightarrow$$
$$\Gamma' \vdash e_2^s\ e^s : \tau^s \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[e_1^s\ e^s] \textbf{ wf} = \Gamma \vdash (E^{s'}[\Box\ e^s])[e_1^s] \textbf{ wf}$$

CASE $(E^s = E^{s'}[v^s\ \Box])$.

$$\Gamma \vdash (E^{s'}[v^s\ \Box])[e_1^s] \textbf{ wf} = \Gamma \vdash E^{s'}[v^s\ e_1^s] \textbf{ wf} \overset{lm}{\Rightarrow}{}^{3} \Gamma' \vdash v^s\ e_1^s : \tau^s \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash v^s : \tau^{s'} \to \tau^s;\ \Gamma' \vdash e_1^s : \tau^{s'} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \tau^{s'} \Rightarrow$$
$$\Gamma' \vdash v^s\ e_2^s : \tau^s \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[v^s\ e_2^s] \textbf{ wf} = \Gamma \vdash (E^{s'}[v^s\ \Box])[e_2^s] \textbf{ wf}$$

Case $(E^s = E'[\lambda x : \Box.e])$.

$$\Gamma \vdash (E'[\lambda x : \Box.e])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E'[\lambda x : e_1^s.e]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash \lambda x : e_1^s.e\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash e\ \mathbf{wf};\ \Gamma' \vdash e_1^s : \mathbf{type} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{type} \Rightarrow$$
$$\Gamma' \vdash \lambda x : e_2^s.e\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma \vdash E'[\lambda x : e_2^s.e]\ \mathbf{wf} = \Gamma \vdash (E'[\lambda x : \Box.e])[e_2^s]\ \mathbf{wf}.$$

Case $(E^s = E^s[\mathbf{if}\ \Box\ \mathbf{then}\ e^s\ \mathbf{else}\ e^s])$.

$$\Gamma \vdash (E'[\mathbf{if}\ \Box\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E'[\mathbf{if}\ e_1^s\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}$$
$$\Gamma' \vdash \mathbf{if}\ e_1^s\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s : \tau^s \overset{lm}{\Rightarrow}{}^{2} \Gamma' \vdash e_1^s : \mathbf{bool};\ \Gamma' \vdash e_3^s : \tau^s;\ \Gamma' \vdash e_4^s : \tau^s \overset{hyp}{\Rightarrow}$$
$$\Gamma' \vdash e_2^s : \tau^s \Rightarrow \Gamma' \vdash \mathbf{if}\ e_2^s\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s : \tau^s \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash E'[\mathbf{if}\ e_2^s\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s]\ \mathbf{wf} = \Gamma \vdash (E'[\mathbf{if}\ \Box\ \mathbf{then}\ e_3^s\ \mathbf{else}\ e_4^s])[e_2^s]\ \mathbf{wf}$$

Case $(E^s = E'[\sim\Box])$.

$$\Gamma \vdash (E'[\sim\Box])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E'[\sim e_1^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash \sim e_1^s\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{2}\ \Gamma' \vdash e_1^s : \mathbf{code}$$
$$\overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{code} \Rightarrow \Gamma' \vdash \sim e_2^s\ \mathbf{wf} \overset{I.H.}{\Rightarrow} \Gamma \vdash E'[\sim e_2^s]\ \mathbf{wf} = \Gamma \vdash (E'[\sim\Box])[e_2^s]\ \mathbf{wf}.$$

Case $(E^s = E^{s'}[\%\Box])$.

$$\Gamma \vdash (E^{s'}[\%\Box])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[\%e_1^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash \%e_1^s : \mathbf{code} \overset{lm}{\Rightarrow}{}^{2}\ \Gamma' \vdash e_1^s : \gamma$$
$$\overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \gamma \Rightarrow \Gamma' \vdash \%e_2^s : \mathbf{code} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[\%e_2^s]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[\%\Box])[e_2^s]\ \mathbf{wf}.$$

Case $(E^s = E^{s'}[\Box \to e^s])$.

$$\Gamma \vdash (E^{s'}[\Box \to e^s])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[e_1^s \to e^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash e_1^s \to e^s : \mathbf{type} \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash e_1^s : \mathbf{type};\ \Gamma' \vdash e^s : \mathbf{type} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{type} \Rightarrow$$
$$\Gamma' \vdash e_2^s \to e^s : \mathbf{type} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[e_2^s \to e^s]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[\Box \to e^s])[e_2^s]\ \mathbf{wf}.$$

Case $(E^s = E^{s'}[\tau \to \Box])$.

$$\Gamma \vdash (E^{s'}[\tau \to \Box])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[\tau \to e_1^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash \tau \to e_1^s : \mathbf{type} \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash \tau : \mathbf{type};\ \Gamma' \vdash e_1^s : \mathbf{type} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{type} \Rightarrow$$
$$\Gamma' \vdash \tau \to e_2^s : \mathbf{type} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[\tau \to e_2^s]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[\tau \to \Box])[e_2^s]\ \mathbf{wf}.$$

Case $(E^s = E^{s'}[\Box =_\tau e^s])$.

$$\Gamma \vdash (E^{s'}[\Box =_\tau e^s])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[e_1^s =_\tau e^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash e_1^s =_\tau e^s : \mathbf{bool} \overset{lm}{\Rightarrow}{}^{2}$$
$$\Gamma' \vdash e_1^s : \mathbf{type};\ \Gamma' \vdash e^s : \mathbf{type} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{type} \Rightarrow$$
$$\Gamma' \vdash e_2^s =_\tau e^s : \mathbf{bool} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[e_2^s =_\tau e^s]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[\Box =_\tau e^s])[e_2^s]\ \mathbf{wf}.$$

Case ($E^s = E^{s'}[v^s =_\tau \square]$).

$$\Gamma \vdash (E^{s'}[v^s =_\tau \square])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[v^s =_\tau e_1^s]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash v^s =_\tau e_1^s : \mathbf{bool} \overset{lm}{\Rightarrow}{}^{2}$$

$$\Gamma' \vdash v^s : \mathbf{type};\ \Gamma' \vdash e_1^s : \mathbf{type} \overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \mathbf{type} \Rightarrow$$

$$\Gamma' \vdash v^s =_\tau e_2^s : \mathbf{bool} \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[v^s =_\tau e_2^s]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[v^s =_\tau \square])[e_2^s]\ \mathbf{wf}.$$

Case ($E^s = E^{s'}[\square[\tau^\alpha]]$).

$$\Gamma \vdash (E^{s'}[\square[\tau^\alpha]])[e_1^s]\ \mathbf{wf} = \Gamma \vdash E^{s'}[e_1^s[\tau^\alpha]]\ \mathbf{wf} \overset{lm}{\Rightarrow}{}^{3}\ \Gamma' \vdash e_1^s[\tau^\alpha] : \tau^s \overset{lm}{\Rightarrow}{}^{2}\ \Gamma' \vdash e_1^s : \forall\alpha.\tau^{s'}$$

$$\overset{hyp}{\Rightarrow} \Gamma' \vdash e_2^s : \forall\alpha.\tau^{s'} \Rightarrow \Gamma' \vdash e_2^s[\tau^\alpha] : \tau^s \overset{I.H.}{\Rightarrow} \Gamma \vdash E^{s'}[e_2^s[\tau^\alpha]]\ \mathbf{wf} = \Gamma \vdash (E^{s'}[\square[\tau^\alpha]])[e_2^s]\ \mathbf{wf}.$$

□

Lemma 5 (Canonical Forms).

(1) *If* $\Gamma \vdash v^s :$ ***type*** *then* $v^s = \tau$.
(2) *If* $\Gamma \vdash v^s : \tau_1^s \to \tau_2^s$ *then* $v^s = \lambda x : \tau_1^s.e^s$ *or* $v^s = f$ *or* $v^s = c^{s^-}$.
(3) *If* $\Gamma \vdash v^s : \forall\alpha.\tau^s$ *then* $v^s = \Lambda\alpha.e^s$.
(4) *If* $\Gamma \vdash v^s :$ ***code*** *then* $v^s = {\prec} e^o {\succ}$.
(5) *If* $\Gamma \vdash v^s : \gamma$ *then* $v^s = c$.

Proof. By induction on the rules of $\Gamma \vdash e^s : \tau^s$. To give a flavor of the proof, consider Item 1. Most rules are vacuously true. For example,

Case ($\Gamma \vdash x : \tau^s$). $x \neq v^s$ so this case is vacuous.

Only two rules are not vacuously true:

Case ($\Gamma \vdash \gamma :$ **type**). $\Gamma \vdash \gamma :$ **type**. $\gamma = \tau$.

Case ($\Gamma \vdash e_1^s \to e_2^s :$ **type**). By the induction hypothesis, $e_1^s = \tau_1$ and $e_2^s = \tau_2$. Thus $\tau_1 \to \tau_2$ is both a value and a τ.

□

Lemma 6 (Decomposition of Well-Formed Terms).

(1) *Let* e *be a term of the code language,* E *a code context, and* Γ *an environment. Then if* $\Gamma \vdash E[e]$ ***wf****, one of the following is true:*
   (a) e *is pure code* (e^o).
   (b) *There is a redex* r *and an evaluation context* $E^{s'}$ *such that* $E[e] = E^{s'}[r]$.
   (c) *There is a term variable* x *and an evaluation context* $E^{s'}$ *such that* $E[e] = E^{s'}[x]$.
   (d) *There is a type variable* α *and an evaluation context* $E^{s'}$ *such that* $E[e] = E^{s'}[\alpha]$.
   (e) *There is a pure code term* e^o *and a code context* E' *such that* $E[e] \equiv E'[{\sim}{\prec} e^o {\succ}]$.
(2) *Let* e^s *be a term of the metalanguage,* E^s *an evaluation context, and* Γ *an environment. Then if* $\Gamma \vdash E^s[e^s]$ ***wf****, one of the following is true:*
   (a) e^s *is a metalanguage value* (v^s).
   (b) *There is a redex* r *and an evaluation context* $E^{s'}$ *such that* $E^s[e^s] = E^{s'}[r]$.

(c) *There is a term variable x and an evaluation context $E^{s'}$ such that $E^{s'}[e^s] = E^{s'}[x]$.*
(d) *There is a type variable α and an evaluation context $E^{s'}$ such that $E^s[e^s] = E^{s'}[\alpha]$.*
(e) *There is a pure code term e^o and a code context E' such that $E^s[e^s] = E'[\sim\prec e^o \succ]$.*

PROOF. By mutual induction over the structure of e and e^s.

CASE ($e = x$). e is pure code.

CASE ($e = c$). e is pure code.

CASE ($e = f$). e is pure code.

CASE ($e = \lambda x : e^s.e'$).

- If $e = \lambda x : \tau.e^o$, then e is pure code.
- Suppose e^s is not a value. Let $E^{s'} = E[\lambda x : \Box.e']$. Then $E[e] = E^{s'}[e^s]$ and by the induction hypothesis the case is true.
- Suppose e^s is a value and e' is not pure code. Then by Lemma 2and Lemma 5, $e^s = \tau$. Let $E' = E[\lambda x : \tau.\Box]$. Then $E[e] = E'[e']$ and by the induction hypothesis, the case is true.

CASE ($e = e_1\ e_2$).

- Suppose e_1 is not pure code. Let $E' = E[\Box\ e_2]$. Then $E[e] = E'[e_1]$, and by the induction hypothesis, the case is true.
- Suppose e_1 is pure code and e_2 is not. Let $E' = E[e_1\ \Box]$. Then $E[e] = E'[e_2]$, and by the induction hypothesis, the case is true.

CASE ($e =$ **if** e_1 **then** e_2 **else** e_3).

- Suppose e_1 is not pure code. Let $E' = E[\textbf{if}\ \Box\ \textbf{then}\ e_2\ \textbf{else}\ e_3]$. Then $E[e] = E'[e_1]$, and by the induction hypothesis the case is true.
- Suppose e_1 is pure code and e_2 is not. Let $E' = E[\textbf{if}\ e_1\ \textbf{then}\ \Box\ \textbf{else}\ e_3]$. Then $E[e] = E'[e_2]$, and by the induction hypothesis the case is true.
- Suppose e_1 and e_2 are pure code and e_3 is not. Let $E' = E[\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ \Box]$. Then $E[e] = E'[e_3]$, and by the induction hypothesis the case is true.

CASE ($e = \sim e^s$).

- Suppose e^s is a value. Then by Lemma 2 and Lemma 5, $e^s = \prec e^o \succ$, and $E[e] = E[\sim\prec e^o \succ]$.
- Suppose e^s is not a value. Let $E^{s'} = E[\sim\Box]$. Then $E[e] = E^{s'}[e^s]$. and by the induction hypothesis, the case is true.

CASE ($e^s = x$). $E^s[e^s] = E^s[x]$.

CASE ($e^s = \alpha$). $E^s[e^s] = E^s[\alpha]$.

CASE ($e^s = c$). e^s is a value.

CASE ($e^s = f$). e^s is a value.

CASE ($e^s = c^s$). e^s is a value.

Case ($e^s = \lambda x : \tau^s . e^s$). e^s is a value.

Case ($e^s = e_1^s\ e_2^s$). By Lemma 3, $\Gamma' \vdash e_1^s\ e_2^s : \tau^s$ for some τ^s, so by Lemma 2, $\Gamma' \vdash e_1^s : \tau^{s'} \to \tau^s$ and $\Gamma' \vdash e_2^s : \tau^{s'}$.

- Suppose e_1^s is not a value. Let $E^{s'} = E^s[\Box\ e_2^s]$. Then $E^s[e^s] = E^{s'}[e_1^s]$ and by the induction hypothesis the case is true.
- Suppose e_1^s is a value and e_2^s is not. Let $E^{s'} = E^s[e_1^s\ \Box]$. Then $E^s[e^s] = E^{s'}[e_2^s]$ and by the induction hypothesis, the case is true.
- Suppose that both e_1^s and e_2^s are values. Then by Lemma 5, $e_1^s\ e_2^s$ is a redex r and $E^s[e^s] = E^s[r]$.

Case ($e^s = \textbf{if}\ e_1^s\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s$).

- Suppose e_1^s is not a value. Then let $E^{s'} = E^s[\textbf{if}\ \Box\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s]$. Then $E^s[e^s] = E^{s'}[e_1^s]$, and by the induction hypothesis the case is true.
- Suppose e_1^s is a value. Then e^s is a redex.

Case ($e^s = \Lambda\alpha . e^s$). e^s is a value.

Case ($e^s = e^{s'}[\tau^\alpha]$).

- Suppose $e^{s'}$ is not a value. Then Let $E^{s'} = E^s[\Box[\tau^\alpha]]$. Then $E^s[e^s] = E^{s'}[e^{s'}]$, and by the induction hypothesis, the case is true.
- Suppose $e^{s'}$ is a value. Then by Lemmas 3,2, and 5, e^s is a redex r and $E^s[e^s] = E^s[r]$.

Case ($e^s = \prec e \succ$).

- Suppose e is pure code. Then e^s is a value.
- Suppose e is not pure code. Then let $E' = E^s[\prec \Box \succ]$. Then $E^s[e^s] = E'[e]$ and by the induction hypothesis, the case is true.

Case ($e^s = \%e^{s'}$). By Lemma 3 and Lemma 2, $\Gamma' \vdash e^s : \gamma$ for some gamma.

- Suppose $e^{s'}$ is a value. Then by Lemma 5, $e^s = c$ and so e^s is a redex r and $E^s[e^s] = E^s[r]$.
- Suppose $e^{s'}$ is not a value. Then let $E^{s'} = E^s[\%\Box]$. Then $E^s[e^s] = E^{s'}[e^{s'}]$ and by the induction hypothesis, the case is true.

Case ($e^s = e_1^s \to e_2^s$). By Lemma 3 and Lemma 2, $\Gamma' \vdash e_1^s : \textbf{type}$ and $\Gamma' \vdash e_2^s : \textbf{type}$.

- Suppose e_1^s is not a value. Then let $E^{s'} = E^s[\Box \to e_2^s]$. Then $E^s[e^s] = E^{s'}[e_1^s]$ and by the induction hypothesis, the case is true.
- Suppose e_1^s is a value and e_2^s is not. By Lemma 5, $e_1^s = \tau$, so let $E^{s'} = E^s[e_1^s \to \Box]$. Then $E^s[e^s] = E^{s'}[e_2^s]$ and by the induction hypothesis, the case is true.
- Suppose both e_1^s and e_2^s are values. Then by Lemma 5, $e_1^s = \tau_1$ and $e_2^s = \tau_2$, so e^s is a value.

Case ($e^s = e_1^{s'} =_\tau e_2^{s'}$). By Lemma 3 and Lemma 2, $\Gamma' \vdash e_1^s : \textbf{type}$ and $\Gamma' \vdash e_2^s : \textbf{type}$.

- Suppose e_1^s is not a value. Then let $E^{s'} = E^s[\Box =_\tau e_2^s]$. Then $E^s[e^s] = E^{s'}[e_1^s]$ and by the induction hypothesis, the case is true.
- Suppose e_1^s is a value and e_2^s is not. Then let $E^{s'} = E^s[e_1^s =_\tau \Box]$ Then $E^s[e^s] = E^{s'}[e_2^s]$ and by the induction hypothesis, the case is true.

- Suppose both e_1^s and e_2^s are values. Then by Lemma 5, $e_1^s = \tau_1$ and $e_2^s = \tau_2$, so e^s is a redex.

□

To prove unique decomposition, we appeal to the set of *potentially stuck terms*, a subset of metalanguage terms.

Potentially Stuck Terms

$$pt \quad ::= \quad x \mid \alpha \mid v^s\ v^s \mid v^s[\tau^\alpha] \mid \%v^s \mid v^s =_\tau v^s$$

Note that the productions for pt are mutually exclusive.

LEMMA 7. *If $\Gamma \vdash pt : \tau^s$ and pt is not a variable x or α, then pt is a redex.*

PROOF. By case analysis on potentially stuck terms. Each case follows from Lemma 5. □

LEMMA 8. *The set of potentially stuck terms pt and metalanguage values v^s are disjoint.*

PROOF. By case analysis on the structure of pt. □

LEMMA 9.

