Multi-Stage Programming: Its Theory and Applications

Walid Taha
B.S. Computer Science and Engineering, 1993, Kuwait University.

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

November 1999

The dissertation "Multi-Stage Programming: Its Theory and Applications" by Walid Taha has been examined and approved by the following Examination Committee:

Tim Sheard
Associate Professor
Thesis Research Adviser

James Hook
Associate Professor

John Launchbury
Associate Professor

David Maier
Professor

Neil D. Jones
Professor, University of Copenhagen

Eugenio Moggi
Professor, University of Genova

Dedication

To Sohair, Mohamed, Nodar, Hanan, and Hamid.

Why is my verse so barren of new pride,

So far from variation or quick change?

Why with the time do I not glance aside

To new-found methods and to compounds strange?

Why write I still all one, ever the same,

And keep invention in a noted weed,

That every word doth almost tell my name,

Showing their birth and where they did proceed?

O, know, sweet love, I always write of you,

And you and love are still my argument;

So all my best is dressing old words new,

Spending again what is already spent:

For as the sun is daily new and old,

So is my love still telling what is told.

Sonnet LXXVI, Shakespeare

Acknowledgements

The creativity, energy, enthusiasm of Prof. Tim Sheard, my advisor, have been a tremendous source of support throughout my four and half years of graduate studies. His guidance and support are the reason why I am completing my dissertation today.

An Arabic proverb says that companions should be chosen before route. This is not easy when the school you want to go to is half-way around the world. Despite that, I must say, I have had excellent company during my stay at OGI. For many memorable times, I am indebted to Davide Anacona, Veronica Baiceanu, Byron Cook, Scott Daniels, Joey Flack, Anita Govindarajan, Phil Galbiati, Ashish and Shailesh Godbole, Brian Hansen, Paul Hosom, Mary Hultine, Jon Inouye, Jeff Lewis, Roz Lindquist, John and Julie Mathews, Emir Pasalic, Matt Saffell, Amol and Alok Vyas (and the Indian Mafia at large), Bennett Vance, Karen Ward, Lisa Walton, Wei Wei, and many other wonderful friends. The many emails and intermittent visits with Tamer Abdel-Rahim, Wael Fatin, Ashraf Hamalawy, Sarah Miller, Wael Masri, and Kelly Schreckengost provided me with lots of sunshine when the Portland clouds tried to protect me from it.

Our library did a miraculous job at responding to my continual requests for inter-library loans, especially as I came closer to graduation and wanted to "tie everything together". A special thanks goes to Mary Hultine for her support and continual interest in my native culture and language. I am also very grateful to Julianne Williams for her promptness and patience in helping me prepare the bibliography for this dissertation.

The systems support staff of CSE have been most accommodating for my many demands during my years at OGI. I would especially like to thank Marion Håkanson for helping me quickly recover from endless goofs, especially the time when I deleted a chapter of my dissertation.

I would like to thank the administrative staff of OGI in general, and of PacSoft in particular. Kelly Atkinson and Therese Fisher have continually done miracles to keep me a happy camper both when I was at OGI and when I was away from OGI during the many business trips that they had always arranged with meticulous care.

Credits and Personal Contribution: I had the good fortune of finding many of the people that I wanted to thank ending up being on my thesis committee. My thesis committee played an active role in the development of this work from beginning to end. I would like to thank Neil Jones for his careful reading of my thesis proposal, and bringing much needed fresh insights into the various formulations of MetaML's semantics. Neil pointed out to me the amazing works of Quine. I would like to thank Eugenio Moggi for his careful reading of my dissertation, and for his many suggestions for improving the technical presentation. I am especially grateful to Eugenio for being critical and also doubtful of the possibility of relating the reduction and big-step semantics. Long before I started working on my thesis, David Maier's class on Automata helped me regain the courage to explore theoretical aspects of programming languages. While I was writing my dissertation, David preempted many of my attempts to massacre the English language. James Hook's classes on Semantics offered a challenge that, as evidenced by this dissertation, I eventually decided to take up. The numerous illuminating discussions with James Hook about both Software Engineering and Theoretical Computer Science have had a significant influence on my work. John Launchbury's classes on Category Theory and Formal Software Development, and his emphasis on having the students work through proofs were the reason I developed an appetite for these two topics. There is a substantial "energy barrier" involved in understanding what these two topics are about, and John is a magician when it comes to helping people understand. I am especially indebted to James Hook and John Launchbury for the many challenging questions that they have posed to me during the years of my dissertation work, and to which I hope they found satisfactory answers in this dissertation.

I would also like to thank Lois Delcambre and Dick Kieburtz, in addition to my examiners, for serving on my thesis proposal committee.

While conducting the research presented in this dissertation, and during its preparation, I have received substantial technical help from a number of people. The idea of manually staged programs and MetaML's original implementation is primarily due to Sheard. Cross-stage persistence and gensym-renaming of free variable were already being used in Sheard's original MetaML implementation. Multi-stage programming with explicit annotations is Sheard's conception. I just gave these things names, and went on to study them formally. Rowan Davies gave me a concrete example of the limitations of the original type system for MetaML. Zine El-Abidine Benaissa guided me in doing "strip-lemma" confluence proofs for an early version of the reduction semantics. The work on an earlier version of the first type system for MetaML is joint with Zine. I also had the great pleasure of working with Eugenio Moggi. Eugenio proposed simplifications to the original type system, and played the key role in developing the category for interpreting MetaML and its extensions. I developed the interpretation of MetaML into this model jointly with him. Eugenio also proposed and spearheaded the study of a language that combined the two logical modalities previously explored by Davies and Pfenning, which lead us to a framework where the importance of closedness could be explained clearly. I had very fruitful discussions with Franklyn Turbak about observational equivalence proofs and Plotkin's paper on call-by-name, call-by-value, and the lambda calculus. Hongwei Xi pointed out to me Takahashi's paper on parallel reduction. I also had very useful discussions with Hongwei on the details of standardization proofs. Discussions with Eugenio Moggi and Sava Kristic greatly sharpened my presentation of the results on the soundness of MetaML's reduction semantics.

The research on MetaML has also benefited from valuable inputs from Don Batory, Koen Claessen, Olivier Danvy, Rowan Davies, Robert Glück, Sam Kamin, Peter Lee, Erik Meijer, Flemming Nielson, Dino Oliva, Frank Pfenning, Amr Sabry, Mark Shields, Yannis Smaragdakis, Carolyn Talcott, Phil Wadler and the members of PacSoft.

Many people have been kind enough to read parts of this dissertation and gave me feedback that improved the presentation and sharpened my thought. I would like to thank Lennart Augustsson, Tito Autry, Byron Cook, Leo Fegaras, Mark Jones, Alex Kotov, Abdel Hamid Taha, and Eelco Visser. All mistakes and all other results in this dissertation are my responsibility.

Parts of the present dissertation build on works by the author and collaborators previously published elsewhere [94, 91, 55, 5].

Funding and Financial Support: I am very grateful to the Oregon Graduate Institute of Technology, and the Pacific Software Research Institute (PacSoft) for generously supporting me financially throughout my doctoral studies, and covering all my expenses related to attending numerous workshops, conferences and summer schools. My stipend was provided by the National Science Foundation (NSF) *Type Safe Program Generators* award #IRI9625462. My tuition was provided by the Air Force Material Command *Phase I & II* and *Phase III* contracts #F19628-93-C-0069 and #F19628-96-C-0161, respectively.

I would also like to thank the University of Copenhagen, Lucent Technologies, and the University of Genova for supporting on different occasions. The University of Copenhagen (DIKU) provided funding for the Region Analysis school, and for a one week research visit in Copenhagen. Lucent Technologies Bell-Laboratories supported me during a most enjoyable summer internship, and also provided funding for my attendance of the Region Analysis school. The University of Genova (DISI) provided funding for a one month research visit in Genova.

Table of Contents

Abstract

Multi-Stage Programming:
Its Theory and Applications

Walid Taha

Supervising Professor: Tim Sheard

MetaML is a statically typed functional programming language with special support for program generation. In addition to providing the standard features of contemporary programming languages such as Standard ML, MetaML provides three staging annotations. These staging annotations allow the construction, combination, and execution of object-programs.

Our thesis is that MetaML's three staging annotations provide a useful, theoretically sound basis for building program generators. This dissertation reports on our study of MetaML's staging constructs, their use, their implementation, and their formal semantics. Our results include an extended example of where MetaML allows us to produce efficient programs, an explanation of why implementing these constructs in traditional ways can be challenging, two formulations of MetaML's semantics, a type system for MetaML, and a proposal for extending MetaML with a type construct for closedness.

The dissertation consolidates a number of previous publications by the author, including MetaML's type systems and big-step semantics. The presentation is new. The proposed solution to an implementation problem and the reduction semantics for MetaML's three staging constructs are also new.

Chapter 1

Introduction

If thought corrupts language,
language can also corrupt thought.

Politics and the English Language,
George Orwell

Program generation is a powerful and pervasive technique for the development of software. It has been used to improve code reuse, product reliability and maintainability, performance and resource utilization, and developer productivity [43, 84, 87, 92]. Despite the success of program generation, there has been little special support for *writing* generators in high-level programming languages such as C++, Java, Standard ML (SML) or Haskell. A host of fundamental problems inherent in program generation can be addressed effectively by a programming language designed specifically to support writing program generators. MetaML is a novel example of such a meta-programming language. This dissertation shows that MetaML is a concise and expressive meta-programming language based on a solid formal foundation.

This chapter begins by explaining the need for special language constructs to support program generation and meta-programming, and the role of MetaML in this context. To outline the scope of this dissertation, we[1] present a basic classification of meta-programming that distinguishes what we call multi-stage programming from other kinds of meta-programming. We relate multi-stage programming to partial evaluation and multi-level languages, state our thesis, and preview the organization of this dissertation.

[1] In this dissertation, I use "we" to refer primarily to myself. Readers are encouraged to include themselves in this reference at their discretion.

1.1 Program Generation as Meta-Programming

Meta-programs are programs that manipulate other programs. *Object-programs* are programs manipulated by other programs. *Program generators* are meta-programs that produce object-programs as their final result. *MetaML* is a general-purpose programming language designed so that object-programs can be manipulated in both a concise and type-safe manner. The notion of type-safety of MetaML is strong: Not only is a meta-program guaranteed to be free of run-time errors, but so are all of its *object-programs*. Furthermore, because MetaML's meta-programming constructs are both concise and expressive, it facilitates a powerful method of developing meta-programs that we call multi-stage programming.

MetaML provides the user with constructs for building, combining and executing object-programs, all within the same statically-typed language and within the same run-time environment. The benefits of MetaML can be illustrated by contrasting meta-programming in MetaML with meta-programming in a mainstream general-purpose language. In particular, MetaML constitutes a clear improvement over a popular general-purpose language along the following qualitative dimensions of meta-programming language design:

1. Support for syntactic correctness and conciseness in specifying object-programs,
2. Support for type correctness of object-programs,
3. Efficient combination of object-programs,
4. Semantic correctness of object-programs, and
5. Support for staging-correctness.

In addition, MetaML enjoys two desirable reflective properties. In the rest of this section, we elaborate on each of these design dimensions and on where MetaML lies in this design space.

Syntactic Correctness and Conciseness: In a general-purpose language, fragments of an object-program are typically encoded using either strings or datatypes. With the string encoding, we represent the code fragment `f (x,y)` simply as "f (x,y)". While constructing and combining fragments represented by strings can be done concisely, deconstructing them is quite verbose. More seriously, there is no *automatically verifiable* guarantee that programs thusly constructed are syntactically correct. For example, "f (,y)" can have the static type `string`, but this clearly does *not* imply that this string represents a syntactically correct program.

With the datatype encoding, we can address the syntactic correctness problem. Without any loss of generality, let us consider SML [49, 72, 67] as a typical general-purpose language. A datatype to represent object-programs can be defined in SML as follows:

```
datatype exp = Variable of string
             | Apply of (exp * exp)
             | Tuple of exp list
             | Function of string * exp.
```

This datatype declaration implements the set of representations of object-programs defined by the following BNF rule:

$$exp := x \mid exp\ exp \mid (exp, ..., exp) \mid \lambda x.exp$$

where x is drawn from a set of identifiers. In the datatype the set of identifiers is implemented by the set of SML strings. Thus, the datatype encoding is essentially the same as what is called "abstract syntax" or "parse trees". The encoding of the fragment f (x,y) in this SML datatype is:

```
Apply (Variable "f",Tuple [Variable "x" ,Variable "y"]).
```

Using a datatype encoding has an immediate benefit: *correct typing* for the meta-program ensures *correct syntax* for all object-programs. Because SML supports pattern matching over datatypes, deconstructing programs becomes easier than with the string representation. However, constructing programs is now more verbose.

In contrast, MetaML provides a construct called Brackets that allows us to specify this code fragment as ⟨f (x,y)⟩ (read *"Bracket f of x comma y"*). This encoding combines the simplicity and conciseness of the string encoding and the strength of the datatype encoding, in that such a Bracketed expression is accepted by the parser of MetaML only as long as what is contained in the Brackets has correct syntax. In other words, with MetaML, *correct syntax* for the meta-program ensures *correct syntax* for all object-programs.

Type Correctness: Even when syntax errors are avoided by using a datatype representation, there is no protection against constructing ill-formed programs that contain type errors, as in the case with the fragment (a,b) (c,d). In particular, with the datatype encoding, all object-programs are represented by one type, exp, which does not provide us with any information about the *type of the object-program represented by an* exp. The string encoding suffers the same problem. MetaML takes advantage of a simple type theoretic device, namely, the *parametric type constructor*. A common example of a parametric type constructor in SML is list, which allows us to have lists with elements of different types, such as [1,2,3,4] and ['a','b','c'], which have type int list and char list, respectively. In MetaML, we use a parametric type constructor ⟨_⟩ for code[2]. Thus, ⟨1⟩ has type

[2] Note that unlike the type constructor for lists, the MetaML code type constructor *cannot* be declared as a datatype in SML. We expand on this point in Section 7.1.3.

⟨int⟩ (read *"code of int"*), ⟨'b'⟩ has type ⟨char⟩, and ⟨fn x ⇒ x⟩ has type ⟨'a → 'a⟩. As we will show in this dissertation, with MetaML, *correct typing* for the meta-program ensures *correct typing* for all object-programs.

Efficient Combination and Operational Control: When constructing the representation of an object-program, it is possible to identify instances where part of the program being constructed can be performed immediately. For example, consider the meta-program:

⟨fn x ⇒ (fn y ⇒ (y,y)) x⟩.

This meta-program evaluates to itself, as does a quoted string. But note that it contains an object-level application that does not depend on any unknown information. We can modify this meta-program slightly so that the application is done while the object-program is being *constructed*, rather than while the object-program is being *executed*. This modification can be accomplished using the another MetaML construct called Escape that allows us to incorporate a code fragment in the context of a bigger code fragment. The improved meta-program is:

⟨fn x ⇒ ~((fn y ⇒ ⟨(~y,~y)⟩) ⟨x⟩)⟩.

When this improved meta-program is evaluated, some useful work will be done. The evaluation of this term proceeds as follows:

1. ⟨fn x ⇒ ~⟨(~⟨x⟩,~⟨x⟩)⟩⟩ ... The application is performed.
2. ⟨fn x ⇒ ~⟨(x,x)⟩⟩ ... The Escaped ⟨x⟩s are spliced into context.
3. ⟨fn x ⇒ (x,x)⟩ ... The Escaped ⟨(x,x)⟩ is spliced into context.

In the presence of recursion in the meta-language, Escapes allow us to perform more substantial computations while constructing the final result, thus yielding more efficient object-programs.

Escaping can be done with both string and datatype encodings easily as long as the object-program is not itself a meta-program, in which case Escaping gets more involved.

Semantic Correctness: With both string and datatype encodings, ensuring that there are no name clashes or inadvertent variable captures is the responsibility of the meta-programmer. For example, consider writing a simple program transformation T that takes an object-level arithmetic expression and returns another object-level function that adds the arithmetic expression to its arguments. For example, for an object-program 1+5 we get an object-program fn x ⇒ x + (1+5). Similarly, for y+z we get fn x ⇒ x + (y+z). It may seem that we can implement this program transformation as:

fun T e = Function ("x", Apply (Variable "+", Tuple [Variable "x", e])).

But this implementation is probably flawed. In particular, for y+x we get fn x ⇒ x + (y+x). This result is not what we would have expected if we had assumed that x is just a "dummy variable" that will never appear in the argument to T. In this case, we say that x was *inadvertently captured.* As we will see in this dissertation, inadvertent capture becomes an especially subtle problem in the presence of recursion.

The *intended* function T can be defined in MetaML as:

fun T e = ⟨fn x ⇒ x+˜e⟩.

Not only is this definition more concise than the one above, it has simpler semantics: x is *never* captured by the result of "splicing-in" e, because the run-time system ensures that all occurrences of x have the expected static-scoping semantics. Inadvertent capture is avoided in MetaML because the language is completely statically scoped, even with respect to object-level variables. This way, naming issues that arise with code generation are automatically managed by MetaML's run-time system.

Staging-Correctness: With both string and datatype encodings, care must be taken to ensure that no meta-program tries to use a variable belonging to one of its object-programs. For example, if we generate a program that uses a local variable x, we would like to ensure that the generator itself does not attempt to "use" the variable x, which will not become bound to anything until the run-time of the generated program. To begin addressing such problems, we must define a notion of level. The level of a term is the number of surrounding Brackets less the number of surrounding Escapes. For example, the term ⟨fn x ⇒ ˜x⟩ is not correct from the staging point of view, because the variable x is bound at level 1, yet we attempt to use it at level 0. Intuitively, this means that we are trying to use x before it is available.

Staging-correctness is a subtle problem in the traditional setting where the object-language is not itself a meta-language. In particular, there is an accidental reason that prevents the staging-correctness problem from manifesting itself in a two-level language where there is no construct for executing code. Let us consider both the string and datatype encodings and study the encoding of the MetaML term ⟨fn x ⇒ ˜x⟩. The encodings yield the following two *untypable* expressions:

1. "fn x ⇒ "^x. This fragment is not well-typed (generally speaking) because x is not a variable in the meta-language, and would be considered to be an "unknown identifier".
2. Function("x", x). This fragment is not well-typed for precisely the same reason.

Thus, the real problem in this term, namely, the incorrect staging, is hidden by the coincidental fact that both encodings are untypable.

With both encodings, we can still show that there are well-typed terms that are *not* correctly staged. Staging-correctness does manifest itself in both encodings when we begin describing multi-level terms such as $\langle\langle$fn x $\Rightarrow$ ~x$\rangle\rangle$. Both encodings of this term are well-typed:

1. "\"fn x ⇒ \"\^x" is a perfectly valid string.
2. Bracket(Function("x", Escape (Var "x"))) is also a perfectly valid exp if we extend the exp datatype with Brackets and Escapes, that is:

```
    datatype exp = Variable of string
                 | Apply of (exp * exp)
                 | Tuple of exp list
                 | Function of string * exp
                 | Bracket of exp
                 | Escape of exp.
```

Now, the staging-correctness problem is not evident at the time the object-program is constructed, but *becomes* evident when the object-program is executed. When executing the program, we get stuck trying to construct an encoding of the untypable term $\langle$fn x $\Rightarrow$ ~x$\rangle$.

The staging-correctness problem is especially subtle when we allow the execution of code, because executing code involves changing the level of terms at run-time. For example, executing $\langle 5\rangle$ we get 5, and the level of 5 has dropped by one. Sheard had postulated that ensuring staging-correctness should be one of the responsibilities of MetaML's type system. This dissertation presents two type systems for MetaML where well-typed MetaML programs are also correctly staged.

Reflection: Reflection is sometimes defined by (the presence of a construct in the language that allows) the execution of object-programs [27, 100, 17, 89], and sometimes by the ability of a language to represent (all of its meta-)programs as object-programs [85, 86]. Both definitions are properties that can be formalized, and both can be interpreted as positive qualities. It is therefore unfortunate the one name is used for two different properties, each of which is important in its own right.

MetaML enjoys instances of both reflective properties[3]. Qualitatively, the first kind of reflection suggests that the meta-language is at least as expressive as the object-language. This kind of reflection is realized in MetaML by incorporating a Run construct to execute object-programs. The

[3] Reification, when defined as the mapping of a value into a representation of that value, is not available in MetaML.

MetaML program run ⟨1+2⟩ returns 3, and ⟨run ⟨⟨1+5⟩⟩⟩ is a valid *multi-level* program. Qualitatively, the second kind of reflection suggests that the object-language is at least as expressive as the meta-language. This kind of reflection is realized in MetaML by allowing any object-program to be itself a meta-program.

To summarize this section, MetaML was designed to solve a host of fundamental problems encountered when writing program generators, thus freeing generator developers from having to continually re-invent the solutions to these problems.

1.2 Meta-Programming for Optimization

Meta-programming is often used to overcome limitations of an existing programming language. Such limitations can either be performance or expressivity problems. The focus of this dissertation is on a semantic basis for improving performance. We make a broad distinction between two ways of improving performance using meta-programming: Translation, and staging. While our work does touch on translation, our focus is on the staging aspect of meta-programming. In this section, we explain the difference between translation and staging.

1.2.1 Meta-Programming as Translation, or Re-mapping Abstract Machines

Abstract machines, whether realized by software or by hardware, vary in speed and resource usage. It is therefore possible to reduce the cost of executing a program by re-mapping it from one abstract machine to another. As hardware machines can be both faster and more space efficient than software ones, such re-mappings commonly involve producing machine or byte code. We will call this technique *translation* to distinguish it from staging (which discussed in the next subsection). Translation is an integral part of the practical compilation of a programming languages, and is typically performed by the *back-end* of a compiler.

Translation involves both inspecting and constructing code. MetaML implementations support some experimental constructs for inspecting code, but they are not the focus of this dissertation. However, we do study various forms of a Run construct which allow the meta-programmer to exploit the full power of the underlying machine. The distinction between Run and generalized code inspection is subtle, but has profound implications on the semantics (see Section 6.3) and the type system (see Section 7.1.3) of a multi-stage programming language.

1.2.2 Meta-Programming as Staging

The goal of staging is to improve a program based on *a priori* information about how it will be used. As the name suggests, *staging* is a program transformation that involves reorganizing the program's execution into stages [42].

The concept of a stage arises naturally in a wide variety of situations. Compilation-based program execution involves two distinct stages: compile-time, and run-time. Generated program execution involves three: generation-time, compile-time, and run-time. For example, consider the Yacc parser generator: first, it reads a grammar and generates C code; second, the generated program is compiled; third, the user runs this compiled program. Both compilation and high-level program generation can be used to reduce the cost of a program's execution. As such, staging provides us with a tool to improve the performance of high-level programs.

Cost Models for Staged Computation Cost models are not an absolute, and are generally dictated by the surrounding environment in which an algorithm, program, or system is to be used or deployed. Staging allows us to take advantage of features of both the inputs to a program and the cost model to improve performance. In particular, while staging may be an optimization under one model, it may not be under another. There are three important classes of cost models under which staging can be beneficial:

– Overall cost is the total cost of all stages, *for most inputs*. This model applies, for example, in implementations of programming languages. The cost of a simple compilation followed by execution is *usually* lower than the cost of interpretation. For example, the program being executed usually contains loops, which typically incur large overhead in an interpreted implementation.
– Overall cost is a *weighted average* of the cost of all stages. The weights reflect the relative frequency at which the result of a stage can be reused. This model is useful in many applications of symbolic computation. Often, solving a problem symbolically, and then graphing the solution at a thousand points can be cheaper than numerically solving the problem a thousand times. This cost model can make a symbolic approach worthwhile even when it is 100 times more expensive than a direct numerical one. Symbolic computation is a form of staged computation where free variables are values that will only become available at a later stage.
– Overall cost is the cost of the *last stage*. This cost model is often just a practical approximation of the previous model, where the relative frequency of executing the last stage is much larger than that for any of the previous stages. To illustrate, consider an embedded system where the *sin* function may be implemented as a large look-up table. The cost of constructing the

table is not relevant. Only the cost of computing the function at run-time is relevant. This observation also applies to optimizing compilers, which may spend an unusual amount of time to generate a high-performance computational library. The cost of optimization is often not relevant to the users of such libraries[4].

The last model seems to be the most commonly referenced one in the literature, and is often described as "there is ample time between the arrival of different inputs", "there is a significant difference between the frequency at which the various inputs to a program change", and "the performance of the program matters only after the arrival of its last input".

Finally, we wish to emphasize that non-trivial performance gains can be achieved using only staging, and without any need for translation. MetaML provides the software developer with a programming language where the staging aspect of a computation can be expressed in a concise manner, both at the level of syntax and types. This way, the programmer does not need to learn a low-level language, yet continues to enjoy many of the performance improvements previously associated only with program generation. Furthermore, when translation is employed in an *implementation* of MetaML, translation too can be exploited by the meta-programmer through using the Run construct.

1.3 Partial Evaluation and Multi-Level Languages

Today, the most sophisticated automated staging systems are partial evaluation systems. Partial evaluation optimizes a program using partial information about some of that program's inputs. Jones introduced off-line partial evaluation to show that partial evaluation can be performed efficiently [41]. An off-line partial evaluator is itself a staged system. First, a Binding-Time Analysis (BTA) annotates the input program to indicate whether each subexpression can be computed at partial-evaluation-time (static), or at run-time (dynamic). Intuitively, only the subexpressions that depend on static inputs can be computed at partial-evaluation time. Second, the annotated program is specialized on the static inputs to produce the new specialized program. MetaML provides a *common language* for illustrating the workings of off-line partial evaluation. For example, we can construct a representation of a program in MetaML. Let us consider a simple MetaML session. We type in:

```
-| val p = ⟨fn x ⇒ fn y ⇒ (x+1)+y⟩;
```

[4] Not to mention the century-long "stages" that were needed to evolve the theory behind many of these libraries.

and the MetaML implementation prints:

```
val p = ⟨fn x ⇒ fn y ⇒ (x+1)+y⟩
      : ⟨int → int → int⟩.
```

If the program p is fed to a partial evaluator, it must first go through BTA. At an implementation level, BTA can be viewed as a source-to-source transformation. Typically, BTA is given a specification of which inputs are static and which inputs are dynamic. For simplicity, let us assume that we are only interested in programs that take two curried arguments, and the first one is static, and the second one is dynamic. Although our MetaML implementation does not provide such a function today, one can, in principle, add a *constant* BTA to MetaML with the following type[5]:

```
-| BTA;
val BTA = -fn-
        : ⟨'a →  'b →  'c⟩ → ⟨ 'a → ⟨ 'b →  'c⟩⟩.
```

Then, to perform BTA, we apply this constant to the source program:

```
-| val ap = BTA p;
val ap = ⟨fn x ⇒ ⟨fn y ⇒ ~(lift (x+1))+y⟩⟩
       : ⟨int → ⟨int → int⟩⟩
```

yielding the "annotated program". The lift function is a secondary annotation that takes a ground value and returns a code fragment containing that value. For now, the reader can view lift simply as being fn x ⇒ ⟨x⟩.

The next step is specialization. It involves running the program on the input term:

```
-| val p5 = (run ap) 5;
val p5 = ⟨fn y ⇒ 6+y⟩
       : ⟨int → int⟩
```

yielding, in turn, the specialized program.

Partial evaluation in general, and off-line partial evaluation in particular, have been the subject of a substantial body of research. Much of our understanding of the applications and the limitations of staged computation has grown out of that literature. The word "multi-stage" itself seems to have been first introduce by Jones *et al.* [40]. In the illustration above, we have taken the view that the output of BTA is simply a two-stage annotated program. This view seems to have been first

[5] BTA cannot be expressed using only the staging constructs that we study in this dissertation. In particular, an analysis such as BTA requires intensional analysis, which is only addressed tangentially in this dissertation.

suggested in the works of Nielson and Nielson [62] and by Gomard and Jones [32], when two-level languages were introduced. Recently, two-level languages have been proposed as an intermediate representation for off-line partial evaluation systems. Glück generalized two-stage off-line partial evaluation to multi-level off-line partial evaluation and introduced multi-level languages. These ideas where the starting point for this dissertation. For example, MetaML is essentially a multi-level language with the addition of Run. This view of MetaML is precisely the sense in which we use the term "multi-stage":

$$\text{Multi-Stage Language} = \text{Multi-Level Language} + \text{Run}.$$

1.4 Multi-Stage Programming with Explicit Annotations

From a software engineering point of view, the novelty of MetaML is that it admits an intuitively appealing method of developing meta-programs. A multi-stage program can be developed in MetaML as follows:

1. A single-stage program is developed, implemented, and tested.
2. The type of the single-stage program is annotated using the code type constructor to reflect the order in which inputs arrive.
3. The organization and data-structures of the program are studied to ensure that they can be used in a staged manner. This analysis may indicate a need for "factoring" some parts of the program and its data structures. This step can be subtle, and can be a critical step towards effective multi-stage programming. Fortunately, it has been thoroughly investigated in the context of partial evaluation where it is known as *binding-time engineering* [40].
4. Staging annotations are introduced to specify explicitly the evaluation order of the various computations of the program. The staged program may then be tested to ensure that it achieves the desired performance.

The method described above, called *multi-stage programming with explicit annotations*, can be summarized by the slogan:

$$\text{A Staged Program} = \text{A Conventional Program} + \text{Staging Annotations}.$$

The conciseness of meta-programs written in MetaML allows us to view meta-programs as simple variations on conventional (that is, "non-meta-") programs. These variations are minor, orthogonal, and localized, compared to writing meta-programs in a general-purpose programming language, a task which would be riddled by the problems described in Section 1.1. Furthermore, staging is accurately reflected in the manifest interfaces (the types) of the MetaML programs.

Because program generation is often used for the purpose of staging, we can widen the scope of applicability of the slogan above by restating it as:

Many a Program Generator = A Conventional Program + Staging Annotations.

1.5 Thesis and Contributions

Our thesis is that MetaML is a well-designed language that is useful in developing meta-programs and program generators. We break down the thesis into three main hypotheses:

H1. MetaML is a useful medium for meta-programming.

H2. MetaML can be placed on a standard, formal foundation whereby staging annotations are viewed as language constructs amenable to the formal techniques of programming languages.