(1) *If $E^s[v^s] = E^{s'}[e^s]$ where $E^s < E^{s'}$, then e^s is a value v^s.*
(2) *If $E^s[v^s] = E'[e]$ where $E^s < E'$, then e is pure code e^o.*
(3) *If $E[e^o] = E'[e]$ where $E < E'$, then e is pure code e^o.*
(4) *If $E[e^o] = E^{s'}[e^s]$ where $E < E'$, then e^s is a value v^s.*

PROOF. By induction on the structure of v^s and e^o.

CASE ($e^o = x$). There is no extension of $E'[x]$.

CASE ($e^o = f$). There is no extension of $E'[f]$.

CASE ($e^o = c$). There is no extension of $E'[c]$.

CASE ($e^o = e_1^o\ e_2^o$). There are two possible immediate extensions of E: $E[e^o] = (E[\Box\ e_2^o])[e_1^o]$ and $E[e^o] = (E[e_1^o\ \Box])[e_2^o]$. By the induction hypothesis, both cases extend as in the lemma.

CASE ($e^o = \lambda x : \tau.e^o$). There are two immediate extensions of E: $E[e^o] = (E[\lambda x : \Box.e^o])[\tau]$ and $E[e^o] = (E[\lambda x : \tau.\Box])[e^o]$. By the induction hypothesis, both cases extend as in the lemma.

CASE ($v^s = c$). There is no extension of $E^s[c]$.

CASE ($v^s = f$). There is no extension of $E^s[f]$.

CASE ($v^s = c^s$). There is no extension of $E^s[c^s]$.

CASE ($v^s = \prec e^o \succ$). There is one immediate extension of E^s: $E^s[v^s] = (E^s[\prec \Box \succ])[e^o]$. The induction hypothesis completes this case.

CASE ($v^s = \tau$). This case is by induction on the structure of τ. If $\tau = \gamma$, then there is no extension of $E^s[\gamma]$. If $\tau = \tau_1 \to \tau_2$ then there are two immediate extensions of E: $E^s[v^s] = (E^s[\Box \to \tau_2])[\tau_1]$ and $E^s[v^s] = (E^s[\tau_1 \to \Box])[\tau_2]$. By induction, each of these cases extends as per the lemma.

Case ($v^s = \lambda x : \tau^s.e^s$). There is no extension of $E^s[\lambda x : \tau^s.e^s]$.

Case ($v^s = \Lambda\alpha.e^s$). There is no extension of $E^s[\Lambda\alpha.e^s]$.

□

Lemma 10.

(1) *If* $E^s[pt] = E^{s'}[e^s]$ *where* $E^s < E^{s'}$, *then* e^s *is a value* v^s.
(2) *If* $E^s[pt] = E'[e]$ *where* $E^s < E'$, *then* e *is pure code* e^o.
(3) *If* $E[\sim\prec e^o \succ] = E'[e]$ *where* $E' < E$, *then* e *is pure code* e^o.
(4) *If* $E[\sim\prec e^o \succ] = E^{s'}[e^s]$ *where* $E < E^{s'}$, *then* e^s *is a value* v^s.

Proof. Every immediate decomposition of a *pt* focuses on a value; the only immediate decomposition of a $\sim\prec e^o \succ$ focuses on pure code. The rest follows by Lemma 9. □

Corollary 1. *If* $e = E^s[pt]$ *or* $e = E[\sim\prec e^o \succ]$, *then there is no decomposition that extends the given context and leads to either kind of term.*

Lemma 11. *Suppose* $E_1^s[e_1^s] = E^s[pt]$ *where* $E_1^s < E^s$, *and* $E_1^s[e_1^s] = E_2^s[e_2^s]$ *where* $E_1^s <_1 E_2^s$ *but* $E_2^s \not< E^s$. *Then* e_2^s *is a value* v^s. *Analogous results are true for each combination of* $E_1[e_1]$, $E_2[e_2]$ *and* $E[\sim\prec e^o \succ]$ *in place of* $E_1^s[e_1^s]$, $E_2^s[e_2^s]$, *and* $E^s[pt]$ *respectively.*

Proof. If $E^s < E_2^s$ etc., then by Lemma 9, e_2^s must be a value, etc. The proof is by cases on the structure of E_1^s and E_1.

Case ($E_1 = \Box$). All decompositions of a term have $\Box$ as a prefix, so this case is vacuously true.

Case ($E_1 = E'[\Box\ e]$). Then $E_1[e_1] = E'[e_1\ e]$. Then for E_2 to not be a prefix of E, $E_2 = E'[e^o\ \Box]$. But this is impossible because by Lemma 9, e_1 cannot be a value since $E_1 < E$.

Case ($E_1 = E'[e^o\ \Box]$). Then $E_1[e_1] = E'[e^o\ e_1]$. Then for E_2 to not be a prefix of E, $E_2 = E^s[\Box\ e_1]$. But since $E_1[e_1] = E_2[e^o]$, then by Lemma 9, any extension of E_2 will focus on a value.

Case ($E_1 = E'[\lambda x : \tau.\Box]$). Then $E_1[e_1] = E'[\lambda x : \tau.e_1]$. To not be a prefix of E, $E_2^s = E'[\lambda x : \Box.e_1]]$. But since $E_1[e_1] = E_2^s[\tau]$, then by Lemma 9, any extension of E_2^s focuses on a value.

Case ($E = E'[\textbf{if}\ \Box\ \textbf{then}\ e_2\ \textbf{else}\ e_3]$). Then $E_1[e_1] = E'[\textbf{if}\ e_1\ \textbf{then}\ e\ \textbf{else}\ e]$. Then to not be a prefix of E, $E_2 = E'[\textbf{if}\ \Box\ \textbf{then}\ e_2\ \textbf{else}\ e_3]$, but this is impossible because by Lemma 9, e_1 cannot be a value.

Case ($E = E'[\textbf{if}\ e^o\ \textbf{then}\ \Box\ \textbf{else}\ e]$). Then $E_1[e_1] = E'[\textbf{if}\ e^o\ \textbf{then}\ e_1\ \textbf{else}\ e_2]]$. Then to not be a prefix of E, $E_2 = E'[\textbf{if}\ \Box\ \textbf{then}\ e_2\ \textbf{else}\ e_3]$, but since $E_1[e_1] = E_2[e^o]$, then by Lemma 9, any extension of E_2 focuses on a value.

Case ($E = E'[\textbf{if}\ e^o\ \textbf{then}\ e^o\ \textbf{else}\ \Box]$). Then $E_1[e_1] = E'[\textbf{if}\ e_1^o\ \textbf{then}\ e_2^o\ \textbf{else}\ e_1]$. Then to not be a prefix of E, $E_2 = E'[\textbf{if}\ \Box\ \textbf{then}\ e_2^o\ \textbf{else}\ e_1]$ or $E_2 = E'[\textbf{if}\ e_1^o\ \textbf{then}\ \Box\ \textbf{else}\ e_1]$ but since $E_1[e_1] = E_2[e_1^o]$ or $E_1[e_1] = E_2[e_2^o]$, then by Lemma 9, any extension of E_2 focuses on a value.

Case ($E_1 = E^{s'}[\prec\Box\succ]$).

Case ($E_1^s = E^{s'}[\Box\ e^s]$). Then $E_1^s[e_1^s] = E^{s'}[e_1^s\ e^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[v^s\ \Box]$. But this is impossible because by Lemma 9, e_1^s cannot be a value.

Case ($E_1^s = E^{s'}[v^s\ \Box]$). Then $E_1^s[e_1^s] = E^{s'}[v^s\ e_1^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\Box\ e_1^s]$. But since $E_1^s[e_1^s] = E_2^{s'}[v^s]$, then by Lemma 9, any extension of $E_2^{s'}$ will focus on a value.

Case ($E_1^s = E'[\lambda x : \Box.e]$). Then $E_1^s[e_1^s] = E'[\lambda x : e_1^s.e]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\Box\ e_1^s]$. But since $E_1^s[e_1^s] = E_2^{s'}[v^s]$, then by Lemma 9, any extension of $E_2^{s'}$ will focus on a value.

Case ($E^s = E^s[\textbf{if}\ \Box\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s]$). Then $E_1^s[e_1^s] = E^s[\textbf{if}\ e_1^s\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s]$. There is no immediate extension of E_1^s that is not extended by E^s.

Case ($E_1^s = E'[\sim\Box]$). Then $E_1^s[e_1^s] = E'[\sim e_1^s]$. There is no immediate extension of E_1^s that is not extended by E^s.

Case ($E_1^s = E^{s'}[\%\Box]$). Then $E_1^s[e_1^s] = E'[\%e_1^s]$. There is no immediate extension of E_1^s that is not extended by E^s.

Case ($E_1^s = E^{s'}[\Box \to e^s]$). Then $E_1^s[e_1^s] = E^{s'}[e_1^s \to e^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\tau \to\ \Box]$. But this is impossible because by Lemma 9, e_1^s cannot be a value.

Case ($E_1^s = E^{s'}[\tau \to \Box]$). Then $E_1^s[e_1^s] = E^{s'}[\tau \to e_1^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\Box \to e_1^s]$. But since $E_1^s[e_1^s] = E_2^{s'}[\tau]$, then by Lemma 9, any extension of $E_2^{s'}$ will focus on a value.

Case ($E_1^s = E^{s'}[\Box =_\tau e^s]$). Then $E_1^s[e_1^s] = E^{s'}[e_1^s =_\tau e^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\tau =_\tau\ \Box]$. But this is impossible because by Lemma 9, e_1^s cannot be a value.

Case ($E_1^s = E^{s'}[v^s =_\tau \Box]$). Then $E_1^s[e_1^s] = E^{s'}[v^s =_\tau e_1^s]$. Then for E_2^s to not be a prefix of E^s, $E_2^s = E^s[\Box =_\tau e_1^s]$. But since $E_1^s[e_1^s] = E_2^{s'}[v^s]$, then by Lemma 9, any extension of $E_2^{s'}$ will focus on a value.

Case ($E_1^s = E^{s'}[\Box[\tau^\alpha]]$). Then $E_1^s[e_1^s] = E^{s'}[e_1^s[\tau^\alpha]]$. There is no immediate extension of E_1^s that is not extended by E^s.

□

Lemma 12 (Unique Decomposition). *If $\Gamma \vdash e$ **wf**, and e is not proper code, then e can be uniquely decomposed into one and only one of $E^s[x]$, $E^s[\alpha]$, $E^s[r]$, or $E[\sim\prec e^o\succ]$.*

Proof. Lemma 6 proves that a decomposition exists. Lemma 11 proves that no other such decomposition exists. Lemma 3 and Lemma 7 prove that non-variable decompositions are redexes. □

Lemma 13.

(1) *For all meta contexts E^s and e^s, $FMV^s(e^s) \subseteq FMV(E^s[e^s])$;*
(2) *For all code contexts E and e, $FMV(e) \subseteq FMV(E[e])$*

Proof. By mutual induction on the structure of E and E^s.

Case ($E = \Box$). $FMV(e) = FMV(\Box[e])$.

Case $(E = E'[\Box\ e'])$.

$$FMV(e) \subseteq FMV(e\ e') = FMV(e) \cup FMV(e') \overset{I.H.}{\subseteq} FMV(E'[e\ e']) = FMV((E'[\Box\ e'])[e]).$$

Case $(E = E'[e^o\ \Box])$.

$$FMV(e) \subseteq FMV(e^o\ e) = FMV(e^o) \cup FMV(e) \overset{I.H.}{\subseteq} FMV(E'[e^o\ e]) = FMV((E'[e^o\ \Box])[e]).$$

Case $(E = E'[\lambda x : \tau.\Box])$.

$$FMV(e) \subseteq FMV(\lambda x : \tau.e) = FMV^s(\tau) \cup FMV(e) \overset{I.H.}{\subseteq} FMV(E'[\lambda x : \tau.e]) = FMV((E'[\lambda x : \tau.\Box])[e]).$$

Case $(E = E'[\textbf{if } \Box \textbf{ then } e_2 \textbf{ else } e_3])$.

$$FMV(e) \subseteq FMV(\textbf{if } e \textbf{ then } e_2 \textbf{ else } e_3) = FMV(e) \cup FMV(e_1) \cup FMV(e_3) \overset{I.H.}{\subseteq} FMV(E'[\textbf{if } e \textbf{ then } e_2 \textbf{ else } e_3]) = FMV((E'[\textbf{if } \Box \textbf{ then } e_2 \textbf{ else } e_3])[e])$$

Case $(E = E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e_2])$.

$$FMV(e) \subseteq FMV(\textbf{if } e^o \textbf{ then } e \textbf{ else } e_2) = FMV(e^o) \cup FMV(e) \cup FMV(e_2) \overset{I.H.}{\subseteq} FMV(E'[\textbf{if } e^o \textbf{ then } e \textbf{ else } e_2]) = FMV((E'[\textbf{if } e^o \textbf{ then } \Box \textbf{ else } e_2])[e])$$

Case $(E = E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])$.

$$FMV(e) \subseteq FMV(\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e) = FMV(e_1^o) \cup FMV(e_2^o) \cup FMV(e) \overset{I.H.}{\subseteq} FMV(E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } e]) = FMV((E'[\textbf{if } e_1^o \textbf{ then } e_2^o \textbf{ else } \Box])[e])$$

Case $(E = E^{s'}[\prec \Box \succ])$.

$$FMV(e) = FMV^s(\prec e \succ) \overset{I.H.}{\subseteq} FMV(E^{s'}[\prec e \succ]) = FMV((E^{s'}[\prec \Box \succ])[e]).$$

Case $(E^s = E^{s'}[\Box\ e^s])$.

$$FMV^s(e^s) \subseteq FMV^s(e^s\ e^{s'}) = FMV^s(e^s) \cup FMV^s(e^{s'}) \overset{I.H.}{\subseteq} FMV(E^{s'}[e^s\ e^{s'}]) = FMV((E^{s'}[\Box\ e^{s'}])[e^s]).$$

Case $(E^s = E^{s'}[v^s\ \Box])$.

$$FMV^s(e^s) \subseteq FMV^s(v^s\ e^s) = FMV^s(v^s) \cup FMV^s(e^s) \overset{I.H.}{\subseteq} FMV(E^{s'}[v^s\ e^s]) = FMV((E^{s'}[v^s\ \Box])[e^s]).$$

Case $(E^s = E'[\lambda x : \Box.e'])$.

$$FMV^s(e^s) \subseteq FMV^s(\lambda x : e^s.e) = FMV^s(e^s) \cup FMV(e) \overset{I.H.}{\subseteq} FMV(E'[\lambda x : e^s.e]) = FMV((E'[\lambda x : \Box.e])[e^s]).$$

Case ($E^s = E^s[\textbf{if } \Box \textbf{ then } e_2^s \textbf{ else } e_3^s]$).

$$FMV^s(e^s) \subseteq FMV^s(\textbf{if } e^s \textbf{ then } e_2^s \textbf{ else } e_3^s) =$$
$$FMV^s(e^s) \cup FMV^s(e_1^s) \cup FMV^s(e_3^s) \overset{I.H.}{\subseteq}$$
$$FMV^s(E'[\textbf{if } e^s \textbf{ then } e_2^s \textbf{ else } e_3^s]) = FMV^s((E'[\textbf{if } \Box \textbf{ then } e_2^s \textbf{ else } e_3^s])[e^s])$$

Case ($E^s = E'[\sim\Box]$).

$$FMV^s(e^s) = FMV(\sim e^s) \overset{I.H.}{\subseteq} FMV(E'[\sim e^s]) = FMV((E'[\sim\Box])[e^s]).$$

Case ($E^s = E^{s'}[\%\Box]$).

$$FMV^s(e^s) = FMV^s(\%e^s) \overset{I.H.}{\subseteq} FMV(E^{s'}[\%e^s]) = FMV((E^{s'}[\%\Box])[e^s]).$$

Case ($E^s = E^{s'}[\Box \to e^s]$).

$$FMV^s(e^s) \subseteq FMV^s(e^s \to e^{s'}) = FMV^s(e^s) \cup FMV^s(e^{s'}) \overset{I.H.}{\subseteq}$$
$$FMV(E^{s'}[e^s \to e^{s'}]) = FMV((E^{s'}[\Box \to e^{s'}])[e^s]).$$

Case ($E^s = E^{s'}[\tau \to \Box]$).

$$FMV^s(e^s) \subseteq FMV^s(\tau \to e^s) = FMV^s(\tau) \cup FMV^s(e^s) \overset{I.H.}{\subseteq}$$
$$FMV(E^{s'}[\tau \to e^s]) = FMV((E^{s'}[\tau \to \Box])[e^s]).$$

Case ($E^s = E^{s'}[\Box =_\tau e^s]$).

$$FMV^s(e^s) \subseteq FMV^s(e^s =_\tau e^{s'}) = FMV^s(e^s) \cup FMV^s(e^{s'}) \overset{I.H.}{\subseteq}$$
$$FMV(E^{s'}[e^s =_\tau e^{s'}]) = FMV((E^{s'}[\Box =_\tau e^{s'}])[e^s]).$$

Case ($E^s = E^{s'}[v^s =_\tau \Box]$).

$$FMV^s(e^s) \subseteq FMV^s(v^s =_\tau e^s) = FMV^s(v^s) \cup FMV^s(e^s) \overset{I.H.}{\subseteq}$$
$$FMV(E^{s'}[v^s =_\tau e^s]) = FMV((E^{s'}[v^s =_\tau \Box])[e^s]).$$

Case ($E^s = E^{s'}[\Box[\tau^\alpha]]$).

$$FMV^s(e^s) = FMV^s(e^s[\tau^\alpha]) \overset{I.H.}{\subseteq} FMV(E^{s'}[e^s[\tau^\alpha]]) = FMV((E^{s'}[\Box[\tau^\alpha]])[e^s]).$$

□

Corollary 2. *If e is meta variable-closed then*

(1) *If $e = E^s[e^s]$ then e^s is meta variable-closed.*
(2) *If $e = E[e]$ then e is meta variable-closed.*

Lemma 14.