H3. MetaML in particular, and multi-level languages in general, can be improved both in their design and implementation by what we have learned while building MetaML's formal foundations.

This dissertation presents the following contributions to support the hypotheses:

Applications of MetaML (H1) We have used MetaML to develop program generators. An important benefit of the approach seems to be its simplicity and transparency. Furthermore, MetaML has been a powerful pedagogical tool for explaining the workings of partial evaluation systems. At the same time, we have also identified some limitations of the approach, and identified ways in which they could be addressed in the future. In Chapter 3 we present a detailed example prototypical of our experience.

A Formal Basis for Multi-Stage Programming (H2) We present a formal semantics and a type system for MetaML, and a common framework that unifies previous proposals for formal foundations for high-level program generation and run-time code generation. We have formalized the semantics in two different styles (big-step style in Chapter 5, and reduction style in Chapter 6) and have developed a type system (Chapter 5) for MetaML that we proved sound.

Improving the Design and Implementation of MetaML (H3) In the process of formalizing the semantics, we have uncovered a variety of subtleties and some flaws in the early implementations of MetaML and proposed remedies for them. Examples of such findings are presented in

Chapter 4. Furthermore, we have identified extensions to the language and showed how they can be incorporated in a type-safe manner. In particular, we present a proposal for extending MetaML with a type constructor for closedness in Chapter 5.

1.6 Organization and Reading Plans

This dissertation is organized into three parts. Part I introduces MetaML, and provides examples of multi-stage programming with explicit annotations. Part II presents the formal semantics and type system that we propose for MetaML. Part III covers related works, a discussion of the results, and concludes the dissertation.

The following is a detailed overview of the three parts.

1.6.1 Part I

Chapter 2 provides the basic background needed for developing multi-stage programs in MetaML, including:

- The intuitive semantics of MetaML's staging annotations, illustrated by some small examples.
- The design principles that have shaped MetaML. We stress the novelty and significance of two principles, called cross-stage persistence and cross-stage safety.
- Simple two-stage examples of multi-stage programming with explicit annotations. The examples illustrates the positive role of types in the development method.
- A three-stage example of multi-stage programming with explicit annotations.

Chapter 3 presents an extended example of multi-stage programming with explicit annotations. This example shows that while developing staged programs can be challenging, borrowing well-known techniques from the area of partial evaluation can yield worthwhile results. We consider a simple term-rewriting system and make a first attempt at staging it. Searching for a staged type for this system suggests that this direct attempt might not yield an optimal result. Indeed, we find that this is the case. We then make a second attempt using a technique that has been exploited by users of off-line partial evaluation systems, and show that this approach yields satisfactory results.

1.6.2 Part II

Chapter 4 summarizes the problems that must be addressed when we wish to implement a multi-stage programming language such as MetaML, and gives examples of how our study of the formal semantics improved our understanding of MetaML implementations. These problems and examples provide the motivation for the theoretical pursuit presented in the rest of Part II. We begin by reviewing the implementation problems that were known when MetaML was first developed, including the basic scoping and typing problems. We then describe the semantics of a simple implementation of subset of MetaML that we call λ-M[6]. This simple implementation is prototypical of how implementations of multi-stage languages are developed in practice. The implementation allows us to point out a new set of problems, including new scoping subtleties, more typing issues, and the need for a better understanding of what MetaML programs can be considered equivalent. This chapter is illustrative of the state of the art in the (potentially verifiable) implementation of multi-stage programming languages.

Chapter 5 presents a basic type system for the λ-M subset of MetaML, together with a proof of the soundness of this type system with respect to a big-step semantics for λ-M. We then argue for extending the type system and present a big-step semantics for the a proposed extension to MetaML that we call λ^{BN}[7]. The chapter presents:

– A big-step semantics. A big-step semantics provides us with a functional semantics. It is a partial function, and therefore resembles an interpreter for our language. Because "evaluation under lambda" is explicit in this semantics, it is a good, realistic model of multi-stage computation. Using only capture-free substitution in such a semantics is the *essence* of static scoping. Furthermore, this semantics illustrates how MetaML violates one of the basic assumptions of many works on programming language semantics, namely, that we are dealing only with closed terms.
– A Type-safety result. We show that a basic type system for λ-M guarantees run-time safety, based on an augmented big-step semantics.
– Closedness types. After explaining an expressivity problem in the basic type system presented for λ-M, we show how this problem can be remedied by introducing a special type for closed values. This extension paves the way for a new and more expressive form of the Run construct.

This chapter represents the state of the art in (untyped) semantics and type systems for multi-stage programming languages.

[6] The letter M stands for MetaML.
[7] The letters B and N stand for Box and Next, respectively, after the names of the logical modalities used in the work of Davies and Pfenning [23, 22]. It should be noted, however, that λ^{BN} no longer has a type for closed code, but rather, a type for closedness.

Chapter 6 presents a reduction semantics for a subset of MetaML that we call λ-U[8]. The chapter presents:

- A reduction semantics. The reduction semantics is a set of directed rewrite rules. Intuitively, the rewrite rules capture the "notions of reduction" in MetaML.
- Subject reduction. We show that each reduction preserves typing under the type system for λ-M.
- Confluence. This result is an indicator of the well-behavedness of our notions of reduction. It states that the result of any two (possibly different) sequences of reduction can always be reduced to a common term.
- Soundness. This result has two parts. First, all what can be achieved by the λ-M big-step semantics, can be achieved by the reductions. Second, applying the reductions to any subterm of a program does not change the termination behavior of the λ-M big-step semantics. In essence, this result establishes that λ-U and λ-M are "equivalent" formulations of the same language.

This chapter presents new results on the untyped semantics of multi-stage programming languages.

1.6.3 Part III

Chapter 7 summarizes related work and positions our contributions in the context of programming languages, partial evaluation, and program generation research. The chapter presents:

- A summary of key developments in multi-level specialization and languages.
- A brief review of the history of quasi-quotations, revisiting Quine's original work and LISP's back-quote and comma mechanism.

In Chapter 8 we appraise our findings, outline directions for future works, and conclude the dissertation.

Appendix A presents some remarks on an intermediate language that we do not develop fully in this dissertation, but we intend to study in more detail in future work.

[8] We call this reduction semantics λ-U to avoid asserting *a priori* that it is equivalent to the big-step semantics (λ-M). The letter U is the last in the sequence R, S, T, U. Our first attempt at a calculus was called λ-R. We have included λ-T in Appendix A because it may have applications in the implementation of multi-stage languages, but it is not as suitable as λ-U for the purpose of equational reasoning.

1.6.4 Reading Plans

The reader interested primarily in the practice of writing program generators, and the relevance of MetaML and multi-stage programming to program generation may find Chapters 1 to 3 to be the most useful. The reader interested in understanding the difficulties in implementing multi-stage languages such as MetaML as extensions of existing programming languages may find Chapter 4 (skipping Section 4.6.2) to be the most useful, and Chapter 2 can serve as a complete introduction.

The reader interested primarily in the formal semantics of multi-level languages may find Chapter 5 and 6 to be the most useful, and Chapter 2 (and Section 4.6.2) can again serve as a complete introduction.

Chapter 7 is primarily for readers interested in becoming more acquainted with the related literature on multi-stage languages.

Chapter 8 is primarily for readers interested in the summary of findings presented in this dissertation and an overview of open problems.

Part I

The Practice of Multi-Stage Programming

Chapter 2

MetaML and Staging

All the world's a stage,
And all the men and women merely players:
They have their exits and their entrances;
And one man in his time plays many parts...

Jacques, Act 2, Scene 7,
As You Like it, Shakespeare

This chapter introduces staging and MetaML. We present MetaML's staging constructs and explain how staging, even at a fairly abstract level, can be useful for improving performance. Then, we explain the key design choices in MetaML and illustrate MetaML's utility in staging three well-known functions.

2.1 MetaML the Conceptual Framework

A good formalism for staging should allow us to explain the concept of staging clearly. In essence, staging is altering a program's order of evaluation in order to change the cost of its execution. MetaML is a good formalism for staging because it provides four staging annotations that can be used to explain such alterations:

1. Brackets $\langle \_ \rangle$ for delaying a computation,
2. Escape ˜_ for combining delayed computations,
3. Run run _ for executing a delayed computation, and
4. Lift lift _ for constructing a delayed computation from a (ground) value.

With just this abstract description of MetaML's annotations, we can explain how one can reduce the cost of a program using staging.

2.1.1 Staging and Reducing Cost

Although, MetaML is a call-by-value (CBV) language, the cost of executing a program under both CBV and call-by-name (CBN) semantics can be reduced by staging. Consider the following computation:

    (fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+(7*2)).

Ignore the fact that part or all of this computation can be performed by an optimizing compiler before a program is executed. Such optimizing compilers constitute an additional level of complexity that we are not concerned with at the moment. Consider a cost model where we count only the number of arithmetic operations performed. We make this choice only for simplicity. This cost model is realistic in situations where the arithmetic operations in the program above stand in place of more costly operations.

Evaluating the program above under CBV semantics proceeds as follows (the cost of each step is indicated by "... *n* arith op(s)"):

1. ((fn x ⇒ x+(7*2)) 9)+((fn x ⇒ x+(7*2)) 13) ... 0 arith ops
2. (9+(7*2))+(13+(7*2)) ... 0 arith ops
3. 50 ... 5 arith ops

The total cost is 5 ops. Evaluating the same computation under CBN semantics proceeds in essentially the same way. Again, the total cost is 5.

2.1.2 Staging Reduces Cost in CBV

In the CBV setting, we can stage our computation as follows:

    (fn f ⇒ (f 9)+(f 13)) (run ⟨fn x ⇒ x+˜(lift (7*2))⟩).

Evaluating this staged computation under CBV semantics proceeds as follows:

1. (fn f ⇒ (f x)+(f y)) (run ⟨fn x ⇒ x+˜(lift 14)⟩) ... 1 arith op
2. (fn f ⇒ (f x)+(f y)) (run ⟨fn x ⇒ x+˜⟨14⟩⟩) ... 0 arith ops
3. (fn f ⇒ (f x)+(f y)) (run ⟨fn x ⇒ x+14⟩) ... 0 arith ops
4. (fn f ⇒ (f x)+(f y)) (fn x ⇒ x+14) ... 0 arith ops
5. (((fn x ⇒ x+14) 9)+((fn x ⇒ x+14) 13)) ... 0 arith ops
6. ((9+14)+(13+14)) ... 0 arith ops
7. 50 ... 3 arith ops

The staged version costs 1 op less.

2.1.3 Staging Reduces Cost in CBN

In the CBN setting, we can stage our computation as follows:

run ⟨(fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+~(lift (7*2)))⟩.

Evaluating the staged computation above under CBN semantics proceeds as follows:

1. run ⟨(fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+~(lift 14))⟩ ... 1 arith ops
2. run ⟨(fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+~⟨14⟩)⟩ ... 0 arith ops
3. run ⟨(fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+14)⟩ ... 0 arith ops
4. (fn f ⇒ (f 9)+(f 13)) (fn x ⇒ x+14) ... 0 arith ops
5. (((fn x ⇒ x+14) 9)+((fn x ⇒ x+14) 13)) ... 0 arith ops
6. (9+14)+(13+14) ... 0 arith ops
7. 50 ... 3 arith ops

The cost is again 1 op less than without staging. It is in this sense that staging gives the programmer control over the evaluation order in a manner that can be exploited to enhance performance.

Having explained the concept of staging, we are now ready to introduce MetaML the programming language.

2.2 MetaML the Programming Language

MetaML is a functional programming language with special constructs for staging programs. In addition to most features of SML, MetaML provides the following special support for multi-stage programming:

- Four staging annotations, which we believe are a good basis for general-purpose multi-stage programming.
- Static type-checking and a polymorphic type-inference system. In MetaML, a multi-stage program is type-checked once and for all before it begins executing, ensuring the safety of all computations in all stages. This feature of MetaML is specially useful in systems where the later stages are executed when the original programmer is no longer around.
- Static scoping for both meta-level and object-level variables.

MetaML implements delayed computations as abstract syntax trees representing MetaML programs. The four MetaML staging constructs are implemented as follows:

1. Brackets ⟨_⟩ construct a code fragment,
2. Escape ˜_ combines code fragments,
3. Run run _ executes a code fragment, and
4. Lift lift _ constructs a code fragment from a ground value, such that the code fragment represents the ground value.

In this section, we explain the intuitive semantics of each of these four constructs.

2.2.1 Brackets

Brackets can be inserted around any expression to delay its execution. MetaML implements delayed expressions by building a representation of the source code. While using the source code representation is not the only way of implementing delayed expressions, it is the simplest. The following short interactive session illustrates the behavior of Brackets in MetaML:

```
-| val result0 = 1+5;
val result0 = 6 : int

-| val code0 = ⟨1+5⟩;
val code0 = ⟨1%+5⟩ : ⟨int⟩.
```

The percentage signs in %+ simply indicates that + is not a free variable. The reader can treat such percentage signs as white space until their significance is explained in Section 2.3.1.

In addition to delaying the computation, Brackets are also reflected in the type. The type in the last declaration is ⟨int⟩, read *"Code of Int"*. The code type constructor is the primary devise that the type system uses for distinguishing delayed values from other values and prevents the user from accidentally attempting unsafe operations such as 1+⟨5⟩.

2.2.2 Escape

Escape allows the combination of smaller delayed values to construct larger ones. This combination is achieved by "splicing-in" the argument of the Escape in the context of the surrounding Brackets:

```
-| val code1 = ⟨(˜code0,˜code0)⟩;
val code1 = ⟨(1%+5,1%+5)⟩ : ⟨int * int⟩.
```

This declaration binds code1 to a new delayed computation, representing a tuple of two arithmetic expressions. Escape combines delayed computations efficiently in the sense that the combination of the subcomponents of the new computation is performed while the new computation is being *constructed*, rather than while it is being *executed*. This subtle distinction is crucial for staging. And as we will see in examples to come, this particular behavior of Escape can make a big difference in the run-time performance of the delayed computation.

2.2.3 Run

Run allows the execution of a code fragment. Having Run in the language is important if we want to use code constructed using the other MetaML constructs, *without going outside the language.* This added expressivity makes Run important from both practical and theoretical points of view. From the practical point of view, having Run in a statically typed language allows us to develop multi-stage systems as part of any program and still be sure that the *whole system* will be free of run-time errors. From the theoretical point of view, it allows us to formalize the problem of safety in multi-stage systems and to address it with formal rigor.

The use of Run in MetaML can be illustrated with the following simple example:

```
-| val code0 = ⟨1+5⟩;
val code0 = ⟨1%+5⟩ : ⟨int⟩

-| val result1 = run code0;
val result1 = 6 : int.
```

2.2.4 Lift

Lift allows us to inject values of ground type into a value of type code. In MetaML, base types are types containing no arrows (function types). Examples of ground types are int, string, bool, and int list:

```
-| val code2 = lift 6 ;
val code2 = ⟨6⟩ : ⟨int⟩.
```

While both Brackets and Lift construct code, Lift does *not* delay its argument. Lift first evaluates its argument, then constructs a representation for this value:

```
-| val code3 = lift 1+5;
val code3 = ⟨6⟩ : ⟨int⟩.
```

Because Lift is implemented by producing a source-level representation of the its operand, Lift cannot be defined for functions: Given an arbitrary function, there is no general way of computing a source-level representation for that function[1]. However, functions can still be delayed using Brackets:

```
-| val result2 = fn x ⇒ x+1;
val result2 = -fn- : int → int

-| val code4 = ⟨result2 5⟩;
val code4 = ⟨%result2 5⟩ : ⟨int⟩

-| val result3 = run code4;
val result3 = 6 : int.
```

In the first declaration, result2 is bound to the function that takes an int and returns an int. The second declaration constructs a piece of code that uses result2 in a delayed context.

2.2.5 The Notion of Level

Determining when an Escaped expression should be performed requires a notion of level. The *level* of a term is the number of surrounding Brackets minus the number of surrounding Escapes. Escapes in MetaML are only evaluated when they are at level one.

MetaML is a multi-level language. This feature is important because it allows us to have multiple distinct stages of execution. For example, we can write expressions such as:

```
-| ⟨⟨5+5⟩⟩;
val it = ⟨⟨5%+5⟩⟩ : ⟨⟨int⟩⟩
```

and the type reflects the number of times the enclosed integer expression is delayed.

Escapes can be also used in object-programs. In such multi-level expressions, we can also use Escapes:

```
-| val code5 = ⟨⟨(5+5,~(code2))⟩⟩;
val code5 = ⟨⟨(5%+5,~(code2))⟩⟩ : ⟨⟨int*int⟩⟩.
```

The Escape is not "performed" when this expression was evaluated, because there are two Brackets surrounding this Escape. We can Run the doubly delayed value code5 as follows:

[1] Recent work on *type-directed partial evaluation* [18] suggests that there are practical ways of deriving the source-level representations for functions at run-time when the executables available at run-time are sufficiently instrumented. Sheard has investigated type-directed partial evaluation in the context of MetaML [81]. The treatment of this subject is, however, beyond the scope of the present work.

```
-| val code6 = run code5;
val code6 = ⟨(5%+5,6)⟩ : ⟨int*int⟩.
```

Run eliminates one Bracket[2], thus lowering the level of the Escape from 2 to 1, and the Escape is performed.

2.3 The Pragmatics of Variables and Levels

Is it reasonable to write a program where a variable is bound at one level and is used at another? On one hand, it seems completely justifiable to write terms such as ⟨sqrt 16.0⟩. It is typical in the formal treatment of programming languages to consider sqrt to be a free variable (bound at level 0). In this case sqrt is bound at level 0 and is used at level 1. On the other hand, we do not wish to allow terms such as ⟨fn x ⇒ ~x⟩ which dictates that x be evaluated (at level 0) *before* it is bound (at level 1). The first term is an example of why *cross-stage persistence* is desirable, and the second term is an example of why violating *cross-stage safety* is undesirable.

2.3.1 Cross-Stage Persistence

We say that a variable is cross-stage persistent when it is bound at one level and is used at a higher level. Permitting this usage of variables means allowing the programmer to take full advantage of all primitives and bindings that are available in the current stage by reusing them in future stages. A percentage sign is printed by the display mechanism to indicate that %a is *not* a variable, but rather, a new *constant.* For example the program

```
let val a = 1+4 in ⟨72+a⟩ end
```

computes the code fragment ⟨72 %+ %a⟩. The percentage sign %_ indicates that the cross-stage persistent variables a and + are bound in the code's local environment. The variable a has been bound during the first stage to the constant 5. The name "a" is printed only to provide a hint to the user about where this new constant originated from.

When %a is evaluated in a later stage, it will return 5 independently of the binding for the variable a in the new context. Arbitrary values (including functions) can be injected into a piece of code using this hygienic binding mechanism.

[2] Despite that, there is a sense in which evaluation and reduction both "preserve level". Such properties will be established as part of the formal treatment of MetaML in Part II of this dissertation.

2.3.2 Cross-Stage Safety

We say the that a variable violates *cross-stage safety* when it is bound at one level and is used at a lower level. This violation occurs in the expression:

```
fn a ⇒ ⟨fn b ⇒ ˜(a+b)⟩.
```

The annotations in this expression dictate computing a+b in the first stage, when the value of b will be available only in the second stage!

Supporting cross-stage persistence means that a type system for MetaML must ensure that "well-typed programs won't go *Wrong*", where going wrong now includes the violation of the cross-stage safety condition, as well as the standard notions of going wrong [48] in statically-typed languages. In our experience, having a type system to screen out such programs is a significant aid in developing a multi-stage program.

We are now ready to see how MetaML can be used to stage some simple functions.

2.4 Staging the List-Membership Function

Using MetaML, the programmer can stage programs by inserting the proper annotations at the right places in the program. The programmer uses these annotations to modify the default strict evaluation order of the program.

Let us consider staging the function that takes a list and a value and searches the list for this value. We begin by writing the single-stage function[3]:

```
(* member1 : ''a → ''a list → bool *)
fun member1 v l =
    if (null l)
      then false
      else if v=(hd l)
           then true
           else member1 v (tl l).
```

We have observed that the possible annotations in a staged version of a program are significantly constrained by its type. This observation suggests that a good strategy for hand-staging a program

[3] In SML, whereas type variables written as 'a are unrestricted polymorphic variables, type variables written as ''a are also polymorphic, but are restricted to *equality types*, that us, types whose elements can be tested for equality. Function types are the prototypical example of a type whose elements *cannot* be tested for equality. Thus, equality type variables cannot be instantiated to functions types, for example.

is to first determine the target type of the desired annotated program. Thus, we will start by studying the type of the single-stage function. Suppose the list parameter is available in the first stage, and the element sought is available later. A natural target type for the staged function is then

$$\langle\,{}''a\rangle \to\ {}''a\ \mathsf{list} \to \langle \mathsf{bool}\rangle.$$

This type reflects the fact that both the first argument and the result of this function will be "late". In other words, only the second argument is available at the current time.

Having chosen a suitable target type, we begin annotating the single-stage program. We start with the whole expression and working inwards until all sub-expressions have been considered. At each step, we try to find the annotations that will "correct" the type of the expression so that the whole function has a type closer to the target type. The following function realizes the target type for the staged member function:

```
(* member2 : ⟨ "a⟩ →  "a list → ⟨bool⟩ *)
fun member2 v l =
    if (null l)
      then ⟨false⟩
      else ⟨if ˜v=˜(lift (hd l))
               then true
               else ˜(member2 v (tl l))⟩.
```

Not all annotations are explicitly dictated by the type. The annotated term ˜(lift (hd l)) has the same type as hd l, but it ensures that hd l is performed during the first stage. Otherwise, all selections of the head element of the list would be delayed until the generated code is Run in a later stage[4].

The Brackets around the branches of the outermost if-expression ensure that the return value of member2 will be a code type ⟨_⟩. The first branch ⟨false⟩ needs no further annotations and makes the return value precisely a ⟨bool⟩. Moving inwards in the second branch, the condition ˜v forces the type of the v parameter to have type ⟨ "a⟩, as planned.

Just like the first branch of the outer if-statement, the inner if-statement must return bool. So, the first branch returns true without any further annotations. But because the recursive call to member2 has type ⟨bool⟩, it must be Escaped. Inserting this Escape also implies that the recursion

[4] More significantly, recursive programs can be annotated in two fundamentally different ways and types do not provide any help in this regard. We expand on this point in Chapter 7.

will be performed in the first stage, which is exactly the desired behavior. Thus, the result of member2 is a recursively-constructed piece of code of type type bool.

Evaluating ⟨fn x ⇒ ~(member2 ⟨x⟩ [1,2,3])⟩ yields:

```
⟨fn d1 ⇒
if d1 %= 1 then true
           else if d1 %= 2 then true
                         else if d1 %= 3 then true
                                       else false⟩.
```

2.5 Staging the Power Function

Computing the powers of a real number is a classic example from the partial evaluation literature [40]. Consider the following definition of exponentiation:

$$\begin{aligned} x^0 &= 1 \\ x^{n+n+1} &= x \times x^{n+n} \\ x^{n+n+2} &= (x^{n+1})^2 \end{aligned}$$

This definition can be used to compute expressions such as 1.7^5. More interestingly, it can be used to compute or "generate" an efficient formula for expressions such as x^{72} where only the exponent is known. We have underlined the term being expanded:

$$\begin{aligned} \underline{x^{72}} &= (\underline{x^{36}})^2 = (((\underline{x^{18}})^2)^2)^2 = ((\underline{x^9})^2)^2 = (((x \times \underline{x^8})^2)^2)^2 \\ &= (((x \times (\underline{x^4})^2)^2)^2)^2 = (((x \times ((\underline{x^2})^2)^2)^2)^2)^2 \\ &= (((x \times (((x \times 1)^2)^2)^2)^2)^2)^2 \end{aligned}$$

This sequence of expansions is an example of symbolic computation, because we are computing with free variables. It is also an example of staged computation, because we are doing useful work in a stage distinct from the final stage of computation.

The computation above can be be programmed in MetaML. First, we write the single-stage "all-inputs are known" function:

```
fun exp (n,x) = (* : int × real → real *)
  if n = 0 then 1.0
          else if even n then sqr (exp (n div 2,x))
                        else x * exp (n - 1,x).
```

Then, we add staging annotations:

```
fun exp' (n,x) = (* : int × ⟨real⟩ → ⟨real⟩ *)
  if n = 0 then ⟨1.0⟩
           else if even n then sqr' (exp' (n div 2,x))
                        else ⟨~x-345 * ~(exp' (n - 1,x))⟩.
```

We also need to stage the function sqr:

```
fun sqr x = (* : real → real *)
  x * x;
```

to

```
fun sqr' x = (* : ⟨real⟩ → ⟨real⟩ *)
  ⟨let val y = ~x in y * y end⟩.
```

The pattern of multiplying a variable by itself after a let-binding let ... in y * y, will be evident in the generated code. Now, we can use exp' to generate the code for x^{72} as follows:

```
- val exp72' = ⟨fn x ⇒ ~(exp' (72,⟨x⟩))⟩;
val exp72' = ⟨fn x ⇒
              let val a =
                 let val b =
                    let val c = x %*
                       let val d =
                          let val e =
                             let val f = x %* 1.0
                             in f %* f end
                          in e %* e end
                       in d %* d end
                    in c %* c end
                 in b %* b end
              in a %* a end⟩ : ⟨real → real⟩.
```

MetaML performs an optimization on let-bindings whereby a term of the form:

$$\text{let val } x = \ (\text{let val } y = e \text{ in } f \text{ end}) \text{ in } g \text{ end}$$

is replaced by

$$\text{let val } y = e \text{ val } x = f \text{ in } g \text{ end}$$

Thus, the code returned by the system is in fact simpler:

```
val exp72' = ⟨fn x ⇒
                let val f = x %* 1.0
                    val e = f %* f
                    val d = e %* e
                    val c = x %* (d %* d)
                    val b = c %* c
                    val a = b %* b
                in a %* a end⟩ : ⟨real → real⟩.
```

Finally, we can re-integrate this specialized function into the run-time system using Run:

```
- val exp72 = run exp72'; (* : real → real *).
```

This new function can now be used anywhere where need to compute x^{72} efficiently. Having Run in the language is essential for allowing the programmer to use an automatically generated function anywhere a hand-written function can be used.

The recursion has been performed in the first stage. Eliminating the function calls involved in recursion can yield a performance improvement for short lists. For longer lists, the size of the generated expression could burden the memory management system.

2.6 Back and Forth: Two Useful Functions on Code Types

While staging programs, we found an interesting pair of functions to be useful:

```
fun back f = ⟨fn x ⇒ ~(f ⟨x⟩)⟩; (* : (⟨ 'a⟩ → ⟨ 'b⟩) → ⟨ 'a →  'b⟩ *)
fun forth f x = ⟨~f ~x⟩;           (* : ⟨ 'a →  'b⟩ → ⟨ 'a⟩ → ⟨ 'b⟩ *).
```

The function ⟨back⟩ takes a function f and constructs a code fragment representing a new function. The body of the new function is the result of unfolding the function f on the argument of the new function. The function forth takes a code fragment f and a code fragment x, and constructs a code fragment representing the application of the first to the second. These two "computational patterns" come up often in the course of multi-stage programmers. We have already used a similar construction to stage the member2 function of type ⟨ "a⟩ → "a list → ⟨bool⟩, within the term ⟨fn x ⇒ ~(member2 ⟨x⟩ [1,2,3])⟩ which has type ⟨ "a → bool⟩.

In our experience, annotating a function to have type ⟨ 'a⟩ → ⟨ 'b⟩ requires less annotations than annotating it to have type ⟨ 'a → 'b⟩ and is often easier to think about. Furthermore, perhaps because we are functional programmers, it seemed more convenient to think of code transformers (functions from code to code) rather than representations of functions (code of functions). Both reasons lead us to avoid writing programs of the latter type except when we need to see the generated code.

The same observations apply to programs with more than two stages. Consider the function:

```
(* back2 : (⟨ 'a⟩ → ⟨⟨ 'b⟩⟩ → ⟨⟨ 'c⟩⟩) → ⟨ 'a → ⟨ 'b →  'c⟩⟩ *)
fun back2 f = ⟨fn x ⇒ ⟨fn y ⇒ ~~(f ⟨x⟩ ⟨⟨y⟩⟩)⟩⟩.
```

This function allows us to write a program which takes a ⟨a⟩ and a ⟨⟨b⟩⟩ as arguments and which produces a ⟨⟨c⟩⟩, and turns it into a three-stage function. Our experience is that such functions have considerably fewer annotations, and are easier to reason about. We expand on this point in the next section.

2.7 Staging the Inner Product: A Three-Stage Example

The function for computing the inner product of two vectors has been studied in several other works [29, 45], and its execution can be usefully separated into three distinct stages:

1. In the first stage, knowing the size of the two vectors offers an opportunity to specialize the inner product function on that size, removing a looping overhead from the body of the function.
2. In the second stage, knowing the first vector offers an opportunity for specialization based on the values in the vector. If the inner product of that vector is to be taken many times with other vectors it can be specialized by removing the overhead of looking up the elements of the first vector each time. Knowing the size and the first vector is exactly the case when computing the multiplication of two matrices. For each row in the first matrix, the dot product of that row will be taken with each column of the second matrix.
3. In the third stage, knowing the second vector, the computation is brought to completion.

We will present three versions of the inner product function. One (iprod) with no staging annotations, the second (iprod2) with two levels of annotations, and the third (iprod3) with two levels of annotations but constructed with the back2 function. In MetaML we quote relational operators such as less-than < and greater-than > because of the possible confusion with Brackets.

```
(* iprod : int → Vector → Vector → int *)
fun iprod n v w =
    if n '>' 0
      then ((nth v n) * (nth w n)) + (iprod (n-1) v w)
      else 0;
```

```
(* iprod2 : int → ⟨Vector → ⟨Vector → int⟩⟩ *)
fun iprod2 n = ⟨fn v ⇒ ⟨fn w ⇒
    ~~(if n '>' 0
        then ⟨⟨(~(lift (nth v ~(lift n))) * (nth w ~(lift (~lift n)))) + (~(~(iprod2 (n-1)) v) w)⟩⟩
        else ⟨⟨0⟩⟩)⟩⟩;

(* iprodF2 : int → ⟨Vector⟩ → ⟨⟨Vector⟩⟩ → ⟨⟨int⟩⟩ *)
fun iprodF2 n v w =
    if n '>' 0
      then ⟨⟨ (~(lift (nth ~v ~(lift n))) * (nth ~~w ~(lift (~lift n)))) + ~~(iprodF2 (n-1) v w)⟩⟩
      else ⟨⟨0⟩⟩;

fun iprod3 n = back2 (iprodF2 n).
```

As we predicted in the last section, the version written with back2 has one less annotation. More importantly, annotating the version written with back2 seems less cumbersome. While this observation is a somewhat subjective, it may benefit future MetaML or multi-stage programming. We emphasize that the type inference mechanism and the interactive environment are especially useful when exploring different approaches to annotation and staging.