(1) *For all meta contexts E^s and e^s, $FTV^s(e^s) \subseteq FTV(E^s[e^s])$;*
(2) *For all code contexts E and e, $FTV(e) \subseteq FTV(E[e])$*

Proof. Same form as Lemma 13. □

COROLLARY 3. *If e is type variable-closed then*

(1) *If $e = E^s[e^s]$ then e^s is type variable-closed.*
(2) *If $e = E[e]$ then e is type variable-closed.*

LEMMA 15.

(1) *If e is closed and $e = E[e']$ then $FCV(e') \subseteq \Gamma(E)$.*
(2) *If e is closed and $e = E^s[e^{s'}]$ then $FCV(e^{s'}) \subseteq \Gamma(E^s)$.*

PROOF. By mutual induction on the structure of E and E^s. □

THEOREM 8 (Progress).

(1) *If e is closed, $\Gamma \vdash e$ **wf**, then e is proper code or there is some e' such that $e \longmapsto e'$ or $e \longmapsto \bot$.*

PROOF. Lemma 14 and Lemma 13 prove that e cannot be decomposed to a meta variable or type variable. Then by Lemma 12, e can be decomposed to a redex $E^s[r]$ or $E[\sim\prec e^o \succ]$. □

LEMMA 16 (Permutation).

(1) *If $\Gamma \vdash e$ **wf** and Γ' is a well-formed permutation of Γ, then $\Gamma' \vdash e$ **wf**.*
(2) *If $\Gamma \vdash e^s : \tau^s$ and Γ' is a well-formed permutation of Γ, then $\Gamma' \vdash e^s \mid \tau^s$.*

PROOF. By mutual induction on the structure of e and e^s. The only interesting cases are for $e = x$, $e^s = x$, and $e^s = \alpha$, and are immediate. □

LEMMA 17 (Weakening).

(1) *If $\Gamma \vdash e$ **wf** and $x \notin FV(\Gamma)$, then $\Gamma, x : \tau^s \vdash e$ **wf**.*
(2) *If $\Gamma \vdash e^s : \tau^s$ and $x \notin FV(\Gamma)$, then $\Gamma, x : \tau^s \vdash e^s : \tau^s$.*

*Analogous results apply for $x :$ **dyn** and $\alpha : *$*

PROOF. By mutual induction on the structure of e and e^s. □

LEMMA 18 (Term Substitution).

(1) *If $\Gamma, x : \tau_1^s \vdash e$ **wf** and $\Gamma \vdash v^s : \tau_1^s$, then $\Gamma \vdash e\,[v^s/x]$ **wf**.*
(2) *If $\Gamma, x : \tau_1^s \vdash e^s : \tau_2^s$ and $\Gamma \vdash v^s : \tau_1^s$, then $\Gamma \vdash e^s\,[v^s/x]^s : \tau_2^s$.*

PROOF. By mutual induction over the structure of e and e^s.

CASE ($e = x$). Since $x\,[v^s/x] = x$, this case is immediate.

CASE ($e = c$). Since $c\,[v^s/x] = c$, this case is immediate.

CASE ($e = \lambda x' : e^s.e'$). Assume wolog that $x' \notin FMV(v^s)$.

$$\Gamma, x : \tau_1^s \vdash \lambda x' : e^s.e' \ \mathbf{wf} \overset{lm}{\Rightarrow} 2$$

$$x' \notin FV(\Gamma, x : \tau_1^s);\ \Gamma, x : \tau_1^s \vdash e^s : \mathbf{type};\ \Gamma, x : \tau_1^s, x' : \mathbf{dyn} \vdash e' \ \mathbf{wf} \overset{lm}{\Rightarrow} 16$$

$$\Gamma, x' : \mathbf{dyn}, x : \tau_1^s \vdash e' \ \mathbf{wf} \overset{lm}{\Rightarrow} 17\ \Gamma, x' : \mathbf{dyn} \vdash v^s \vdash \tau_1^s \overset{I.H.}{\Rightarrow}$$

$$\Gamma \vdash e^s\,\{v^s/x\}^s : \mathbf{type};\ \Gamma, x' : \mathbf{dyn} \vdash e'\,\{v^s/x\}\ \mathbf{wf} \Rightarrow$$

$$\Gamma \vdash \lambda x' : (e^s\,\{v^s/x\}^s).(e'\,\{v^s/x\})\ \mathbf{wf} = \Gamma \vdash (\lambda x' : e^s.e')\,\{v^s/x\}\ \mathbf{wf}$$

CASE $(e = e_1\ e_2)$.

$$\Gamma, x : \tau_1^s \vdash e_1\ e_2\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e_1\ \mathbf{wf};\ \Gamma, x : \tau_1^s \vdash e_2\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash e_1\,[v^s/x]\ \mathbf{wf};\ \Gamma \vdash e_2\,[v^s/x]\ \mathbf{wf} \Rightarrow$$
$$\Gamma \vdash (e_1\,[v^s/x])\ (e_2\,[v^s/x])\ \mathbf{wf} = \Gamma \vdash (e_1\ e_2)\,[v^s/x]\ \mathbf{wf}.$$

CASE $(e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)$.

$$\Gamma, x : \tau^s \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow}$$
$$\Gamma, x : \tau^s \vdash e_1\ \mathbf{wf};\ \Gamma, x : \tau^s \vdash e_2\ \mathbf{wf};\ \Gamma, x : \tau^s \vdash e_3\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash e_1\,[v^s/x]\ \mathbf{wf};\ \Gamma \vdash e_2\,[v^s/x]\ \mathbf{wf};\ \Gamma \vdash e_3\,[v^s/x]\ \mathbf{wf} \Rightarrow$$
$$\Gamma \vdash \mathbf{if}\ e_1\,[v^s/x]\ \mathbf{then}\ e_2\,[v^s/x]\ \mathbf{else}\ e_3\,[v^s/x]\ \mathbf{wf} =$$
$$\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3)\,[v^s/x]\ \mathbf{wf}$$

CASE $(e = {\sim} e^s)$.

$$\Gamma, x : \tau_1^s \vdash {\sim} e^s\ \mathbf{wf} \overset{lm\ 2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e^s : \mathbf{code}$$
$$\overset{I.H.}{\Rightarrow} \Gamma \vdash e^s\,[v^s/x]^s : \mathbf{code} \Rightarrow \Gamma \vdash {\sim} e^s\,[v^s/x]^s\ \mathbf{wf}$$
$$= \Gamma \vdash ({\sim} e^s)\,[v^s/x]\ \mathbf{wf}.$$

CASE $(e^s = x')$. Suppose $x' \equiv x$. Then $\tau_1^s = \tau_2^s$ and $\Gamma, x : \tau_1^s \vdash x\,[v^s/x]^s : \tau_1^s = \Gamma, x : \tau_1^s \vdash v^s : \tau_1^s$, which is true by assumption. Suppose $x' \not\equiv x$. Then $x'\,[v^s/x]^s = x'$ and the result is immediate.

CASE $(e^s = \alpha)$. Since $\alpha\,[v^s/x]^s = \alpha$, this case is immediate.

CASE $(e^s = c)$. Since $c\,[v^s/x]^s = c$, this case is immediate.

CASE $(e^s = c^s)$. Since $c^s\,[v^s/x]^s = c^s$, this case is immediate.

CASE $(e^s = \lambda x' : \tau_3^s.e^s)$. Assume wolog that $x' \notin FMV(v^s)$. Then

$$\Gamma, x : \tau_1^s \vdash \lambda x' : \tau_3^s.e^s : \tau_3^s \to \tau_4^s.\ \overset{lm\ 2}{\Rightarrow} x' \notin FV(\Gamma, x : \tau_1^s);\ \Gamma, x : \tau_1^s, x' : \tau_3^s \vdash e^s : \tau_4^s \overset{lm\ 16}{\Rightarrow}$$
$$\Gamma, x' : \tau_3^s, x : \tau_1^s \vdash e^s : \tau_4^s \overset{lm\ 17}{\Rightarrow} \Gamma, x' : \tau_3^s \vdash v^s : \tau_1^s \overset{I.H.}{\Rightarrow}$$
$$\Gamma, x' : \tau_3^s \vdash e^s\,[v^s/x]^s : \tau_4^s \Rightarrow$$
$$\Gamma \vdash \lambda x' : \tau_3^s.e^s\,[v^s/x]^s : \tau_3^s \to \tau_4^s = \Gamma \vdash (\lambda x' : \tau_3^s.e^s)\,[v^s/x]^s : \tau_3^s \to \tau_4^s$$

CASE $(e^s = e_1^s\ e_2^s)$.

$$\Gamma, x : \tau_1^s \vdash e_1^s\ e_2^s : \tau_2^s \overset{lm\ 2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e_1^s : \tau^{s'} \to \tau_2^s;\ \Gamma, x : \tau_1^s \vdash e_2^s : \tau^{s'}$$
$$\overset{I.H.}{\Rightarrow} \Gamma \vdash e_1^s\,[v^s/x]^s : \tau^{s'} \to \tau_2^s;\ \Gamma \vdash e_2^s\,[v^s/x]^s : \tau^{s'}$$
$$\Rightarrow \Gamma \vdash (e_1^s\,[v^s/x]^s)\ (e_2^s\,[v^s/x]^s) : \tau_2^s$$
$$= \Gamma \vdash (e_1^s\ e_2^s)\,[v^s/x]^s : \tau_2^s$$

CASE ($e^s = \textbf{if } e_1^s \textbf{ then } e_2^s \textbf{ else } e_3^s$).

$$\Gamma, x : \tau^s \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ wf} \overset{lm.2}{\Rightarrow}$$

$$\Gamma, x : \tau^s \vdash e_1 : \textbf{bool}\, \Gamma, x : \tau^s \vdash e_2 : \tau^s;\ \Gamma, x : \tau^s \vdash e_3 : \tau^s \overset{I.H.}{\Rightarrow}$$

$$\Gamma \vdash e_1\,[v^s/x]^s : \textbf{bool};\ \Gamma \vdash e_2\,[v^s/x]^s : \tau^s;\ \Gamma \vdash e_3\,[v^s/x]^s : \tau^s \Rightarrow$$

$$\Gamma \vdash \textbf{if } e_1\,[v^s/x]^s \textbf{ then } e_2\,[v^s/x]^s \textbf{ else } e_3\,[v^s/x]^s : \tau^s =$$

$$\Gamma \vdash (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3)\,[v^s/x]^s : \tau^s$$

CASE ($e^s = \Lambda\alpha.e^s$).

$$\Gamma, x : \tau_1^s \vdash \Lambda\alpha.e^s : \tau_2^s \overset{lm.2}{\Rightarrow}$$

$$\alpha \notin FV(\Gamma, x : \tau_1^s);\ \tau_2^s = \forall\alpha.\tau^{s'};\ \Gamma, x : \tau_1^s, \alpha : * \vdash e^s : \tau^{s'} \overset{lm.16}{\Rightarrow}$$

$$\Gamma, \alpha : *, x : \tau_1^s \vdash e^s : \tau^{s'} \overset{lm.17}{\Rightarrow} \Gamma, \alpha : * \vdash v^s : \tau_1^s \overset{I.H.}{\Rightarrow} \Gamma, \alpha : * \vdash e^s\,[v^s/x]^s : \tau^{s'} \Rightarrow$$

$$\Gamma \vdash \Lambda\alpha.e^s\,[v^s/x]^s : \tau_2^s = \Gamma \vdash (\Lambda\alpha.e^s)\,[v^s/x]^s : \tau_2^s$$

CASE ($e^s = e^s[\tau^\alpha]$).

$$\Gamma, x : \tau_1^s \vdash e^s[\tau^\alpha] : \tau_2^s \overset{lm.2}{\Rightarrow}$$

$$\Gamma, x : \tau_1^s \vdash e^s : \forall\alpha.\tau^{s'};\ \tau_2^s = \tau^{s'}\,\{\tau^\alpha/\alpha\}^\tau \overset{I.H.}{\Rightarrow} \Gamma \vdash e^s\,[v^s/x]^s : \forall\alpha.\tau^{s'} \Rightarrow$$

$$\Gamma \vdash (e^s\,[v^s/x]^s)[\tau^\alpha] : \tau_2^s = \Gamma \vdash (e^s[\tau^\alpha])\,[v^s/x]^s : \tau_2^s$$

CASE ($e^s = \prec e \succ$).

$$\Gamma, x : \tau_1^s \vdash \prec e \succ : \textbf{code} \overset{lm.2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e \textbf{ wf} \overset{I.H.}{\Rightarrow}$$

$$\Gamma \vdash e\,[v^s/x] \textbf{ wf} \Rightarrow \Gamma \vdash \prec e\,[v^s/x] \succ : \textbf{code} = \Gamma \vdash \prec e \succ\,[v^s/x]^s : \textbf{code}$$

CASE ($e^s = \%e^s$).

$$\Gamma, x : \tau_1^s \vdash \%e^s : \textbf{code} \overset{lm.2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e^s : \gamma \overset{I.H.}{\Rightarrow}$$

$$\Gamma \vdash e^s\,[v^s/x]^s : \gamma \Rightarrow \Gamma \vdash \%e^s\,[v^s/x]^s : \gamma = \Gamma \vdash (\%e^s)\,[v^s/x]^s : \gamma$$

CASE ($e^s = e_1^s \to e_2^s$).

$$\Gamma, x : \tau_1^s \vdash e_1^s \to e_2^s : \textbf{type} \overset{lm.2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e_1^s : \textbf{type};\ \Gamma, x : \tau_1^s \vdash e_2^s : \textbf{type}$$

$$\overset{I.H.}{\Rightarrow} \Gamma \vdash e_1^s\,[v^s/x]^s : \textbf{type};\ \Gamma \vdash e_2^s\,[v^s/x]^s : \textbf{type}$$

$$\Rightarrow \Gamma \vdash (e_1^s\,[v^s/x]^s) \to (e_2^s\,[v^s/x]^s) : \textbf{type}$$

$$= \Gamma \vdash (e_1^s \to e_2^s)\,[v^s/x]^s : \textbf{type}$$

CASE ($e^s = e_1^s =_\tau e_2^s$).

$$\Gamma, x : \tau_1^s \vdash e_1^s =_\tau e_2^s : \textbf{bool} \overset{lm.2}{\Rightarrow} \Gamma, x : \tau_1^s \vdash e_1^s : \textbf{type};\ \Gamma, x : \tau_1^s \vdash e_2^s : \textbf{type}$$

$$\overset{I.H.}{\Rightarrow} \Gamma \vdash e_1^s\,[v^s/x]^s : \textbf{type};\ \Gamma \vdash e_2^s\,[v^s/x]^s : \textbf{type}$$

$$\Rightarrow \Gamma \vdash (e_1^s\,[v^s/x]^s) =_\tau (e_2^s\,[v^s/x]^s) : \textbf{bool} = \Gamma \vdash (e_1^s =_\tau e_2^s)\,[v^s/x]^s : \textbf{bool}$$

□

LEMMA 19 (Type Substitution).

(1) *If* $\Gamma, \alpha : *, \Gamma' \vdash e$ ***wf*** *and* $\Gamma \vdash \tau^\alpha :$ ***type****, then* $\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e \{\tau^\alpha/\alpha\}$ ***wf****.*

(2) *If* $\Gamma, \alpha : *, \Gamma' \vdash e^s : \tau_2^s$ *and* $\Gamma \vdash \tau^\alpha :$ ***type****, then* $\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e^s \{\tau^\alpha/\alpha\}^s : \tau_2^s \{\tau^\alpha/\alpha\}^\tau$.

PROOF. By mutual induction over the structure of e and e^s.

CASE ($e = x$). Since $x \{\tau^\alpha/\alpha\} = x$, this case is immediate.

CASE ($e = c$). Since $c \{\tau^\alpha/\alpha\} = c$, this case is immediate.

CASE ($e = \lambda x : e^s.e'$). Assume wolog that and $x \notin FMV^s(v^s)$.

$$\Gamma, \alpha : *, \Gamma' \vdash \lambda x : e^s.e' \ \mathbf{wf} \overset{lm_2}{\Rightarrow}$$

$$x \notin FV(\Gamma, \alpha : *, \Gamma');\ \Gamma, \alpha : *, \Gamma' \vdash e^s : \mathbf{type};\ \Gamma, \alpha : *, \Gamma', x : \mathbf{dyn} \vdash e' \ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e^s \{\tau^\alpha/\alpha\}^s : \mathbf{type};\ \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau), x : \mathbf{dyn} \vdash e' \{\tau^\alpha/\alpha\} \ \mathbf{wf} \Rightarrow$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \lambda x : (e^s \{\tau^\alpha/\alpha\}^s).(e' \{\tau^\alpha/\alpha\}) \ \mathbf{wf} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (\lambda x : e^s.e') \{\tau^\alpha/\alpha\} \ \mathbf{wf}$$

CASE ($e = e_1\ e_2$).

$$\Gamma, \alpha : *, \Gamma' \vdash e_1\ e_2 \ \mathbf{wf} \overset{lm_2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e_1 \ \mathbf{wf};\ \Gamma, \alpha : *, \Gamma' \vdash e_2 \ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_1 \{\tau^\alpha/\alpha\} \ \mathbf{wf};\ \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_2 \{\tau^\alpha/\alpha\} \ \mathbf{wf} \Rightarrow$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1 \{\tau^\alpha/\alpha\})\ (e_2 \{\tau^\alpha/\alpha\}) \ \mathbf{wf} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1\ e_2) \{\tau^\alpha/\alpha\} \ \mathbf{wf}.$$

CASE ($e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$).

$$\Gamma, \alpha : * \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \ \mathbf{wf} \overset{lm_2}{\Rightarrow}$$

$$\Gamma, \alpha : * \vdash e_1 \ \mathbf{wf};\ \Gamma, \alpha : * \vdash e_2 \ \mathbf{wf};\ \Gamma, \alpha : * \vdash e_3 \ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma \vdash e_1 \{\tau^\alpha/\alpha\} \ \mathbf{wf};\ \Gamma \vdash e_2 \{\tau^\alpha/\alpha\} \ \mathbf{wf};\ \Gamma \vdash e_3 \{\tau^\alpha/\alpha\} \ \mathbf{wf} \Rightarrow$$
$$\Gamma \vdash \mathbf{if}\ e_1 \{\tau^\alpha/\alpha\} \ \mathbf{then}\ e_2 \{\tau^\alpha/\alpha\} \ \mathbf{else}\ e_3 \{\tau^\alpha/\alpha\} \ \mathbf{wf} =$$
$$\Gamma \vdash (\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \{\tau^\alpha/\alpha\} \ \mathbf{wf}$$

CASE ($e = {\sim} e^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash {\sim} e^s \ \mathbf{wf} \overset{lm_2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e^s : \mathbf{code} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e^s \{\tau^\alpha/\alpha\}^s : \mathbf{code} \Rightarrow \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash {\sim} e^s \{\tau^\alpha/\alpha\}^s \ \mathbf{wf} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash ({\sim} e^s) \{\tau^\alpha/\alpha\} \ \mathbf{wf}.$$

CASE ($e^s = x$). $\Gamma, \alpha : *, \Gamma' \vdash x : \tau^s \overset{lm_2}{\Rightarrow} x : \tau^s \in (\Gamma, \alpha : *, \Gamma')$. By $(\Gamma, \alpha : *, \Gamma')$ **wf** $\alpha \in \tau^s \Rightarrow x : \tau^s \in \Gamma'$, and $\alpha \notin \tau^s \Rightarrow x : \tau^s \{\tau^\alpha/\alpha\}^\tau = \tau^s$, so $\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash x : \tau^s \{\tau^\alpha/\alpha\}^\tau$.

CASE ($e^s = \alpha'$). Suppose $\alpha' \equiv \alpha$. Then $\tau_1^s = \tau_2^s =$ **type** and $\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \alpha \{\tau^\alpha/\alpha\}^s :$ **type** $= \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \tau^\alpha :$ **type**, which is true by Lemma 17. Suppose $\alpha' \not\equiv \alpha$. Then $\alpha' \{\tau^\alpha/\alpha\}^s = \alpha'$ and $\alpha' : * \in \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau)$. It follows that $\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \alpha' :$ **type**.

CASE ($e^s = c$). Since $c\left\{\tau^\alpha/\alpha\right\}^s = c$, this case is immediate.

CASE ($e^s = c^s$). Since $c^s\left\{\tau^\alpha/\alpha\right\}^s = c^s$, and this case is immediate.

CASE ($e^s = \lambda x : \tau_3^s.e^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash \lambda x : \tau_3^s.e^s : \tau_2^s \overset{lm}{\Rightarrow} 2$$

$$x \notin FV(\Gamma, \alpha : *, \Gamma');\ \tau_2^s = \tau_3^s \to \tau^{s'};\ \Gamma, \alpha : *, \Gamma', x : \tau_3^s \vdash e^s : \tau^{s'} \overset{I.H.}{\Rightarrow}$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau), x : \tau_3^s\left\{\tau^\alpha/\alpha\right\}^\tau \vdash e^s\left\{\tau^\alpha/\alpha\right\}^s : \tau^{s'}\left\{\tau^\alpha/\alpha\right\}^\tau \Rightarrow$$

$$\Gamma \vdash \lambda x : \tau_3^s\left\{\tau^\alpha/\alpha\right\}^\tau.e^s\left\{\tau^\alpha/\alpha\right\}^s : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau =$$

$$\Gamma \vdash (\lambda x : \tau_3^s.e^s)\left\{\tau^\alpha/\alpha\right\}^s : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau.$$

CASE ($e^s = e_1^s\ e_2^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash e_1^s\ e_2^s : \tau_2^s \overset{lm}{\Rightarrow} 2\ \Gamma, \alpha : *, \Gamma' \vdash e_1^s : \tau^{s'} \to \tau_2^s;\ \Gamma, \alpha : *, \Gamma' \vdash e_2^s : \tau^{s'} \overset{I.H.}{\Rightarrow}$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash e_1^s\left\{\tau^\alpha/\alpha\right\}^s : \tau^{s'} \to \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau;$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash e_2^s\left\{\tau^\alpha/\alpha\right\}^s : \tau^{s'}\left\{\tau^\alpha/\alpha\right\}^\tau \Rightarrow$$

$$\Gamma \vdash (e_1^s\left\{\tau^\alpha/\alpha\right\}^s)\ (e_2^s\left\{\tau^\alpha/\alpha\right\}^s) : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau = \Gamma \vdash (e_1^s\ e_2^s)\left\{\tau^\alpha/\alpha\right\}^s : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau$$

CASE ($e^s = \textbf{if}\ e_1^s\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s$).

$$\Gamma, \alpha : * \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3\ \textbf{wf} \overset{lm}{\Rightarrow} 2$$

$$\Gamma, \alpha : * \vdash e_1 : \textbf{bool}\ \Gamma, \alpha : * \vdash e_2 : \tau^s;\ \Gamma, \alpha : * \vdash e_3 : \tau^s \overset{I.H.}{\Rightarrow}$$

$$\Gamma \vdash e_1\left\{\tau^\alpha/\alpha\right\}^s : \textbf{bool};\ \Gamma \vdash e_2\left\{\tau^\alpha/\alpha\right\}^s : \tau^s;\ \Gamma \vdash e_3\left\{\tau^\alpha/\alpha\right\}^s : \tau^s \Rightarrow$$

$$\Gamma \vdash \textbf{if}\ e_1\left\{\tau^\alpha/\alpha\right\}^s\ \textbf{then}\ e_2\left\{\tau^\alpha/\alpha\right\}^s\ \textbf{else}\ e_3\left\{\tau^\alpha/\alpha\right\}^s : \tau^s =$$

$$\Gamma \vdash (\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)\left\{\tau^\alpha/\alpha\right\}^s : \tau^s$$

CASE ($e^s = \Lambda\alpha'.e^s$). Assume wolog that $\alpha' \notin FTV^s(v^s)$. Then

$$\Gamma, \alpha : *, \Gamma' \vdash \Lambda\alpha'.e^s : \forall\alpha'.\tau^{s'} \overset{lm}{\Rightarrow} 2$$

$$\alpha \notin FV(\Gamma, \alpha : *, \Gamma');\ \Gamma, \alpha : *, \Gamma', \alpha' : * \vdash e^s : \tau^{s'} \overset{I.H.}{\Rightarrow}$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau), \alpha' : * \vdash e^s\left\{\tau^\alpha/x\right\}^s : \tau^{s'}\left\{\tau^\alpha/\alpha\right\}^\tau \Rightarrow$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash \Lambda\alpha'.e^s\left\{\tau^\alpha/x\right\}^s : \forall\alpha'.\tau^{s'}\left\{\tau^\alpha/\alpha\right\}^\tau =$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash (\Lambda\alpha'.e^s)\left\{\tau^\alpha/x\right\}^s : (\forall\alpha'.\tau^{s'})\left\{\tau^\alpha/\alpha\right\}^\tau.$$

CASE ($e^s = e^s[\tau_2^\alpha]$).

$$\Gamma, \alpha : *, \Gamma' \vdash e^s[\tau_2^\alpha] : \tau_2^s \overset{lm}{\Rightarrow} 2\ \Gamma, \alpha : *, \Gamma' \vdash e^s : \forall\alpha'.\tau^{s'};$$

$$\Gamma, \alpha : *, \Gamma' \vdash \tau_2^\alpha : \textbf{type};\ \tau_2^s = \tau^{s'}\left\{\tau_2^\alpha/\alpha'\right\}^\tau \overset{I.H.}{\Rightarrow}$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash e^s\left\{\tau^\alpha/\alpha\right\}^s : \forall\alpha'.\tau^{s'}\left\{\tau^\alpha/\alpha\right\}^\tau;$$

$$\Gamma, (\Gamma'\left\{\tau^\alpha/\alpha\right\}^\tau) \vdash \tau_2^\alpha\left\{\tau^\alpha/\alpha\right\}^\tau : \textbf{type} \Rightarrow$$

$$\Gamma \vdash (e^s\left\{\tau^\alpha/\alpha\right\}^s)[\tau_2^\alpha\left\{\tau^\alpha/\alpha\right\}^\tau] : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau = \Gamma \vdash (e^s[\tau_2^\alpha])\left\{\tau^\alpha/\alpha\right\}^s : \tau_2^s\left\{\tau^\alpha/\alpha\right\}^\tau$$

Case ($e^s = \prec e \succ$).

$$\Gamma, \alpha : *, \Gamma' \vdash \prec e \succ : \mathbf{code} \overset{lm\ 2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e\ \mathbf{wf} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e \{\tau^\alpha/\alpha\}\ \mathbf{wf} \Rightarrow \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \prec e \{\tau^\alpha/\alpha\} \succ : \mathbf{code} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \prec e \succ \{\tau^\alpha/\alpha\}^s : \mathbf{code}$$

Case ($e^s = \%_0 e^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash \%_0 e^s : \mathbf{code} \overset{lm\ 2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e^s : \gamma \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e^s \{\tau^\alpha/\alpha\}^s : \gamma \Rightarrow \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash \%_0 e^s \{\tau^\alpha/\alpha\}^s : \mathbf{code} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (\%_0 e^s) \{\tau^\alpha/\alpha\}^s : \mathbf{code}$$

Case ($e^s = e_1^s \to e_2^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash e_1^s \to e_2^s : \mathbf{type} \overset{lm\ 2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e_1^s : \mathbf{type};\ \Gamma, \alpha : *, \Gamma' \vdash e_2^s : \mathbf{type} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_1^s \{\tau^\alpha/\alpha\}^s : \mathbf{type};\ \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_2^s \{\tau^\alpha/\alpha\}^s : \mathbf{type} \Rightarrow$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1^s \{\tau^\alpha/\alpha\}^s) \to (e_2^s \{\tau^\alpha/\alpha\}^s) : \mathbf{type} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1^s \to e_2^s) \{\tau^\alpha/\alpha\}^s : \mathbf{type}$$

Case ($e^s = e_1^s =_\tau e_2^s$).

$$\Gamma, \alpha : *, \Gamma' \vdash e_1^s =_\tau e_2^s : \mathbf{bool} \overset{lm\ 2}{\Rightarrow} \Gamma, \alpha : *, \Gamma' \vdash e_1^s : \mathbf{type};\ \Gamma, \alpha : *, \Gamma' \vdash e_2^s : \mathbf{type} \overset{I.H.}{\Rightarrow}$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_1^s \{\tau^\alpha/\alpha\}^s : \mathbf{type};\ \Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash e_2^s \{\tau^\alpha/\alpha\}^s : \mathbf{type} \Rightarrow$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1^s \{\tau^\alpha/\alpha\}^s) =_\tau (e_2^s \{\tau^\alpha/\alpha\}^s) : \mathbf{bool} =$$
$$\Gamma, (\Gamma' \{\tau^\alpha/\alpha\}^\tau) \vdash (e_1^s =_\tau e_2^s) \{\tau^\alpha/\alpha\}^s : \mathbf{bool}$$

□

Lemma 20. *If* $\Gamma \vdash \sim \prec e^o \succ$ ***wf*** *then* $\Gamma \vdash e^o$ ***wf****.*

Proof. By applications of Lemma 2. □

Lemma 21. *If* $\Gamma \vdash r : \tau^s$ *and* $r \to_{E^s} e'$ *then* $\Gamma \vdash e^s$ ***wf***

Proof. By cases on $r \to_{E^s} e'$. Most of the cases are straightforward applications of inversion.

Case ($(\lambda x : \tau^s . e^s)\ v^s \to_{E^s} e^s\ [v^s/x]^s$). Consequence of Lemma 18

Case ($(\Lambda\alpha . e^s)[\tau^\alpha] \to_{E^s} e^s \{\tau^\alpha/\alpha\}^s$). Consequence of Lemma 19

Case ($f\ c \to_{E^s} \delta(f, c)$). This follows from Lemma 2 and from the restrictions on f, c, and δ.

□

Theorem 9 (Preservation). *If* e *is closed,* $\Gamma \vdash e$ ***wf****, and* $e \longmapsto e'$*, then* $\Gamma \vdash e'$ ***wf****.*

Proof. If $e \longmapsto e'$, then there are three cases

Case ($e = E^s[r]$ and $r \to e^s$). This can be proved by cases on $\to$. The $r = (\lambda x : \tau^s . e^s)\ v^s$ is a consequence of Lemma 3, Lemma 21, and Lemma 4. The $r = (\Lambda\alpha . e^s)[\tau^\alpha]$ case is a consequence of Lemma 3, Lemma 21, and Lemma 4. The rest of the cases are straightforward

Case ($e = E^s[\textbf{typeof } \prec e^o \succ]$). $E^s[\textbf{typeof } \prec e^o \succ] \longmapsto E^s[\tau]$, and both the redex and contractum have type **type**. The case follows from Lemma 4.

Case ($e = E[\sim \prec e^o \succ]$). Consequence of Lemma 3, Lemma 20, and Lemma 4.

□

APPENDIX B

Surface Language Metatheory and Proofs

Symbol Classes

γ ::= ⟨type constant⟩
x ::= ⟨variable⟩
c ::= ⟨value constant⟩
f ::= ⟨function constant⟩

Terms

pgm ::= e (program)

e ::= $x \mid \lambda x : e^s.e \mid e\ e$ (code language)
| $\mathbf{let}\ x = e\ \mathbf{in}\ e$
| $\mathbf{let\ meta}\ x = e^s\ \mathbf{in}\ e$
| $\mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e$
| $c \mid f$
| $e^s[e^s]$

e^s ::= $x \mid \alpha \mid \lambda x : \tau^s.e^s \mid e^s\ e^s$ (metalanguage)
| $\mathbf{let}\ x = e^s\ \mathbf{in}\ e^s$
| $\mathbf{if}\ e^s\ \mathbf{then}\ e^s\ \mathbf{else}\ e^s$
| $c \mid f$
| $\gamma \mid \gamma^? \mid \rightarrow^? \mid \mathbf{dom} \mid \mathbf{cod} \mid e^s =_\tau e^s$
| $\mathbf{typeof}$
| $\prec e \succ$
| $e^s \rightarrow e^s$
| $\mathbf{fgen}\ [\overline{x_i}](x : \mathbf{code}\ \tau^x).e^s$
| $\mathbf{fgen}\ [\overline{\alpha_i}](x : \mathbf{meta}\ \tau^\alpha).e^s$

e^o ::= $x \mid \lambda x : \tau.e^o \mid e^o\ e^o \mid \mathbf{let}\ x = e^o\ \mathbf{in}\ e^o$
| $c \mid f \mid \mathbf{if}\ e^o\ \mathbf{then}\ e^o\ \mathbf{else}\ e^o$

Types

τ ::= $\gamma \mid \tau \rightarrow \tau$ (type)
τ^x ::= $x \mid \gamma \mid \tau^x \rightarrow \tau^x$ (type pattern)
τ^α ::= $\alpha \mid \gamma \mid \tau^\alpha \rightarrow \tau^\alpha$ (type schema)

Parameters

τ^s ::= $\gamma \mid \alpha \mid \mathbf{code}$ (meta types)
| $\mathbf{type}$
| $\tau^s \rightarrow \tau^s$
| $[\overline{x_i}](\mathbf{code}\ \tau^x) \Rightarrow \tau^\sim$
| $[\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \Rightarrow \tau^\sim$

$\tau^\sim$ ::= $\gamma \mid \mathbf{code}$ (splice types)

ε ::= $x : \tau^s \mid x : \mathbf{dyn} \mid \alpha : *$ (environment bindings)
Γ ::= $\overline{\varepsilon_i}$ (translation environment)