Another benefit of MetaML is providing us with a way to visualize multi-stage computation, in essence, by running them stage by stage and inspecting the intermediate results. By testing iprod3 on some inputs we can immediately see the results:

```
-| val f1 = iprod3 3;
f1 = ⟨fn d1 ⇒
          ⟨fn d5 ⇒
              (~(lift (%nth d1 3)) %* (%nth d5 3)) %+
              (~(lift (%nth d1 2)) %* (%nth d5 2)) %+
              (~(lift (%nth d1 1)) %* (%nth d5 1)) %+
              0⟩⟩
   : ⟨Vector → ⟨Vector → int⟩⟩.
```

When f1 is Run, it returns a function. When this function is applied to a vector, it builds another piece of code. This building process includes looking up each element in the first vector and splicing-in the actual value using the Lift operator. Lift is especially useful when we wish to inspect the generated code:

```
val f2 = (run f1) [1,0,4];
f2: ⟨Vector → int⟩ =
⟨fn d1 ⇒ (4 %* (%nth d1 3)) %+
          (0 %* (%nth d1 2)) %+
          (1 %* (%nth d1 1)) %+ 0⟩.
```

The actual values of the first array appear in the code, and the access function nth appears applied to the second vector d1.

This code still does not take full advantage of all the information known in the second stage. For example, we may wish to eliminate multiplications by 0 or 1. Such multiplications can be optimized without knowing the value of the second operand. To this end, we write a "smarter" function add which given a vector index i, an integer x from the first vector, and a piece of code y representing the second vector, constructs a piece of code which adds the result of the x and y multiplication to the code-valued fourth argument e.

```
(* add : int → int → ⟨Vector⟩ → ⟨int⟩ → ⟨int⟩ *)
fun add i x y e =
   if x=0 then e
        else if x=1 then ⟨(nth ~y ~(lift i)) + ~e⟩
                 else ⟨(~(lift x) * (nth ~y~(lift i))) + ~e⟩.
```

This specialized function can now be used to build an improved version of the iprodF2 function:

```
(* iprodFS2 : int → ⟨Vector⟩ → ⟨⟨Vector⟩⟩ → ⟨⟨int⟩⟩ *)
fun iprodFS2 n v w =
   if n = 1 then ⟨add n (nth ~v n) ~w ⟨0⟩⟩
        else ⟨add n (nth ~v n) ~w ~(iprodFS2 (n-1) v w)⟩;

fun iprodS2 n = back2 (iprodFS2 n).
```

Now let us look at the result of the first stage computation:

```
val f3 = iprodS2 3;
f3: ⟨Vector → ⟨Vector → int⟩⟩ =
⟨fn d1 ⇒
   ⟨fn d5 ⇒
      ~(%add 3 (%nth d1 3) ⟨d5⟩
            ⟨ ~(%add 2 (%nth d1 2) ⟨d5⟩
                   ⟨~(%add 1 (%nth d1 1) ⟨d5⟩
                          ⟨0⟩)⟩)⟩)⟩⟩.
```

This code is linear in the size of the vector. If we had actually in-lined the calls to add it would be exponential. Controlling code explosion is a side benefit of the sharing provided by cross-stage persistent constants such as add.

Now let us observe the result of the second stage computation:

```
val f4 = (run f3) [1,0,4];
f4: ⟨Vector → int⟩ = ⟨fn d1 ⇒ (4 %* (%nth d1 3)) %+ (%nth d1 1) %+ 0⟩.
```

Now only the multiplications that contribute to the answer remain. If the vector is sparse, then this sort of optimization can have dramatic effects.

Finally, note that while iprodFS2 is a three-stage function, the terms in the body of this function involve only two levels of terms. Such implicit increase in the number of stages is a prototypical example of how multi-stage programs develop naturally from the composition of two-stage programs, or any lower-number-of stages programs.

Chapter 3

Staging a Simple Term-Rewriting Implementation

And let this world no longer be a stage
To feed contention in a lingering act;

Northumberland, Act 1, Scene1,
King Henry IV, Shakespeare

In this chapter we present an extended example of applying the method of multi-stage programming with explicit annotations. Starting with a single-stage term-rewriting function, we annotate it to derive a two-stage function. We point out some limitations of this direct attempt to stage the function and show how re-implementing the function in continuation-passing style (CPS) improves the generated code substantially. Rewriting the program into CPS is an example of what is known in the partial evaluation literature as *binding-time improvement*.

3.1 Term-Rewriting

Dershowitz [24] defines a *term rewriting system* as a set of directed rules. Each rule is made of a left-hand side and a right-hand side. A rule may be applied to a term t if a sub-term s of t matches the left-hand side under some substitution σ. A rule is applied by replacing s with s', where s' is the result of applying the substitution σ to the right-hand side. We call the modified t term t', and say *"t rewrites (in one step) to t′ "*, and write $t \longrightarrow t'$. The choice of which rule to apply is made non-deterministically. As an example, the rules for a Monoid [24] are:

$$x + 0 \longrightarrow_{r_1} x$$

$$0 + x \longrightarrow_{r_2} x$$

$$x + (y + z) \longrightarrow_{r_3} (x + y) + z$$

Variables x, y, and z in such rules can each match any term. If a variable occurs more than once on the left-hand side of a rule, all occurrences must match identical terms. These rules allow us to have derivations such as:

$$\begin{array}{rl} & (a+b)+\underline{(0+(d+e))} \\ \longrightarrow & \text{by } r_2, \sigma = [(d+e)/x] \\ & \underline{(a+b)+(d+e)} \\ \longrightarrow & \text{by } r_3, \sigma = [(a+b)/x, d/y, e/z] \\ & ((a+b)+d)+e \end{array}$$

where the sub-term s being rewritten has been underlined.

Generally, such rules do not change over the life of a term-rewriting system. At the same time, the basic form of a matching function is a simultaneous traversal of a term and the left-hand side of the rule it is being matched against. The invariance of the rules during the normal lifetime of the rewriting system offers an opportunity for staging: We can specialize matching over the rules in a first stage, and eliminate the overhead of traversing the left-hand side of the rules. Not only that, but we will see that we can also remove a significant amount of administrative computations involved in constructing and applying the substitution σ. We expect that this staging would significantly reduce the cost of rewriting a term.

3.2 A Single-Stage Implementation

In this section, we present the implementation of part of a simple term-rewriting system.

3.2.1 An Implementation of Terms and Patterns

Both terms and patterns can be represented using the following MetaML type:

```
datatype term = Var of string
              | Int of int
              | Op of term * string * term.
```

The constructors of this type are Var, Int and Op, corresponding to variables, integers, and applications of binary operators. Thus, we can represent terms such as $t_1 = (a+b)+(d+e)$ and $t_2 = ((a+b)+d)+e$ as:

```
val t1 = Op(Op(Var "a","+",Var "b"),"+",Op(Var "d","+",Var "e"));
val t2 = Op(Op(Op(Var "a","+",Var "b"),"+",Var "d"),"+",Var "e").
```

We will represent rewrite rules as ordered pairs of terms. For example, the rules for a Monoid are represented as:

```
val r1 = (Op(Var "x","+",Int 0),Var "x");
val r2 = (Op(Int 0,"+",Var "x"),Var "x");
val r3 = (Op(Var "x","+",Op(Var "y","+",Var "z")),
          Op(Op(Var "x","+",Var "y"), "+", Var "z")).
```

We will represent substitutions as lists of pairs of strings and terms. For example, the substitution $[(a+b)/x, d/y, e/z]$ is represented as:

```
val s1 = [("x",Op(Var"a","+",Var"b")),
          ("y",Var"d"),
          ("z",Var"e")].
```

Finally, because matching can succeed or fail, we will use an instance of the following simple datatype for the return value of matching:

```
datatype 'a option = Nothing | Just of 'a.
```

The option type constructor provides a mechanism whereby one can write functions that can either succeed and return a value of an arbitrary type 'a, or fail to return such a value and simply return the value Nothing.

3.2.2 An Implementation of Matching

We focus on the function that tries to match a candidate term with a left-hand side of a rule. We assume that separate helper functions take care of applying this function to all sub-terms of the candidate term. The single-stage implementation of this matching function is as follows:

```
fun match1 pat msigma t =
case msigma of
  Nothing ⇒ Nothing
| Just (sigma) ⇒
(case pat of
   Var u ⇒ (case find u sigma of
               Nothing ⇒ Just ((u,t) :: sigma)
             | Just w ⇒ if w = t then Just sigma else Nothing)
|  Int n ⇒ (case t of
               Int u ⇒ if u=n then msigma else Nothing
            |  _ ⇒ Nothing)
|  Op(t11,s1,t12) ⇒ (case t of
                       Op (t21,s2,t22) ⇒ (if s2 = s1
                                              then (match1 t11
                                                       (match1 t12 msigma t22) t21)
                                              else Nothing)
                     | _ ⇒ Nothing)).
```

The match1 function takes a pattern pat, a substitution msigma, a candidate term t, and tries to compute a substitution that would instantiate the pattern to the term. First, match1 inspects msigma. If msigma is Nothing, this means that a failure has occurred in a previous match, and we simply propagate this failure by returning Nothing. Otherwise, a case analysis on the pattern pat is performed. If the pattern is a variable, the substitution is extended appropriately after checking for consistency with the new binding. We check consistency by looking up the name u in the environment sigma. Then, if there is no binding for that name, then we simply extend the environment. If there is already a binding for that name, and it is the same as the one we wish to add, then there is no need to perform the extension. If there is already a binding for that name, but it is different from the one we wish to add, then we fail.

If the pattern is an integer, then the term must also be an integer with the same value. In this case the original substitution is returned. If the pattern is an operator, we check that the term is also an operator with the same operation, and then the right- and left-hand sides of the pattern and the term are recursively matched, possibly extending the substitution. In all other cases, match1 returns Nothing to indicate that it has failed.

The match1 function has type:

$$\text{term} \rightarrow (\text{string} * \text{term})\ \text{list option} \rightarrow \text{term} \rightarrow (\text{string} * \text{term})\ \text{list option}$$

The two occurrences of option in the type can be explained as follows: First, match1 returns a substitution option, because it can fail to match the pattern to the term. Such failure occurs in all places where match1 returns Nothing. Second, the result of one call to match1 gets fed-back into another call to match1, hence the incoming and outgoing substitutions must have exactly the same type. This *threading* of the substitution happens in the case of an Op pattern:

```
...
(match1 t11
(match1 t12 msigma t22) t21)
... .
```

But matching the right operand t22 can also fail, in which case the whole match should fail. So, if match1 is passed an invalid substitution Nothing, then the outer case-statement immediately returns Nothing, thus correctly propagating the failure of matching t22.

3.2.3 Helper Functions

To use match1, we need some helper functions:

```
fun find v [] = Nothing
  | find v ((n,t)::bs) = if v=n then Just t else find v bs.
```

This function takes a variable name and a substitution and returns the term corresponding to this name, if any. Again, we use the option type for dealing with failure. Applying the substitution is performed by the following function:

```
fun subst1 sigma pat =
case pat of
  Var v ⇒ (case find v sigma of Just w ⇒ w)
| Int i ⇒ Int i
| Op(t1,s,t2) ⇒ Op (subst1 sigma t1, s, subst1 sigma t2).
```

This function takes a substitution and a pattern, and applies the substitution to the pattern. The function works by recursively traversing the pattern and replacing any variables by the corresponding binding in the substitution msigma. Applying a rewrite rule is performed by the following function:

```
fun rewrite1 (lhs,rhs) t =
case match1 lhs (Just []) t of
  Nothing ⇒ t
| Just (sigma) ⇒ subst1 sigma rhs.
```

This function takes a rule and a term, and uses match1 to try to compute a unifying substitution. If matching is successful, it applies the substitution using subst1. Otherwise, it returns the term unchanged.

3.2.4 What to Expect from Staging

Often, we can appraise the success of staging qualitatively, without need for quantitative benchmarking. Such an appraisal can be achieved by inspecting an intermediate result of a staged computation, comparing it to what can be achieved by hand-programming a specialized version of the desired program, based on the contextual information that is available. For example, a good hand-crafted version of rewrite1 specialized for the rule $r_1 : x + 0 \to x$ is as follows:

```
fun rewriteR1 term =
case term of
  Op (t1,s,t2) ⇒ if s = "+" then (case t2 of Int n ⇒ if n=0 then t1 else term
                                            | _ ⇒ term)
                            else term
| _ ⇒ term.
```

This function takes a term, checks if it is a binary operation, the operation is an addition, the second argument is an integer, and the integer is 0, then returns the first argument. If any of the checks fails, it returns the term unchanged. This function does not pay the interpretive overhead of traversing the pattern, and the effect of the substitution operation has also been computed in advance.

In the rest of this chapter, we will study how far MetaML can take us towards turning our single-stage program into a generator that produces such highly specialized code.

3.3 A First Attempt at Staging

Defer no time, delays have dangerous ends;

Reignier, Act 3, Scene 2,
King Henry VI, Shakespeare

In this section we analyze the type of the match1 function and develop a target type for a two-stage matching function based on match1. We then present the two-stage matching function. Finally, we inspect the generated code and contrast it to the goal that we have set in Section 3.2.4.

3.3.1 Designing the Multi-Stage Type

We have already decided that we will consider the candidate term to be available only in the second stage. This choice means that a first approximation of our target for the type of the staged function is:

$$\text{term} \to (\text{string} * \text{term})\ \text{list option} \to \langle\text{term}\rangle \to (\text{string} * \text{term})\ \text{list option}.$$

But based on our knowledge of the algorithm, we know that the substitution returned by the match function must contain sub-terms of the term being matched. Thus, the resulting substitution is going to be a *partially-static datatype*. In particular, what we will get back is a skeleton of a list of pairs, where the first element in the pair is a string, but the second element will not yet be known. Thus, a second approximation of the desired type is:

$$\mathsf{term} \to (\mathsf{string} * \mathsf{term})\ \mathsf{list\ option} \to \langle\mathsf{term}\rangle \to (\mathsf{string} * \langle\mathsf{term}\rangle)\ \mathsf{list\ option}.$$

Going back to the algorithm, recall that we *thread* the substitution through two recursive calls to the match function in the case of matching binary operations. This threading means that, in a staged implementation of match1, the result of the term match1 t12 msigma t22 will also have to have type (string * ⟨term⟩) list option. The change in the type of this term forces us to change the type of the substitution *passed* to the staged match function:

$$\mathsf{term} \to (\mathsf{string} * \langle\mathsf{term}\rangle)\ \mathsf{list\ option} \to \langle\mathsf{term}\rangle \to (\mathsf{string} * \langle\mathsf{term}\rangle)\ \mathsf{list\ option}.$$

The "If" Problem The type above looks promising. Unfortunately, there is a subtlety in our matching algorithm that forces us to reflect more delays in the staged type. We say unfortunately because having more delays in the type usually suggests that we will not be able to stage this function as effectively as we would like. The problem with our algorithm is that the option part of the return value depends on testing the equality of terms in the substitutions. In particular, testing the consistency of a new binding with the existing substitution is performed by equating terms. This test is carried out in the statement:

```
...
if w = t then Just sigma else Nothing
... .
```

When the condition of an if-statement is delayed, the result of the whole if-statement must also be considered as delayed. The reason is that we will not know which branch to pursue until all the information for the condition becomes available. Thus, the return type for the staged match function will have to be ⟨_ option⟩. We are left with the following target type:

$$\mathsf{term} \to \langle(\mathsf{string} * \mathsf{term})\ \mathsf{list\ option}\rangle \to \langle\mathsf{term}\rangle \to \langle(\mathsf{string} * \mathsf{term})\ \mathsf{list\ option}\rangle$$

This type indicates that only the pattern is inspected in the first stage and the result is a specialized function that can be Run in the second stage. In the following subsection, we will see how well we can take advantage of this information in a multi-stage program.

3.3.2 A Two-Stage Implementation

Following a similar process to that described for staging the list membership function (Section 2.4), we arrive at the following two-stage version of match1:

```
fun match2 pat msigma t =
⟨case ˜msigma of
   Nothing ⇒ Nothing
|  Just (sigma) ⇒
˜(case pat of
    Var u ⇒ ⟨case find u sigma of
                 Nothing ⇒ Just ((u,˜t) :: sigma)
               | Just w ⇒ if w = ˜t then Just sigma else Nothing⟩
|   Int n ⇒ ⟨case ˜t of
                Int u ⇒ if u=˜(lift n) then ˜msigma else Nothing
              | _ ⇒ Nothing⟩
|   Op(t11,s1,t12) ⇒ ⟨case ˜t of
                        Op (t21,s2,t22) ⇒ (if s2 = s1
                                           then ˜(match2 t11
                                                  (match2 t12 msigma ⟨t22⟩) ⟨t21⟩)
                                           else Nothing)
    | _ ⇒ Nothing⟩)⟩.
```

Note that if we erase the annotations from the source for match2, we get the source for match1. We will also need to stage the helper function rewrite1:

```
fun rewrite2 (lhs,rhs) t =
⟨case ˜(match2 lhs ⟨Just []⟩ t) of
   Nothing ⇒ ˜t
|  Just (sigma) ⇒ subst1 sigma rhs⟩.
```

We are now ready to inspect the code generated by this function and to appraise its quality.

3.3.3 Using the Two-Stage Implementation

We can use the rewrite2 function to generate code specialized for $r_1 : x + 0 \to x$ as follows:

```
-| rewrite2 r1;
val it = fn : ⟨term⟩ → ⟨term⟩

-| ⟨fn x ⇒ ˜(it ⟨x⟩)⟩;
val it =
⟨fn a ⇒
(case (case a of
        Op(e,d,c) ⇒ if d %= %s1'1230
                     then (case (case c of
                                              Int h ⇒ if h %= 0 then Just ([]) else Nothing
                                            | _ ⇒ Nothing) of
                                        Nothing ⇒ Nothing
                                      | Just f ⇒ (case %find %u'1237 f of
                                                      Nothing ⇒ Just (::((%u'1237,e),f))
                                                    | Just g ⇒ if g %= e then Just f else Nothing))
                                  else Nothing
                 | _ ⇒ Nothing) of
              Nothing ⇒ a
            | Just b ⇒ %subst1 b %rhs)⟩
          : ⟨term → term⟩.
```

The traversal of the pattern has been performed. But compared to the target code we wrote by hand, there are still too many nested case-statements, and the call to subst1 and the test for consistency with the substitution were not performed.

Closer inspection of the generated code shows that it ought be possible to reduce the nested case-statements by some meaning-preserving transformations. If the outer case-statements could be "pushed" through the inner ones, then we would be able to simplify all the values at the leaves of the inner if-statements. In particular, at every point where we return Nothing we ought to be able to simply return a, and every point where we return Just σ, we ought to be able to return the result of applying the substitution subst σ rhs.

3.4 A Second Attempt

It is not unusual that the first attempt at staging does not yield the best results. Even when staging is automated, as is the case with BTA, it is common for users of offline partial evaluators to restructure their programs to help BTA yield better results [39]. Just because we place binding-time annotations manually does not exempt us from this requirement. But because MetaML allows us to "touch, see, and experiment" with both the explicit annotations and the code produced, it helps us better understand the problem of staging a particular algorithm. We believe that a strength of MetaML is the mental model it provides for reasoning about multi-stage computation.

3.4.1 An Implementation of Matching in Continuation-Passing Style

One possible approach to improving the generated code is to rewrite match1 in *continuation-passing style* (CPS). This technique has been found to be quite useful in partial evaluation systems [15]. Intuitively, a *continuation* is a function that specifies how the computation will "continue". A function written in CPS always takes a continuation k as an argument, and applies k to the results it would normally just return. When a CPS function calls another CPS function, it passes k as the new continuation to the second function. Furthermore, whenever there is any sequencing of function calls involved, a function in CPS must explicitly pass around and extend the continuation k as necessary.

This subtle change to the form of the algorithm gives us an extra degree of freedom that provides a new opportunity for a better design for our multi-stage type, and in turn, for a better multi-stage program.

The CPS matching function takes a continuation k of type (string * term) list option → 'a. Informally, we can read this type as saying that the continuation is some "consumer of substitutions". The CPS matching function is defined as follows:

```
fun matchK1 pat k msigma t =
case (msigma) of
  Nothing ⇒ k Nothing
| Just (sigma) ⇒
(case pat of
   Var u ⇒ (case find u sigma of
                  Nothing ⇒ k (Just ((u,t) :: sigma))
                | Just w ⇒ if w = t then k (Just sigma) else k Nothing)
 | Int n ⇒ case t of
                Int u ⇒ if u= n then k msigma else k Nothing
              | _ ⇒ k Nothing
 | Op(p11,s1,p12) ⇒ case t of
                         Op(t21,s2,t22) ⇒ if s1 = s2
                                           then (matchK1 p11
                                                   (fn msig ⇒ matchK1 p12 k msig t22)
                                                   msigma t21)
                                           else k Nothing
 | _ ⇒ k Nothing).
```

The threading of the substitution in the case of Op patterns has been replaced by one call to matchK1, which itself takes the second call to matchK1 as part of its continuation. More importantly, the continuation has been explicitly placed on the *branches* of the if-expression, as opposed to just around the whole expression. This minor variation will have a significant effect on what can be achieved by staging the matching function.

To use a function written in CPS, one must provide an initial continuation. Intuitively, the initial continuation is the final step of the computation. The initial continuation can simply be the identity function, specifying no further computation on the final result. In the case of the matchK1 function, however, we need a more interesting continuation: the application of the final substitution to the right-hand side of the rewrite rule. To pass this continuation to matchK1, we write a new version of rewrite1 as follows:

```
fun rewriteK1 (lhs,rhs) =
fn t ⇒ (matchK1 lhs
   (fn Nothing ⇒ t
     | Just s ⇒ substK2 s rhs) (Just []) t).
```

The continuation that we pass to matchK1 first inspects the final result of matching. If it was a failure, the result is returned unchanged. Otherwise, the substitution is applied to the right-hand side of the rewrite rule. As usual, the call to matchK1 is also passed an empty substitution and the term.

3.4.2 Re-Designing the Multi-Stage Type

The type of the CPS matching function is:

$$((\mathsf{string} * \mathsf{term})\,\mathsf{list}\,\mathsf{option} \rightarrow\ '\mathsf{a}) \rightarrow (\mathsf{string} * \mathsf{term})\,\mathsf{list}\,\mathsf{option} \rightarrow \mathsf{term} \rightarrow\ '\mathsf{a}$$

This function has a polymorphic type because nothing constrains the value returned by the continuation k. But note that we should not read the "return" type of this function as 'a, but rather as the type of the argument of the continuation, which is the first occurrence of the type (string * term) list option.

In contrast to the first attempt at designing a multi-stage type for the matching function, nothing forces us to "return" a delayed option type. Recall from Section 3.3.1 that the difficulty in achieving a type for the staged function was caused by the if-statement that tested for the equality of two terms. This test forced the whole if-expression to be delayed. Now, the dependencies have changed:

```
...
if u=n then k msigma else k Nothing
... .
```

The unavailability of the condition of the if-statement no longer matters. Because we have an explicit handle on the continuation, we do not have to "freeze" both branches of the if-statement. Instead *we can pursue both possible ways in which the computation might continue.* Pursuing the continuations could mean eliminating many superfluous Nothing and Just constructs. For example, the initial continuation passed to matchK1 by rewriteK1 is a case analysis over the result. As such the initial continuation *eliminates* the constructs of the option type. Furthermore, the only time we extend the continuation is when we are matching binary operators:

```
...
(matchK1 p11
 (fn msig ⇒ matchK1 p12 k msig t22)
 msigma t21)
... .
```

As long as match continues to "consume" the option constructors of sigma, the new continuation will also be a "consumer". This observation suggests that the delayed computations generated by both branches of the if-statement may be free of any option constructors. At the same time, eliminating these constructors should coincide with eliminating the complex nested case-statements that infested the output of our first attempt at staging the match function.

In summary, now there is no reason why we cannot achieve the type:

$$((\mathsf{string} * \langle\mathsf{term}\rangle)\,\mathsf{list\,option} \rightarrow \langle\,'\mathsf{a}\rangle) \rightarrow (\mathsf{string} * \langle\mathsf{term}\rangle)\,\mathsf{list\,option} \rightarrow \langle\mathsf{term}\rangle \rightarrow \langle\,'\mathsf{a}\rangle.$$

3.4.3 A Two-Stage Continuation-Passing Style Implementation

Adding staging annotations to matchK1 we arrive at the following two-stage function:

```
fun matchK2 pat k msigma t =
case (msigma) of
  Nothing ⇒ k Nothing
| Just (sigma) ⇒
(case pat of
   Var u ⇒ (case find u sigma of
                Nothing ⇒ k (Just ((u,t) :: sigma))
              | Just w ⇒ ⟨if ~w = ~t then ~(k (Just sigma)) else ~(k Nothing)⟩)
 | Int n ⇒ ⟨case ~t of
               Int u ⇒ if u= ~(lift n) then ~(k msigma) else ~(k Nothing)
             | _ ⇒ ~(k Nothing)⟩
 | Op(p11,s1,p12) ⇒ ⟨case ~t of
                        Op(t21,s2,t22) ⇒ if ~(lift s1) = s2
                                         then ~(matchK2 p11
                                                (fn msig ⇒ matchK2 p12 k msig ⟨t22⟩)
```

```
                                              msigma ⟨t21⟩)
                                         else ˜(k Nothing)
  | _ ⇒ ˜(k Nothing)⟩).
```

The type of the substitutions must change: The skeleton of the list and the first projection of each binding (the variable names) are now available in the first stage. Thus, we can rewrite the substitution function to take advantage of this information:

```
fun substK2 sigma pat =
case pat of
  Var v ⇒ (case find v sigma of Just w ⇒ w)
| Int i ⇒ ⟨Int ˜(lift i)⟩
| Op(t1,s,t2) ⇒ ⟨Op (˜(substK2 sigma t1), ˜(lift s), ˜(substK2 sigma t2))⟩.
```

Finally, we also need to annotate the CPS rewriting function:

```
fun rewriteK2 (lhs,rhs) =
⟨fn t ⇒ ˜(matchK2 lhs
    (fn Nothing ⇒ ⟨t⟩
      | Just s ⇒ substK2 s rhs) (Just []) ⟨t⟩)⟩.
```

3.4.4 Using the Two-Stage Continuation-Passing Style Implementation

The output of the CPS rewrite function for the rule r_1 is as follows:

```
-| rewriteK2 r1;
val it = ⟨(fn a ⇒
case a of
  Op(d,c,b) ⇒ if "+" %= c then (case b of Int e ⇒ if e%=0 then d else a | _ ⇒ a)
                              else a
| _ ⇒ a)⟩
: ⟨term → term⟩.
```

The generated code is both compact and free of reducible nested case-statements, calls to subst, and tests on the consistency of the substitution. Furthermore, it is virtually identical to the idealized code we presented for rewriteR1 in Section 3.2.4: It contains no superfluous redices, and rather, only those that really depend on the parameter to the variable a.

The code generated for the more involved r_3 is also concise:

```
-| rewriteK2 r3;
val it = ⟨fn a ⇒
```

```
(case a of
   Op(d,c,b) ⇒ if "+" %= c then (case b of
                                         Op(g,f,e) ⇒ if "+" %= f then Op(Op(d,"+",g),"+",e)
                                                          else a
                                       | _ ⇒ a)
                             else a
 | _ ⇒ a))
: ⟨term → term⟩.
```

3.4.5 Remark on the Role of MetaML in Multi-Stage Programming

If the reader is not familiar with type systems and with programming languages such as SML, it may appear that designing the type for the staged version (Sections 3.3.1 and 3.4.2) is a black art. What we have described in this chapter as "designing the multi-stage type" is essentially what binding-time analysis (BTA) performs automatically. MetaML provides pedagogical help in "removing the magic" from BTA in two ways:

1. It provides a formal semantic basis for multi-stage programs. This foundation is needed, for example, to prove that MetaML can indeed have a "staging-sensitive" type system, and that this type system ensures that well-typed programs are indeed safe.
2. It provides a system for executing multi-stage programs. For example, having the type inference system and the interactive section provided by the implementation gives the programmer an automatic tool that will point out *all* the problems that we described in the last subsection. Thus, the programmer can begin learning multi-stage programming by trial-and-error. The interactive type system will always tell you when you try to do something unsafe!

In our experience, the feedback provided by the type system helped greatly in developing the right insights and intuitions about the structure of the algorithms that we were interested in staging. Even after a programmer has gained the necessary expertise and has become fluent at staging programs, the type inference system will automatically check the programmer's annotations to ensure that they are indeed sound.

3.4.6 Remark on the Specialization of Term-Rewriting Systems

Bondorf seems to have been the first to systematically study the application of partial evaluation to term-rewriting systems [7]. For the reader interested in the specialization of term-rewriting systems, we recommend consulting Bondorf's thesis [8].

Part II

The Theory of Multi-Stage Programming

Chapter 4

How Do We Implement MetaML? Or, the Basic Semantic Concerns

I test my bath before I sit,
And I'm always moved to wonderment
That what chills the finger not a bit
Is so frigid upon the fundament.

Samson Agonistes, Ogden Nash

In previous chapters, we have introduced MetaML, staging, multi-stage programming with explicit annotations, cross-stage persistence, and cross-stage safety. In this chapter we explain the basic challenges in implementing and typing a multi-stage programming language such as MetaML. We break down these problems into two sets: Ones that were known when the early implementations of MetaML were developed, and new ones that we recognized while using these implementations. This chapter is illustrative of the state of the art in the (potentially verifiable) implementation of multi-stage programming languages.