FIGURE 1. Surface Language Syntax

$\boxed{\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks}}$ Translate Metalanguage

$$\frac{x : \tau^s \in \Gamma}{\Gamma \vdash x : \tau^s \rightsquigarrow x} \qquad \frac{\alpha : * \in \Gamma}{\Gamma \vdash \alpha : \mathbf{type} \rightsquigarrow \alpha} \qquad \frac{}{\Gamma \vdash \gamma : \mathbf{type} \rightsquigarrow \gamma}$$

$$\frac{}{\Gamma \vdash \mathbf{dom} : (\mathbf{type} \to \mathbf{type}) \rightsquigarrow \mathbf{dom}} \qquad \frac{}{\Gamma \vdash \mathbf{typeof} : (\mathbf{code} \to \mathbf{type}) \rightsquigarrow \mathbf{typeof}}$$

$$\frac{}{\Gamma \vdash \mathbf{cod} : (\mathbf{type} \to \mathbf{type}) \rightsquigarrow \mathbf{cod}} \qquad \frac{}{\Gamma \vdash \to^? : (\mathbf{type} \to \mathbf{bool}) \rightsquigarrow \to^?}$$

$$\frac{}{\Gamma \vdash \gamma^? : (\mathbf{type} \to \mathbf{bool}) \rightsquigarrow \gamma^?} \qquad \frac{\Gamma \vdash e_1^s : (\tau_1^s \to \tau_2^s) \rightsquigarrow e_1^{ks} \quad \Gamma \vdash e_2^s : \tau_1^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s\ e_2^s : \tau_2^s \rightsquigarrow e_1^{ks}\ e_2^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{type} \rightsquigarrow e_1^{ks} \quad \Gamma \vdash e_2^s : \mathbf{type} \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s \to e_2^s : \mathbf{type} \rightsquigarrow e_1^{ks} \to e_2^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{type} \rightsquigarrow e_1^{ks} \quad \Gamma \vdash e_2^s : \mathbf{type} \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s =_\tau e_2^s : \mathbf{bool} \rightsquigarrow e_1^{ks} =_\tau e_2^{ks}}$$

$$\frac{\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks} \quad \Gamma \vdash e_1^s : \tau_1^s \rightsquigarrow e_1^{ks} \quad \Gamma, x : \tau_1^s \vdash e_2^s : \tau_2^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash \mathbf{let}\ x = e_1^s\ \mathbf{in}\ e_2^s : \tau_2^s \rightsquigarrow (\lambda x : \tau_1^{ks}.e_2^{ks})\ e_1^{ks}}$$

$$\frac{\Gamma \vdash e_1^s : \mathbf{bool} \rightsquigarrow e_1^{ks} \quad \Gamma \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks} \quad \Gamma \vdash e_3^s : \tau^s \rightsquigarrow e_3^{ks}}{\Gamma \vdash \mathbf{if}\ e_1^s\ \mathbf{then}\ e_2^s\ \mathbf{else}\ e_3^s : \tau^s \rightsquigarrow \mathbf{if}\ e_1^{ks}\ \mathbf{then}\ e_2^{ks}\ \mathbf{else}\ e_3^{ks}} \qquad \frac{\mathit{type}(c) = \gamma}{\Gamma \vdash c : \gamma \rightsquigarrow c}$$

$$\frac{\mathit{type}(f) = \gamma_1 \to \gamma_2}{\Gamma \vdash f : \gamma_1 \to \gamma_2 \rightsquigarrow f} \qquad \frac{\Gamma \vdash e \rightsquigarrow e^k}{\Gamma \vdash \prec e \succ : \mathbf{code} \rightsquigarrow \prec e^k \succ}$$

$$\frac{\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks} \quad \Gamma, x : \tau_1^s \vdash e^s : \tau_2^s \rightsquigarrow e^{ks}}{\Gamma \vdash \lambda x : \tau_1^s.e^s : \tau_1^s \to \tau_2^s \rightsquigarrow \lambda x : \tau_1^{ks}.e^{ks}}$$

$$\frac{\dfrac{\overline{x_i} \vdash \tau^x\ \mathbf{wf}}{e_i^{ks} = x_i \mathit{cpat}(\tau^x, x_\tau)} \qquad \dfrac{x_\tau \notin FV(e^s)}{\Gamma, \overline{x_i : \mathbf{type}}, x : \mathbf{code} \vdash e^s : \tau^\sim \rightsquigarrow e^{ks}}}{\begin{array}{l}\Gamma \vdash \mathbf{fgen}\ [\overline{x_i}](x : \mathbf{code}\ \tau^x).e^s : [\overline{x_i}](\mathbf{code}\ \tau^x) \to \tau^\sim \\ \quad \rightsquigarrow \lambda x : \mathbf{code}. \\ \qquad (\lambda x_\tau : \mathbf{type}. \\ \qquad\ \overline{((\lambda x_i : \mathbf{type}.\ e^{ks}\ )\ e_i^{ks})}) \\ \qquad \mathbf{typeof}\ x)\end{array}}$$

$$\frac{\overline{\alpha_i} \vdash \tau^\alpha\ \mathbf{wf} \quad \Gamma, \overline{\alpha_i : *}, x : \tau^\alpha \vdash e^s : \tau^\sim \rightsquigarrow e^{ks}}{\begin{array}{l}\Gamma \vdash \mathbf{fgen}\ [\overline{\alpha_i}](x : \mathbf{meta}\ \tau^\alpha).e^s : [\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \to \tau^\sim \\ \quad \rightsquigarrow \overline{\Lambda \alpha_i}.\lambda x : \tau^\alpha.e^{ks}\end{array}}$$

FIGURE 2. Translation to Kernel Language, Part 1

$\boxed{\Gamma \vdash e \rightsquigarrow e^k}^{\dagger}$ Translate Code Language

$$\frac{x : \mathbf{dyn} \in \Gamma}{\Gamma \vdash x \rightsquigarrow x} \qquad \frac{x : \tau^{\sim} \in \Gamma}{\Gamma \vdash x \rightsquigarrow [\![\tau^{\sim}]\!] x} \qquad \frac{\Gamma \vdash e^s : \mathbf{type} \rightsquigarrow e^{ks} \qquad \Gamma, x : \mathbf{dyn} \vdash e \rightsquigarrow e^k}{\Gamma \vdash \lambda x : e^s . e \rightsquigarrow \lambda x : e^{ks} . e^k}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma \vdash e_2 \rightsquigarrow e_2^k}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1^k\ e_2^k} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma, x : \mathbf{dyn} \vdash e_2 \rightsquigarrow e_2^k}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow (\lambda x : \mathbf{typeof}\ \prec e_1^k \succ . e_2^k)\ e_1^k}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1^k \qquad \Gamma \vdash e_2 \rightsquigarrow e_2^k \qquad \Gamma \vdash e_3 \rightsquigarrow e_3^k}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rightsquigarrow \mathbf{if}\ e_1^k\ \mathbf{then}\ e_2^k\ \mathbf{else}\ e_3^k} \qquad \frac{}{\Gamma \vdash c \rightsquigarrow c} \qquad \frac{}{\Gamma \vdash f \rightsquigarrow f}$$

$$\frac{\begin{array}{c}\Gamma \vdash x : [\overline{\alpha_i}](\mathbf{meta}\ \tau^{\alpha}) \to \tau^{\sim} \rightsquigarrow x \\ \Gamma \vdash e^o : \tau_0^{\alpha} \rightsquigarrow e^{ks} \qquad \overline{\tau_i^{\alpha} = \alpha_i mpat(\tau^{\alpha}, \tau_0^{\alpha})}\end{array}}{\Gamma \vdash x\ e^o \rightsquigarrow [\![\tau^{\sim}]\!](x\overline{[\tau_i^{\alpha}]}\ e^{ks})}$$

$$\frac{\Gamma \vdash x : [\overline{x_i}](\mathbf{code}\ \tau^x) \to \tau^{\sim} \rightsquigarrow x \qquad \Gamma \vdash e \rightsquigarrow e^k}{\Gamma \vdash x\ e \rightsquigarrow [\![\tau^{\sim}]\!](x \prec e^k \succ)}$$

$$\frac{\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks} \qquad \Gamma, x : \tau^s \vdash e \rightsquigarrow e^k \qquad \Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}}{\Gamma \vdash \mathbf{let}\ \mathbf{meta}\ x = e^s\ \mathbf{in}\ e \rightsquigarrow \sim((\lambda x : \tau^{ks} . \prec e^k \succ)\ e^{ks})}$$

$$\frac{\Gamma \vdash e_1^s : \tau_1^s \to \tau^{\sim} \rightsquigarrow e_1^{ks} \qquad \Gamma \vdash e_2^s : \tau_1^s \rightsquigarrow e_2^{ks}}{\Gamma \vdash e_1^s[e_2^s] \rightsquigarrow [\![\tau^{\sim}]\!](e_1^{ks}\ e_2^{ks})}$$

$$^{\dagger}[\![\tau^{\sim}]\!] = \begin{cases} \sim & \text{if } \tau^{\sim} = \mathbf{code} \\ \sim\% & \text{if } \tau^{\sim} = \gamma \end{cases}$$

FIGURE 3. Translation to Kernel Language, Part 2

$\boxed{mpat(\tau^\alpha, \tau^\alpha) = \sigma}$ Match Type Schema

$$\frac{}{mpat(\alpha, \tau^\alpha) = [\tau^\alpha/\alpha]} \qquad \frac{}{mpat(\gamma, \tau^\alpha) = \emptyset}$$

$$\frac{mpat(\tau^\alpha_{11}, \tau^\alpha_{21}) = \sigma_1 \qquad mpat(\tau^\alpha_{12}, \tau^\alpha_{22}) = \sigma_2}{mpat(\tau^\alpha_{11} \to \tau^\alpha_{12}, \tau^\alpha_{21} \to \tau^\alpha_{22}) = \sigma_1 \circ \sigma_2}$$

$\boxed{cpat(\tau^x, x) = \sigma}$ Match Type Pattern

$$\frac{}{cpat(x', x) = [x/x']} \qquad \frac{}{cpat(\gamma, x) = \emptyset}$$

$$\frac{cpat(\tau^x_1, x) = \overline{[e^s_{i_1}/x_{i_1}]} \qquad cpat(\tau^x_2, x) = \overline{[e^s_{i_2}/x_{i_2}]}}{cpat(\tau^x_1 \to \tau^x_2, x) = \overline{[\mathbf{dom}\ e^s_{i_1}/x_{i_1}]} \circ \overline{[\mathbf{cod}\ e^s_{i_2}/x_{i_2}]}}$$

FIGURE 4. Pattern Matching Metafunctions

$\boxed{\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}}$ Translate Meta Type

$$\frac{\alpha : * \in \Gamma}{\Gamma \vdash \alpha \rightsquigarrow \alpha} \qquad \frac{}{\Gamma \vdash \gamma \rightsquigarrow \gamma} \qquad \frac{}{\Gamma \vdash \mathbf{code} \rightsquigarrow \mathbf{code}} \qquad \frac{}{\Gamma \vdash \mathbf{type} \rightsquigarrow \mathbf{type}}$$

$$\frac{\Gamma \vdash \tau^s_1 \rightsquigarrow \tau^{ks}_1 \qquad \Gamma \vdash \tau^s_2 \rightsquigarrow \tau^{ks}_2}{\Gamma \vdash \tau^s_1 \to \tau^s_2 \rightsquigarrow \Gamma \vdash \tau^{ks}_1 \to \tau^{ks}_2} \qquad \frac{\overline{x_i} \vdash \tau^x\ \mathbf{wf} \qquad \Gamma \vdash \tau^\sim \rightsquigarrow \tau^{ks}}{\Gamma \vdash [\overline{x_i}](\mathbf{code}\ \tau^x) \Rightarrow \tau^\sim \rightsquigarrow \mathbf{code} \to \tau^{ks}}$$

$$\frac{\overline{\alpha_i} \vdash \tau^\alpha\ \mathbf{wf} \qquad \Gamma, \overline{\alpha_i : *} \vdash \tau^\sim \rightsquigarrow \tau^{ks}}{\Gamma \vdash [\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \Rightarrow \tau^\sim \rightsquigarrow \overline{\forall \alpha_i}.\tau^\alpha \to \tau^{ks}}$$

$\boxed{\overline{x_i} \vdash \tau^x\ \mathbf{wf}}$ Well-formed Type Pattern

$\boxed{\overline{\alpha_i} \vdash \tau^x\ \mathbf{wf}}$ Well-formed Type Schema

$$\frac{\overline{x_i} \vdash \tau^x : \emptyset}{\overline{x_i} \vdash \tau^x\ \mathbf{wf}} \qquad \frac{x \notin \overline{x_i}, \overline{x_j}}{\overline{x_i}, x, \overline{x_j} \vdash x : \overline{x_i}, \overline{x_j}} \qquad \frac{}{\overline{x_i} \vdash \gamma : \overline{x_i}}$$

$$\frac{\overline{x_i} \vdash \tau^x_1 : \overline{x_i}' \qquad \overline{x_i}' \vdash \tau^x_2 : \overline{x_i}''}{\overline{x_i} \vdash \tau^x_1 \to \tau^x_2 : \overline{x_i}''}$$

FIGURE 5. Translation to Kernel Language, Part 4

$\boxed{\Gamma \rightsquigarrow \Gamma^k}$ Translate Type Environment

$$\frac{}{\emptyset \rightsquigarrow \emptyset} \qquad \frac{\Gamma \rightsquigarrow \Gamma^k \qquad \Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}}{\Gamma, x : \tau^s \rightsquigarrow \Gamma^k, x : \tau^{ks}} \qquad \frac{\Gamma \rightsquigarrow \Gamma^k}{\Gamma, x : \mathbf{dyn} \rightsquigarrow \Gamma^k, x : \mathbf{dyn}}$$

$$\frac{\Gamma \rightsquigarrow \Gamma^k}{\Gamma, \alpha : * \rightsquigarrow \Gamma^k, \alpha : *}$$

FIGURE 6. Translation to Kernel Language, Part 5

LEMMA 22 (Inversion of Translation).

(1) *If* $\Gamma \vdash x : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = x$ *and* $(x : \tau^s) \in \Gamma$.
(2) *If* $\Gamma \vdash \alpha : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \alpha$, $\tau^s = \textbf{\textit{type}}$, *and* $(\alpha : *) \in \Gamma$.
(3) *If* $\Gamma \vdash \gamma : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \gamma$ *and* $\tau^s = \textbf{\textit{type}}$.
(4) *If* $\Gamma \vdash \textbf{\textit{dom}} : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \textbf{\textit{dom}}$ *and* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{type}}$.
(5) *If* $\Gamma \vdash \textbf{\textit{cod}} : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \textbf{\textit{cod}}$ *and* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{type}}$.
(6) *If* $\Gamma \vdash \gamma^? : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \gamma^?$ *and* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{bool}}$.
(7) *If* $\Gamma \vdash \to^? : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \to^?$ *and* $\tau^s = \textbf{\textit{type}} \to \textbf{\textit{bool}}$.
(8) *If* $\Gamma \vdash \textbf{\textit{typeof}} : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = \textbf{\textit{typeof}}$ *and* $\tau^s = \textbf{\textit{code}} \to \textbf{\textit{type}}$.
(9) *If* $\Gamma \vdash c : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = c$ *and* $\tau^s = \gamma$ *for* $type(c) = \gamma$.
(10) *If* $\Gamma \vdash f : \tau^s \rightsquigarrow e^{ks}$, *then* $e^{ks} = f$ *and* $\tau^s = \gamma_1 \to \gamma_2$ *for* $type(f) = \gamma_1 \to \gamma_2$.
(11) *If* $\Gamma \vdash \textbf{\textit{if}}\ e_1^s\ \textbf{\textit{then}}\ e_2^s\ \textbf{\textit{else}}\ e_3^s : \tau^s \rightsquigarrow e^{ks}$ *then* $\Gamma \vdash e_1^s : \textbf{\textit{bool}} \rightsquigarrow e_1^{ks}$, $\Gamma \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks}$, $\Gamma \vdash e_3^s : \tau^s \rightsquigarrow e_3^{ks}$, *and* $e^{ks} = \textbf{\textit{if}}\ e_1^{ks}\ \textbf{\textit{then}}\ e_2^{ks}\ \textbf{\textit{else}}\ e_3^{ks}$.
(12) *If* $\Gamma \vdash \prec e \succ : \tau^s \rightsquigarrow e^{ks}$, *then* $\tau^s = \textbf{\textit{code}}$, $\Gamma \vdash e \rightsquigarrow e^k$, *and* $e^{ks} = \prec e^k \succ$.
(13) *If* $\Gamma \vdash e_1^s \to e_2^s : \tau^s \rightsquigarrow e^{ks}$, *then* $\tau^s = \textbf{\textit{type}}$, $\Gamma \vdash e_1^s : \textbf{\textit{type}} \rightsquigarrow e_1^{ks}$, $\Gamma \vdash e_2^s : \textbf{\textit{type}} \rightsquigarrow e_2^{ks}$, *and* $e^{ks} = e_1^{ks} \to e_2^{ks}$.
(14) *If* $\Gamma \vdash e_1^s =_\tau e_2^s : \tau^s \rightsquigarrow e^{ks}$, *then* $\tau^s = \textbf{\textit{bool}}$, $\Gamma \vdash e_1^s : \textbf{\textit{type}} \rightsquigarrow e_1^{ks}$, $\Gamma \vdash e_2^s : \textbf{\textit{type}} \rightsquigarrow e_2^{ks}$, *and* $e^{ks} = e_1^{ks} =_\tau e_2^{ks}$.
(15) *If* $\Gamma \vdash e_1^s\ e_2^s : \tau^s \rightsquigarrow e^{ks}$, *then there is a type* $\tau^{s'}$ *such that* $\Gamma \vdash e_1^s : \tau^{s'} \to \tau^s \rightsquigarrow e_1^{ks}$ *and* $\Gamma \vdash e_2^s : \tau^{s'} \rightsquigarrow e_2^{ks}$ *and* $e^{ks} = e_1^{ks}\ e_2^{ks}$.
(16) *If* $\Gamma \vdash \lambda x : \tau_1^s . e^{s'} : \tau^s \rightsquigarrow e^{ks}$, *then there is a type* $\tau^{s'}$ *such that* $\Gamma, x : \tau_1^s \vdash e^{s'} : \tau^{s'} \rightsquigarrow e^{ks'}$, $\tau^s = \tau_1^s \to \tau^{s'}$, $\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks}$, *and* $e^{ks} = \lambda x : \tau_1^{ks} . e^{ks}$; $x \notin FV(\Gamma)$.
(17) *If* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1^s\ \textbf{\textit{in}}\ e_2^s : \tau^s \rightsquigarrow e^{ks}$ *then* $\Gamma \vdash e_1^s : \tau_1^s \rightsquigarrow e_1^{ks}$, $\Gamma, x : \tau_1^s \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks}$, $\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks}$, *and* $e^{ks} = (\lambda x : \tau_1^{ks} . e_2^{ks})\ e_1^{ks}$.
(18) *If* $\Gamma \vdash \textbf{\textit{fgen}}\ [\overline{x_i}](\textbf{\textit{code}}\ \tau^x) . e^s : \tau^s \rightsquigarrow e^{ks}$, *then* $\overline{x_i} \vdash \tau^x$ *wf*, $\Gamma, \overline{x_i : \textbf{\textit{type}}}, x : \textbf{\textit{code}} \vdash e^{s'} : \tau^\sim \rightsquigarrow e^{ks'}$, $\tau^s = [\overline{x_i}](\tau^x) \to \tau^\sim$, *and*

$$\begin{aligned} e^{ks} = \lambda x : \textbf{\textit{code}}.\ & \\ & (\lambda x_\tau : \textbf{\textit{type}}. \\ & \quad \overline{((\lambda x_i : \textbf{\textit{type}}.\ e^{ks'}\ )\ e_i^{ks})}) \\ & \textbf{\textit{typeof}}\ x), \end{aligned}$$

where $x_\tau \notin FV(e^{s'})$, *and* $e_i^{ks} = x_i cpat(\tau^x, x_\tau)$ *for each* i.
(19) *If* $\Gamma \vdash \textbf{\textit{fgen}}\ [\overline{\alpha_i}](\textbf{\textit{meta}}\ \tau^\alpha) . e^s : \tau^s \rightsquigarrow e^{ks}$, *then* $\overline{\alpha_i} \vdash \tau^\alpha$ *wf*, $\Gamma, \overline{\alpha_i : *}, x : \tau^\alpha \vdash e^{s'} : \tau^\sim \rightsquigarrow e^{ks'}$, $\tau^s = [\overline{\alpha_i}](\tau^\alpha) \to \tau^\sim$, *and* $e^{ks} = \overline{\Lambda \alpha_i} . \lambda x : \tau^\alpha . e^{ks'}$.
(20) *If* $\Gamma \vdash x \rightsquigarrow e^k$, *then one and only one of the following is true:*
  (a) $e^k = x$ *and* $(x : \textbf{\textit{dyn}}) \in \Gamma$;
  (b) *For some* $\tau^\sim$, $e^k = [\![\tau^\sim]\!] x$ *and* $(x : \tau^\sim) \in \Gamma$;
(21) *If* $\Gamma \vdash \lambda x : e^s . e' \rightsquigarrow e^k$, *then* $\Gamma \vdash e^s : \textbf{\textit{type}} \rightsquigarrow e^{ks}$, $\Gamma, x : \textbf{\textit{dyn}} \vdash e' \rightsquigarrow e^{k'}$, *and* $e^k = \lambda x : e^{ks} . e^{k'}$; $x \notin FV(\Gamma)$.
(22) *If* $\Gamma \vdash e_1\ e_2 \rightsquigarrow e^k$, *then one and only one of the following is true:*
  (a) $\Gamma \vdash e_1 \rightsquigarrow e_1^k$, $\Gamma \vdash e_2 \rightsquigarrow e_2^k$, *and* $e^k = e_1^k\ e_2^k$.
  (b) $e_1 = x$, $e_2 = e^o$, $\Gamma \vdash x : [\overline{\alpha_i}](\textbf{\textit{meta}}\ \tau^\alpha) \to \tau^\sim \rightsquigarrow x$, $\Gamma \vdash e^o : \tau_0^\alpha \rightsquigarrow e^{ks}$, *and* $e^k = [\![\tau^\sim]\!](x \overline{[\tau_i^\alpha]}\ e^{ks})$ *where* $\tau_i^\alpha = \alpha_i mpat(\tau^\alpha, \tau_0^\alpha)$ *for each* i.