4.1 Known Concerns

We begin by explaining the need for renaming of bound variables in quoted expressions and why such renaming is necessary for implementing static scoping. We also explain why retaining static scoping is desirable from the point of view of the design goals of MetaML. We then present the syntax and type system for a toy functional language and explain how the type system must be changed to accommodate extending this toy language with staging annotations.

4.1.1 Scoping and the h Function

A name-generating function ("gensym") can be used to avoid accidental name capture in multi-level languages. The use of such a function has been associated with use of back-quote and comma in LISP ever since they were introduced [88]. The back-quote and comma mechanism is an early ancestor of Brackets and Escape (See Section 7.4.2.) Various works have studied and explained the need for run-time renaming of bound variables [13]. In what follows, we present a brief explanation of the need for renaming, and why it must be performed at run-time. To illustrate this point, we will introduce a simple function called h that we use throughout this chapter.

The Need for Renaming In a multi-level language, object-level bound variables must be renamed. Consider the function T from Chapter 1:

```
fun T e = ⟨fn x ⇒ x + ˜e⟩.
```

The function T takes an expression e and returns a new expression representing a function that takes an argument x and adds it to e. We can use T as follows:

```
val e = ⟨fn y ⇒ ˜(T ⟨y⟩)⟩.
```

And the result is:

```
⟨fn y ⇒ fn x ⇒ x + y⟩.
```

If we do not rename bound variables while constructing a piece of code, we run into the following problem. If, in e, we had called the local variable x instead of y, then we would get the result:

```
⟨fn x ⇒ fn x ⇒ x + x⟩.
```

This term represents a function that is very different from the first one, because it simply ignores its first argument, and returns the result of adding the second argument to itself. We consider this dynamic capture of names unsatisfactory, because we wish to maintain a relatively simple relation between our multi-level programs and their "unannotated counterparts". In particular, if we erase the annotations from T and e we get:

```
fun T' e = fn x ⇒ x + e
val e' = fn x ⇒ T' x.
```

The standard semantics for a CBV language does not confuse the two occurrences of the name x in this declaration. We can check this fact using an implementation of SML:

```
- e' 4 5;
val it = 9 : int.
```

Renaming Cannot Be Performed at Compile-Time If we consider only examples such as the function T, it may seem enough to rename variables in a program *before* we begin executing it. But renaming before execution is not enough: Bound variables in object-terms must be repeatedly renamed at run-time. To illustrate this point, we will present two versions of a function that we call h. The first, single-stage version is called h1, and second, two-stage version is called h2. The single-stage h1 function is defined as follows:

```
fun h1 n z = if n=0 then z else (fn x ⇒ (h1 (n-1) x+(z))) n.
```

This function takes two integer arguments. If the first argument is 0, then it simply returns the second argument. Otherwise, it makes a recursive call to itself after subtracting one from the first argument and adding the first argument to the second. Note however that we use a local variable x as a local name for the second argument n. This fact is inconsequential to the result of h1, but will be significant when we analyze the staged version h2:

```
fun h2 n z = if n=0 then z else ⟨(fn x ⇒ ˜(h2 (n-1) ⟨ x+(˜z)⟩)) n⟩.
```

Note that any renaming of this one variable before run-time is inconsequential: It will still be one variable. Without renaming at run-time, the application h2 3 ⟨4⟩ evaluates to:

$$\underline{\langle(\mathsf{fn}\ x \Rightarrow \underline{((\mathsf{fn}\ x \Rightarrow \underline{((\mathsf{fn}\ x \Rightarrow x+(x+(x+4)))\ 1)})\ 2)})\ 3\rangle}$$

and all the occurrences of the name x in the arithmetic expression refer to the argument from the innermost binding. But this accidental capture of all references by the innermost binder should not take place. If we follow the execution of the program closely, we find that each occurrence of an x should be associated to a different binding in the following manner:

$$\underline{\langle(\mathsf{fn}\ x_1 \Rightarrow \underline{((\mathsf{fn}\ x_2 \Rightarrow \underline{((\mathsf{fn}\ x_3 \Rightarrow x_3+(x_2+(x_1+4)))\ 1)})\ 2)})\ 3\rangle}.$$

Again, the reader can verify that the result of evaluating this last code fragment is more closely related to the "unannotated" semantics of the h function.

In Section 4.3.2, we will show how this renaming is performed in the implementation.

4.1.2 Typing Functional Languages and MetaML (λ and λ-M)

Most functional programming languages are inspired by the lambda calculus [1]. In order to begin developing the semantics and type system for MetaML, we focus on a minimal subset of MetaML that can be studied as an extension of the lambda calculus. In what follows, we introduce the syntax and type system of a lambda calculus extended to include integers, and then explain how the type judgement must be extended in order to be able to express the notion of staging.

The λ Language: Terms and Type System A basic lambda calculus [1] extended to include integer constants has the following syntax:

$$e \in E := i \mid x \mid \lambda x.e \mid e\,e$$

where E is the set of expressions. Expressions can be integers from the set $\mathbb{I}$, lambda-abstractions, free occurrences of variables x, or applications $e\,e$ of one expression to another. A lambda-abstraction $\lambda x.e$ introduces the variable x so that it can occur free in the expression e, which is called the *body* of the lambda term.

Type systems allow us to restrict the set of programs that we are interested in (For an introduction, see [2, 10, 12, 36, 104].) For example, in the language presented above, we might not be interested in programs that apply integers to other expressions. Thus, a common restriction on application is to allow only terms that have a *function type* to be applied. Types have the following syntax:

$$\tau \in T := \mathsf{int} \mid \tau \rightarrow \tau$$

The two productions stand for integers and for function types, respectively. This simple language includes type terms such as:

- int,
- $\mathsf{int} \rightarrow \mathsf{int}$,
- $\mathsf{int} \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$, usually written simply as $\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$, and
- $(\mathsf{int} \rightarrow \mathsf{int}) \rightarrow \mathsf{int}$.

To build a type system, we usually also need a notion of a type environment (or type assignment). Such environments simply associate variable names to types, and can be represented as follows[1]:

$$\Gamma \in D := [] \mid x : \tau; \Gamma$$

A type system for the λ language can be specified by a judgment $\Gamma \vdash e : \tau$ where $e \in E$, $\tau \in T$, $\Gamma \in D$. Intuitively, the type judgement will associate the term int, which up until now was mere syntax, with integers, and the type term $\tau_1 \rightarrow \tau_2$ with partial functions that take an argument of type τ_1 and may return a result of type τ_2 (or diverge). Because terms can contain free variables[2],

[1] Technically, elements of D represent (or implement) finite mappings of names to types. For this implementation to be correct, we also need the additional assumption of having a variable occur at most once in the environment.

[2] Free variables must be addressed even if we are considering only closed terms. In particular, typing judgements are generally defined by induction on the structure of the expression that we wish to assign a type to. In the case of lambda-abstraction, the judgement must go "under the lambda". In the body of this lambda-abstraction, the bound variable introduced by the lambda is free. To give a concrete example, try to see what happens when we wish to establish that the (closed expression) $\lambda x.x$ has type $\mathsf{int} \rightarrow \mathsf{int}$ under the empty environment $[]$.

the type judgement involves an environment that simply associates the free variables in the given term with a single type.

The type system, or more precisely, the derivability of the typing judgment is defined by induction on the structure of the term e as follows:

$$\frac{}{\Gamma \vdash i : \mathsf{int}}\,\text{Int}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\,\text{Var} \qquad \frac{x : \tau'; \Gamma \vdash e : \tau}{\Gamma \vdash \lambda x.e : \tau' \to \tau}\,\text{Lam} \qquad \frac{\Gamma \vdash e_2 : \tau' \quad \Gamma \vdash e_1 : \tau' \to \tau}{\Gamma \vdash e_1\ e_2 : \tau}\,\text{App}$$

The rule for Integers (Int) says that an integer i can have type int under all environments. The rule for variables (Var) says that a variable x has type τ under all environments where x was bound to type τ. The rule for lambda-abstractions (Lam) says that an abstraction can have an arrow type from τ' to τ under all environments as long as the body of the abstraction can be shown to have type τ when the environment is extended with a binding of the variable x with the type τ'. Finally, the rule for application (App) says that an application can have type τ under all environments where the operator can be shown to have type $\tau' \to \tau$ and the argument to have type τ'.

The λ-M Language: Terms and Type System The first step to extending the λ language with staging constructs is adding Brackets, Escape, and Run[3] to the syntax:

$$e \in E := i \mid x \mid e\,e \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e.$$

The second step is extending the type system. Extending the type system involves extending the type terms to include a type term for code:

$$\tau \in T := b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle$$

We will also need to extend the type environments to keep track of the level at which a variable is bound. To represent environments we will need a representation of the naturals:

$$n, m \in \mathbb{N} := 0 \mid n+$$

With a slight abuse of notation, we will also write:

$$\begin{aligned} n + 0 \quad &:= n \\ n + (m+) &:= (n + m)+ \end{aligned}$$

[3] We drop Lift from the rest of our treatment because it does not introduce any unexpected complications.

The two productions stand for 0 and for "next" number, respectively. Now we can present the type environments that will be used for typing λ-M:

$$\Gamma \in D := [] \mid x : \tau^n ; \Gamma$$

To be able to address the issue of cross-stage safety in a type system, we introduce a level-index into the typing judgment. The type judgment is thus extended to have the form $\Gamma \vdash^n e : \tau$ where $e \in E$, $\tau \in T$, $\Gamma \in D$, and $n \in N$. The judgement $\Gamma \vdash^n e : \tau$ is read *"e has type τ at level n under environment Γ"*. It is important to note that the level n is part of the judgement and *not* part of the type.

$$\begin{aligned} \tau \in T &:= b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle \\ \Gamma \in D &:= [] \mid x : \tau^n ; \Gamma \end{aligned}$$

Note that we have also extended type assignments to keep track of the level at which a variable is bound. The level index on the judgement combined with the level-annotations on variables will be used to reject terms where a variable violates cross-stage safety.

The type judgment is defined by induction on the structure of the term e as follows[4]:

$$\frac{}{\Gamma \vdash^n i : \mathsf{int}}\,\text{Int}$$

$$\frac{\Gamma(x) = \tau^{(n-m)}}{\Gamma \vdash^n x : \tau}\,\text{Var} \qquad \frac{x : \tau'^n ; \Gamma \vdash^n e : \tau}{\Gamma \vdash^n \lambda x.e : \tau' \to \tau}\,\text{Lam} \qquad \frac{\Gamma \vdash^n e_2 : \tau' \quad \Gamma \vdash^n e_1 : \tau' \to \tau}{\Gamma \vdash^n e_1\ e_2 : \tau}\,\text{App}$$

$$\frac{\Gamma \vdash^{n+} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\,\text{Brk} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+}\ \tilde{}e : \tau}\,\text{Esc} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \mathsf{run}\ e : \tau}\,\text{Run}$$

The rule for Integers (Int) allows us to assign the type int to any integer i, at any level n, and under any environment Γ.

The rule for variables (Var) allows us to assign any type τ to any variable x at any level n under any environment where x has been associated with the same type τ and with any level *less than or equal to n*.

The rule for lambda-abstractions (Lam) works essentially in the same way as for the lambda language. The noteworthy difference is that it associates the level n with the variable x when it is used to extend the environment. The lambda rule is the only rule where level bindings are introduced into the environments, and is essential for ensuring cross-stage safety.

[4] Warning: We have not yet proved whether these rules are sound or not. The soundness of these rules is discussed in Section 4.4.2.

The rule for application (App) is also essentially the same as before, the only difference being that we type the subexpressions under the same level n as the whole expression.

The rule for Brackets (Bra) "introduces" the code type, and at the same time, increments the level at which we type its subexpression to $n+$. This change to the level index on the judgement ensures that the level-index on our type system is counting the number of surround Brackets for any subexpression consistently.

The rule for Escapes (Esc) "eliminates" the code type, and at the same time, decrements the level at which we type its subexpression to n. This change to the level index on the judgement ensures that the level-index on our type system is counting the number of surrounding Escapes for any subexpression consistently. Note that the type system does not allow Escapes to occur at level 0.

The rule for Run (Run) "eliminates" the code type without any constraints on the level or the environment.

4.2 A Simple Approach to Implementing a Functional Language

4.2.1 Values

An elegant implementation of a toy functional language can be developed around a datatype containing functions (See for example [75].) An element of this datatype represents the *value* of a term in the toy functional language.

A datatype containing functions can be declared in SML as follows:

```
datatype value = VI of int               (* Integer values *)
               | VF of value → value     (* Function values *).
```

This datatype allows us to pass around values corresponding to the interpretation of functions such as

1. Identity: `fn x ⇒ x`,
2. Function composition: `fn f ⇒ fn g ⇒ fn x ⇒ f (g x)`,
3. Square: `fn x ⇒ x * x`, or
4. Factorial: `fn x ⇒ x!`, where $*$ and ! are the integer multiplication and factorial functions.

These values can be represented in the datatype `value`:

```
val id       = VF (fn x ⇒ x);
val compose  = VF (fn (VF f) ⇒ VF (fn (VF g) ⇒ VF (fn x ⇒ f (g x))));
val square    = VF (fn (VI x) => (VI (x*x)));
val bang      = let fun fact n = if n=0 then 1 else n*(fact (n-1)
                  in VF (fn (VI x) => (VI (fact x))) end.
```

The last example illustrates how this encoding allows us to take features of the meta-language (in this case SML) and embed them into the value domain of our toy object-language. In the factorial example, we are taking advantage of the following features of the meta-language:

- recursion,
- conditionals,
- arithmetic operations,

to introduce factorial into the object-language.

4.2.2 Expressions

Just as we implemented values in the datatype value, a datatype to implement the expressions of our language can be declared in SML as follows:

```
datatype exp = EI of int              (* Integers *)
             | EA of exp * exp        (* Applications *)
             | EL of string * exp     (* Lambda-abstractions *)
             | EV of string           (* Variables *).
```

Closed terms such as fn x ⇒ x and fn f ⇒ fn g ⇒ fn x ⇒ f (g x) can be encoded into the datatype exp as follows:

```
val ID        = EL ("x", EV "x");
val COMPOSE = EL ("f", EL ("g", EL ("x", EA (EV "f", EA (EV "g", EV "x"))))).
```

There are no closed terms in the syntax of our (typed) object-language that can express the functions square and bang[5]. However, we will see how they can expressed as open terms using a sufficiently rich environment.

4.2.3 Environments

The job of our interpreter will be to take terms such as ID and COMPOSE and produce the values id and compose, respectively. We can write a very concise interpreter ev0 having the following type:

$$\text{exp} \rightarrow \text{env} \rightarrow \text{value}$$

[5] We are not aware of any proof of this statement. Negative results on expressivity are generally a substantial challenge. Thus the reader should take this statement as nothing more than folklore.

where env is the type of an environment. The environment associates a value to each free variable in the expression being evaluated. For simplicity, we will take env to simply be string → value, that is, we will represent environments by a function that takes a variable name and returns a value. All we need to support this simple implementation of environments is to define the empty environment, and an environment extension function:

```
exception NotBound
val env0 = fn x ⇒ raise NotBound

fun ext env x v = fn y ⇒ if x=y then v else env y.
```

The empty environment is a function that takes a variable and raises an exception to indicate that this variable is not bound. Raising this exception is one of the ways in which our semantics "can go *Wrong*" and it is the role of the type system to ensure that any well-typed program does not go wrong when it is being evaluated (or interpreted).

The environment extension function takes an environment, a variable name, and a value, and returns a new function of type string → value (which is the new environment). This function returns v when it is applied to x and returns env applied to y otherwise.

4.2.4 Interpretation

We can now define a CBV interpreter for the λ language in a mere six lines:

```
fun ev0 env e =
(case e of
   EI i ⇒ VI i
 | EA (e1,e2) ⇒ (case (ev0 env e1, ev0 env e2) of (VF f, v) ⇒ f v)
 | EL (x,e1) ⇒ VF (fn v ⇒ ev0 (ext env x v) e1)
 | EV x ⇒ env x).
```

This interpreter is easy to use. By equational reasoning at the level of the meta-language (SML), we can see that the interpretation for ID and COMPOSE under the empty environment env0 should produce the values id and compose, respectively. But because these values contain function types, they are not printable, so we cannot "see" the output. We can, however, see the output of the interpreter when the result is an integer. For example, we would expect the object-term (fn x ⇒ x) 5 to evaluate to 5. The encoding of the term is simply EA (ID, EI 5). Applying the interpreter ev0 to this term under the empty environment env0 produces the expected result:

```
- ev0 env1 (EA (EL ("x",EV "x"), EI 5));
val it = VI 5 : value.
```

4.2.5 Introducing Constants

The implementation technique described above is appealing, partly because it allows us to introduce a rich set of constants into our language by simply extending the environment, rather than modifying the interpreter itself. Such constants can include an addition function or even a fixed-point operator (allowing us to implement recursion).

We can easily extend the object-language without modifying the syntax, the set of values, or the interpreter itself. A lot of expressivity can be added to the object-language by simply extending the initial environment under which the terms of the language are evaluated. For example, we can define the following meta-level constants:

```
val plus  = VF (fn (VI x) => VF (fn (VI y) => VI (x+y)));
val minus = VF (fn (VI x) => VF (fn (VI y) => VI (x-y)));
val times = VF (fn (VI x) => VF (fn (VI y) => VI (x*y))).
```

With these constants, we can express terms that evaluate to square. Such terms can contain free variables that are bound in an extended environment. For example, we can construct an environment binding a variable called * to the function value times:

```
val env1 = ext env0 "*" times.
```

We encode the term fn x => x*x using the following open term[6]:

```
val SQUARE = EL ("x", EA (EA (EV "*", EV "x"), EV "x")).
```

Now we can use ev0 to evaluate square 5:

```
- ev0 env0 (EA (SQUARE, EI 5));
val it = VI 25 : value.
```

Which produces an encoding of the integer value 25, as expected.

Introducing Conditionals through a Constant We can introduce a conditional statement into the language through a constant. In general, the use of conditional statements can be replaced by using the following function:

```
val Z = fn n => fn tf => fn ff => if n=0 then tf 0 else ff n.
```

[6] Our convention is that x*y is syntactic sugar for (* x) y, where * is a free variable. This convention is different from the SML convention where the same term is syntactic sugar for * (x,y). Using the SML convention would require introducing tuples or at least pairs into our toy language.

This function takes an integer n, a function from integers to booleans, and a second function from integers to booleans. If the integer is 0, it is simply passed to the first function[7] and the result is returned. Otherwise, the integer is passed to the second function and the result is returned. So, in general, we can replace any expression

$$\mathsf{case}\ e_1\ \mathsf{of}\ 0 \Rightarrow e_2 \mid \mathsf{n} \Rightarrow e_3$$

by an expression with one less conditional statement:

$$\mathsf{Z}\ e_1\ (\mathsf{fn}\ 0 \Rightarrow e_2)\ (\mathsf{fn}\ \mathsf{n} \Rightarrow e_3)$$

The function Z can be represented in the datatype value as:

```
val ifzero = VF (fn (VI n) ⇒ VF (fn (VF tf) ⇒ VF (fn (VF ff) ⇒ if n=0 then tf (VI 0)
                                                                  else ff (VI n)))).
```

Introducing Recursion through a Constant Recursion can also be introduced into the language through a fixed-point function[8]. In essence, the function we need is[9]:

```
val Y = let fun Y' f = f (fn v ⇒ (Y' f) v) in Y' end.
```

Now, a declaration:

```
fun f n = ... f ...
```

can be replaced by another declaration containing one fewer recursive expression:

```
val f = Y (fn f ⇒ fn n ⇒ ... f ...)
```

The function Y can be implemented in the datatype value as:

```
val recur = let fun recur' (VF f) =
                    f (VF (fn v ⇒
                          case (recur' (VF f)) of VF fp ⇒ fp v))
            in VF recur' end.
```

Now, equipped with a way of expressing conditionals and recursion, we can express a wide variety of interesting programs in our toy functional language.

[7] If our meta-language (SML) and our toy language were CBN rather than CBV, the conditional statement would have been slightly simpler. In particular, we would not need to pass the 0 to the first function. In a CBV language, this otherwise obsolete argument is needed to maintain correct termination behavior. But because both the MetaML implementation and SML are CBV, we chose the definitions presented in this chapter.

[8] Again, in a CBN setting, the definition of the fixed-point operator would have been slightly simpler.

[9] Y is the name usually used for the fixed-point combinator, of which this function is an instance.

4.2.6 Expressing and Executing the h Function

The h1 function of Section 4.1.1 can be expressed in our language. To demonstrate this, we will first show how we can construct an environment that contains all the necessary ingredients for expressing this function, and then we will show how each use of a non-λ construct can be replaced by the use of a constant from this environment.

All the meta-level concepts presented above can be introduced directly into the object-language by including them in the initial environment:

```
val env1 = ext (ext (ext (ext (ext env0 "+" plus) "-" minus) "*" times) "Z" ifzero) "Y" recur.
```

Given this extended environment, we can express and evaluate terms such as the h function discussed above. The interesting part of this task is encoding the term into our representation of terms. In a complete interpreter, this encoding is performed by a parser. Here we do it by clarifying the mechanics of the interpreter. The original declaration was:

```
fun h1 n z = if n=0 then z else (fn x ⇒ (h1 (n-1) (x+z))) n.
```

To avoid the need for tuples in our language, we minimize the parameters of the recursive function declaration to the one that the recursion really depends on:

```
fun h1 n = fn z ⇒ if n=0 then z else (fn x ⇒ (h1 (n-1) (x+z))) n.
```

Using the Y function we turn recursion into lambda abstractions, applications, and a conditional:

```
val h1 = Y (fn h1' ⇒ fn n ⇒ fn z ⇒ if n=0 then z else
                              (fn x ⇒ (h1' (n-1) (x+z))) n).
```

Similarly, using the function Z, we turn the if-statement used in the function h1 into lambda-abstractions and applications:

```
val h1 = Y (fn h1' ⇒ fn n ⇒ fn z ⇒
                                  Z n (fn _ ⇒ z) (fn n ⇒ (fn x ⇒ (h1' (n-1) (x+z))) n)).
```

Finally, we encode this SML program into the datatype for representing the syntax of our toy language:

```
- val H1 = EA (EV "Y", EL ("h1'", ...));

- val IT = EA (EA (H1, EI 3), EI 4);

- val answer = ev0 env1 IT;
val answer = VI 10 : value.
```

where the final result represents the integer value 10.

4.3 Extending the Simple Implementation with Staging Annotations

The first two implementations of MetaML were based on an interpreter similar to the one presented above. The hypothesis was that we can introduce staging constructs into the language without adding too much complexity to the interpreter. In this section, we will discuss what is involved in extending this simple implementation with MetaML's staging annotations. We will show that a number of subtle problems arise, some of which are not easily addressed. The significance of these difficulties comes from the fact that extending an existing implementation of a programming language to include staging annotations is prototypical of how multi-stage languages are developed in practice (See for example [3, 26, 84, 97].) As such, the existence of these problems underlines the potential dangers of this widespread practice and the importance of studying the formal semantics of multi-stage languages.

4.3.1 Values and Expressions

To extend the interpreter to deal with a multi-stage language such as λ-M of Section 4.1.2, we must enrich the expression datatype exp with new variants to represent the constructs Brackets, Escape, Run, and cross-stage persistent constants. We must also enrich the value datatype value with a variant to represent code fragments. The definitions of exp and value will now be mutually recursive because we can have expressions in values and values in expressions:

```
datatype exp       = EI of int              (* Integers *)
                   | EA of exp * exp        (* Applications *)
                   | EL of string * exp     (* Lambda-abstractions *)
                   | EV of string           (* Variables *)
                   | EB of exp              (* Brackets *)
                   | ES of exp              (* Escape *)
                   | ER of exp              (* Run *)
                   | EC of value            (* Cross-stage constants *)

and      value     = VI of int              (* Integer Values *)
                   | VF of value → value    (* Function Values *)
                   | VC of exp;             (* Expressions (object-code) *).
```

We now have all the infrastructure necessary for defining an interpretation function.

4.3.2 Environments and Bound Variable Renaming

To address the renaming problem described in the Section 4.1.1, the extended interpreter must explicitly rename bound variables in object-programs. One way of performing the renaming is to use the environment to carry the new names that we assign to object-level bound variables at run-time. This technique is used in the interpreter of Jones *et al.* [40]. To perform this renaming, we use a stateful function NextVar that computes a "new name" for a variable:

```
val ctr = ref 0;

fun NextVar s = let val _ = (ctr := (!ctr +1)) in s^(Integer.makestring (!ctr)) end.
```

To distinguish between the instances when we are using the environment to carry around real values and when we are simply using it to implement renaming, we generalize our interpreter by weakening the type of environments. One way of achieving this weakening is by changing the type of the environment to string → exp instead of string → value. This change in types is not too drastic, as our datatypes exp and val are mutually recursive, and are essentially of the same expressivity. Using exp instead of val simply makes our interpreter more concise. Now, normal bindings of a name x to a value v will be replaced by bindings to the expression EC v.

For reasons that will become apparent in the next section, it will also be useful to change the default behavior of the empty environment on variables. In particular, the empty environment will now simply return a variable expression for any variable:

```
fun env0 x = EV x.
```

Now, we extend our interpretation function to deal with Brackets, Run, and cross-stage constants as follows:

```
fun ev1 env e =
(case e of
   EI i ⇒ VI i
 | EA (e1,e2) ⇒ (case (ev1 env e1, ev1 env e2) of (VF f, v) ⇒ f v)
 | EL (x,e1) ⇒ VF (fn v ⇒ ev1 (ext env x (EC v)) e1)
 | EV x ⇒ (case (env x) of EC v ⇒ v)
 | EB e1 ⇒ VC (eb1 1 env e1)
 | ER e1 ⇒ (case (ev1 env e1) of VC e2 ⇒ ev1 env0 e2)
 | EC v ⇒ v).
```

The first four cases are essentially the same as before. The only difference is that the variable case requires the variable to be associated with a constant in the environment. If the variable is not associated with a constant, a run-time error occurs.

We postpone explaining the case of Brackets momentarily, as their interpretation uses the rebuilding function eb1. Note also that there is no case for Escapes, because Escapes are not treated by the evaluation function, but rather, the rebuilding function.

The case of Run is interpreted by first interpreting the argument to get a code fragment and then re-interpreting this code fragment in the empty environment. Cross-stage persistent constants are interpreted as simply the value they carry.

The case of Brackets makes a call to a *rebuilding* function eb1. The primary role of rebuilding is to evaluate any Escapes at level 1. Evaluating Escapes is performed by traversing the inside of a Bracketed expression until a level 1 Escape is encountered, at which point, the Escaped expression is evaluated, and the result of this evaluation is "spliced-into" the current context. The rebuilding function eb1 is defined in mutual recursion with ev1:

```
and eb1 n env e =
(case e of
   EI i ⇒ EI i
 | EA (e1,e2) ⇒ EA (eb1 n env e1, eb1 n env e2)
 | EL (x,e1) ⇒ let val x' = NextVar x in EL (x', eb1 n (ext env x (EV (x'))) e1) end
 | EV y ⇒ env y
 | EB e1 ⇒ EB (eb1 (n+1) env e1)
 | ES e1 ⇒ (if n=1 then (case (ev1 env e1) of VC e ⇒ e) else ES (eb1 (n-1) env e1))
 | ER e1 ⇒ ER (eb1 n env e1)
 | EC v ⇒ EC v).
```

The parameter n is the level of the expression being rebuilt. Rebuilding integer expressions leaves them unchanged. Rebuilding all other expressions (except Escape at level 1) involves rebuilding the subexpressions with an appropriate correction to the level parameter. For example, rebuilding the subexpression of a Bracket expression requires adding one to the level. Rebuilding the subexpressions of an Escape at level higher than 1 requires subtracting one from the level. The level parameter is never changed otherwise. Because of the way we chose to type our environment, rebuilding variables works in a simple way: If the environment is carrying a new name, then it simply replaces the old one. If the environment is carrying a "real" value, then it is already equipped with a surround EC, which is exactly the form we need to have a value so that it can replace a variable in an expression.

Rebuilding a lambda term involves producing a new name for the bound variable and extending the environment with this new name so that the old name is replaced with the new one when it is encountered.

4.3.3 Introducing Constants

We can use extended initial environments in essentially the same manner as before. To accommodate the change in the type of environments, we modify the way we extend the environment slightly:

```
val env1 = ext (ext (ext (ext (ext env0
            "+"  (EC plus))
            "-"  (EC minus))
            "*"  (EC times))
            "Z"  (EC ifzero))
            "Y"  (EC recur).
```

Evaluating the encoding of h1 4 4 produces exactly the same result as before.

4.3.4 Expressing and Executing the Staged h Function

Now we can express the staged h function in our toy language:

```
fun h2 n z = if n=0 then z else ⟨(fn x ⇒ ˜(h2 (n-1) ⟨ x+(  z⟩) ) 1⟩.
```

We modify this definition slightly to make it amenable to encoding into exp:

```
val h2 = Y (fn h2' ⇒ fn n ⇒ fn z ⇒
                    Z n (fn _ ⇒ z)
                        (fn n ⇒ ⟨(fn x ⇒ ˜(h2' (n-1) ⟨x+ ˜z⟩)) n ⟩)).
```

Now the definition contains only applications, abstraction, variables, and integers, all of which we can express in the syntax of our language. It also contains the free variables Y, Z, and +, all of which we already know how to include in the initial environment. The encoding of this function is essentially:

```
- val H2 = EA (EV "Y", EL ("h2'", ... ));

- val IT = EA (EA (H2, EI 3), EB (EI 4));

- val answer = ev1 env1 IT;
val it = VC (EA (EL ("x1",EA (EL ("x2",EA ...),EI 2)),EI 3)) : value.
```

corresponding to the result we expected:

$\langle$(fn x_1 ⇒ ((fn x_2 ⇒ ((fn x_3 ⇒ x_3+(x_2+(x_1+4))) 1)) 2)) 3$\rangle$.