(c) $e_1 = x$, $\Gamma \vdash x : [\overline{x_i}](\textbf{\textit{code}}\ \tau^x) \to \tau^\sim \rightsquigarrow x$, $\Gamma \vdash e_2 \rightsquigarrow e_2^k$, *and* $e^k = [\![\tau^\sim]\!](x\ \prec e_2^k \succ)$.

(23) *If* $\Gamma \vdash \textbf{\textit{if}}\ e_1\ \textbf{\textit{then}}\ e_2\ \textbf{\textit{else}}\ e_3 \rightsquigarrow e^k$ *then* $\Gamma \vdash e_1 \rightsquigarrow e_1^k$, $\Gamma \vdash e_2 \rightsquigarrow e_2^k$, $\Gamma \vdash e_3 \rightsquigarrow e_2^k$, *and* $e^k = \textbf{\textit{if}}\ e_1^k\ \textbf{\textit{then}}\ e_2^k\ \textbf{\textit{else}}\ e_3^k$.

(24) *If* $\Gamma \vdash e_1^s[e_2^s] \rightsquigarrow e^k$ *then* $\Gamma \vdash e_1^s : \tau^s \to \tau^\sim \rightsquigarrow e_1^{ks}$, $\Gamma \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks}$, *and* $e^k = [\![\tau^\sim]\!](e_1^{ks}\ e_2^{ks})$.

(25) *If* $\Gamma \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 \rightsquigarrow e^k$ *then* $\Gamma \vdash e_1 :\rightsquigarrow e_1^k$, $\Gamma, x : \textbf{\textit{dyn}} \vdash e_2 \rightsquigarrow e_2^k$, *and* $e^k = (\lambda x : \textbf{\textit{typeof}}\ \prec e_1^k \succ .e_2^k)\ e_1^k$

(26) *If* $\Gamma \vdash \textbf{\textit{let meta}}\ x = e^s\ \textbf{\textit{in}}\ e' \rightsquigarrow e^k$ *then* $\Gamma \vdash e^s : \tau_1^s \rightsquigarrow e^{ks}$, $\Gamma, x : \tau_1^s \vdash e' \rightsquigarrow e^{k'}$, $\Gamma \vdash \tau_1^s \rightsquigarrow \tau_1^{ks}$, *and* $e^k = \sim((\lambda x : \tau_1^{ks}.\ \prec e^{k'} \succ)\ e^{ks})$.

If $\Gamma \vdash c \rightsquigarrow e^k$ *then* $e^k = c$.
If $\Gamma \vdash f \rightsquigarrow e^k$ *then* $e^k = f$.

Proof. By cases on derivations of $\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks}$ and $\Gamma \vdash e \rightsquigarrow e^k$. □

Lemma 23. *If* Γ ***wf*** *then*

(1) $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$ *is a function to* τ^{ks}.
(2) *if* $x : \tau^s \in \Gamma$ *then* $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$.
(3) $\Gamma \rightsquigarrow \Gamma^k$ *and if* $x : \tau^s \in \Gamma$ *then* $x : \tau^{ks} \in \Gamma^k$ *where* $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$.
(4) $\Gamma \vdash \tau^\sim \rightsquigarrow \tau^\sim$.

Lemma 24. *If* $\Gamma^k \vdash e^{ks} : \tau^\sim$ *then* $\Gamma^k \vdash^k [\![\tau^\sim]\!]e^{ks}$ ***wf***.

Theorem 10 (Well-Typing Preservation).

(1) *If* $\Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks}$, $\Gamma \rightsquigarrow \Gamma^k$, *and* $\Gamma \vdash \tau^s \rightsquigarrow \tau^{ks}$, *then* $\Gamma^k \vdash^k e^{ks} : \tau^{ks}$;
(2) *If* $\Gamma \vdash e \rightsquigarrow e^k$, *and* $\Gamma \rightsquigarrow \Gamma^k$, $\Gamma^k \vdash^k e^k$ ***wf***.

Proof. By mutual induction on the structure of e^s and e.

Case ($e = x$). If $\Gamma \vdash x \rightsquigarrow e^k$ then there are two cases:

(1) $e^k = x$; $x : \mathbf{dyn} \in \Gamma \overset{lm}{\Rightarrow}^{23} x : \mathbf{dyn} \in \Gamma^k \Rightarrow \Gamma^k \vdash^k x$ **wf**.
(2) $e^k = [\![\tau^\sim]\!]x$; $(x : \tau^\sim) \in \Gamma \overset{lm}{\Rightarrow}^{23} x : \tau^\sim \in \Gamma^k \Rightarrow \Gamma^k \vdash^k x : \tau^\sim \overset{lm}{\Rightarrow}^{24} \Gamma^k \vdash^k [\![\tau^\sim]\!]x$ **wf**.

Case ($e = c$). Immediate.

Case ($e = f$). Immediate.

Case ($e = \lambda x : e^s.e$).

$$\Gamma \vdash \lambda x : e^s.e \rightsquigarrow \lambda x : e^{ks}.e^k \overset{lm}{\Rightarrow}^{22} \Gamma \vdash e^s : \mathbf{type} \rightsquigarrow e^{ks};\ \Gamma, x : \mathbf{dyn} \vdash e \rightsquigarrow e^k \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k \vdash^k e^{ks} : \mathbf{type};\ \Gamma^k, x : \mathbf{dyn} \vdash^k e^k\ \mathbf{wf} \Rightarrow \Gamma^k \vdash^k \lambda x : e^{ks}.e^k\ \mathbf{wf}.$$

Case ($e = e_1\ e_2$). This case splits into three subcases.

(1)

$$\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1^k\ e_2^k \overset{lm}{\Rightarrow}^{22} \Gamma \vdash e_1 \rightsquigarrow e_1^k;\ \Gamma \vdash e_2 \rightsquigarrow e_1^k \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k \vdash^k e_1^k\ \mathbf{wf};\ \Gamma^k \vdash^k e_2^k\ \mathbf{wf} \Rightarrow \Gamma^k \vdash^k e_1^k\ e_2^k\ \mathbf{wf}.$$

(2)

$$\Gamma \vdash x\ e^o \rightsquigarrow [\![\tau^\sim]\!](x[\overline{\tau_i^\alpha}]\ e^{ks}) \overset{lm}{\Rightarrow}{}^{22} \Gamma \vdash x : [\overline{\alpha_i}](\mathbf{meta}\ \tau^\alpha) \Rightarrow \tau^\sim \rightsquigarrow x;$$
$$\Gamma \vdash e^o : \tau_0^\alpha \rightsquigarrow e^{ks};\ \tau_i^\alpha = \alpha_i mpat(\tau^\alpha, \tau_0^\alpha) \overset{I.H.}{\Rightarrow} \Gamma^k \vdash^k x : \overline{\forall \alpha_i}.\tau^\alpha \to \tau^\sim;$$
$$\Gamma^k \vdash^k e^{ks} : \tau_0^\alpha \Rightarrow \Gamma^k \vdash^k x[\overline{\tau_i^\alpha}] : \tau_0^\alpha \to \tau^\sim \Rightarrow \Gamma^k \vdash^k x[\overline{\tau_i^\alpha}]\ e^{ks} : \tau^\sim \Rightarrow$$
$$\Gamma^k \vdash^k [\![\tau^\sim]\!](x[\overline{\tau_i^\alpha}]\ e^{ks})\ \mathbf{wf}.$$

(3)

$$\Gamma \vdash x\ e \rightsquigarrow [\![\tau^\sim]\!](x\ \prec e^k \succ) \overset{lm}{\Rightarrow}{}^{22} \Gamma \vdash x : [\overline{x_i}](\mathbf{code}\ \tau^x) \Rightarrow \tau^\sim \rightsquigarrow x;$$
$$\overline{x_i} \vdash \tau^x\ \mathbf{wf};\ \Gamma \vdash e \rightsquigarrow e^k \overset{lm}{\Rightarrow}{}^{22} \Gamma^k \vdash x : \mathbf{code} \to \tau^\sim;\ \Gamma^k \vdash^k e^k\ \mathbf{wf} \Rightarrow$$
$$\Gamma^k \vdash^k \prec e^k \succ : \mathbf{code} \Rightarrow \Gamma^k \vdash^k x\ \prec e^k \succ : \tau^\sim \Rightarrow$$
$$\Gamma^k \vdash^k [\![\tau^\sim]\!](x\ \prec e^k \succ)\ \mathbf{wf}.$$

Case ($e = \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3$). Straightforward.

Case ($e = e_1^s[e_2^s]$).

$$\Gamma \vdash e_1^s[e_2^s] \rightsquigarrow [\![\tau^\sim]\!](e_1^{ks}\ e_2^{ks}) \overset{lm}{\Rightarrow}{}^{22} \Gamma \vdash e_1^s : \tau^s \to \tau^\sim \rightsquigarrow e_1^{ks};\ \Gamma \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks} \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k \vdash^k e_1^{ks} : \tau^{ks} \to \tau^\sim;\ \Gamma^k \vdash^k e_2^{ks} : \tau^{ks} \Rightarrow \Gamma^k \vdash^k e_1^{ks}\ e_2^{ks} : \tau^\sim \Rightarrow$$
$$\Gamma^k \vdash^k [\![\tau^\sim]\!](e_1^{ks}\ e_2^{ks})\ \mathbf{wf}.$$

Case ($e = \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$).

$$\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \rightsquigarrow (\lambda x : \mathbf{typeof}\ \prec e_1^k \succ e_2^k)\ e_1^k \overset{lm}{\Rightarrow}{}^{22} \Gamma \vdash e_1 \rightsquigarrow e_1^k;$$
$$\Gamma, x : \mathbf{dyn} \vdash e_2 \rightsquigarrow e_2^k \overset{I.H.}{\Rightarrow} \Gamma^k \vdash e_1^k\ \mathbf{wf};\ \Gamma, x : \mathbf{dyn} \vdash e_2^k\ \mathbf{wf} \Rightarrow$$
$$\Gamma^k \vdash \prec e_1^k \succ : \mathbf{code} \Rightarrow \Gamma^k \vdash \mathbf{typeof}\ \prec e_1^k \succ : \mathbf{type} \Rightarrow$$
$$\Gamma^k \vdash (\lambda x : \mathbf{typeof}\ \prec e_1^k \succ e_2^k)\ e_1^k\ \mathbf{wf}.$$

Case ($e = \mathbf{let\ meta}\ x = e^s\ \mathbf{in}\ e$).

$$\Gamma \vdash \mathbf{let\ meta}\ x = e^s\ \mathbf{in}\ e \rightsquigarrow (\lambda x : \tau^{ks}.\ \prec e^k \succ)\ e^{ks} \overset{lm}{\Rightarrow}{}^{22} \Gamma \vdash e^s : \tau^s \rightsquigarrow e^{ks};$$
$$\Gamma, x : \tau^s \vdash e \rightsquigarrow e^k \overset{I.H.}{\Rightarrow} \Gamma^k \vdash^k e^{ks} : \tau^{ks};\ \Gamma^k, x : \tau^{ks} \vdash^k e^k\ \mathbf{wf} \Rightarrow$$
$$\Gamma^k, x : \tau^{ks} \vdash^k \prec e^k \succ : \mathbf{code} \Rightarrow \Gamma^k \vdash^k \lambda x : \tau^{ks}.\ \prec e^k \succ : \tau^{ks} \to \mathbf{code} \Rightarrow$$
$$\Gamma^k \vdash^k (\lambda x : \tau^{ks}.\ \prec e^k \succ)\ e^{ks} : \mathbf{code} \Rightarrow \Gamma^k \vdash^k \sim((\lambda x : \tau^{ks}.\ \prec e^k \succ)\ e^{ks})\ \mathbf{wf}$$

Case ($e^s = x$).

$$\Gamma \vdash x : \tau^s \rightsquigarrow x \overset{lm}{\Rightarrow}{}^{22} x : \tau^s \in \Gamma \overset{lm}{\Rightarrow}{}^{23} x : \tau^{ks} \in \Gamma^k \Rightarrow \Gamma^k \vdash^k x : \tau^{ks}.$$

Case ($e^s = \alpha$).

$$\Gamma \vdash \alpha : \mathbf{type} \rightsquigarrow \alpha \overset{lm}{\Rightarrow}{}^{22} \alpha : * \in \Gamma \overset{lm}{\Rightarrow}{}^{23} \alpha : * \in \Gamma^k \Rightarrow \Gamma^k \vdash^k \alpha : \mathbf{type}.$$

Case ($e^s = c$).

$$\Gamma \vdash c : \gamma \rightsquigarrow c \overset{lm}{\Rightarrow}{}^{22} type(c) = \gamma \Rightarrow \Gamma^k \vdash^k c : \gamma.$$

Case $(e^s = f)$.

$$\Gamma \vdash f : \gamma_1 \to \gamma_2 \rightsquigarrow f \overset{lm}{\Rightarrow}^{22} type(f) = \gamma_1 \to \gamma_2 \Rightarrow \Gamma^k \vdash^k f : \gamma_1 \to \gamma_2.$$

Case $(e^s = c^s)$. Straightforward.

Case $(e^s = \lambda x : \tau^s.e^s)$.

$$\Gamma \vdash \lambda x : \tau^s.e^s : \tau^s \to \tau^{s'} \rightsquigarrow \lambda x : \tau^{ks}.e^{ks} \overset{lm}{\Rightarrow}^{22} \Gamma, x : \tau^s \vdash e^s : \tau^{s'} \rightsquigarrow e^{ks} \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k, x : \tau^{ks} \vdash^k e^{ks} : \tau^{ks'} \Rightarrow \Gamma^k \vdash^k \lambda x : \tau^{ks}.e^{ks} : \tau^{ks} \to \tau^{ks'}.$$

Case $(e^s = e_1^s\ e_2^s)$. Straightforward.

Case $(e^s = \textbf{if}\ e_1^s\ \textbf{then}\ e_2^s\ \textbf{else}\ e_3^s)$. Straightforward.

Case $(e^s = e_1^{s'} \to e_2^{s'})$. Straightforward.

Case $(e^s = e_1^{s'} =_\tau e_2^{s'})$. Straightforward.