4.4 New Concerns

One of our hypotheses is that studying the formal semantics of MetaML can improve our understanding of this language, and in turn, allow us to improve implementations of MetaML. It is difficult to explain accurately how studying the formal semantics of a language enhances our understanding of how the language can and should be implemented. In some cases, studying the formal semantics helps us solve problems and in other cases, it helps us identify problems. Developing the type systems presented in the rest of this dissertation are examples of where our study helped us solve a problem. It has also been the case that our scrutiny of the MetaML implementation has brought to light previously unknown problems. This section describes two new anomalies identified in the course of our study, and which constitute ample justification for the pursuit of a rigorous semantics of MetaML.

4.4.1 Scoping, Cross-Stage Persistence, and Hidden Free-Variables

Cross-stage persistence gets in the way of the bound-variable renaming strategy used in the interpreter above. In particular, ev1 performs renaming using the rebuilding function itself. This usage of rebuilding is a limited kind of substitution, namely one variable name for another. But cross-stage persistent constants can be functions. Because we cannot "traverse" functions, rebuilding does not "go inside" cross-stage persistent constants. Thus, the renaming strategy fails. We call this the *hidden free-variable* problem.

How did the Formal Semantics Help? (I) Identifying this failure came as a direct result of studying the formal semantics of MetaML. We observed this problem while developing an early version of the reduction semantics presented in Chapter 6. We may have run into a similar problem while testing the existing implementation, but when examining an implementation that does not (yet) have an associated formal specification, it is hard to distinguish between what is merely an implementation mistake and what is a more fundamental problem with our understanding of how the language *should* be implemented. The reduction semantics allowed us to see clearly that using functions to represent the values of functions can make performing the renaming difficult. Indeed, shortly after making this observation, we synthesised the following concrete counter-example that caused the MetaML implementation to exhibit anomalous behavior:

```
- val puzzle = ⟨fn a ⇒ ~((fn x ⇒ ⟨x⟩) (fn x ⇒ ⟨a⟩)) 0⟩;
- (run puzzle) 5.
```

As we will see in the rest of the dissertation, the term is well typed, and under fairly simple specifications of MetaML semantics, should evaluate to $\langle 5 \rangle$. In the implementations however, we get a different result.

To see the problem, note that the Escaped computation constructs a code fragment containing a cross-stage persistent function, which itself contains a dynamic variable in its body. Thus, the code fragment we get back to "splice" into context contains a cross-stage persistent constant carrying a value that itself contains a free object-level variable. The conceptual error in the design of ev1 and eb1 was that we assumed that values carried by cross-stage persistent constants are "fully developed" in the sense that they do not need further processing.

The function ev1 is a faithful model of the implementations, and so we will use ev1 to illustrate puzzle problem. The application of the result of Running the puzzle on 5 presented above can be encoded and evaluated as follows:

```
- val PUZZLE = (EA(ER(EB(EL("a",EA(ES(EA(EL("x",EB(EV "x")),
                    EL("x",EB (EV "a")))), EI 0)))), EI 5));

- ev1 env0 PUZZLE
val it = VC (EV "a1") : value.
```

The obscure result represents $\langle a_1 \rangle$: A bound variable has escaped from the scope of its binding abstraction. This anomaly should not occur in a statically scoped language, where all occurrences of a variable should remain syntactically inside the scope of their binder.

How did the Formal Semantics Help? (II) In the next section, we present a current proposal for modifying the implementation to deal with this problem. This proposal is based primarily on an operational view of how λ-M implementation presented in this chapter can be corrected. Both this problem and the proposed solution were identified *during* the development of the formal semantics of MetaML. We present our proposal not just because we expect it to solve the problem, but also because the sheer complexity of the solution is our final argument for the need to study the formal semantics MetaML and multi-stage programming languages. The puzzle problem does *not* arise at all if we simply use a standard (formal) notion of substitution, as is done in all the formulations of MetaML semantics in the rest of part II.

4.4.2 Typing and the Failure of Cross-Stage Safety

As was the case with the scoping problem above, we ran into a number of obscure "bugs" involving the use of the Run construct. We documented these as simply being implementation mistakes. It was not until we presented a version of a formal soundness proof of MetaML's type system that Rowan Davies pointed out that soundness of the type system does not hold. Rowan presented the following expression:

$$\langle \mathsf{fn}\ \mathsf{x} \Rightarrow \tilde{}(\mathsf{run}\ (\mathsf{run}\ \langle\langle \mathsf{x} \rangle\rangle))\rangle$$

This expression is well-typed under the type systems presented in Section 4.1.2, but it is not well-behaved. Encoding and executing this expression in the implementation presented above leads to a run-time error.

The root cause of the problem presented in this subsection is that Run dynamically changes the level of a term. In the expression above, x starts off at level 2. When the expression is executed, the level of x drops to 0. An attempt is therefore made to evaluate x by looking up an associated value in the environment, but no such value is available yet.

The problem presented in this subsection is a result of a flaw in the type system rather than in the "implementation" of MetaML. In Chapter 5, we will present two solutions to this problem and explain why we prefer one of these solutions to the other.

4.5 Covers: A Plausible Solution to the Hidden Free-Variables Problem

Before we go on to describe more subtle reasons for studying the formal semantics of MetaML, we will consider a possible solution to the problem of hidden free-variables. We will see how the solution itself is quite complex, and even though the solution may seem plausible, formally verifying its correctness remains a non-trivial task.

Because cross-stage persistent constants carry "values", it may seem that rebuilding does not need to "go inside" cross-stage persistent constants. As we mentioned earlier, this fallacy arises because one generally expects "values" to be fully developed terms, thus requiring no further processing. This confusion results from the fact that cross-stage persistent constants carry (SML) values of type value. Unfortunately, not all things represented by our value datatype are *values in the sense of well-formed interpretations of* λ*-M terms.* The counter-example puzzle presented at the end of the previous subsection is evidence of this problem. The result of evaluating PUZZLE using ev1 can be represented in value, but it does not correspond to anything that we accept as a MetaML *value.*

As we hinted earlier, the problem with evaluating PUZZLE is that renaming does not go inside cross-stage persistent constants. Making renaming go inside cross-stage persistent constants is tricky because it is not obvious how we can rename free variables *inside function values* in the value datatype. The solution we propose here is based on what we have called a *cover*. Intuitively, a cover allows us to perform a substitution on a datatype containing functions. Alternatively, a cover can be viewed as a delayed environment. The essential idea is to perform substitution on non-function terms in the normal manner and then to *cover* functions by making the functions themselves apply the substitution to their own results whenever these results become available. This way, a free variable that has been eliminated by a substitution (or a renaming) should never be able to escape from the scope of its binding occurrence.

We define two mutually recursive functions to cover both expressions and values as follows:

```
fun CoverE env e =
(case e of
   EI i ⇒ EI i
 | EA (e1,e2) ⇒ EA (CoverE env e1, CoverE env e2)
 | EL (y,e1) ⇒ EL (y,CoverE env e1)
 | EV y ⇒ env y
 | EB e1 ⇒ EB (CoverE env e1)
 | ES e1 ⇒ ES (CoverE env e1)
 | ER e1 ⇒ ER (CoverE env e1)
 | EC v ⇒ EC (CoverV env v))
and CoverV env v =
(case v of
   VI i ⇒ VI i
 | VF f ⇒ VF ((CoverV env) o f)
 | VC e ⇒ VC (CoverE env e)).
```

We revised the definition of our interpreter as follows:

```
fun ev2 env e =
(case e of
   EI i ⇒ VI i
 | EA (e1,e2) ⇒ (case (ev2 env e1, ev2 env e2) of (VF f, v) ⇒ f v)
 | EL (x,e1) ⇒ VF (fn v ⇒ ev2 (ext env x (EC v)) e1)
 | EV x ⇒ (case (env x) of EC v ⇒ v)
 | EB e1 ⇒ VC (eb2 1 env e1)
 | ER e1 ⇒ (case (ev2 env e1) of VC e2 ⇒ ev2 env0 e2)
 | EC v ⇒ CoverV env v)
and eb2 n env e =
(case e of
   EI i ⇒ EI i
| EA (e1,e2) ⇒ EA (eb2 n env e1, eb2 n env e2)
| EL (x,e1) ⇒ let val x' = NextVar x in EL (x', eb2 n (ext env x (EV (x'))) e1) end
| EV y ⇒ env y
| EB e1 ⇒ EB (eb2 (n+1) env e1)
| ES e1 ⇒ (if n=1 then (case (ev2 env e1) of VC e ⇒ e) else ES (eb2 (n-1) env e1))
| ER e1 ⇒ ER (eb2 n env e1)
| EC v ⇒ EC (CoverV env v)).
```

The only changes to the evaluation and rebuilding functions are in the cases of cross-stage persistent constants: Cross-stage persistent constants are covered using the current environment before they are returned. Cross-stage persistent variables are covered even during rebuilding, to address the possibility of them being added to the environment and then moved under another dynamic lambda, which could then incorrectly capture a dynamic variable that originated from the cross-stage persistent constant.

Now, evaluating the term PUZZLE under this semantics produces the expected result:

```
- ev2 env0 PUZZLE;
val it = VC (EC (VI 5)) : value.
```

Finally, we point out that the accumulation of such wrappers can lead to unnecessary performance degradation, especially for cross-stage persistent constants which do not contain any code. This problem can be alleviated by postponing the application of covers until a code fragment is actually encountered. This idea is similar in spirit to what can be done with calculi of explicit substitutions [4]. As of yet, this optimization is still unexplored.

4.6 More Concerns

The concerns described in the last section are "bugs": they are instances where the implementation and the type system break. There are other more qualitative concerns that we have identified. We present two in this section:

1. The need for validating certain run-time optimizations of object-code, and
2. The seeming existence of interesting intrinsic properties that the MetaML code type might enjoy.

We revisit the first concern at the end of Chapter 6.

4.6.1 Optimization on Generated Code

While the interpretation presented above was considered sufficient for executing MetaML programs, it was known that the code generated by such programs would contain some superfluous computations. Not only can these superfluous computations make it more costly to execute the generated programs, but it can also make the code larger and hence harder for humans to understand. In this section, we explain the need for these optimizations.

Safe β Reduction It is not uncommon that executing a multi-stage program will result in the construction of many applications that we would like to eliminate. Consider the following example:

```
val g = ⟨fn x ⇒ x * 5⟩;
val h = ⟨fn x ⇒ (~g x) - 2⟩.
```

If we use the interpreter presented above, the declaration for `h` evaluates to `⟨fn d1 ⇒ ((fn d2 ⇒ d2 * 5) d1) - 2⟩`. But the MetaML implementation returns `⟨fn d1 ⇒ (d1 * 5) - 2⟩` because it attempts to perform a kind of *safe* β reduction whenever a piece is code is Escaped into another. Generally, a β reduction is *safe* if it does not affect semantics properties, such as termination[10]. There is one safe case which is particularly easy to recognize: An application of a lambda-abstraction to a constant or a variable. We would like to know that such an application can always be reduced symbolically without affecting termination. Furthermore, restricting this optimization to the cases when the argument is a small constant or a variable allows us to avoid the possibility of code explosion.

This rebuilding-time optimization can be easily incorporated into the interpretation by modifying the application case in the rebuilding function `eb2`:

```
...
| EA (e1,e2) ⇒ case (eb2 n env e1, eb2 n env e2) of
                    (EL (x,e3), EI i) ⇒ eb2 n e3 (ext env0 x (EI 1))
                  | (EL (x,e3), EV x) ⇒ eb2 n e3 (ext env0 x (EV x))
                  | (e4,e5) ⇒ EA (e4,e5)
... .
```

This optimization requires that the β rule is expected to hold at all levels. Verifying this claim is not trivial. In fact, without a simple formal semantics for MetaML, this claim is practically impossible to verify. We return to the issue of β in Chapter 6.

[10] Respecting termination behavior is sufficient if the only effect in the language is termination. In richer languages, a safe β reduction may also need to respect other semantic properties.

Safe Rebuilding Rebuilding expressions in a multi-level language can be more costly than necessary. The following example is somewhat unnatural, but it allows us to exhibit a behavior that can arise in more "natural" programs:

$\mathsf{val\ a} = \langle\langle 5\rangle\rangle;\ \mathsf{val\ b} = \langle\langle\tilde{}\,\tilde{}\,\mathsf{a}\rangle\rangle.$

If we use the interpreter presented above, the declaration for b evaluates to $\langle\langle\tilde{}\langle 5\rangle\rangle\rangle$. Computing this result involved rebuilding $\langle 5\rangle$ and, in turn, rebuilding 5. Then, Running b would involve the rebuilding of 5 again. Our concern is that the last rebuilding is redundant because we know that rebuilding has already been performed before. If in place of 5 we had a larger expression, the cost of this redundancy could be substantial.

The MetaML implementation alleviates this problem by checking the result of rebuilding inside an Escape at levels greater than 1. If the result of rebuilding inside an Escape was a Bracketed expression, the Escape and the Brackets are eliminated, and the result is just the expression itself. This optimization would be correct if $\tilde{}\,\langle e\rangle$ were always equal to e. But again, this equality is not obvious, and cannot be supported or refuted without a rigourous definition of the semantics of MetaML. Furthermore, such a semantics must be fairly simple for any proofs to be practical and reliable. Again, our need to determine whether this optimization was sound or not is further motivation for seeking an *equational theory* for MetaML. We will outline an elementary equational theory at the end of the Chapter 6 which would support this optimization.

Again, this optimization can be incorporated into the rebuilding function by changing the Escape case of eb2 as follows:

```
...
| ES e1 ⇒ (if n=1 then (case (ev2 env e1) of VC e ⇒ e)
                 else case (eb2 (n-1) env e1) of
                        EB e2 ⇒ e2
                      | e3 ⇒ ES e3
... .
```

Finally, because the two optimization described in this subsection eliminate some redices that the user might expect to see in the generated code, these optimization could, in principle, make it hard to understand *why* a particular program was generated. In our experience, the resulting smaller, simpler programs have been easier to understand and seemed to make the optimizations worthwhile.

4.6.2 A Conjecture on an Isomorphism

All great truths begin as blasphemies.

George Bernard Shaw

We conclude this chapter by describing a simple yet controversial observation that we made shortly after we began *programming* in MetaML[11]. The purpose of this section is not only to present this observation, but to emphasize that programming in MetaML has illuminated the way for deep insights into the nature of multi-stage programming.

We say that there is an *isomorphism* between two types when there is a pair of functions f and g that go back and fourth between the two types and the composition of the two functions is equal to identity (See for example Di Cosmo [25].) More precisely, we are referring to the situation where two terms f and g represent functions, and the representation of their compositions is provably equal to $\lambda x.x$ in an equational theory[12]. A simple example of two isomorphic types are 'a * 'b and 'b * 'a. In this example, the pair of functions f and g are identical and are fn (x,y) ⇒ (y,x). While working with the implementation of MetaML, we began to wonder if the functions back and forth (of Section 2.6) are two such functions. They are not. Let us recall the definition of the two functions:

```
fun back f = ⟨fn x ⇒ ˜(f ⟨x⟩)⟩; (* : (⟨ 'a⟩ → ⟨ 'b⟩) → ⟨ 'a →  'b⟩ *)
fun forth f x = ⟨˜f ˜x⟩;          (* : ⟨ 'a →  'b⟩ → ⟨ 'a⟩ → ⟨ 'b⟩ *).
```

The reason these function do not form an isomorphism is that if back is applied to a non-terminating function, it fails to terminate (instead of returning the code of a non-terminating function). But at the same time, functions that take a code fragment and diverge immediately thereafter are not likely to be useful functions. In particular, "code fragments" in MetaML are *not* an inductive structure: The programmer cannot take apart code fragments or "look inside" them in any way other than by Running them[13]. It is therefore not clear why a *useful* function that takes such a code fragment would fail to terminate. Thus, we continued to search for a counter-example, that

[11] This section may be safely skipped by the reader *not* interested in the details of the formal semantics of MetaML. The section requires familiarity with the formal notion of an equational theory.

[12] It is important to emphasize that we are taking an equational-theoretic view, and not a view where we are concerned with domain-theoretic interpretations of the types. The latter view is interesting, but is not our present subject.

[13] Close inspection of the interpreter presented in this chapter will reveal that the environment can indeed be enriched with constants that would allow us to take apart a piece of code. This possibility is an artifact of the implementation, and not what the language studied in this dissertation is *intended* to do. The fragile nature of the distinction between what is implemented (and is therefore formal) and what is intended (and is therefore possibly still informal) made it harder for us to make the argument presented in this subsection. This difficulty was therefore further incentive to seek alternative (stronger) formulations of MetaML's semantics. We present instances of such semantics in the rest of Part II.

is, a *useful* MetaML function for which the composition of the two functions back and forth is not identity. To date, we have not found such a function.

Thus, we conjectured that back and forth form an isomorphism between two *interesting* subsets of the types $\langle$ 'a$\rangle \to \langle$ 'b$\rangle$ and $\langle$ 'a $\to$ 'b$\rangle$. These subsets must exclude, for example, non-terminating functions in the type $\langle$ 'a$\rangle \to \langle$ 'b$\rangle$. This choice is not too restrictive, because such functions seem generally uninteresting (no useful induction can be performed on a $\langle \_ \rangle$ type).

Finally, Danvy *et al.* have observed that two-level η-expansions can be used for binding-time improvement [20, 21]. The two functions, back and forth are closely related to two-level η-expansions. The application of multi-level expansions for improving staged programs has not been studied explicitly in this dissertation, and remains an important open question. The relation between our two functions and two-level η-expansion has been further motivation for us to validate our conjecture on type isomorphisms.

Chapter 5

Big-Step Semantics, Type Systems, and Closedness

The art of research is the ability to look at the details, and see the passion.

Daryl Zero, *The Zero Effect*

This chapter presents a formal semantics for MetaML in the big-step style, and a type system that we have proven to be type-safe with respect to the big-step semantics. The resulting language, albeit useful as is, has an expressivity limitation. We show how this shortcoming can be overcome by explicating the notion of a Closed value at the level of types. On these grounds, we propose that MetaML be extended with a new type for Closed values, and present a big-step semantics and a sound type system for the proposal.

This chapter represents the state of the art in (untyped) semantics and type systems for multi-stage programming languages.

5.1 A Big-Step Semantics for CBV and CBN λ

Formalizing a *big-step semantics* (see for example Gunter [33]) allows us to specify the semantics as a function that goes directly from expressions to values. We begin by reviewing the big-step semantics for the λ language.

Recall from Chapter 4 that syntax for the λ language is as follows:

$$e \in E := i \mid x \mid \lambda x.e \mid e\,e.$$

The CBV big-step semantics for λ is specified by a partial function $\_ \hookrightarrow \_ : E \rightarrow E$, where E is the set of λ expressions[1]:

$$\frac{}{i \hookrightarrow i}\,\text{Int} \qquad \frac{}{\lambda x.e \hookrightarrow \lambda x.e}\,\text{Lam} \qquad \frac{e_1 \hookrightarrow \lambda x.e \quad e_2 \hookrightarrow e_3 \quad e[x := e_3] \hookrightarrow e_4}{e_1\ e_2 \hookrightarrow e_4}\,\text{App.}$$

Note that there are terms for which none of the rules in this semantics can apply (either because they get "stuck" or the semantics "goes into an infinite loop").

The rule for integers says that they evaluate to themselves. The rule for lambda-abstractions says that they too evaluate to themselves. The rule for applications says that they are evaluated by evaluating the operator to get a lambda-abstraction, substituting the result of evaluating the operand into the body of the lambda-abstraction, and evaluating the result of the substitution. The definition of substitution is standard and is denoted by $e_1[x := e_2]$ for the capture-free substitution[2] of e_2 for the free occurrences of x in e_1. This semantics is a partial function associating at most one unique value to any expression in its domain.

Note that there is no need for an environment that keeps track of bindings of variables: whenever a value is available for a variable, we immediately substitute the value for the variable. This substitution is performed in the rule for applications. It is possible to implement the λ language directly by mimicking the big-step semantics. We should point out, however, that a direct implementation based on this big-step semantics would be somewhat inefficient, as every application would require a traversal of the body of the lambda-abstraction. Most realistic implementation do not involve traversing terms at run-time to perform substitution, and thus, are more similar in spirit to the simple interpreter discussed in Chapter 4.

The Closedness Assumption The semantics presented above for the λ language is fairly standard, but it contains an important assumption that will be violated when we extend the language to a multi-level one. In particular, the big-step semantics above has no rule for evaluating variables. The key observation is that evaluating a *closed* λ term using this big-step semantics does not involve evaluating open sub-terms. This claim can be established as a property of the derivation tree induced by this definition of the semantics. The proof proceeds by induction on the height of the derivation: The claim is true in the base cases of integers and lambda-abstractions, and it

[1] Typically, such a semantics is defined only for closed terms. We do not impose this restriction on our semantics.

[2] Capture-free substitution means that no free variables of e_2 are captured by bound variables in e_1.

is true by induction in the case of applications. In the case of application, we also need to have established that evaluating a closed expression returns a closed expression.

5.2 A Big-Step Semantics for CBV λ-M

Recall from Chapter 4 that the terms of λ-M are:

$$e \in E := i \mid x \mid \lambda x.e \mid e\, e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e.$$

To define the big-step semantics, we employ a finer classification of expressions. For example, the evaluation of a term $\tilde{}e$ does not interest us because Escapes should not occur at top level. Thus, we introduce *expression families*[3]:

$$\begin{array}{lll} e^0 & \in E^0 & := x \mid \lambda x.e^0 \mid e^0\, e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+} & \in E^{n+} & := x \mid \lambda x.e^{n+} \mid e^{n+}\, e^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}e^n \mid \mathsf{run}\ e^{n+}. \end{array}$$

Lemma 5.2.1 (Basic Properties of Expressions Families) $\forall n \in \mathbb{N}$.

1. $E^n \subseteq E$
2. $E^n \subseteq E^{n+}$
3. $\forall e_1 \in E^n, e_2 \in E^0.\, e_1[x := e_2] \in E^n$

Proof. All parts of this lemma are proven by easy inductions.

We illustrate the proof of the first part of this lemma. We prove that:

$$\forall n \in \mathbb{N}.\, \forall e \in E^n.\, e \in E.$$

The proof proceeds by induction on the derivation of $e \in E^n$. If $e \equiv x$ then $x \in E$ by definition of $\in E$. If $e \equiv e_1\, e_2$ then by the definition of $\in E^n$ we know that $e_1, e_2 \in E^n$. By the induction hypothesis, we have $e_1, e_2 \in E$. By the definition of $\in E$ we have $e_1\, e_2 \in E$. The treatment of the rest cases proceeds in the same manner.

The second and the third parts are similar. The third part is by induction on the derivation of $e_1 \in E^n$. $\square$

[3] This presentation of the sets of expressions and values is a slight abuse of notation. The definition of level annotated terms is "essentially" BNF in that it defines a set of terms by simple induction. Technically, this set is defined by induction on the height of a set membership judgment $e \in E^n$, and properties of this set are established by induction on the height of the derivation of this judgment. This can be expressed in more traditional notation as follows:

$$\frac{}{x \in E^n} \quad \frac{e \in E^n}{\lambda x.e \in E^n} \quad \frac{e_1, e_2 \in E^n}{e_1\, e_2 \in E^n} \quad \frac{e \in E^{n+}}{\langle e \rangle \in E^n} \quad \frac{e \in E^n}{\tilde{}e \in E^{n+}} \quad \frac{e \in E^n}{\mathsf{run}\ e \in E^n}.$$

We will use the BNF notation as a shorthand for such definitions. The shorthand is especially convenient for defining the sets of *workable* and *stuck* terms presented in Chapter 6.

Syntax:

$$e \in E := i \mid x \mid e\,e \mid \lambda x.e \mid \langle e \rangle \mid {\sim}e \mid \mathsf{run}\; e$$

Big-Step Rules:

$$\frac{}{i \overset{n}{\hookrightarrow} i}\;\text{Int} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e}\;\text{Lam} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} e_3 \quad e[x := e_3] \overset{0}{\hookrightarrow} e_4}{e_1\; e_2 \overset{0}{\hookrightarrow} e_4}\;\text{App}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\; e_1 \overset{0}{\hookrightarrow} e_3}\;\text{Run} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x}\;\text{Var+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_3 \quad e_2 \overset{n+}{\hookrightarrow} e_4}{e_1\; e_2 \overset{n+}{\hookrightarrow} e_3\; e_4}\;\text{App+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2}\;\text{Lam+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}\;\text{Brk} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\mathsf{run}\; e_1 \overset{n+}{\hookrightarrow} \mathsf{run}\; e_2}\;\text{Run+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{{\sim}e_1 \overset{n++}{\hookrightarrow} {\sim}e_2}\;\text{Esc++} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{{\sim}e_1 \overset{1}{\hookrightarrow} e_2}\;\text{Esc}$$

Fig. 5.1. The (Coarse) CBV Big-Step Semantics for λ-M

The CBV big-step semantics for λ-M is specified by a partial function $\_ \overset{n}{\hookrightarrow} \_ : E^n \to E^n$. We proceed by first defining the *coarse* function $\_ \overset{n}{\hookrightarrow} \_ : E \to E$, and then show that we can restrict the type of this function to arrive at the *fine* function $\_ \overset{n}{\hookrightarrow} \_ : E^n \to E^n$. Figure 5.1 summarizes the coarse CBV big-step semantics for λ-M[4]. Taking n to be 0, we can see that the first three rules correspond to the rules of λ.

The rule for Run at level 0 says that an expression is Run by first evaluating it to get a Bracketed expression, and then evaluating the Bracketed expression. The rule for Brackets at level 0 says that they are evaluated by rebuilding the expression they surround at level 1: *Rebuilding*, or "evaluating at levels higher than 0", eliminates level 1 Escapes. Rebuilding is performed by traversing expressions while correctly keeping track of level. Thus rebuilding simply traverses a term until a level 1 Escape is encountered, at which point the evaluation function is invoked in the Esc rule. The Escaped expression must yield a Bracketed expression, and then the expression itself is returned.

[4] For regularity, we use $\_ \overset{0}{\hookrightarrow} \_$ instead $\_ \hookrightarrow \_$. This way, both evaluation and rebuilding as (described in Chapter 4) are treated as one partial function that takes a natural number as an extra argument. The extra argument can still be used to distinguish between evaluation (the extra argument is 0) and rebuilding (the extra argument is greater than zero).

An immediate benefit of having such a semantics is that it provides us with a formal way of finding what result an implementation of MetaML *should* return for a given expression. For example, it is easy to compute the result of evaluating the application of the result of Running puzzle to 5 (see Section 4.4.1):

$$(\mathsf{run}\ \langle \lambda a.\tilde{}((\lambda x.\langle x\rangle)\ (\lambda x.\langle a\rangle))\ 0\rangle)\ 5 \overset{0}{\hookrightarrow} \langle 5\rangle.$$

5.2.1 Basic Properties of Big-Step Semantics

Next we establish some properties of the operational semantics. Values are a subset of terms that denote the results of computations. Because of the relative nature of Brackets and Escapes, it is important to use a family of sets for values, indexed by the level of the term, rather than just one set. Values are defined as follows:

$$\begin{aligned} v^0 \in V^0 &:= \lambda x.e^0 \mid \langle v^1\rangle \\ v^1 \in V^1 &:= x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2\rangle \mid \mathsf{run}\ \ v^1 \\ v^{n++} \in V^{n++} &:= x \mid v^{n++}\ v^{n++} \mid \lambda x.v^{n++} \mid \langle v^{n+++}\rangle \mid \tilde{}v^{n+} \mid \mathsf{run}\ \ v^{n++} \end{aligned}$$

Intuitively, level 0 values are what we get as a result of evaluating a term at level 0, and level $n+$ values are what we get from rebuilding a term at level $n+$. Thus, the set of values has three important properties: First, a value at level 0 can be a lambda-abstraction or a Bracketed value, reflecting the fact that lambda-abstractions and terms representing code are both considered acceptable results from a computation. Second, values at level $n+$ can contain applications such as $\langle(\lambda y.y)\ (\lambda x.x)\rangle$, reflecting the fact that computations at these levels can be deferred. Finally, there are no level 1 Escapes in level 1 values, reflecting the fact that having such an Escape in a term would mean that evaluating the term has not yet been completed. Evaluation is not complete, for example, in terms like $\langle\tilde{}(f\ x)\rangle$.

The following lemma establishes a simple yet important property of λ-M:

Lemma 5.2.2 (Strong Value Reflection for Untyped Terms) $\forall n \in \mathbb{N}$.

$$V^{n+} = E^n.$$

Proof. By simple induction. □

The lemma has two parts: One saying that every element of in a set of (code) values is also an element of a set of expressions, and the other, saying the converse. As we mentioned in Chapter 1, both of these properties can be interpreted as positive qualities of a multi-level language. The first part tells us that every object-program (value) can be viewed as a meta-program, and the second part tells us that every meta-program can viewed as an object-program (value). Having established reflection, it is easy to verify that the big-step semantics at level n ($e \overset{n}{\hookrightarrow} v$) always returns a value $v \in V^n$:

Lemma 5.2.3 (Basic Properties of Big-Step Semantics) $\forall n \in \mathbb{N}$.

1. $V^n \subseteq V^{n+}$,
2. $\forall e, e' \in E^n. \, e \overset{n}{\hookrightarrow} e' \Longrightarrow e' \in V^n$.

Proof. Part 1 is by a simple induction on the derivation of $v \in V^n$ to prove that:

$$\forall n \in \mathbb{N}. \forall v \in V^n. \, v \in V^{n+}.$$

Part 2 is also a simple induction on the derivation of $e \overset{n}{\hookrightarrow} e'$. Reflection (Lemma 5.2.2) is needed in the case of Run. □

Corollary 5.2.4 (Level Preservation) $\forall n \in \mathbb{N}. \forall e_1 \in E^n. \forall e_2 \in E$.

$$e_1 \overset{n}{\hookrightarrow} e_2 \Longrightarrow e_2 \in E^n.$$

Proof. Noting that $V^0 \subseteq E^0$ and $V^{n+} = E^n \subseteq E^{n+}$, this result is immediate from the previous lemma. □

Remark 5.2.5 (Fine Big-Step Function) *In the rest of the dissertation, we will only be concerned with the fine big-step semantic function* $\_ \overset{n}{\hookrightarrow} \_ : E^n \to E^n$ *for* λ-*M. We will also refer to it simply as the big-step semantics.*

The Closedness Assumption Violated The semantics is standard in its structure, but note it has the unusual feature that it manipulates *open* terms. In particular, rebuilding goes "under lambda" in the rule Lam+, and Escape at level 1 re-invokes evaluation during rebuilding. Thus, even though a closed term such as $\langle \lambda x. \tilde{}\langle x \rangle \rangle$ evaluates to $\langle \lambda x.x \rangle$ (that is $\langle \lambda x. \tilde{}\langle x \rangle \rangle \overset{0}{\hookrightarrow} \langle \lambda x.x \rangle$) the derivation of this evaluation involves the sub-derivation $\langle x \rangle \overset{0}{\hookrightarrow} \langle x \rangle$ which itself is the evaluation of the open term $\langle x \rangle$.

5.3 A Basic Type System for MetaML

As in Chapter 4, we introduce a level-index into the typing judgment. The type judgment is thus extended to have the form $\Gamma \vdash^n e : \tau$ where $e \in E$, $\tau \in T$, $\Gamma \in D$, and $n \in N$. The judgement $\Gamma \vdash^n e : \tau$ is read *"e has type τ at level n under environment Γ"*. Again, it is important to note that the level n is part of the judgement and *not* part of the type. Figure 5.2 summarizes the λ-M type system that we study in this section.

Syntax:

$$e \in E := i \mid x \mid e\, e \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e$$

Types and Type Environments:

$$\begin{array}{l} \tau \in T := b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle \\ \Gamma \in D := [] \mid x : \tau^n ; \Gamma \end{array}$$

Operations on Type Environments:

$$\Gamma^+(x) \stackrel{\Delta}{=} \tau^{n+} \text{ where } \Gamma(x) = \tau^n$$

Type Rules:

$$\frac{}{\Gamma \vdash^n i : \mathsf{int}}\ \text{Int}$$

$$\frac{\Gamma(x) = \tau^{(n-m)}}{\Gamma \vdash^n x : \tau}\ \text{Var} \qquad \frac{x : \tau'^n ; \Gamma \vdash^n e : \tau}{\Gamma \vdash^n \lambda x.e : \tau' \to \tau}\ \text{Lam} \qquad \frac{\Gamma \vdash^n e_2 : \tau' \quad \Gamma \vdash^n e_1 : \tau' \to \tau}{\Gamma \vdash^n e_1\, e_2 : \tau}\ \text{App}$$

$$\frac{\Gamma \vdash^{n+} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\ \text{Brk} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+} \tilde{}e : \tau}\ \text{Esc} \qquad \frac{\Gamma^+ \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \mathsf{run}\ e : \tau}\ \text{Run}$$

Fig. 5.2. The λ-M Type System

5.3.1 Basic Properties of Type System

We will now present some technical lemmas for λ-M that are needed to establish type-safety. These lemmas establish the basic properties of the type system that are needed for establishing more interesting properties:

Lemma 5.3.1 (Weakening) $\forall n, m \in \mathbb{N}, e \in E, \Gamma \in D, \tau, \sigma \in T.$

$$\Gamma \vdash^n e : \tau \wedge x \notin dom(\Gamma) \wedge x \notin \mathrm{FV}(e) \Longrightarrow \Gamma ; x : \sigma^m \vdash^n e : \tau.$$

Proof. By induction on the structure of the derivation $\Gamma \vdash^n e : \tau$. □

A key lemma needed for proving type-safety of the system is the following:

Lemma 5.3.2 (Persistence of Typability) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T. \forall e \in E^n.$

$$\Gamma \vdash^n e : \tau \Longleftrightarrow \Gamma^+ \vdash^{n+} e : \tau.$$

Proof. Each direction of the implication is by a simple induction on the height of the typing judgement in the assumption. The forward direction is routine. The backward direction is also straightforward, but the case of Escape is notable. In particular, we only prove that $\Gamma^+ \vdash^{n++} \tilde{}e : \tau$ implies $\Gamma \vdash^{n+} \tilde{}e : \tau$, because $\tilde{}e$ is only in E^{n+} and not in E^0. □

Notation 5.3.3 *We write Γ_1, Γ_2 for the environment determined by the (disjoint) union of the two environments $\Gamma_1 \in D$ and $\Gamma_2 \in D$.*

The following lemma says that any term remains typable when we reduce the level of any of the variables in the environment under which the term is typable.

Lemma 5.3.4 (Co-Promotion) $\forall n \in \mathbb{N}, e \in E, \Gamma_1, \Gamma_2 \in D, \tau \in T.$

$$\Gamma_1, \Gamma_2^{+} \vdash^n e : \tau \Longrightarrow \Gamma_1, \Gamma_2 \vdash^n e : \tau.$$

Proof. By a simple induction on the derivation of $\Gamma_1, \Gamma_2 \vdash^n e : \tau$. □

We will now introduce a simple notation that we have found useful for writing the kind of proofs used in this dissertation.

Notation 5.3.5 (Proofs on Derivations) *Whenever convenient, proofs will be laid out in 2-dimensions to reflect that we are relating one derivation tree to another. Symbols such as ${}_R\Uparrow$, $\overset{R}{\Longleftrightarrow}$, and ${}_R\Downarrow$ will be used for (one- or two-way) implications, where R is the list of rules used to achieve this implication. For rules, we instantiate R with $\vdash$ when we have applied the rules of the type system, $\pm$ for arithmetic, and IH for use of the induction hypothesis.*

Proofs start from the left and proceed either up, down, or right. We go up or down depending on the normal orientation of the particular rule that is being used in an implication.

Horizontal implications are aligned with their precedents and antecedents.

Lemma 5.3.6 (Substitution) $\forall n, m \in \mathbb{N}, \Gamma_1, \Gamma_2 \in D, \tau, \sigma \in T. \forall e_1 \in E^n, e_2 \in E^m.$

$$\Gamma_1, \Gamma_2; x : \sigma^m \vdash^n e_1 : \tau \wedge \Gamma_2 \vdash^m e_2 : \sigma \Longrightarrow \Gamma_1, \Gamma_2 \vdash^n e_1[x := e_2] : \tau.$$

Proof. By induction on the derivation of the judgement $\Gamma_1, \Gamma_2; x : \sigma^m \vdash^n e_1 : \tau$.

- Variables I: $e_1 \equiv y \not\equiv x$, we already know $\Gamma_1, \Gamma_2; x : \sigma^m \vdash^n e_2 : \tau$ and by weakening we have $\Gamma_1, \Gamma_2 \vdash^n e_2 : \tau$.
- Variables II: $e_1 \equiv x$, then from the typing rule for variables we know that $n = m + m'$ and that $\sigma \equiv \tau$, for some natural number m'. We already know $\Gamma_1 \vdash^m e_2 : \sigma$. By weakening we have $\Gamma_1, \Gamma_2 \vdash^m e_2 : \sigma$. By persistence of typability we have $\Gamma_1^{m'}, \Gamma_2^{m'} \vdash^{m+m'} e_2 : \sigma$. By m' applications of co-promotion we have $\Gamma_1, \Gamma_2 \vdash^{m+m'} e_2 : \sigma$ and we are done.
- Lambda-abstraction I: $e_1 \equiv \lambda y.e$, and $y \not\equiv x$. By Barendregt's convention, $y \notin \mathrm{FV}(e_2)$.

$$\vdash\Uparrow \frac{\Gamma_1, \Gamma_2; x : \sigma^m; y : {\tau_3}^n \vdash^n e : \tau_4}{\Gamma_1, \Gamma_2; x : \sigma^m \vdash^n \lambda y.e : \tau_3 \to \tau_4} \quad \overset{IH}{\Longrightarrow} \quad \frac{\Gamma_1, \Gamma_2; y : {\tau_3}^n \vdash^n e[x := e_2] : \tau_4}{\Gamma_1, \Gamma_2 \vdash^n \lambda y.e[x := e_2] : \tau_3 \to \tau_4} {}_{\vdash, :=}\Downarrow$$

- Lambda-abstraction II: $e_1 \equiv \lambda y.e$, and $y \equiv x$. By weakening, we have $\Gamma_1, \Gamma_2 \vdash^n \lambda y.e : \tau_3 \to \tau_4$ and we are done.

– Applications:

$$\vdash\Uparrow \frac{\begin{array}{l}\Gamma_1,\Gamma_2;x:\sigma^m \vdash^n e_3 : \tau_1 \to \tau \\ \Gamma_1,\Gamma_2;x:\sigma^m \vdash^n e_4 : \tau_1\end{array}}{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^n e_3\ e_4 : \tau} \begin{array}{c}\overset{IH}{\Longrightarrow} \\ \overset{IH}{\Longrightarrow} \\ \end{array} \frac{\begin{array}{l}\Gamma_1,\Gamma_2 \vdash^n e_3[x:=e_2] : \tau_1 \to \tau \\ \Gamma_1,\Gamma_2 \vdash^n e_4[x:=e_2] : \tau_1\end{array}}{\Gamma_1,\Gamma_2 \vdash^n (e_3\ e_4)[x:=e_2] : \tau} \vdash,:=\Downarrow$$

– Brackets:

$$\vdash\Uparrow \frac{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^{n+1} e : \tau}{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^n \langle e\rangle : \langle\tau\rangle} \overset{IH}{\Longrightarrow} \frac{\Gamma_1,\Gamma_2 \vdash^{n+1} e[x:=e_2] : \tau}{\Gamma_1,\Gamma_2 \vdash^n \langle e\rangle[x:=e_2] : \langle\tau\rangle} \vdash,:=\Downarrow$$

– Escapes:

$$\vdash\Uparrow \frac{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^{n} e : \langle\tau\rangle}{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^{n+1} \tilde{\ }e : \tau} \overset{IH}{\Longrightarrow} \frac{\Gamma_1,\Gamma_2 \vdash^{n} e[x:=e_2] : \langle\tau\rangle}{\Gamma_1,\Gamma_2 \vdash^{n+} \tilde{\ }e[x:=e_2] : \tau} \vdash,:=\Downarrow$$

– Run:

$$\vdash\Uparrow \frac{\Gamma_1^{+1},\Gamma_2^{+1};x:\sigma^{m+1} \vdash^{n} e : \langle\tau\rangle}{\Gamma_1,\Gamma_2;x:\sigma^m \vdash^{n} \mathsf{run}\ e : \tau} \overset{IH}{\Longrightarrow} \frac{\Gamma_1^{+1},\Gamma_2^{+1} \vdash^{n} e[x:=e_2] : \langle\tau\rangle}{\Gamma_1,\Gamma_2 \vdash^{n} \mathsf{run}\ e[x:=e_2] : \tau} \vdash,:=\Downarrow$$

□

5.4 Type-Safety

While the statement and proof of the Type-Safety lemma for λ-M will seem quite straightforward, we cannot over-emphasize the importance of defining carefully what is meant by type-safety for a particular language. We specify the occurrence of a run-time error by first augmenting the big-step semantics to return an error value err when an undesirable condition takes place. Thus, the type of the big-step semantic function becomes $\_ \overset{n}{\hookrightarrow} \_ : E^n \to (E^n \cup \{\mathsf{err}\})$. Second, we introduce an additional set of rules to propagate this value if it is returned by sub-computations. The first set of rules that must be added to the big-step semantics is as follows:

$$\frac{}{x \overset{0}{\hookrightarrow} \mathsf{err}}\ \text{VarErr} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_3 \quad e_3 \not\equiv \lambda x.e}{e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}}\ \text{AppErr}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \langle e\rangle}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}}\ \text{SRnErr} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \langle e\rangle}{\tilde{\ }e_1 \overset{1}{\hookrightarrow} \mathsf{err}}\ \text{Esc1Err}.$$

We also add propagation rules that simply return an error if the result of a sub-computation is an error.

The most subtle of these error cases to identify has been the one for variables. In particular, it is very easy to forget to add this rule as an error-emitting case, and then it becomes very hard to see why it is still possible to "prove" incorrect type-safety theorems.

Finally, it is instructive to note that there is a slight redundancy in this typed treatment of λ-M:

Lemma 5.4.1 $\forall n \in \mathbb{N}, \forall e \in E, \Gamma \in D, \tau \in T.$

$$\Gamma \vdash^n e : \tau \Longrightarrow e \in E^n.$$

Proof. By a simple induction on the derivation of $\Gamma \vdash^n e : \tau$. □

Thus, if we are only interested in typed terms, we do not (strictly speaking) need the family classification on expressions. However, as we will see in Chapter 6, this classification is significant to the treatment of the untyped language.

Reflection for a typed multi-level language can be viewed as the existence of a correspondence between programs $\emptyset \vdash^0 e : \tau$ and program representations $\emptyset \vdash^0 \langle v \rangle : \langle \tau \rangle$. This property holds for λ-M. It is an instance of the following result:

Lemma 5.4.2 (Strong Value Reflection for Typed Terms) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T, v \in V^{n+}, e \in E^n.$

1. $\Gamma^+ \vdash^{n+} v : \tau \Longrightarrow \Gamma \vdash^n v : \tau,$
2. $\Gamma \vdash^n e : \tau \Longrightarrow (\Gamma^+ \vdash^{n+} e : \tau \wedge e \in V^{n+}).$

Proof. Part 1 is by induction on the derivation of $\Gamma^+ \vdash^{n+} v : \tau$, and case analysis on $v \in V^{n+}$. Part 2 is by induction on the derivation of $\Gamma \vdash^n e : \tau$. □

Lemma 5.4.3 (Basic Property of Augmented Semantics) $\forall n \in \mathbb{N}. \forall e, e' \in E^n.$

$$e \overset{n}{\hookrightarrow} e' \Longrightarrow e' \in (V^n \cup \{\mathsf{err}\}).$$

Proof. By a simple induction on the derivation of $e \overset{n}{\hookrightarrow} e'$. Reflection (Lemma 5.2.2) is needed in the case of Run. □

The Closedness Assumption Refined As we have pointed out earlier, an important feature in the λ language is that reductions always operate on closed terms. We have also pointed out that this assumption is violated in the big-step semantics of λ-M, because rebuilding goes "under lambda". A crucial observation allows us to continue the formal development of MetaML in the usual manner: All terms are closed with respect to variables bound at level 0. This observation dictates the general form of the statement of the Type Preservation Theorem:

Theorem 5.4.4 (Type Preservation for CBV λ-M) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T. e \in E^n, v \in V^n.$

$$\Gamma^+ \vdash^n e : \tau \wedge e \overset{n}{\hookrightarrow} v \Longrightarrow \Gamma^+ \vdash^n v : \tau.$$

Proof. By induction on the derivation of $e \overset{n}{\hookrightarrow} v$. The case for application uses Substitution. The case analysis proceeds as follows:

- Variables I: $n = 0$ is vacuous, because we can never derive the type judgment in this case.
- Variables II: $n > 0$ is trivial because $x \overset{n+}{\hookrightarrow} x$.
- Lambda-abstraction I: Interestingly, no induction is needed here. In particular, $\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e$. By definition, $\lambda x.e \in V^0$, and from the premise $\Gamma^+ \vdash^0 \lambda x.e : \tau$.
- Lambda-abstraction II: Straightforward induction:

$$\vdash\Uparrow\ \frac{\Gamma^+; x : \tau_1^n \vdash^n e : \tau}{\Gamma^+ \vdash^n \lambda x.e : \tau_1 \to \tau}\ \hookrightarrow\Uparrow\ \frac{e \overset{n+}{\hookrightarrow} e'}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.e'}\ \overset{IH}{\Longrightarrow}\ \frac{e' \in V^{n+}}{\lambda x.e' \in V^{n+}}\ v\Downarrow\ \frac{\Gamma^+; x : \tau_1^n \vdash^n e' : \tau}{\Gamma^+ \vdash^n \lambda x.e' : \tau_1 \to \tau}\ \vdash\Downarrow$$

Application I and Run I follow a more involved but similar pattern. The other cases are by straightforward induction.

- Applications I: First, we use the induction hypothesis (twice) , which gives us a result that we can use with the substitution lemma (:=):

$$\begin{array}{lll}
\Gamma^+ \vdash^n e_1 : \tau_1 \to \tau & e_1 \overset{n}{\hookrightarrow} \lambda x.e \quad \overset{IH}{\Longrightarrow} & \dfrac{\Gamma^+; x : \tau_1^n \vdash^n e : t}{\Gamma^+ \vdash^n \lambda x.e : \tau_1 \to \tau}\ \vdash\Uparrow \\
\vdash\Uparrow \dfrac{\Gamma^+ \vdash^n e_2 : \tau_1}{\Gamma^+ \vdash^n e_1\ e_2 : \tau} & e_2 \overset{n}{\hookrightarrow} e_2' \quad \overset{IH}{\Longrightarrow} & \dfrac{\Gamma^+ \vdash^n e_2' : \tau_1}{\Gamma^+ \vdash^n e[x := e_2'] : \tau}\ {:=}\Downarrow \\
 & \hookrightarrow\Uparrow \dfrac{e[x := e_2'] \overset{n}{\hookrightarrow} e'}{e \overset{n}{\hookrightarrow} e'} &
\end{array}$$

Note that we we are also using the judgment $\Gamma^+; x : \tau_1^n \vdash^n e : \tau$ when we apply the substitution lemma. Then, based on this information about $e[x := e_2]$ we apply the induction hypothesis for the third time to get $e' \in V^n$ and $\Gamma^+ \vdash^n e' : \tau$.
- Applications II:

$$\vdash\Uparrow\ \frac{\Gamma^+ \vdash^n e_1 : \tau_1 \to \tau \qquad \Gamma^+ \vdash^n e_2 : \tau_1}{\Gamma^+ \vdash^n e_1\ e_2 : \tau}\ \hookrightarrow\Uparrow\ \frac{e_1 \overset{n+}{\hookrightarrow} e_1' \qquad e_2 \overset{n+}{\hookrightarrow} e_2'}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_1'\ e_2'}\ \overset{IH}{\Longrightarrow}\ \frac{e_1' \in V^{n+} \qquad e_2' \in V^{n+}}{e_1'\ e_2' \in V^{n+}}\ v\Downarrow\ \frac{\Gamma^+ \vdash^n e_1' : \tau_1 \to \tau \qquad \Gamma^+ \vdash^n e_2' : \tau_1}{\Gamma^+ \vdash^n e_1'\ e_2' : \tau}\ \vdash\Downarrow$$

- Bracket:

$$\vdash\Uparrow\ \frac{\Gamma^+ \vdash^{n+} e : \tau}{\Gamma^+ \vdash^n \langle e \rangle : \langle \tau \rangle}\ \hookrightarrow\Uparrow\ \frac{e \overset{n+}{\hookrightarrow} e'}{\langle e \rangle \overset{n+}{\hookrightarrow} \langle e' \rangle}\ \overset{IH}{\Longrightarrow}\ \frac{e' \in V^n}{\langle e' \rangle \in V^{n+}}\ v\Downarrow\ \frac{\Gamma^+ \vdash^{n+} e' : \tau}{\Gamma^+ \vdash^n \langle e' \rangle : \langle \tau \rangle}\ \vdash\Downarrow$$

- Escape I:

$$\vdash\Uparrow\ \frac{\Gamma^+ \vdash^0 e : \langle \tau \rangle}{\Gamma^+ \vdash^1 \tilde{}\, e : \tau}\ \hookrightarrow\Uparrow\ \frac{e \overset{0}{\hookrightarrow} \langle e' \rangle \overset{IH}{\Longrightarrow} \dfrac{e' \in V^1}{\langle e' \rangle \in V^0}\ v\Uparrow\ \dfrac{\Gamma^{++} \vdash^1 e' : \tau}{\Gamma^{++} \vdash^0 \langle e' \rangle : \langle \tau \rangle}\ \vdash\Uparrow}{e \overset{0}{\hookrightarrow} e''}$$

- Escape II:

$$\vdash\Uparrow\ \frac{\Gamma^+ \vdash^n e : \langle \tau \rangle}{\Gamma^+ \vdash^{n+} \tilde{}\, e : \tau}\ \hookrightarrow\Uparrow\ \frac{e \overset{n+}{\hookrightarrow} e'}{\tilde{}\, e \overset{n+}{\hookrightarrow} \tilde{}\, e'}\ \overset{IH}{\Longrightarrow}\ \frac{e' \in V^n}{\tilde{}\, e' \in V^{n+}}\ v\Downarrow\ \frac{\Gamma^+ \vdash^n e' : \langle \tau \rangle}{\Gamma^+ \vdash^{n+} \tilde{}\, e' : \tau}\ \vdash\Downarrow$$

– Run I: Similar to application, except that we use persistence of typability. First, we apply the induction hypothesis once, then reconstruct the type judgment of the result:

$$\vdash\Uparrow\frac{\Gamma^{++}\vdash^0 e:\langle\tau\rangle}{\Gamma^{+}\vdash^0 \mathsf{run}\ e:\tau}\qquad e\overset{0}{\hookrightarrow}\langle e'\rangle \overset{IH}{\Longrightarrow} \frac{\Gamma^{++}\vdash^1 e':\tau}{\Gamma^{++}\vdash^0 \langle e'\rangle:\langle\tau\rangle}\vdash\Uparrow \qquad \hookrightarrow\Uparrow\frac{e'\downarrow\overset{0}{\hookrightarrow} e''}{e\overset{0}{\hookrightarrow} e''}$$

By applying persistence of typability to the top-most result in we get $\Gamma^{+}\vdash^0 e':\tau$. Applying the induction hypothesis again we $\Gamma^{+}\vdash^0 e'':\tau$.

– Run II:

$$\vdash\Uparrow\ \frac{\Gamma^{++}\vdash^n e:\langle\tau\rangle}{\Gamma^{+}\vdash^n \mathsf{run}\ e:\tau}\hookrightarrow\Uparrow\ \frac{e\overset{n+}{\hookrightarrow} e'}{\mathsf{run}\ e\overset{n+}{\hookrightarrow}\mathsf{run}\ e'}\ \overset{IH}{\Longrightarrow}\ \frac{e'\in V^{n+}}{\mathsf{run}\ e'\in V^{n+}}\,V\Downarrow\ \frac{\Gamma^{++}\vdash^n e':\langle\tau\rangle}{\Gamma^{+}\vdash^n \mathsf{run}\ e':\tau}\vdash\Downarrow$$

□

Theorem 5.4.5 (Type-Safety for CBV λ-M) $\forall n\in\mathbb{N},\Gamma\in D,\tau\in T.\forall e\in E^n, v\in V^n.$

$$\Gamma^{+}\vdash^n e:\tau\wedge e\overset{n}{\hookrightarrow} v\Longrightarrow v\not\equiv \mathsf{err}.$$

Proof. Follows directly from Type Preservation. □

5.5 A Big-Step Semantics for CBN λ-M

The difference between the CBN semantics and the CBV semantics for λ-M is only in the evaluation rule for application at level 0. For CBN, this rule becomes

$$\frac{e_1\overset{0}{\hookrightarrow}\lambda x.e\quad e[x:=e_2]\overset{0}{\hookrightarrow} v}{e_1\ e_2\overset{0}{\hookrightarrow} v}\ \text{App–CBN.}$$

Figure 5.3 summarizes the full semantics. The Type Preservation proof need only be changed for the application case.

Theorem 5.5.1 (Type Preservation for CBN λ-M) $\forall n\in\mathbb{N},\Gamma\in D,\tau\in T.\, e\in E^n, v\in V^n$

$$\Gamma^{+}\vdash^n e:\tau\wedge e\overset{n}{\hookrightarrow} v\Longrightarrow \Gamma^{+}\vdash^n v:\tau.$$

5.6 A Limitation of the Basic Type System: Re-integration

The basic type system presented in the previous chapter has a problem that renders it unsuitable for supporting multi-stage programming with explicit annotations: It cannot type some simple yet very useful terms. For example, in Chapter 2, we re-integrated the dynamically generated function as follows:

Syntax:

$$e \in E := i \mid x \mid e\,e \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e$$

Big-Step Rules:

$$\frac{}{i \overset{n}{\hookrightarrow} i}\ \text{Int} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e}\ \text{Lam} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \lambda x.e \\ e[x := e_2] \overset{0}{\hookrightarrow} e_3 \end{array}}{e_1\ e_2 \overset{0}{\hookrightarrow} e_3}\ \text{App–CBN}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3}\ \text{Run} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x}\ \text{Var+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_3 \quad e_2 \overset{n+}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_3\ e_4}\ \text{App+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2}\ \text{Lam+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}\ \text{Brk} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+}{\hookrightarrow} \mathsf{run}\ e_2}\ \text{Run+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \tilde{}e_2}\ \text{Esc++} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2}\ \text{Esc}$$

Fig. 5.3. The (Coarse) CBN Big-Step Semantics for λ-M

```
- val exp72 = run exp72'; (* : real → real *).
```

But this declaration was only typable in the faulty type system presented in Chapter 4, and not in type system presented above. In fact, we cannot even type the following simple sequence of declarations in the above type system:

```
- val a = ⟨1⟩;
- val b = run a;
```

This sequence is not typable because (with the standard interpretation of let) it corresponds to the lambda term $(\lambda a.\mathsf{run}\ a)\,\langle 1 \rangle$, which is not typable. Without overcoming this problem, we cannot achieve multi-stage programming "as advertised" in Chapters 1 to 3.

Understanding this problem and its significance requires understanding the relation between the types and the method of multi-stage programming. To this end, we begin by an analysis of the types of the six major artifacts produced at the end of the six main steps of the method of multi-stage programming. It is then clear that the type system does not allow for a general way of conducting the last step of the method. We then argue that the root of the problem lies in the lack of an effective mechanism for tracking free variables. We propose a solution to this problem at the level of types, and show how the new types can be a basis for a refined method that can be supported by a provably sound type system.

5.6.1 Types of the Artifacts of Multi-Stage Programming

The main steps of multi-stage programming are:

1. Write the conventional program

$$\mathit{program} : t_S \rightarrow t_D \rightarrow t$$

   where t_S is the type of the "static" or "known" parameters, t_D is the type of the "dynamic", or "unknown" parameters, and t is the type of the result of the program.
2. Add staging annotations to the program to derive

$$\mathit{annotated\_program} : t_S \rightarrow \langle t_D \rangle \rightarrow \langle t \rangle$$

3. Compose the annotated program with an unfolding combinator $\mathsf{back} : (\langle A \rangle \rightarrow \langle B \rangle) \rightarrow \langle A \rightarrow B \rangle$

$$\mathit{code\_generator} : t_S \rightarrow \langle t_D \rightarrow t \rangle$$

4. Construct or read the static *inputs*:

$$s : t_S$$

5. Apply the code generator to the static inputs to get

$$\mathit{specialized\_code} : \langle t_D \rightarrow t \rangle$$

6. Run the specialized code to re-introduce the generated function as a first-class value in the current environment:

$$\mathit{specialized\_program} : t_D \rightarrow t$$

All the steps of the method except the last one can be carried out within the type system presented in the previous chapter. The last step, however, is problematic.

5.6.2 The Problem of Abstracting Run

The root of the expressivity problem described above seems to be that there is no general type safe way for going from a MetaML value of code type $\langle A \rangle$ to a MetaML value of type A. At the level of language constructs, MetaML provides a construct Run. Run is a *construct* that allows the execution of a code fragment and has the type rule:

$$\frac{\Gamma^{+} \vdash^{n} e : \langle \tau \rangle}{\Gamma \vdash^{n} \mathsf{run}\ e : \tau}\ \text{Run}$$

For example, $\mathsf{run}\ \langle 3+4\rangle$ is well-typed and evaluates to 7. But Run still has limited expressivity. In particular, it is not a function, and cannot be turned into a function, because the lambda-abstraction $\lambda x.\mathsf{run}\ x$ is not typable using the type system of Section 5.2. Without such a function code fragments declared at top-level can never be executed using well-typed terms. At the same time, adding a function such as $\mathsf{unsafe\_run} : \langle A\rangle \to A$ breaks the safety of the type system presented in the previous section, because it is equivalent to reintroducing the faulty Run rule presented in Chapter 4. (Thus, the same counterexample to type safety applies.)

Despite a long search, we have not been able to find reasonable type systems where a *function* $\mathsf{unsafe\_run} : \langle A\rangle \to A$ can exist[5]. Thus we are inclined to believe that a single parametric type constructor $\langle \_ \rangle$ for code does not allow for a natural way of executing code. This observation can be interpreted as saying that "generated code" cannot be easily integrated with the rest of the run-time system.

A Closer Look at What Goes Wrong Operationally, a code fragment of type $\langle A\rangle$ can contain "free dynamic variables". Because the original code type of MetaML does not provide us with any information as to whether or not there are "free dynamic variables" in the fragment, there is no way of ensuring that this code fragment can be safely executed.

Thus, there is a need for a finer typing mechanism that provides a means for reasoning about free variables. As this observation holds in a very minimal language for multi-stage programming language, we believe that it is very likely to hold for many multi-stage languages.

5.7 The λ^{BN} Language. Or Adding Closedness to λ-M

Our proposal is to add a special type constructor to mark Closed terms to MetaML. Closed terms evaluate to closed values, that is, values containing no free variables. The viability of this proposal will be demonstrated by adding a Closed type to λ-M, and presenting a provably sound type system. The extended language is called λ^{BN} and adds the Closed type $[\_]$ to the types of λ-M:

$$\begin{aligned} \tau \in T &:= b \mid \tau_1 \to \tau_2 \mid \langle\tau\rangle \mid [\tau] \\ \Gamma \in D &:= [] \mid x : \tau^n ; \Gamma \end{aligned}$$

[5] It is tempting here to say that we were searching for a "first-class function", but any function in a functional language becomes a "first-class citizen", and an operator that is not a function is not a first-class citizen. Looking for a way to fit an operator into a functional language as a function is the same as looking for a way to fit the operator into the functional language as a first-class citizen.

The λ^{BN} language refines λ-M by adding constructs for marking and un-marking Closed terms, replacing Run with a new construct called Safe-run, and by providing an explicit form of cross-stage persistence for only closed values:

$$e \in E := c \mid x \mid \lambda x.e \mid e\, e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{close}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{open}\ e \mid \mathsf{safe\_run}\ e \mid \mathsf{up}\ e$$

The first production allows the use of any set of constants such as integers, strings, and so on. The next three productions are the standard ones in a λ-calculus. Bracket and Escape are the same as we have seen before. The Close-with construct will assert (when we have imposed a type system on these terms) that e is closed except for a set of variables x_i, each of which is bound to a Closed term e_i. Open allows us to forget the Closedness assertion on e. Safe-run executes a Closed Code fragment and returns a Closed result. Finally, Up allows us to use any Closed expression at a higher level, thus providing cross-stage persistence for Closed values.