Case $(e^s = \textbf{fgen}\ [\overline{x_i}](x : \textbf{code}\ \tau^x).e^s)$.

$$\Gamma \vdash \textbf{fgen}\ [\overline{x_i}](x : \textbf{code}\ \tau^x).e^s : [\overline{x_i}](\tau^x) \to \tau^\sim \rightsquigarrow$$
$$\lambda x : \textbf{code}.$$
$$(\lambda x_\tau : \textbf{type}.$$
$$\overline{((\lambda x_i : \textbf{type}.\ e^{ks}\ )\ e_i^{ks})})$$
$$\textbf{typeof}\ x) \overset{lm}{\Rightarrow}^{22}$$

$$\overline{x_i} \vdash \tau^x\ \textbf{wf};\ \Gamma, \overline{x_i : \textbf{type}}, x : \textbf{code} \vdash e^s : \tau^\sim \rightsquigarrow e^{ks};\ e_i^{ks} = x_i cpat(\tau^x, x_\tau) \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k, \overline{x_i : \textbf{type}}, x : \textbf{code} \vdash e^{ks} : \tau^\sim; \Rightarrow \lambda x : \textbf{code}.$$
$$(\lambda x_\tau : \textbf{type}.$$
$$\overline{((\lambda x_i : \textbf{type}.\ e^{ks}\ )\ e_i^{ks})})$$
$$\textbf{typeof}\ x) : \textbf{code} \to \tau^\sim$$

Case $(e^s = \textbf{fgen}\ [\overline{\alpha_i}](x : \textbf{meta}\ \tau^\alpha).e^s)$.

$$\Gamma \vdash \textbf{fgen}\ [\overline{\alpha_i}](x : \textbf{meta}\ \tau^\alpha).e^s : [\overline{\alpha_i}](\tau^\alpha) \to \tau^\sim \rightsquigarrow \overline{\Lambda\alpha_i}.\lambda x : \tau^\alpha.e^{ks} \overset{lm}{\Rightarrow}^{22}$$
$$\overline{\alpha_i} \vdash \tau^\alpha\ \textbf{wf};\ \Gamma, \overline{\alpha_i : *}, x : \tau^\alpha \vdash e^s : \tau^\sim \rightsquigarrow e^{ks} \overset{I.H.}{\Rightarrow}$$
$$\Gamma^k, \overline{\alpha_i : *}, x : \tau^\alpha \vdash^k e^{ks} : \tau^\sim \Rightarrow \Gamma^k \vdash^k \overline{\Lambda\alpha_i}.\lambda x : \tau^\alpha.e^{ks} : \overline{\forall\alpha_i}.\tau^\alpha \to \tau^\sim.$$

Case $(e^s = \textbf{let}\ x = e_1^s\ \textbf{in}\ e_2^s)$.

$$\Gamma \vdash \textbf{let}\ x = e_1^s\ \textbf{in}\ e_2^s : \tau^s \rightsquigarrow (\lambda x : \tau_1^{ks}.e_2^{ks})\ e_1^{ks} \overset{lm}{\Rightarrow}^{22} \Gamma \vdash e_1^s : \tau_1^s \rightsquigarrow e_1^{ks};$$
$$\Gamma, x : \tau_1^s \vdash e_2^s : \tau^s \rightsquigarrow e_2^{ks} \overset{I.H.}{\Rightarrow} \Gamma^k \vdash^k e_1^{ks} : \tau_1^{ks};\ \Gamma^k, x : \tau_1^{ks} \vdash^k e_2^{ks} : \tau^{ks} \Rightarrow$$
$$\Gamma^k \vdash^k \lambda x : \tau_1^{ks}.e_2^{ks} : \tau_1^{ks} \to \tau^{ks} \Rightarrow \Gamma^k \vdash^k (\lambda x : \tau_1^{ks}.e_2^{ks})\ e_1^{ks} : \tau^{ks}.$$

□

APPENDIX C

Kernel Language Implementation

The following is the full listing for the PLT Redex model of the kernel metaprogramming language.

```
;;
;; kernel.ss - a PLT Redex implementation of the metaprogramming
;; kernel language
;;

;; Author: Ronald Garcia


;; (language (nonterminal-name rhs-pattern ...) ...)
(module kernel mzscheme
  (require (planet "reduction-semantics.ss" ("robby" "redex.plt" 3 9))
           (planet "gui.ss" ("robby" "redex.plt" 3 9)))


(define meta-k
  (language
   (g int bool)
   (x variable-not-otherwise-mentioned)
   (a variable-not-otherwise-mentioned)
   (c number #t #f)
   (f add1 zero? not sub1)
   (bop + < - *)

   ;; metalanguage-specific constants
   (g? int? bool?)
   (cs- g? ->? dom cod typeof)
   (cs g cs-)

   ;; e - code language
   (e x (lam (x e^s) e) (e e) c f (if e e e) (splice e^s)
      (bop e e))


   ;; e^o - pure code
   (e^o x (lam (x t) e^o) (e^o e^o) c f (if e^o e^o e^o)
```

```
        (bop e^o e^o))

   ;; e^s - metalanguage
   (e^s x (lam (x t^s) e^s) (e^s e^s) (fix (x t^s) e^s)
        (tlam a e^s)
        (tapp e^s t^a)
        c f
        (if e^s e^s e^s)
        cs
        (code e)
        (csp e^s)
        (-> e^s e^s)
        (=t e^s e^s)
        (bop e^s e^s))

   ;; v^s - metalanguage values
   (v^s c f cs- (code e^o) t (lam (x t^s) e^s) (tlam a e^s))
   (v^s+ v^s x (fix (x t^s) e^s))

   (t g (-> t t))
   (t^a a g (-> t^a t^a))
   (t^s a g code type (-> t^s t^s) (forall (a) t^s))

   ;; abort expression
   (bottom bottom)

   ;; Contexts
   (E^s hole (E^s e^s) (v^s E^s) (code E)
        (if E^s e^s e^s) (csp E^s) (-> E^s e^s) (-> t E^s)
        (=t E^s e^s) (=t v^s E^s) (tapp E^s t^a)
        (bop E^s e^s) (bop v^s E^s))

   ;; The named hole for E makes the splicing rule possible
   (E (hole spl) (E e) (e^o E) (lam (x E^s) e) (lam (x t) E)
      (splice E^s) (if E e e) (if e^o E e) (if e^o e^o E)
      (bop E e) (bop e^o E))
   ))

;; (reduction-relation meta-kernel
;;   (--> lhs-pattern consequence) ...)
```

```
(define red
  (reduction-relation
   meta-k

   ;; binary operations
   (--> (in-hole E_1 (+ number_1 number_2))
        (in-hole E_1 ,(+ (term number_1) (term number_2)))
        "+")
   (--> (in-hole E_1 (- number_1 number_2))
        (in-hole E_1 ,(- (term number_1) (term number_2)))
        "-")
   (--> (in-hole E_1 (< number_1 number_2))
        (in-hole E_1 ,(< (term number_1) (term number_2)))
        "<")
   (--> (in-hole E_1 (* number_1 number_2))
        (in-hole E_1 ,(* (term number_1) (term number_2)))
        "*")

   ;; unary operations
   (--> (in-hole E_1 (add1 number_1))
        (in-hole E_1 ,(add1 (term number_1)))
        "add1")

   (--> (in-hole E_1 (sub1 number_1))
        (in-hole E_1 ,(sub1 (term number_1)))
        "sub1")

   (--> (in-hole E_1 (zero? number_1))
        (in-hole E_1 ,(zero? (term number_1)))
        "zero?")

   (--> (in-hole E_1 (not #t))
        (in-hole E_1 #f)
        "not-t")
   (--> (in-hole E_1 (not #f))
        (in-hole E_1 #t)
        "not-f")
   ;; ...


   (--> (in-hole E_1 (if #t e^s_1 e^s_2))
        (in-hole E_1 e^s_1)
        "if-t")
   (--> (in-hole E_1 (if #f e^s_1 e^s_2))
        (in-hole E_1 e^s_2)
        "if-f")
```

```
(--> (in-hole E_1 ((lam (x_1 t^s_1) e^s_1) v^s_1))
     (in-hole E_1 (mesubst (x_1 v^s_1 e^s_1)))
     "lam")

(--> (in-hole E_1 (tapp (tlam a_1 e^s_1) t^a_1))
      (in-hole E_1 (tesubst (a_1 t^a_1 e^s_1)))
      "tlam")

(--> (in-hole E_1 (fix (x_1 t^s_1) e^s_1))
     (in-hole E_1 (mesubst (x_1 (fix (x_1 t^s_1) e^s_1) e^s_1)))
     "fix")

(--> (in-hole E_1 (csp c_1))
     (in-hole E_1 (code c_1))
     "csp")

(--> (in-hole E_1 (int? t_1))
     (in-hole E_1 ,(eq? (term t_1) (term int)))
     "int?")

(--> (in-hole E_1 (bool? t_1))
     (in-hole E_1 ,(eq? (term t_1) (term bool)))
     "bool?")

(--> (in-hole E_1 (->? (-> t_1 t_2)))
     (in-hole E_1 #t)
     "->?-t")
(--> (in-hole E_1 (->? t_1))
     (in-hole E_1 #f)
     "->?-f"
     (side-condition
      (not (test-match meta-k (-> t_2 t_3) (term t_1)))))

(--> (in-hole E_1 (dom (-> t_1 t_2)))
     (in-hole E_1 t_1)
     "dom")
(--> (in-hole E_1 (dom t_1))
     (in-hole E_1 t_1)
     "dom-t"
     (side-condition
      (not (test-match meta-k (-> t_2 t_3) (term t_1)))))

(--> (in-hole E_1 (cod (-> t_1 t_2)))
     (in-hole E_1 t_2)
     "cod")
```

```
   (--> (in-hole E_1 (cod t_1))
        (in-hole E_1 t_1)
        "cod-t"
        (side-condition
         (not (test-match meta-k (-> t_2 t_3) (term t_1)))))

   (--> (in-hole E_1 (=t t_1 t_2))
        (in-hole E_1 ,(equal? (term t_1) (term t_2)))
        "=t")

   (--> (in-hole E_1 (typeof (code e^o_1)))
        (in-hole E_1 (compute-type-of (E_1 (typeof (code e^o_1)))))
        "typeof"
        (side-condition
         (test-match meta-k t_1
           (term (compute-type-of (E_1 (typeof (code e^o_1))))))))

   (--> (in-hole E_1 (typeof (code e^o_1)))
        bottom
        "notype"
        (side-condition
         (eq? (term (compute-type-of (E_1 (typeof (code e^o_1)))))
              (term bottom))))


   (--> (in-named-hole spl E_1 (splice (code e^o_1)))
        (in-hole E_1 e^o_1)
        "splice")

   ))


;;
;; The substitution functions
;;

;; mesubst x v^s+ e^s - substitute a  value into a metaterm
(define-metafunction mesubst
  meta-k
  ;; replace x_1
  [(x_1 v^s+_1 x_1) v^s+_1]

  ;; x_1 and x_2 are different, so don't replace
  [(x_1 v^s+_1 x_2) x_2
   (side-condition (not (eq? (term x_1) (term x_2))))]
```

```
;; ignore type variables
[(x_1 v^s+ a_1) a_1]

;; x_1 bound, so don't continue in body...
[(x_1 v^s+_1 (lam (x_1 t^s_1) e^s_1))
 (lam (x_1 t^s_1) e^s_1)]

;; general purpose capture avoiding case
[(x_1 v^s+_1 (lam (x_2 t^s_1) e^s_1))
 ,(term-let ([x_new
              (variable-not-in (term (v^s+_1 e^s_1)) (term x_2))])
    (term
     (lam (x_new t^s_1)
          (mesubst (x_1 v^s+_1 (mesubst (x_2 x_new e^s_1)))))))]

;; x_1 bound, so don't continue in body...
[(x_1 v^s+_1 (fix (x_1 t^s_1) e^s_1))
 (fix (x_1 t^s_1) e^s_1)]

;; general purpose capture avoiding case
[(x_1 v^s+_1 (fix (x_2 t^s_1) e^s_1))
 ,(term-let ([x_new
              (variable-not-in (term (v^s+_1 e^s_1)) (term x_2))])
    (term
     (fix (x_new t^s_1)
          (mesubst (x_1 v^s+_1 (mesubst (x_2 x_new e^s_1)))))))]


;; term application
[(x_1 v^s+_1 (e^s_1 e^s_2))
 ((mesubst (x_1 v^s+_1 e^s_1)) (mesubst (x_1 v^s+_1 e^s_2)))]

;; conditionals
[(x_1 v^s+_1 (if e^s_1 e^s_2 e^s_3))
 (if (mesubst (x_1 v^s+_1 e^s_1))
     (mesubst (x_1 v^s+_1 e^s_2))
     (mesubst (x_1 v^s+_1 e^s_3)))]

;; type abstraction
[(x_1 v^s+_1 (tlam a_1 e^s_1))
 (tlam a_1 (mesubst (x_1 v^s+_1 e^s_1)))]

;; type application
[(x_1 v^s+_1 (tapp e^s_1 t^a_1))
 (tapp (mesubst (x_1 v^s+_1 e^s_1)) t^a_1)]
```

```
  ;; basic constants
  [(x_1 v^s+_1 c_1)  c_1]

  ;; function constants
  [(x_1 v^s+_1 f_1)  f_1]

  ;; metaconstants
  [(x_1 v^s+_1 cs_1)  cs_1]

  ;; type equality
  [(x_1 v^s+_1 (=t e^s_1 e^s_2))
   (=t (mesubst (x_1 v^s+_1 e^s_1)) (mesubst (x_1 v^s+_1 e^s_2)))]

  ;; arrow expressions
  [(x_1 v^s+_1 (-> e^s_1 e^s_2))
   (-> (mesubst (x_1 v^s+_1 e^s_1)) (mesubst (x_1 v^s+_1 e^s_2)))]

  ;; binary operations
  ;; arrow expressions
  [(x_1 v^s+_1 (bop_1 e^s_1 e^s_2))
   (bop_1 (mesubst (x_1 v^s+_1 e^s_1))
          (mesubst (x_1 v^s+_1 e^s_2)))]

  ;; code expressions
  [(x_1 v^s+_1 (code e_1))
   (code (mcsubst (x_1 v^s+_1 e_1)))]

  ;; csp expressions
  [(x_1 v^s+_1 (csp e^s_1))
   (csp (mesubst (x_1 v^s+_1 e^s_1)))])



;; mcsubst x v^s+ e - substitute a value into a code term
(define-metafunction mcsubst
  meta-k
  ;; let it alone
  [(x_1 v^s+ x_2) x_2]


  ;; general purpose capture avoiding case
  [(x_1 v^s+_1 (lam (x_2 e^s_1) e_1))
   ,(term-let ([x_new
                (variable-not-in (term (v^s+_1 e_1)) (term x_2))])
      (term
       (lam (x_new (mesubst (x_1 v^s+_1 e^s_1)))
```

```
            (mcsubst (x_1 v^s+_1 (cvcsubst (x_2 x_new e_1)))))))]


  [(x_1 v^s+_1 (e_1 e_2))
   ((mcsubst (x_1 v^s+_1 e_1)) (mcsubst (x_1 v^s+_1 e_2)))]

  [(x_1 v^s+_1 (if e_1 e_2 e_3))
   (if (mcsubst (x_1 v^s+_1 e_1))
       (mcsubst (x_1 v^s+_1 e_2))
       (mcsubst (x_1 v^s+_1 e_3)))]

  [(x_1 v^s+_1 (bop_1 e_1 e_2))
   (bop_1 (mcsubst (x_1 v^s+_1 e_1))
       (mcsubst (x_1 v^s+_1 e_2)))]


  [(x_1 v^s+_1 c_1)  c_1]

  [(x_1 v^s+_1 f_1)  f_1]

  [(x_1 v^s+_1 (splice e^s_1))
   (splice (mesubst (x_1 v^s+_1 e^s_1)))])



;; cvesubst x x e^s - substitute a code variable into a metaterm
(define-metafunction cvesubst
  meta-k

  ;; ignore meta variables
  [(x_1 x_n x_2) x_2]

  ;; ignore type variables
  [(x_1 x_n a_1) a_1]

  ;; general purpose capture avoiding case
  [(x_1 x_n (lam (x_2 t^s_1) e^s_1))
   ,(term-let ([x_new
                (variable-not-in (term (x_n e^s_1)) (term x_2))])
      (term
       (lam (x_new t^s_1)
            (cvesubst (x_1 x_n (mesubst (x_2 x_new e^s_1)))))))]

  ;; general purpose capture avoiding case
  [(x_1 x_n (fix (x_2 t^s_1) e^s_1))
   ,(term-let ([x_new
```

```
              (variable-not-in (term (x_n e^s_1)) (term x_2))])
    (term
     (fix (x_new t^s_1)
          (cvesubst (x_1 x_n (mesubst (x_2 x_new e^s_1)))))))]


;; term application
[(x_1 x_n (e^s_1 e^s_2))
 ((cvesubst (x_1 x_n e^s_1)) (cvesubst (x_1 x_n e^s_2)))]

;; conditionals
[(x_1 x_n (if e^s_1 e^s_2 e^s_3))
 (if (cvesubst (x_1 x_n e^s_1))
     (cvesubst (x_1 x_n e^s_2))
     (cvesubst (x_1 x_n e^s_3)))]

;; type abstraction
[(x_1 x_n (tlam a_1 e^s_1))
 (tlam a_1 (cvesubst (x_1 x_n e^s_1)))]

;; type application
[(x_1 x_n (tapp e^s_1 t^a_1))
 (tapp (cvesubst (x_1 x_n e^s_1)) t^a_1)]

;; basic constants
[(x_1 x_n c_1)  c_1]

;; function constants
[(x_1 x_n f_1)  f_1]

;; metaconstants
[(x_1 x_n cs_1)  cs_1]

;; type equality
[(x_1 x_n (=t e^s_1 e^s_2))
 (=t (cvesubst (x_1 x_n e^s_1)) (cvesubst (x_1 x_n e^s_2)))]

;; arrow expressions
[(x_1 x_n (-> e^s_1 e^s_2))
 (-> (cvesubst (x_1 x_n e^s_1)) (cvesubst (x_1 x_n e^s_2)))]

;; binary operations
[(x_1 x_n (bop_1 e^s_1 e^s_2))
 (bop_1 (cvesubst (x_1 x_n e^s_1)) (cvesubst (x_1 x_n e^s_2)))]

;; code expressions
```

```
  [(x_1 x_n (code e_1))
   (code (cvcsubst (x_1 x_n e_1)))]

  ;; csp expressions
  [(x_1 x_n (csp e^s_1))
   (csp (cvesubst (x_1 x_n e^s_1)))])

;; cvcsubst x x e - substitute a code variable into a code term
(define-metafunction cvcsubst
  meta-k

  ;; replace x_1
  [(x_1 x_n x_1) x_n]

  ;; x_1 and x_2 are different, so don't replace
  [(x_1 x_n x_2) x_2
   (side-condition (not (eq? (term x_1) (term x_2))))]

  ;; x_1 bound, so don't continue in body...
  [(x_1 x_n (lam (x_1 e^s_1) e^s_2))
   (lam (x_1 (cvesubst (x_1 x_n e^s_1))) e^s_2)]


  ;; general purpose capture avoiding case
  [(x_1 x_n (lam (x_2 e^s_1) e_1))
   ,(term-let ([x_new
                (variable-not-in (term (x_n e_1)) (term x_2))])
      (term
       (lam (x_new (cvesubst (x_1 x_n e^s_1)))
            (cvcsubst (x_1 x_n (cvcsubst (x_2 x_new e_1)))))))]


  [(x_1 x_n (e_1 e_2))
   ((cvcsubst (x_1 x_n e_1)) (cvcsubst (x_1 x_n e_2)))]

  [(x_1 x_n (if e_1 e_2 e_3))
   (if (cvcsubst (x_1 x_n e_1))
       (cvcsubst (x_1 x_n e_2))
       (cvcsubst (x_1 x_n e_3)))]

  [(x_1 x_n (bop_1 e_1 e_2))
   (bop_1 (cvcsubst (x_1 x_n e_1))
       (cvcsubst (x_1 x_n e_2)))]

  [(x_1 x_n c_1)  c_1]
```

```
  [(x_1 x_n f_1)  f_1]

  [(x_1 x_n (splice e^s_1))
   (splice (cvesubst (x_1 x_n e^s_1)))])



;; tesubst a  e^s - substitute a type schema into a metaterm
(define-metafunction tesubst
  meta-k
  ;; replace a_1 with e_1
  [(a_1 t^a_1 a_1) t^a_1]

  ;; a_1 and a_2 are different, so don't replace
  [(a_1 t^a_1 a_2) a_2
   (side-condition (not (eq? (term a_1) (term a_2))))]

  ;; ignore term variables
  [(a_1 t^a x_1) x_1]

  ;; a_1 bound, so don't continue in body...
  [(a_1 t^a_1 (tlam a_1 e^s_1))
   (tlam a_1 e^s_1)]

  ;; general purpose capture avoiding case
  [(a_1 t^a_1 (tlam a_2 e^s_1))
   ,(term-let ([a_new
                (variable-not-in (term (t^a_1 e^s_1)) (term a_2))])
      (term
       (tlam a_new
            (tesubst (a_1 t^a_1 (tesubst (a_2 a_new e^s_1)))))))]

  ;; just go into fix
  [(a_1 t^a_1 (fix (x_1 t^s_1) e^s_1))
   (fix (x_1 (ttsubst (a_1 t^a_1 t^s_1)))
      (tesubst (a_1 t^a_1 e^s_1)))]


  ;; term application
  [(a_1 t^a_1 (e^s_1 e^s_2))
   ((tesubst (a_1 t^a_1 e^s_1)) (tesubst (a_1 t^a_1 e^s_2)))]

  ;; conditionals
  [(a_1 t^a_1 (if e^s_1 e^s_2 e^s_3))
   (if (tesubst (a_1 t^a_1 e^s_1))
       (tesubst (a_1 t^a_1 e^s_2))
```

```
      (tesubst (a_1 t^a_1 e^s_3)))]

  ;; term abstraction
  [(a_1 t^a_1 (lam (x_1 t^s_1) e^s_1))
   (lam (x_1 (ttsubst (a_1 t^a_1 t^s_1)))
      (tesubst (a_1 t^a_1 e^s_1)))]

  ;; type application
  [(a_1 t^a_1 (tapp e^s_1 t^a_2))
   (tapp (tesubst (a_1 t^a_1 e^s_1)) (ttsubst (a_1 t^a_1 t^a_2)))]

  ;; basic constants
  [(a_1 t^a_1 c_1)  c_1]

  ;; function constants
  [(a_1 t^a_1 f_1)  f_1]

  ;; metaconstants
  [(a_1 t^a_1 cs_1)  cs_1]

  ;; type equality
  [(a_1 t^a_1 (=t e^s_1 e^s_2))
   (=t (tesubst (a_1 t^a_1 e^s_1)) (tesubst (a_1 t^a_1 e^s_2)))]

  ;; arrow expressions
  [(a_1 t^a_1 (-> e^s_1 e^s_2))
   (-> (tesubst (a_1 t^a_1 e^s_1)) (tesubst (a_1 t^a_1 e^s_2)))]

  ;; binary operations
  [(a_1 t^a_1 (bop_1 e^s_1 e^s_2))
   (bop_1 (tesubst (a_1 t^a_1 e^s_1)) (tesubst (a_1 t^a_1 e^s_2)))]

  ;; code expressions
  [(a_1 t^a_1 (code e_1))
   (code (tcsubst (a_1 t^a_1 e_1)))]

  ;; csp expressions
  [(a_1 t^a_1 (csp e^s_1))
   (csp (tesubst (a_1 t^a_1 e^s_1)))])


;; tcsubst x t^a e - substitute a type schema into a code term
(define-metafunction tcsubst
  meta-k
  ;; replace x_1 with e_1
  [(a_1 t^a x_2) x_2]
```

```
  ;; term abstraction
  [(a_1 t^a_1 (lam (x_1 e^s_1) e_1))
   (lam (x_1 (tesubst (a_1 t^a_1 e^s_1)))
      (tcsubst (a_1 t^a_1 e_1)))]

  [(a_1 t^a_1 (e_1 e_2))
   ((tcsubst (a_1 t^a_1 e_1)) (tcsubst (a_1 t^a_1 e_2)))]

  [(a_1 t^a_1 (if e_1 e_2 e_3))
   (if (tcsubst (a_1 t^a_1 e_1))
       (tcsubst (a_1 t^a_1 e_2))
       (tcsubst (a_1 t^a_1 e_3)))]

  [(a_1 t^a_1 (bop_1 e_1 e_2))
   (bop_1 (tcsubst (a_1 t^a_1 e_1))
          (tcsubst (a_1 t^a_1 e_2)))]

  [(a_1 t^a_1 c_1)  c_1]

  [(a_1 t^a_1 f_1)  f_1]

  [(a_1 t^a_1 (splice e^s_1))
   (splice (tesubst (a_1 t^a_1 e^s_1)))])


;; ttsubst a t^a t^s - substitute a type schema into a meta-type.
(define-metafunction ttsubst
  meta-k
  ;; replace a_1 with e_1
  [(a_1 t^a_1 a_1) t^a_1]

  ;; a_1 and a_2 are different, so don't replace
  [(a_1 t^a_1 a_2) a_2
   (side-condition (not (eq? (term a_1) (term a_2))))]

  ;; ignore base types
  [(a_1 t^a_1 g) g]

  [(a_1 t^a_1 code) code]

  [(a_1 t^a_1 type) type]

  [(a_1 t^a_1 (-> t^s_1 t^s_2))
   (-> (ttsubst (a_1 t^a_1 t^s_1)) (ttsubst (a_1 t^a_1 t^s_2)))]
```

```
  [(a_1 t^a_1 (forall (a_1) t^s_1))
   (forall (a_1) t^s_1)]


  ;; general purpose capture avoiding case
  [(a_1 t^a_1 (forall (a_2) t^s_1))
   ,(term-let ([a_new
                (variable-not-in (term (t^a_1 t^s_1)) (term a_2))])
      (term
       (forall (a_new)
            (ttsubst (a_1 t^a_1 (ttsubst (a_2 a_new t^s_1)))))))])


;; given a metacode context and a typeof expression,
;; compute the type of the code in the argument to typeof.
(define-metafunction compute-type-of
  meta-k
  [(E_1 (typeof (code e^o_1)))
   ,(term-let ([any_1 (make-gamma (term E_1))])
      (term (type-of (any_1 e^o_1))))])



;; Extract all the relevant lambda bindings from the context.
(define (make-gamma E)

  (let ([m
         ;; Use test-match to extract all the (x_1 t_1) pairs
         ;; in the current context.
         ;; This relies on the outside-in ordering of matches.
         (test-match meta-k
             (in-named-hole spl E_2
                            (in-hole (lam (x_1 t_1) E_1) hole))
             E)])
    (if m
        (reverse ;; because we want them in inside-out order
         (map
          (lambda (m)
            (let ([ribs (bindings-table (mtch-bindings m))])
              (let ([alist (map cons (map rib-name ribs)
                                (map rib-exp ribs))])
                (list (cdr (assq (term x_1) alist))
                      (cdr (assq (term t_1) alist))))))) m))
        '())))
```

```
;; typeof :: (Gamma e^o) -> t
(define-metafunction type-of
  meta-k
  [(any_1 #t) bool]
  [(any_1 #f) bool]
  [(any_1 number_1) int]
  [(any_1 add1) (-> int int)]
  [(any_1 zero?) (-> int bool)]
  [(any_1 not) (-> bool bool)]
  [(any_1 (+ e_1 e_2))
   int
   (side-condition (eq? (term (type-of (any_1 e_1))) (term int)))
   (side-condition (eq? (term (type-of (any_1 e_2))) (term int)))]
  [(any_1 (- e_1 e_2))
   int
   (side-condition (eq? (term (type-of (any_1 e_1))) (term int)))
   (side-condition (eq? (term (type-of (any_1 e_2))) (term int)))]
  [(any_1 (< e_1 e_2))
   bool
   (side-condition (eq? (term (type-of (any_1 e_1))) (term int)))
   (side-condition (eq? (term (type-of (any_1 e_2))) (term int)))]
  [(any_1 (if e_1 e_2 e_3))
   (type-of (any_1 e_2))
   (side-condition (eq? (term (type-of (any_1 e_1))) (term bool)))
   (side-condition (equal? (term (type-of (any_1 e_2)))
                           (term (type-of (any_1 e_3)))))]
  [(((x_1 t_1) ...) x_2)
   ,(cadr (assq (term x_2) (term ((x_1 t_1) ...))))
   (side-condition (assq (term x_2) (term ((x_1 t_1) ...))))]
  [(any_1 (e_1 e_2))
   ,(term-let ([t_2
                ((term-match/single meta-k
                   [(-> t_1 t_2) (term t_2)])
                 (term (type-of (any_1 e_1))))])
       (term t_2))
   (side-condition (test-match meta-k
                     (-> t_1 t_2) (term (type-of (any_1 e_1)))))
   (side-condition
    (term-let ([t_1
                ((term-match/single meta-k
                   [(-> t_1 t_2) (term t_1)])
                 (term (type-of (any_1 e_1))))])
       (eq? (term t_1) (term (type-of (any_1 e_2))))))]
   [(((x_1 t_1) ...) (lam (x_2 t_2) e_1))
       ,(term-let ([t_3 (term
```

```
                        (type-of (((x_2 t_2) (x_1 t_1) ...) e_1)))])
         (term (-> t_2 t_3)))]
   ;; no type
   [(any_1 any_2) bottom])



(define (show-trace term) (traces meta-k red term))
(define (red* term) (apply-reduction-relation* red term))
(define (k-red term) (car (red* term)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Examples programs
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(k-red '(splice ((lam (f code)
           (code (lam (x int) (* x (splice f)))))
      (code (+ x 7)))))
;; (lam (x1 int) (* x1 (+ x 7)))



(k-red
'(splice (((tapp (tlam a
               (lam (f (-> a bool))
                    (lam (c a)
                         (if (f c)
                             (code (lam (t a)
                                        (lam (f a) t)))
                             (code (lam (t a)
                                        (lam (f a) f)))))))
         int) zero?) 5)))
;; (lam (t int) (lam (f2 int) f2))


(k-red
 '(splice ((lam (n int) (code (* 7 (splice (csp (+ n 4)))))) 6)))
;; (* 7 10)


(k-red
 '(splice ((lam (f (-> code code))
                 (code (* (splice (f (code 2)))
                          (splice (f (code 3))))))
```

```
            (lam (x code) (code (+ 5 (splice x)))))))
;; (* (+ 5 2) (+ 5 3))


(k-red
 '(lam (x int) ((lam (y (typeof (code (zero? x)))) (* x 4)) #f)))
;; (lam (x int) ((lam (y bool) (* x 4)) #f))


(k-red
 '(splice ((lam (ctsum int)
                (code ((lam (rtsum int)
                            (- rtsum (splice (csp ctsum))))
                       (+ 5 3))))
          (+ 5 3))))
;; ((lam (rtsum1 int) (- rtsum1 8)) (+ 5 3))


(k-red
 '(splice ((lam (pow (-> int (-> int int)))
              (csp ((pow 5) 7)))
          (fix (p (-> int (-> int int)))
               (lam (m int)
                    (lam (n int)
                         (if (zero? n)
                             1
                             (* m ((p m) (sub1 n))))))))))
;; 78125




(k-red
 '(splice ((lam (pow (-> code (-> int code)))
              (code (lam (m int) (splice ((pow (code m)) 7)))))
          (fix (p (-> code (-> int code)))
               (lam (m code)
                    (lam (n int)
                         (if (zero? n)
                             (code 1)
                             (code (* (splice m)
                                      (splice ((p m) (sub1 n))))))))))))
;; (lam (m4 int) (* m4 (* m4 (* m4 (* m4 (* m4 (* m4 (* m4 1))))))))


(k-red
```

```
 '(splice ((lam (numargs (-> type code))
                (csp (numargs (-> int (-> bool int)))))
           (fix (n (-> type code))
                (lam (t type)
                     (if (->? t)
                         (add1 (n (cod t)))
                         0))))))
;; 2


) ;; module meta-k
```