5.7.1 Big-Step Semantics for λ^{BN}

The big-step semantics of λ^{BN} is very similar to that of λ-M, and is summarized in Figure 5.4. The first two evaluation rules are those of evaluation in the λ language. The next rule says that evaluating Bracketed expression is done by rebuilding the expression. The next two rules are new, and specify the semantics of the Closedness annotations.

The evaluation rule for Close-with says that it first evaluates the expressions in the with-clause, and then substitutes these results in place of the variables in the body of the Close-with. The result of the substitution is then evaluated, and returned as the final result. The rule for Open says that it simply evaluates its argument to get a Closed result, and returns that result. The next two rules are also new, and specify the semantics of two new operations that exploit a useful interaction between the Closed and Code types.

The definition for rebuilding is essentially the same as before, with level annotations being changed only in the cases of Brackets and Escapes.

Note that this semantics does not explicitly specify any renaming on bound object-level variables when we are rebuilding code. The capture-free substitution performed in the application rule takes care of all the necessary renaming. For example, the expression

$$\langle \lambda x.\tilde{}((\lambda z.\langle \lambda x.x + \tilde{}z \rangle)\langle x \rangle)\rangle$$

evaluates at level 0 to:

$$\langle \lambda x.\lambda y.y + x \rangle.$$

Syntax:

$$e \in E := c \mid x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \,\tilde{}\, e \mid \mathsf{close}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{open}\ e \mid \mathsf{safe\_run}\ e \mid \mathsf{up}\ e$$

Shorthands:

$$\begin{array}{rcl} \mathsf{close}\ e & \textit{for} & \mathsf{close}\ e\ \mathsf{with}\ \emptyset \\ \mathsf{close}\ e\ \mathsf{with}\ x_i = e_i & \textit{for} & \mathsf{close}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \end{array}$$

Big-Step Rules at level 0 (Evaluation):

$$\frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e}\ \text{Lam} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x := v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2}\ \text{App}$$

$$\frac{e \overset{1}{\hookrightarrow} v}{\langle e \rangle \overset{0}{\hookrightarrow} \langle v \rangle}\ \text{Brk}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i := v_i] \overset{0}{\hookrightarrow} v}{\mathsf{close}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \mathsf{close}\ v}\ \text{Clos} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ v}{\mathsf{open}\ e \overset{0}{\hookrightarrow} v}\ \text{Opn}$$

$$\frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v' \rangle \quad v' \overset{0}{\hookrightarrow} v}{\mathsf{safe\_run}\ e \overset{0}{\hookrightarrow} \mathsf{close}\ v}\ \text{SRn} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ v'}{\mathsf{up}\ e \overset{1}{\hookrightarrow} \mathsf{close}\ v'}\ \text{Up}$$

Big-Step Rules at level $n+1$ (Rebuilding):

$$\frac{}{c \overset{n+}{\hookrightarrow} c}\ \text{Const+}$$

$$\frac{}{x \overset{n+}{\hookrightarrow} x}\ \text{Var+} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v}\ \text{Lam+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+}{\hookrightarrow} v_1\ v_2}\ \text{App+}$$

$$\frac{e \overset{n++}{\hookrightarrow} v}{\langle e \rangle \overset{n+}{\hookrightarrow} \langle v \rangle}\ \text{Brk+} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\,\tilde{}\, e \overset{1}{\hookrightarrow} v}\ \text{Esc1} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\,\tilde{}\, e \overset{n++}{\hookrightarrow} \,\tilde{}\, v}\ \text{Esc}$$

$$\frac{e_i \overset{n+}{\hookrightarrow} v_i}{\mathsf{close}\ e\ \mathsf{with}\ x_i = e_i \overset{n+}{\hookrightarrow} \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i}\ \text{Clo+} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{open}\ e \overset{n+}{\hookrightarrow} \mathsf{open}\ v}\ \text{Opn+}$$

$$\frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{safe\_run}\ e \overset{n+}{\hookrightarrow} \mathsf{safe\_run}\ v}\ \text{SRn+} \qquad \frac{e \overset{n+}{\hookrightarrow} v'}{\mathsf{up}\ e \overset{n++}{\hookrightarrow} \mathsf{up}\ v'}\ \text{Up+}$$

Fig. 5.4. The CBV Big-Step Semantics for λ^{BN}

Because capture-free substitution renames the bound variable when it goes inside a lambda, there was no inadvertent capture of the variable x in the value $\langle \mathsf{x} \rangle$ during substitution.

It is also worth noting that there is a rule for rebuilding constants, which says that they remain unchanged. There are, however, no evaluation rules for Escapes, because Escapes are only intended to occur inside Brackets, and are meaningless at level 0. When the language is extended with specific constants (and specific rules for evaluating them), one will have to ensure that these rules do not violate any properties (such as Type-Safety, for example).

5.7.2 Type System for λ^{BN}

Typing judgments for λ^{BN} have the form $\Gamma \vdash^n e : \tau$, where $\Gamma \in G$ and n is a natural number called the level of the term. The *level* of the term is the number of Brackets surrounding this term less the number of Escapes surrounding this term. Figure 5.5 summarizes the type system for λ^{BN}. The rule for a constant says that it has the type associated with that constant. The next three rules are essentially the same as before, but note that the variable rule no longer allows "implicit" cross-stage persistence. Now, variables can only be used at the level at which they are bound. But we will see how one of the rules allows us to achieve a restricted form of cross-stage persistence. The next two rules are for Brackets and Escape, and are exactly the same as before. The next two rules are new, and specify the typing of the Closedness annotations. The rule for Close-with says it is typable if all the bindings in the with-clause are Closed, and the term in the body of the Close-with is typable at level 0 assuming that all the variables in the with-clause are available at level 0. In essence, this ensures that a Closed expression can only contain variables that are themselves bound to Closed expressions. The rule for open simply forgets the Closed type. The rule for Safe-run allows us to eliminate the Code type when it occurs under the Closed type. Finally, the rule for Up allows us to lift any Closed value from any level to the next, thus providing us with a limited form of cross-stage persistence: cross-stage persistence for Closed values.

5.7.3 Remark on Up, Covers, and Performance

It is worth noting that if we build an implementation along the lines of Chapter 4 for a language that only has cross-stage persistence for Closed values, there should be no need for the covers discussed in Section 4.5. In particular, cross-stage persistent constants in such an implementation can never carry free variables. To see this claim, recall that:

1. covers are used to perform substitution on functional values,

Syntax:

$$e \in E := c \mid x \mid \lambda x.e \mid e\,e \mid \langle e \rangle \mid \tilde{}e \mid \textsf{close}\ e\ \textsf{with}\ \{x_i = e_i | i \in m\} \mid \textsf{open}\ e \mid \textsf{safe\_run}\ e \mid \textsf{up}\ e$$

Types and Type Environments:

$$\begin{array}{l} \tau \in T := b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle \mid [\tau] \\ \Gamma \in D := [] \mid x : \tau^n ; \Gamma \end{array}$$

Operations on Type Environments:

$$\Gamma^+(x) \stackrel{\Delta}{=} \tau^{n+} \text{ where } \Gamma(x) = \tau^n$$

Type Rules:

$$\frac{}{\Gamma \vdash^n c : \tau_c}\text{Const}$$

$$\frac{\Gamma(x) = \tau^n}{\Gamma \vdash^n x : \tau}\text{Var} \quad \frac{\Gamma; x : \tau_1^n \vdash^n e : \tau_2}{\Gamma \vdash^n \lambda x.e : (\tau_1 \to \tau_2)}\text{Lam} \quad \frac{\Gamma \vdash^n e_1 : (\tau_1 \to \tau_2) \quad \Gamma \vdash^n e_2 : \tau_1}{\Gamma \vdash^n e_1\ e_2 : \tau_2}\text{App}$$

$$\frac{\Gamma \vdash^{n+} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\text{Brk} \quad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+} \tilde{}e : \tau}\text{Esc}$$

$$\frac{\Gamma \vdash^n e_i : [\tau_i] \quad \{x_i : [\tau_i]^0 | i \in m\} \vdash^0 e : \tau}{\Gamma \vdash^n \textsf{close}\ e\ \textsf{with}\ x_i = e_i : [\tau]}\text{Clo} \quad \frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^n \textsf{open}\ e : \tau}\text{Opn}$$

$$\frac{\Gamma \vdash^n e : [\langle \tau \rangle]}{\Gamma \vdash^n \textsf{safe\_run}\ e : [\tau]}\text{SRn} \quad \frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^{n+} \textsf{up}\ e : [\tau]}\text{Up}$$

Fig. 5.5. The λ^{BN} Type System

2. functional values arise only inside cross-stage persistent constants,
3. Closed cross-stage persistent constants do not contain free variables.

For example, the puzzle term of Chapter 4

$$\langle \mathsf{fn\ a} \Rightarrow \tilde{~}((\mathsf{fn\ x} \Rightarrow \langle \mathsf{x}\rangle)\ (\mathsf{fn\ x} \Rightarrow \langle \mathsf{a}\rangle))\ 0\rangle$$

is no longer acceptable by the type system, because a cross-stage persistent variable such as x in the first lambda-abstraction must be of Closed type, and at the same time, a is a free variable in the second lambda-abstraction and so the lambda-abstraction cannot have Closed type.

As covering involves adding an extra function composition and a latent traversal of code every time a cross-stage persistent variable is evaluated (independently of whether it contains any hidden free-variables or not), covering is a costly operation. The observations above also suggest that covering can be avoided completely if we restrict cross-stage persistence to Closed values.

5.7.4 Basic Properties of Type System

We now present some technical lemmas for λ^{BN} needed to establish type-safety.

Lemma 5.7.1 (Weakening) $\forall n, m \in \mathbb{N}, e \in E, \Gamma \in D, \tau, \sigma \in T.$

$$\Gamma \vdash^n e : \tau \wedge x \notin dom(\Gamma) \wedge x \notin \mathrm{FV}(e) \Longrightarrow \Gamma; x : \sigma^m \vdash^n e : \tau.$$

Proof. Same as for λ-M. □

Lemma 5.7.2 (Substitution) $\forall n, m \in \mathbb{N}, e_1, e_2 \in E, \Gamma_1, \Gamma_2 \in D, \tau, \sigma \in T.$

$$\Gamma_1, \Gamma_2; x : \sigma^m \vdash^n e_1 : \tau \wedge \Gamma_2 \vdash^m e_2 : \sigma \Longrightarrow \Gamma_1, \Gamma_2 \vdash^n e_1[x := e_2] : \tau.$$

Proof. Same as Substitution for λ-M. □

5.7.5 Type-Safety

The first set of rules that must be added to the big-step semantics is as follows:

$$\frac{}{x \overset{0}{\hookrightarrow} \mathsf{err}}\ \mathrm{VarErr} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_3 \quad e_3 \not\equiv \lambda x.e}{e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}}\ \mathrm{AppErr}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \mathsf{close}\ v}{\mathsf{open}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}}\ \mathrm{OpnErr} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \mathsf{close}\ \langle v\rangle}{\mathsf{safe\_run}\ e_1 \overset{0}{\hookrightarrow} \mathsf{err}}\ \mathrm{SRnErr}\,.$$

$$\frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \mathsf{close}\ v}{\mathsf{up}\ e_1 \overset{1}{\hookrightarrow} \mathsf{err}}\ \mathrm{UpErr} \qquad \frac{e_1 \overset{0}{\hookrightarrow} e_2 \quad e_2 \not\equiv \langle v\rangle}{\tilde{~}e_1 \overset{1}{\hookrightarrow} \mathsf{err}}\ \mathrm{Esc1Err}$$

Again, the propagation rules simply return an error if the result of a sub-computation is an error.

5.7.6 Basic Properties of Big-Step Semantics

Values for λ^{BN} are defined as follows:

$$\begin{aligned}
v^0 \in V^0 \quad &:= \lambda x.e \mid \langle v^1 \rangle \mid \mathsf{close}\ v^0 \\
v^1 \in V^1 \quad &:= c \mid x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i^1 \mid \mathsf{open}\ v^1 \mid \mathsf{safe\_run}\ v^1 \\
v^{n++} \in V^{n++} &:= c \mid x \mid v^{n++}\ v^{n++} \mid \lambda x.v^{n++} \mid \langle v^{n+++} \rangle \mid \tilde{}v^{n+} \mid \\
&\qquad \mathsf{up}\ v^{n+} \mid \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i^{n++} \mid \mathsf{open}\ v^{n++} \mid \mathsf{safe\_run}\ v^{n++}
\end{aligned}$$

Lemma 5.7.3 (Values) $\forall n \in \mathbb{N}$.

1. $v \in V^n \Longrightarrow v \in V^{n+}$
2. $\forall e, e' \in E^n.\, e \overset{n}{\hookrightarrow} e' \Longrightarrow e' \in (V^n \cup \{\mathsf{err}\})$

Proof. Just as for λ-M. □

The key property to establish for λ^{BN} is that a value produced by evaluating an expression of Closed type will actually be a *closed* value. The following lemma formalizes this claim by saying that a value of Closed type is typable under the empty environment:

Lemma 5.7.4 (Closedness) $\forall v \in V^0, \Gamma \in D, \tau \in T.$

$$\Gamma \vdash^0 v : [\tau] \Longrightarrow \emptyset \vdash^0 v : [\tau].$$

Proof. Immediate from the definition of values. □

Lemma 5.7.5 (Strong Value Reflection for Typed Terms) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T, v \in V^{n+}, e \in E^n$.

1. $\Gamma^+ \vdash^{n+} v : \tau \Longrightarrow \Gamma \vdash^n v : \tau,$
2. $\Gamma \vdash^n e : \tau \Longrightarrow (\Gamma^+ \vdash^{n+} e : \tau \wedge e \in V^{n+}).$

Proof. Just as for λ-M. □

Theorem 5.7.6 (Type Preservation for CBV λ^{BN}) $\forall n \in \mathbb{N}, e \in E, \Gamma \in D, \tau \in T. v \in V^n$

$$\Gamma^+ \vdash^n e : \tau \wedge e \overset{n}{\hookrightarrow} v \Longrightarrow \Gamma^+ \vdash^n v : \tau.$$

Proof. By induction on the derivation of $e \overset{n}{\hookrightarrow} v$. The case for application uses Substitution. The case for Up involves Closedness, Reflection, and Weakening, in addition to applying the induction hypothesis. The case for Safe-run involves Reflection. □

Theorem 5.7.7 (Type-Safety for CBV λ^{BN}) $\forall n \in \mathbb{N}, e \in E, \Gamma \in D, \tau \in T. \forall v \in V^n$.

$$\Gamma^+ \vdash^n e : \tau \wedge e \overset{n}{\hookrightarrow} v \Longrightarrow v \not\equiv \mathsf{err}.$$

Proof. Follows directly from Type Preservation. □

5.7.7 CBN λ^{BN}

As for λ-M, the difference between the CBN semantics and the CBV semantics for λ^{BN} is only in the evaluation rule for application at level 0. For CBN, the application rule becomes

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e[x := e_2] \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}\ \text{App–CBN}$$

Again, the Type Preservation proof need only be changed for the application case.

Theorem 5.7.8 (Type Preservation for CBV λ^{BN}) $\forall n \in \mathbb{N}, \forall e \in E, \Gamma \in D, \tau \in T. v \in V^n$

$$\Gamma^+ \vdash^n e : \tau \wedge e \overset{n}{\hookrightarrow} v \Longrightarrow \Gamma^+ \vdash^n v : \tau.$$

Theorem 5.7.9 (Type-Safety for CBN λ^{BN}) $\forall n \in \mathbb{N}, e \in E, \Gamma \in D, \tau \in T. \forall v \in V^n.$

$$\Gamma^+ \vdash^n e : \tau \wedge e \overset{n}{\hookrightarrow} v \Longrightarrow v \not\equiv \mathsf{err}.$$

5.8 Refining the Types

The crucial insight presented in this chapter is that there are useful type systems where a function $\mathsf{safe\_run} : [\langle A \rangle] \to [A]$ exists. Safe-run has the same operational behavior that $\mathsf{unsafe\_run}$ was intended to achieve, namely *running* code. The difference is *only* in the typing of this function. In a nutshell, $\mathsf{safe\_run}$ allows the programmer to exploit the fact that *closed code can be safely executed.*

5.8.1 Refining the Method

We propose a refinement of multi-stage programming with explicit assertions about *closedness*, and where these assertions are checked by the type system:

1. Write the conventional program (exactly the same as before)

$$\mathit{program} : t_S \to t_D \to t.$$

2. Add staging *and Closedness* annotations to the program to achieve

$$\mathit{closed\_annotated\_program} : [t_S \to \langle t_D \rangle \to \langle t \rangle].$$

   Almost the same as before. The difference is that the programmer must now use the Closed type constructor [_] to demonstrate to the type system that the annotated program does not introduce any "free dynamic variables". This new requirement means that, in constructing the annotated program, the programmer will only be allowed to use Closed values.

3. Compose the annotated program with an unfolding combinator to get

$$closed\_code\_generator : [t_S \to \langle t_D \to t\rangle].$$

   Now back must itself be Closed if we are to use it inside a Closed value, that is, we use a slightly different combinator closed_back : $[(\langle A\rangle \to \langle B\rangle) \to \langle A \to B\rangle]$

4. Turn the Closed code-generator into

$$generator\_of\_closed\_code : [t_S] \to [\langle t_D \to t\rangle].$$

   This new program is exhibited by applying a combinator closed_apply : $[A \to B] \to [A] \to [B]$.

5. Construct or read the static inputs *as Closed values*:

$$cs : [t_S]$$

   This step is similar to multi-stage programming with explicit annotations. However, requiring the input to be Closed is much more specific than in the original method. Thus, we now have to make sure that all combinators used in constructing this value are themselves Closed.

6. Apply the code generator to the static inputs to get

$$closed\_specialized\_code : [\langle t_D \to t\rangle].$$

7. Run the result above to get:

$$closed\_specialized\_program : [t_D \to t].$$

   This step exploits an interaction between the Closed and Code types in our type system. The step is performed by applying a function safe_run : $[\langle A\rangle] \to [A]$.

8. Forget that the specialized program is Closed:

$$specialized\_program : t_D \to t.$$

   The step is performed by applying a function open:$[A] \to A$.

The full development of various multi-stage programming examples from previous chapter can be expressed in λ^{BN}.

5.9 Staging the Power Function Revisited

Recall the power function that we staged in Section 2.5:

```
fun exp (n,x) = (* : int × real → real *)
  if n = 0 then 1.0
          else if even n then sqr (exp (n div 2,x))
                         else x * exp (n - 1,x).
```

Staging this function in λ^{BN} is essentially the same as staging it in MetaML. The difference is that we surround the MetaML-style staged function with Closedness annotations. We use Close-with to mark a term as Closed, and we use Open to "forget" the Closedness of the free variables that we wish to use in defining the Closed term. Thus, the program is annotated as follows:

```
val exp'' = (* : [int × ⟨real⟩ → ⟨real⟩] *)
  close let val even = open even'' (* : int → bool *)
          val sqr' = open sqr'' (* : ⟨real⟩ → ⟨real⟩ *)
          fun exp' (n,x) = (* : int × ⟨real⟩ → ⟨real⟩ *)
            if n = 0 then ⟨1.0⟩
                    else if even n then sqr' (exp' (n div 2,x))
                                   else ⟨~x * ~(exp' (n - 1,x))⟩;
        in exp' end
  with even'' = even'', sqr'' = sqr''.
```

where even'' and sqr'' are versions of the even and the sqr' functions Closed in the same manner as above[6]. For example, sqr'' is defined as:

```
val sqr'' = (* : [⟨real⟩ → ⟨real⟩] *)
  close let fun sqr' x = (* : ⟨real⟩ → ⟨real⟩ *)
            ⟨let val y = ~x in y * y end⟩;
        in sqr' end.
```

Note that the fun exp' part of the val exp'' declaration is exactly the text for the exp function staged in MetaML. (See Section 2.5.)

In general, we have to explicitly Close all values that we wish to use in constructing a bigger Closed value, with the exception of primitive operations. Programs produced in this way are somewhat verbose, but we believe that this problem can be alleviated by a careful separation of Closed and non-Closed values in the environment. The study of this separation is left for future work.

[6] The operationally unnecessary re-binding with even'' = even'', sqr'' = sqr'' is needed for type-checking (see type system). Operationally, it is no different from writing $(\lambda y.(\lambda x.e))\ y\ x$.

5.10 The Refined Method is Intuitively Appealing

MetaML's original type system (Section 4.1.2) has one Code type constructor, which tries to combine the features of open and closed code type constructors: The constructor was supposed to allow us to "evaluate under lambda" (thus work with open code) and to run code (for which it must be closed). This *combination* leads to the typing problem discussed in Section 4.4.2. In contrast, λ^{BN}'s type system incorporates separate open-code and closed-value type constructors, thereby providing *correct semantics* for the following natural and desirable functions:

1. open : $[\tau] \to \tau$. This function allows us to forget the Closedness of its argument. The λ^{BN} language has no function of the inverse type $\tau \to [\tau]$.
2. up : $[\tau] \to \langle[\tau]\rangle$. This function corresponds to cross-stage persistence for Closed values. In fact, it embeds any Closed value into a code fragment, including values of functional type. Such a function does not exist in $\lambda^{\bigcirc}$ [22]. At the same time, λ^{BN} has no function of the inverse type $\langle[\tau]\rangle \to [\tau]$, reflecting the fact that there is no general way of going backwards.
3. $\mathsf{safe\_run}$: $[\langle\tau\rangle] \to [\tau]$. This function allows us to execute a Closed piece of code to get a Closed value. It can be viewed as the essence of the interaction between the Bracket and the Closed types.

Chapter 6

Reduction Semantics

I am the one who was seduced by The Impossible.
I saw the moon, I jumped high, high in the sky.
Reached it — or not; what do I care?
Now that my heart was quenched with joy!

Quatrians, Salah Jaheen

In this chapter we begin by explaining why defining a reduction semantics for MetaML is challenging. We then present a strikingly simple reduction semantics for MetaML that is confluent, and is sound with respect to the big-step semantics.

This chapter presents new results on the untyped semantics of multi-stage programming languages.

6.1 A Reduction Semantics for CBN and CBV λ

A formal semantics, in general, provides us with a means for going from arbitrary expressions to values, with the provision that certain expressions may not have a corresponding value. An important conceptual tool for the study of a programming language is a reduction semantics. A reduction semantics is a set of rewrite rules that formalize the "notions of reduction" for a given language. Having such a semantics can be useful in developing an equational theory[1]. We will first review how this semantics can be specified for the λ language of Section 4.1.2.

[1] In our experience, it has also been the case that studying such a semantics has helped us in developing the first type system presented in Chapter 5. It is likely that a reduction semantics can be helpful in developing a type system. In particular, an important property of an appropriate type system is that it should remain invariant under reductions (Subject Reduction). Because reduction semantics are often simple, they can help language designers eliminate many inappropriate type systems.

Recall that the set of expressions and the set of values for the λ language can be defined as follows:

$$\begin{aligned} e \in E &:= i \mid x \mid \lambda x.e \mid e\, e \\ v \in V &:= i \mid \lambda x.e. \end{aligned}$$

In order, the productions for expressions are for integers, lambda abstractions, and applications. Values for this language are integers and lambda-abstractions.

Intuitively, expressions are "commands" or "computations", and values are the "answers", "acceptable results" or simply "expressions that require no further evaluation". Note that we allow any value to be used as an expression with no computational content. In order to build a mechanism for going from expressions to values, we need to specify a formal rule for eliminating both variables and applications from a program. In a *reduction semantics*, (see for example Barendregt [1]) this elimination process is specified by introducing rewrite rules called "notions of reduction". The well-known β rule helps us eliminate both applications and variables at the same time:

$$(\lambda x.e_1)\, e_2 \longrightarrow_\beta e_1[x := e_2].$$

This rule says that the application of a lambda-abstraction to an expression can be simplified to the substitution the expression into the body of the lambda-abstraction. The CBN semantics is based on this rule. A similar rule is used for CBV:

$$(\lambda x.e)\, v \longrightarrow_{\beta_v} e[x := v],$$

where the argument is restricted to be a CBV value[2], thus forcing it be evaluated before it is passed to the function. The MetaML implementation is CBV, but we will simply use β in the rest of this chapter, emphasizing the applicability of the work to a CBN language[3].

Using the β rule, we build a new relation $\longrightarrow$ (with no subscript) that allows us to perform this rewrite on any subexpressions. (See for example Section 3.1.) More formally, for any two expressions $C[e]$ and $C[e']$ which are identical everywhere but in exactly one hole filled with e and e', respectively, we can say:

$$e \longrightarrow_\beta e' \Longrightarrow C[e] \longrightarrow C[e']$$

When there is more than one rule in our reduction semantics, the left hand side of this condition is the disjunction of the rewrites from e to e' using *any* of the rules in our rewrite system. Thus

[2] CBV values are slightly different from CBN values, most notably, in that CBV values typically include variables also. Note also that this distinction only arises for reduction semantics, and not for big-step semantics.

[3] It should be noted that, due to time limitations, we have only formally verified confluence and soundness for CBN MetaML, and not for CBV MetaML. However, we expect these properties to hold.

the relation $\longrightarrow$ holds between any two terms if exactly one of their subterms is rewritten using any of the rules in our reduction semantics.

6.1.1 Coherence and Confluence

Two important concepts that will be central to this chapter are coherence and confluence (For confluence, see for example Barendregt [1]). Recall from Section 3.1 that a term-rewriting system is non-deterministic. Therefore, depending on the order in which we apply the rules, we might get different results. When this is the case, our semantics could reduce a program e to either 0 or 1. We say a reduction semantics is *coherent* when any path that leads to a ground value leads to the same ground value. A semantics that lacks coherence is not satisfactory for a deterministic programming language.

Intuitively, knowing that a rewriting system is *confluent* tells us that the reductions can be applied in any order, without affecting the set of results that we can reach by applying more reductions. Thus, confluence of a term-rewriting system is a way of ensuring coherence. Conversely, if we lose coherence, we lose confluence.

We now turn to the problem of how to extend the reduction semantics of λ to a multi-stage language.

6.2 Extending the Reduction Semantics for λ

A first attempt at extending the set of expressions and values of λ to incorporate the basic staging constructs of MetaML yields the following set of expressions and values:

$$e \in E := i \mid x \mid \lambda x.e \mid e\,e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e,$$

and we add the following two rules to the β rule:

$$\tilde{}\langle e \rangle \longrightarrow_E e$$
$$\mathsf{run}\ \langle e \rangle \longrightarrow_R e.$$

But there are several reasons why this naive approach is unsatisfactory. In the rest of this chapter, we will explain the problems with this approach, and explore the space of possible improvements to this semantics.

6.3 Intensional Analysis Conflicts with β on Raw MetaML Terms

Our first observation is that there is a conflict between the β rule and supporting intensional analysis. Support for intensional analysis means adding constructs to MetaML that would allow a program to inspect a piece of code, and possibly change its execution based on either the structure or content of that piece of code. This conflict is an example of a high-level insight that resulted from studying the formal semantics of MetaML. In particular, MetaML was developed as a *meta-programming language*, and while multi-stage programming does not need to concern itself with how the code type is represented, the long-term goals of the MetaML project have at one time included support for intensional analysis. The idea is that intensional analysis could be used, for example, to allow programers to write their own optimizers for code.

It turns out that such intensional analysis is in direct contention with allowing the β-rule on object-code (that is, at levels higher than 0). To illustrate the interaction between the β rule and intensional analysis, assume that we have a minimal extension to core MetaML that tests a piece of code to see if it is an application. This extension can be achieved using a simple hypothetical construct with the following semantics:

```
-| IsApp ⟨(fn x ⇒ x) (fn y ⇒ y)⟩;
val it = true : bool.
```

Allowing β on object-code then means that ⟨(fn x ⇒ x) (fn y ⇒ y)⟩ can be replaced by ⟨fn y ⇒ y⟩. Such a reduction could be performed by an optimizing compiler, and could be justifiable, because it eliminates a function call in the object-program. But such an "optimization" would have a devastating effect on the semantics of MetaML. In particular, it would also allow our language to behave as follows:

```
-| IsApp ⟨(fn x ⇒ x) (fn y ⇒ y)⟩;
val it = false : bool.
```

When the reduction is performed, the argument to IsApp is no longer an application, but simply the lambda term ⟨fn y ⇒ y⟩. In other words, allowing both intensional analysis and object-program optimization implies that we can get the result false just as well as we can get the result true. This example illustrates a problem of coherence of MetaML's semantics with the presence of β reduction at higher levels, and code inspection. While this issue is what first drew our attention to the care needed in specifying what equalities should hold in MetaML, there are more subtle concerns that are of direct relevance to multi-stage programming, even in the absence of intensional analysis.

6.4 Level-Annotated MetaML Terms and Expression Families

In order to control the applicability of the β at various levels, we developed the notion of *level-annotated terms*. Level-annotated terms carry around a natural number at the leaves to reflect the level of the term. Such terms keep track of meta-level information (the level of a subterm) in the terms themselves, so as to give us finer control over where different reductions are applicable.

Level-annotated terms induce an infinite family of sets $E^0, E^1, E^2, \ldots$ where each annotated term lives. The *family* of level-annotated expressions and values is defined as follows:

$$
\begin{array}{lll}
e^0 & \in E^0 & := i^0 \mid x^0 \mid \lambda x.e^0 \mid e^0 e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\
e^{n+} & \in E^{n+} & := i^{n+} \mid x^{n+} \mid \lambda x.e^{n+} \mid e^{n+} e^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}\, e^{n} \mid \mathsf{run}\ e^{n+} \\
v^0 & \in V^0 & := i^0 \mid \lambda x.e^0 \mid \langle v^1 \rangle \\
v^1 & \in V^1 & := i^1 \mid x^1 \mid \lambda x.v^1 \mid v^1 v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1 \\
v^{n++} & \in V^{n++} & := i^{n++} \mid x^{n++} \mid \lambda x.v^{n++} \mid v^{n++} v^{n++} \mid \lambda x.v^{n++} \mid \langle v^{n+++} \rangle \mid \tilde{}\, v^{n+} \mid \mathsf{run}\ v^{n++}.
\end{array}
$$

The key difference between level-annotated terms and raw terms is in the "leaves", namely, the variables and integers. In level-annotated terms, variables and integers explicitly carry around a natural number that represents their level. For all other constructs, we can simply infer the level of the whole term by looking at the subterm. For Brackets and Escapes, the obvious "correction" to levels is performed.