Ronald Garcia

Department of Computer Science
Lindley Hall 215
150 S. Woodlawn Ave.
Bloomington, IN 47405-7104
(812) 855-4829 (FAX)

1724 East Hillside Drive, Apt B
Bloomington, IN 47401
Tel: (203) 241-1643
Email: garcia@cs.indiana.edu
Url: `http://osl.iu.edu/~garcia/`

Interests Programming language design, domain-specific languages, and software engineering.

Education

Ph.D., Computer Science, Indiana University.
Thesis: *Static Computation and Reflection* (Committee: Andrew Lumsdaine, Chair, Daniel P. Friedman, Amr Sabry, R. Kent Dybvig, Charles Livingston), 2008
Ph.D. Minor: Mathematics.

M.S., Electrical Engineering, University of Notre Dame, 1999.

B.S., Electrical Engineering, University of Notre Dame, 1997.

Awards Notre Dame Department of Electrical Engineering Graduate Student Fellowship.
Hubert Schlafly Undergraduate Minority Engineering Scholarship.

Refereed Journal Articles

1. Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An Extended Comparative Study of Language Support for Generic Programming. In *Journal of Functional Programming*, 17 (2), 145–205. 2007.
2. Ronald Garcia and Andrew Lumsdaine. MultiArray: A C++ Library for Generic Programming with Arrays. In *Software—Practice and Experience*, 35 (2), 159-188. 2005.

Refereed Conference Papers Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A Comparative Study of Language Support for Generic Programming. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*. October 2003.

Refereed Workshop Papers Ronald Garcia and Andrew Lumsdaine. Type Classes Without Types. In *Scheme 2005: The 2005 ACM SIGPLAN Workshop on Scheme and Functional Programming*. September 2005.

Technical Reports

1. Jeremy Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++. Report number N1758=05-0018. 2005.
2. Ronald Garcia, Jeremy Siek, Ruj Akavipat, Samuel Chun, David W. Mack, Heather Roinestad, and Daniel P. Friedman. Using ParentheC to Transform Scheme Programs to C. December 2003. Available at http://www.cs.indiana.edu/l/www/classes/c311/ParentheC.pdf.

Presentations

1. Computing While Compiling: Reasons and Methods for Compile-time Metaprogramming. University of Oregon. May 2008.
2. Computing While Compiling: Reasons and Methods for Compile-time Metaprogramming. Rice University. May 2008.

3. Computing While Compiling: Reasons and Methods for Compile-time Metaprogramming. Argonne National Laboratory. January 2008.
4. Computing While Compiling: Reasons and Methods for Compile-time Metaprogramming. Wesleyan University. December 2007.
5. Compile-time Metaprogramming. Connecticut College. December 2007.
6. Static Computation and Reflection: Practice and Theory. At the 2007 LogicBlox Inc. Researchers Symposium. September 2007.
7. Static Computation and Reflection: Practice and Theory. At the 2007 Summer School on Generative and Transformational Techniques in Software Engineering, Participants Workshop. July 2007
8. A Principled Approach to Compile-time Metaprogramming. Poster presented at the CRA/CDC Programming Languages Summer School. May 2007
9. Type Classes Without Types. At the 2005 ACM SIGPLAN Workshop on Scheme and Functional Programming. September 2005.
10. A Comparative Study of Language Support for Generic Programming. At the 2003 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03). October 2003.

Research Experience

Open Systems Laboratory at Indiana University 2001-2008
Graduate Research Assistant Bloomington, IN
Investigating tools and techniques for developing generic libraries and domain-specific optimizations. Investigating language mechanisms for compile-time meta-programming.

Laboratory for Scientific Computing at Notre Dame 2000-2001
Graduate Research Assistant South Bend, IN
Developed a generic library for multi-dimensional arrays in C++, now a part of the Boost C++ Library collection.

Processor-in-Memory Group at Notre Dame 1997-2000
Graduate Research Assistant South Bend, IN
Assisted in laying out the data path for an at-the-sensamp microprocessor design. Fabricated at Lincoln Labs. Helped design an architecture for a processor-in-memory based microprocessor. Developed several instruction-set simulators during the architecture refinement process.

Design Automation Laboratory at Notre Dame 1995-1997
Undergraduate Research Assistant South Bend, IN
Helped develop a software system for simultaneous optimization of fabrication process and circuit layout for integrated circuits.

Industry Experience

PDF Solutions, Inc. Summer 1997
Summer Intern San Jose, CA
Software Engineering. Developed infrastructure tools for use in production software. Performed maintenance and feature enhancement on a commercial product.

PDF Solutions, Inc. Summer 1996
Summer Intern San Jose, CA
Various software development roles on production and research applications.

Delco Electronics Summer 1995
Summer Intern – Parts Reliability Kokomo, IN

Carried out experiments to characterize the information provided by electrostatic discharge, latch-up, and combined test systems.

Service

Boost C++ Libraries Project 2000–present
Developed the MultiArray library and contributed to others. Currently serving as review manager coordinator. Responsible for arranging and reporting on Boost library reviews.

Task Force on Undergraduate Curriculum and Women 2003
This task force investigated how Indiana University's undergraduate computer science curriculum and related departmental processes affected the enrollment and retention of women. We provided an advisory report to the computer science department.

Referee for conferences and journals

- European Symposium on Programming 2006;
- Higher-Order and Symbolic Computing;
- Science of Computer Programming;
- ACM Workshop on Partial Evaluation and Program Manipulation 2008
- ACM Transactions on Mathematical Software

Software

The Boost MultiArray Library (available at `www.boost.org`).