Note that whenever we "go inside" a Bracket or an Escape, the index of the expression set is changed in accordance with the way the level changes when we "go inside" a Bracket or an Escape.

6.5 Escapes Conflict with β on Annotated MetaML terms

There is a problematic interaction between the β rule at higher levels and Escape. In particular, β does not preserve the syntactic categories of level annotated terms. Consider the following term:

$$\langle (\mathsf{fn\ x} \Rightarrow \tilde{}\,\mathsf{x}^0)\ \tilde{}\langle 4^0 \rangle \rangle.$$

The level of the whole term is 0. If we allow the β rule at higher levels, this term can be reduced to:

$$\langle \tilde{}\,\tilde{}\langle 4^0 \rangle \rangle.$$

This result contains two nested Escapes. Thus, the level of the whole term can no longer be 0. The outer Escape corresponds to the Bracket, but what about the inner Escape? Originally, it corresponded to the same Bracket, but after the β reduction, what we get is an expression that cannot be read in the same manner as the original term.

6.6 Substitution Conflicts with β on Level 0 Annotated Terms

One possibility for avoiding the problem above is to limit β to level 0 terms:

$$(\lambda x.e_1^0)\, e_2^0 \longrightarrow_\beta e_1^0[x := e_2^0].$$

At first, this approach seems appealing because it makes the extension of MetaML with code inspection operations less problematic. But consider the following term:

$(\mathsf{fn\ x} \Rightarrow \langle \mathsf{x}^1\rangle)\ (\mathsf{fn\ y} \Rightarrow (\mathsf{fn\ x} \Rightarrow \mathsf{x}^0)\ 5^0).$

There are two possible β reductions at level 0 in this term. The first is the outermost application, and the second is the application inside the argument. If we do the first application, we get the following result:

$\langle \mathsf{fn\ y} \Rightarrow (\mathsf{fn\ x} \Rightarrow \mathsf{x}^1)\ 5^1\rangle.$

The level annotations need to be adjusted after substitution. (See Section A.1.1.) But first note that there are no β reductions at level 0 left in this term. If we do the second application first, we get

$(\mathsf{fn\ x} \Rightarrow \langle \mathsf{x}^1\rangle)\ (\mathsf{fn\ y} \Rightarrow 5^0).$

and then we can still go back and perform the outermost application to get:

$\langle \mathsf{fn\ y} \Rightarrow 5^1\rangle.$

Again, in the presence of code inspection, this example illustrates an incoherence problem. But even in the absence of code inspection, we still lose the *confluence* of our reductions, despite the fact that we have sacrificed β reductions at higher-levels. Intuitively, the example above illustrates that cross-stage persistence arises naturally in untyped MetaML terms, and that cross-stage persistence makes it hard to limit β to level 0 in a consistent (that is, confluent) way. In the example above, applying the lift-like term $\mathsf{fn\ x} \Rightarrow \langle \mathsf{x}\rangle$ to a function causes all the redices in the body of that function to be frozen.

6.7 A Reduction Semantics for CBN λ-U

No Nightingale did ever chaunt
More welcome notes to weary bands
Of travellers in some shady haunt,
Among Arabian sands:
A voice so thrilling ne'er was heard
In spring-time from the Cuckoo-bird,
Breaking the silence of the seas
Among the farthest Hebrides.

The Solitary Reaper, William Wordsworth

The syntax of λ-U consists of the set of *raw* expressions and values defined as follows:

$$\begin{array}{lll}
e^0 & \in E^0 & := v \mid x \mid e^0\, e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\
e^{n+} & \in E^{n+} & := \langle e^n \rangle \mid i \mid x \mid e^{n+}\, e^{n+} \mid \lambda x.e^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}\, e^n \mid \mathsf{run}\ e^{n+} \\
v & \in V & := i \mid \lambda x.e^0 \mid \langle e^0 \rangle.
\end{array}$$

The essential subtlety in this definition is in the last production in the set of values: Inside the Brackets of a code value, what is needed is simply *an expression of level 0.*

The CBN notions of reduction of λ-U are simply:

$$\begin{array}{rl}
(\lambda x.e_1^0)\, e_2^0 & \longrightarrow_{\beta_U} e_1^0[x := e_2^0] \\
\tilde{}\langle e^0 \rangle & \longrightarrow_{E_U} e^0 \\
\mathsf{run}\ \langle e^0 \rangle & \longrightarrow_{R_U} e^0.
\end{array}$$

Just like the rules for λ, these rules are intended to be applied in any context. This calculus allows us to apply the β rule to any expression that *looks like* a level 0 application. By restricting the body of the lambda term and its argument to be in E^0, the λ-U language avoids the conflict between Escapes and β that we discussed earlier on in this chapter, because level 0 terms are free of top-level Escapes.

6.7.1 Subject Reduction

Figure 6.1 summarizes the language CBN λ-U that we present and study in this section. Note that CBN λ-U also enjoys the cleanliness of being defined in terms of the standard notion of substitution.

It is fairly straightforward to establish that λ-U preserves typability, where the notion of typing is that of the type system presented in Section 5.3:

Syntax:

$$\begin{array}{llll} e^0 & \in E^0 & := v \mid x \mid e^0\, e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+} & \in E^{n+} & := \langle e^n \rangle \mid i \mid x \mid e^{n+}\, e^{n+} \mid \lambda x.e^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}\, e^n \mid \mathsf{run}\ e^{n+} \\ v & \in V & := i \mid \lambda x.e^0 \mid \langle e^0 \rangle \end{array}$$

Reductions:

$$\begin{array}{rl} (\lambda x.e_1^0)\, e_2^0 & \longrightarrow_{\beta_U} e_1^0[x := e_2^0] \\ \tilde{}\langle e^0 \rangle & \longrightarrow_{E_U} e^0 \\ \mathsf{run}\ \langle e^0 \rangle & \longrightarrow_{R_U} e^0 \end{array}$$

Fig. 6.1. The CBN λ-U Language

Theorem 6.7.1 (Subject Reduction for CBN λ-U) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T. \forall e_1, e_2 \in E^n.$

$$\Gamma \vdash^n e_1 : \tau \wedge e_1 \longrightarrow e_2 \Longrightarrow \Gamma \vdash^n e_2 : \tau.$$

Proof. The proof builds on the basic properties of the type system that we have already established in Chapter 5. There are only three cases that need to be considered:

- $e_3, e_4 \in E^0$ and $e_1 = (\lambda x.e_3)\ e_4 \longrightarrow e_3[x := e_4] = e_2$. Then by $\in E^0$ we know that $e_1 \in E^0$. We are given $\Gamma \vdash^n (\lambda x.e_3)\ e_4 : \tau$. Then, by typing, we know $\Gamma \vdash^n e_4 : \sigma$ and $\Gamma \vdash^n (\lambda x.e_3) : \sigma \to \tau$. Again, by typing, we also know $\Gamma; x : \sigma^n \vdash^n e_3 : \tau$. By substitution we know that $\Gamma \vdash^n e_3[x := e_4] : \tau$ and we are done.
- $e_3 \in E^0$ and $e_1 \equiv \mathsf{run}\ \langle e_3 \rangle \longrightarrow e_3 \equiv e_2$. We are given $\Gamma \vdash^n \mathsf{run}\ \langle e_3 \rangle : \tau$. By typing we get $\Gamma^+ \vdash^n \langle e_3 \rangle : \langle \tau \rangle$, and then $\Gamma^+ \vdash^{n+} e_3 : \tau$. By persistence of typability we get $\Gamma \vdash^n e_3 : \tau$ and we are done.
- $e_3 \in E^0$ and $e_1 \equiv \mathsf{run}\ \langle e_3 \rangle \longrightarrow e_3 \equiv e_2$. We are given $\Gamma \vdash^{n+} \tilde{}\langle e_3 \rangle : \tau$. By typing we get $\Gamma \vdash^n \langle e_3 \rangle : \langle \tau \rangle$, and then $\Gamma \vdash^{n+} e_3 : \tau$ and we are done.

□

6.8 Confluence

Establishing the confluence property in the presence of the β rule can be involved, largely because substitution can duplicate redices, and establishing that these redices are not affected by substitution can be non-trivial. Barendregt presents a number of different ways for proving confluence, and discusses their relative merits [1]. Recently, Takahashi has produced a concise yet highly rigorous technique for proving confluence, and demonstrated its application in a variety of settings, including proving some subtle properties of reduction systems such as standardization [96]. The basic idea that Takahashi promotes is the use of an explicit notion of a parallel reduction. While the idea goes back to the original and classic (yet unpublished) works of Tait and Martin-Löf, Takahashi emphasizes that the rather verbose notion of residuals (see Barendregt [1], for example,) can be completely avoided.

In this section, we present a proof of the confluence of CBN λ-U that follows closely the development in the introduction to Takahashi's paper. The CBN reductions of λ-U do not introduce any notable complications to the proof, and it is as simple, concise, and rigorous as the one presented by Takahashi.

Definition 6.8.1 (Context) *A* Context *is an expression with exactly one hole* [].

$$C \in \mathbb{C} := [] \mid \lambda x.C \mid C\,e \mid e\,C \mid \langle C \rangle \mid \tilde{\ }C \mid \mathsf{run}\ C.$$

We write $C[e]$ *for the expression resulting from replacing ("filling") the hole* [] *in the context* C *with the expression* e*.*

Lemma 6.8.2 (Basic Property of Contexts) $\forall C \in \mathbb{C}. \forall e \in E.$

$$C[e] \in E.$$

Proof. By an induction on the derivation of $C \in \mathbb{C}$. □

Remark 6.8.3 *Filling a hole in a context can involve variable capture, in the sense that given* $C \equiv \lambda x.[]$, $C[x] \equiv \lambda x.x$*, and the binding occurrence of* x *in* C *is not renamed.*

Definition 6.8.4 (Parallel Reduction) *The* parallel reduction *relation* $\_ \gg \_ \subseteq E \times E$ *is defined as follows:*

1. $$\frac{}{x \gg x}$$
2. $$\frac{e_1 \gg e_2}{\lambda x.e_1 \gg \lambda x.e_2}$$
3. $$\frac{e_1 \gg e_3 \quad e_2 \gg e_4}{e_1 e_2 \gg e_3 e_4}$$
4. $$\frac{e_1^0 \gg e_3 \quad e_2^0 \gg e_4^0}{(\lambda x.e_1^0)e_2^0 \gg e_3[x := e_4^0]}$$
5. $$\frac{e_1 \gg e_2}{\langle e_1 \rangle \gg \langle e_2 \rangle}$$
6. $$\frac{e_1 \gg e_2}{\tilde{\ }e_1 \gg \tilde{\ }e_2}$$

7. $$\frac{e_1^0 \gg e_2^0}{\tilde{}\langle e_1^0\rangle \gg e_2^0}$$

8. $$\frac{e_1 \gg e_2}{\textsf{run}\ e_1 \gg \textsf{run}\ e_2}$$

9. $$\frac{e_1^0 \gg e_2^0}{\textsf{run}\ \langle e_1^0\rangle \gg e_2^0}\ .$$

Remark 6.8.5 (Idempotence) *By ignoring rules 4,7, and 9 in the definition above, it is easy to see that* $e \gg e$.

Lemma 6.8.6 (Parallel Reduction Properties) $\forall e_1 \in E$.

1. $\forall e_2 \in E.\, e_1 \longrightarrow e_2 \Longrightarrow e_1 \gg e_2$
2. $\forall e_2 \in E.\, e_1 \gg e_2 \Longrightarrow e_1 \longrightarrow^* e_2$
3. $\forall e_3 \in E.\, e_2, e_4 \in E^0.\, e_1 \gg e_3, e_2^0 \gg e_4^0 \Longrightarrow e_1[y := e_2^0] \gg e_3[y := e_4^0]$.

Proof. The first is proved by induction on the context of the redex, and the second and third by induction on e_1. □

Remark 6.8.7 *From 1 and 2 above we can see that that* $\gg^* = \longrightarrow^*$.

The Church-Rosser theorem [1] for $\longrightarrow$ follows from Takahashi's property [96]. The statement of Takahashi's property uses the following notion.

Definition 6.8.8 (Star Reduction) *The* star reduction *function* $\_^* : E \to E$ *is defined as follows:*

1. $x^* \equiv x$
2. $(\lambda x.e_1)^* \equiv \lambda x.e_1^*$
3. $(e_1 e_2)^* \equiv e_1^* e_2^*$ *if* $e_1 e_2 \not\equiv (\lambda x.e_3^0)e_4^0$
4. $((\lambda x.e_1^0)e_2^0)^* \equiv (e_1^0)^*[x := (e_2^0)^*]$
5. $\langle e_1\rangle^* \equiv \langle e_1^*\rangle$
6. $(\tilde{}e_1)^* \equiv \tilde{}(e_1^*)$ *if* $\tilde{}e_1 \not\equiv \tilde{}\langle e_2^0\rangle$
7. $(\tilde{}\langle e_1^0\rangle)^* \equiv (e_1^0)^*$
8. $(\textsf{run}\ e_1)^* \equiv \textsf{run}\ (e_1^*)$ *if* $\textsf{run}\ e_1 \not\equiv \textsf{run}\ \langle e_2^0\rangle$
9. $(\textsf{run}\ \langle e_1^0\rangle)^* \equiv (e_1^0)^*$.

Remark 6.8.9 *By a simple induction on* e, *we can see that* $e \gg e^*$.

Theorem 6.8.10 (Takahashi's Property) $\forall e_1, e_2 \in E$.

$$e_1 \gg e_2 \Longrightarrow e_2 \gg e_1^*.$$

Proof. By induction on e_1. □

The following two results then follow in sequence:

Notation 6.8.11 (Relation Composition) *For any two relations* $\oplus$ *and* $\otimes$*, we write* $a \oplus b \otimes c$ *as a shorthand for* $(a \oplus b) \wedge (b \otimes c)$*.*

Lemma 6.8.12 (Parallel Reduction is Diamond) $\forall e_1, e, e_2 \in E.$

$$e_1 \ll e \gg e_2 \Longrightarrow (\exists e' \in E.\, e_1 \gg e' \ll e_2).$$

Proof. Take $e' = e^*$ and use Takahashi's property. □

Theorem 6.8.13 (CBN λ-U is Confluent) $\forall e_1, e, e_2 \in E.$

$$e_1 \longleftarrow^* e \longrightarrow^* e_2 \Longrightarrow (\exists e' \in E.\, e_1 \longrightarrow^* e' \longleftarrow^* e_2).$$

6.9 The Soundness of CBN λ-U Reductions under CBN λ-M Big-Steps

In this section, we show that CBN λ-U reductions preserve observational equivalence[4], where our notion of observation is simply the termination behavior of the level 0 λ-M big-step evaluation.

Recall from Chapter 5, Figure 5.3, that the CBN λ-M semantics is specified by a partial function $\_ \overset{n}{\hookrightarrow} \_ : E^n \rightarrow E^n$ as follows:

$$\frac{}{i \overset{n}{\hookrightarrow} i}\ \text{Int} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e}\ \text{Lam} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \lambda x.e \\ e[x := e_2] \overset{0}{\hookrightarrow} e_3 \end{array}}{e_1\ e_2 \overset{0}{\hookrightarrow} e_3}\ \text{App}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3}\ \text{Run} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x}\ \text{Var+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_3 \quad e_2 \overset{n+}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_3\ e_4}\ \text{App+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2}\ \text{Lam+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}\ \text{Brk} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+}{\hookrightarrow} \mathsf{run}\ e_2}\ \text{Run+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\tilde{}e_1 \overset{n++}{\hookrightarrow} \tilde{}e_2}\ \text{Esc++} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle}{\tilde{}e_1 \overset{1}{\hookrightarrow} e_2}\ \text{Esc.}$$

[4] A reduction semantics for a lambda calculus is generally not "equal" to a big-step semantics. For example, the reduction semantics for the lambda calculus can do "reductions under lambda", and the big-step semantics generally does not. The reader is referred to textbooks on the semantics for more detailed discussions [50, 103].

Definition 6.9.1 (Level 0 Termination) $\forall e \in E^0.$

$$e \Downarrow \overset{\Delta}{=} (\exists v \in V^0.\, e \overset{0}{\hookrightarrow} v).$$

Definition 6.9.2 (Observational Equivalence) *We define* $\approx_n \in E^n \times E^n$ *as follows:* $\forall n \in \mathbb{N}.\, \forall e_1, e_2 \in E^n.$

$$e_1 \approx_n e_2 \overset{\Delta}{=} \forall C \in \mathbb{C}.\, C[e_1], C[e_2] \in E^0 \Longrightarrow (C[e_1]\Downarrow \iff C[e_2]\Downarrow).$$

Remark 6.9.3 *The definition says that two terms are observationally equivalent exactly when they can be interchanged in every level 0 term without affecting the level 0 termination behavior of the term.*

Notation 6.9.4 *We will drop the U subscript from $\longrightarrow_U$ in the rest of this chapter.*

Theorem 6.9.5 (CBN λ-U Reduction is Sound under λ-M Big-Steps) $\forall n \in \mathbb{N}.\, \forall e_1, e_2 \in E^n.$

$$e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx_n e_2.$$

Proof. By the definition of $\approx_n$, to prove our goal

$$\forall n \in \mathbb{N}.\, \forall e_1, e_2 \in E^n.\, e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx_n e_2$$

is to prove

$$\forall n \in \mathbb{N}, C \in \mathbb{C}.\, \forall e_1, e_2 \in E^n.\, e_1 \longrightarrow e_2 \wedge C[e_1], C[e_2] \in E^0 \Longrightarrow (C[e_1]\Downarrow \iff C[e_2]\Downarrow).$$

Noting that by the compatibility of $\longrightarrow$, we know that $\forall n \in \mathbb{N}, C \in \mathbb{C}.\, \forall e_1, e_2 \in E^n.\, e_1 \longrightarrow e_2 \Longrightarrow C[e_1] \longrightarrow C[e_2]$, it is sufficient to prove a stronger statement:

$$\forall n \in \mathbb{N}, C \in \mathbb{C}.\, \forall e_1, e_2 \in E^n.\, C[e_1] \longrightarrow C[e_2] \wedge C[e_1], C[e_2] \in E^0 \Longrightarrow (C[e_1]\Downarrow \iff C[e_2]\Downarrow).$$

Noting further that $\forall n \in \mathbb{N}, C \in \mathbb{C}.\, \forall a, b \in E^n.\, a \equiv C[b] \in E^0 \Longrightarrow a \in E^0$, it is sufficient to prove an even stronger statement:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow (e_1\Downarrow \iff e_2\Downarrow).$$

This goal can be broken down into two parts:

S1

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow (e_1\Downarrow \Longrightarrow e_2\Downarrow),$$

and

S2

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow (e_2 \Downarrow\ \Longrightarrow e_1 \Downarrow\ ).$$

Let us consider S1. By definition of termination, it says:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0.\, e_1 \overset{0}{\hookrightarrow} v) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v)).$$

We will show that big-step evaluation is included in reduction (Lemma 6.9.6). Thus, to prove S2 it is enough to prove:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0.\, e_1 \longrightarrow^* v) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v)).$$

Confluence (Theorem 6.8.13) tell us that any two reduction paths are joinable, so we can weaken our goal as follows:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0, e_3 \in E.\, e_1 \longrightarrow^* v \longrightarrow^* e_3 \wedge e_2 \longrightarrow^* e_3) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v))$$

We will show (Lemmas 6.9.15 and Remark 6.8.7) that any reduction that starts from a value can only lead to a value (at the same level). Thus we can weaken further:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v, v_3 \in V^0.\, e_1 \longrightarrow^* v \longrightarrow^* v_3 \wedge e_2 \longrightarrow^* v_3) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v))$$

In other words, we already know that e_2 reduces to a value, and the question is really whether it *evaluates* to a value. Formally:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v_3 \in V^0.\, e_2 \longrightarrow^* v_3) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v)).$$

In fact, the original assumption is no longer necessary, and we will prove:

T1

$$\forall e_2 \in E^0.\, ((\exists v_3 \in V^0.\, e_2 \longrightarrow^* v_3) \Longrightarrow (\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v)).$$

Now consider S2. By definition of termination, it says:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0.\, e_2 \overset{0}{\hookrightarrow} v) \Longrightarrow (\exists v \in V^0.\, e_1 \overset{0}{\hookrightarrow} v)).$$

Again, by the inclusion of evaluation in reduction, we can weaken:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0.\, e_2 \longrightarrow^* v) \Longrightarrow (\exists v \in V^0.\, e_1 \overset{0}{\hookrightarrow} v)).$$

Given the first assumption in this statement we can also say:

$$\forall e_1, e_2 \in E^0.\, e_1 \longrightarrow e_2 \Longrightarrow ((\exists v \in V^0.\, e_1 \longrightarrow^* v) \Longrightarrow (\exists v \in V^0.\, e_1 \overset{0}{\hookrightarrow} v)),$$

and we no longer need the assumption as it is sufficient to show:

T2

$$\forall e_1 \in E^0. ((\exists v \in V^0. e_1 \longrightarrow^* v) \Longrightarrow (\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v)).$$

But note that T1 and T2 are identical goals. They state:

T

$$\forall e \in E^0. ((\exists v \in V^0. e \longrightarrow^* v) \Longrightarrow (\exists v \in V^0. e \overset{0}{\hookrightarrow} v)).$$

This statement is a direct consequence of Lemma 6.9.7. □

It is easy to show that $e^0 \overset{0}{\hookrightarrow} v \Longrightarrow e^0 \longrightarrow^* v$, as it follows directly from the following result:

Lemma 6.9.6 (CBN λ-M is in CBN λ-U) $\forall n \in \mathbb{N}. \forall e \in E^n, v \in V^n.$

$$e \overset{n}{\hookrightarrow} v \Longrightarrow e \longrightarrow^* v.$$

Proof. By a straightforward induction on the height of the judgement $e \overset{n}{\hookrightarrow} v$. □

What is harder to show is the "converse", that is, that $e^0 \longrightarrow^* v \Longrightarrow (\exists v' \in V^0. e^0 \overset{0}{\hookrightarrow} v')$. It is a consequence of the following stronger result:

Lemma 6.9.7 (CBN λ-U is in CBN λ-M) $\forall e \in E^0, v_1 \in V^0.$

$$e \longrightarrow^* v_1 \Longrightarrow (\exists v_3 \in V^0. e \overset{0}{\hookrightarrow} v_3 \longrightarrow^* v_1).$$

Proof. We arrive at this result by an adaptation of Plotkin's proof for a similar result for the CBV and CBN lambda calculi [68]. The main steps in the development are:

1. We strengthen our goal to become:

$$e \longrightarrow^* v_1 \Longrightarrow (\exists v_3 \in V^0. e \overset{0}{\hookrightarrow} v_3 \longrightarrow^* v_1).$$

2. We define a *left reduction function* $\overset{n}{\longmapsto}$ (Definition 6.9.10) such that (Lemma 6.9.13): $\forall e \in E^0, v \in V^0.$

$$e \overset{0}{\longmapsto}{}^* v \Longleftrightarrow e \overset{0}{\hookrightarrow} v$$

and $\forall e_1, e_2 \in E^0. e_1 \overset{0}{\longmapsto} e_2 \Longrightarrow e_1 \longrightarrow e_2$ (Lemma 6.9.12). Thus, big-step evaluation (or simply evaluation) is exactly a chain of left reductions that ends in a value.

3. Our goal is restated as:

$$e \longrightarrow^* v_1 \Longrightarrow (\exists v_3 \in V^0. e \overset{0}{\longmapsto}{}^* v_3 \longrightarrow^* v_1).$$

4. For technical reasons, the proofs are simpler if we use a parallel reduction relation $\gg$ (Definition 6.9.14) similar to the one introduced in the last section. Our goal is once again restated as:

$$e \gg^* v_1 \Longrightarrow (\exists v_3 \in V^0.\, e \overset{0}{\longmapsto}{}^* v_3 \gg^* v_1).$$

5. The left reduction function induces a very fine classification on terms (Definition 6.9.8). In particular, any term $e \in E^n$ must be exactly one of the following three (Lemma 6.9.9):
   (a) a *value* $e \in V^n$,
   (b) a *workable* $e \in W^n$, or
   (c) a *stuck* $e \in S^n$,

   where membership in each of these three sets is defined inductively over the structure of the term. We write v^n, w^n and s^n to refer to a member of one of the three sets above, respectively. Left reduction at level n is a total function exactly on the members of the set W^n (Lemma 6.9.11). Thus, left reduction is strictly undefined on non-workables, that is, it is undefined on values and on stuck terms. Furthermore, if the result of any parallel reduction is a value, the source must have been either a value or a workable (Lemma 6.9.15). We will refer to this property of parallel reduction as *monotonicity.*
6. Using the above classification, we break our goal into two cases, depending on whether the starting point is a value or a workable:
   G1 $\forall v_1, v \in V^0.$

$$v \gg^* v_1 \Longrightarrow (\exists v_3 \in V^0.\, v = v_3 \gg^* v_1),$$

   G2 $\forall w \in W^0, v \in V^0.$

$$w \gg^* v_1 \Longrightarrow (\exists v_3 \in V^0.\, w \overset{0}{\longmapsto}{}^+ v_3 \gg^* v_1).$$

   It is obvious that G1 is true. Thus, G2 becomes the current goal.
7. By the monotonicity of parallel reduction, it is clear that all the intermediate terms in the reduction chain $w^0 \gg^* v_1^0$ are either workables or values. Furthermore, workables and values do not interleave, and there is exactly one transition from workables to values in the chain. Thus, this chain can be visualized as follows:

$$w_1^0 \gg w_2^0 \gg ...w_{k-1}^0 \gg w_k^0 \gg v^0 \gg^* v_1^0.$$

   We prove that the transition $w_k^0 \gg v^0$ can be replaced by an evaluation (Lemma 6.9.18):
   R1 $\forall w \in W^0, v \in V^0.$

$$w \gg v \Longrightarrow (\exists v_2 \in V^0.\, w \overset{0}{\longmapsto}{}^+ v_2 \gg v).$$

   With this lemma, we know that we can replace the chain above by one where the evaluation involved in going from the last workable to the first value is explicit:

$$w_1^0 \gg w_2^0 \gg ...w_{k-1}^0 \gg w_k^0 \overset{0}{\longmapsto}{}^+ v_2^0 \gg^* v_1^0.$$

What is left is then to "push back" this information about the last workable in the chain to the very first workable in the chain. This is achieved by a straightforward iteration (by induction over the number of k of workables in the chain) of a result that we prove (Lemma 6.9.20):
R2 $\forall w_1, w_2 \in W^0, v_1 \in V^0.$

$$w_1 \gg w_2 \overset{0}{\longmapsto}^{+} v_1 \Longrightarrow (\exists v_2 \in V^0.\, w_1 \overset{0}{\longmapsto}^{+} v_2 \gg v_1).$$

With this result, we are able to move the predicate $\_ \overset{0}{\longmapsto}^{+} v_3^0 \gg^* v^0$ all the way back to the first workable in the chain. This step can be visualized as follows. With one application of R2 we have the chain:

$$w_1^0 \gg w_2^0 \gg ... w_{k-1}^0 \overset{0}{\longmapsto}^{+} v_3^0 \gg^* v_1^0,$$

and with $k-2$ applications of R2 we have:

$$w_1^0 \overset{0}{\longmapsto}^{+} v_{k+1}^0 \gg^* v_1^0,$$

thus completing the proof.

$\square$

In the rest of this section, we present the definitions and lemmas mentioned above. It should be noted that proving most of the lemmas mentioned above require generalizing the level from 0 to n. In the rest of the development, we present the generalized forms, which can be trivially instantiated to the statements mentioned above.

6.9.1 A Basic Classification of Terms

Definition 6.9.8 (Classes) *We define three judgements on raw (that is, type-free) classes: Values V^n, Workables W^n, and Stuck terms S^n. The four sets are defined as follows:*

$$\begin{array}{lll}
v^0 & \in V^0 & := \lambda x.e^0 \mid \langle v^1 \rangle \\
v^1 & \in V^1 & := x \mid \lambda x.v^1 \mid v^1 v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1 \\
v^{n++} & \in V^{n++} & := x \mid \lambda x.v^{n++} \mid v^{n++} v^{n++} \mid \langle v^{n+++} \rangle \mid \tilde{}\, v^{n+} \mid \mathsf{run}\ v^{n++} \\
w^0 & \in W^0 & := (\lambda x.e^0)\ e^0 \mid w^0\ e^0 \mid \mathsf{run}\ \langle e^0 \rangle \mid \mathsf{run}\ w^0 \mid \langle w^1 \rangle \\
w^1 & \in W^1 & := \lambda x.w^1 \mid w^1\ e^1 \mid v^1\ w^1 \mid \langle w^2 \rangle \mid \tilde{}\, w^0 \mid \tilde{}\, \langle e^0 \rangle \\
w^{n++} & \in W^{n++} & := \lambda x.w^{n++} \mid w^{n++}\ e^{n++} \mid v^{n++}\ w^{n++} \mid \langle w^{n+++} \rangle \mid \tilde{}\, w^{n+} \\
s^0 & \in S^0 & := x \mid s^0\ e^0 \mid \langle v^1 \rangle\ e^0 \mid \langle s^1 \rangle \mid \mathsf{run}\ \lambda x.e^0 \mid \mathsf{run}\ s^0 \\
s^1 & \in S^1 & := \lambda x.s^1 \mid s^1\ e^1 \mid v^1\ s^1 \mid \langle s^2 \rangle \mid \tilde{}\, s^0 \mid \tilde{}\, \lambda x.e^0 \mid \mathsf{run}\ s^1 \\
s^{n++} & \in S^{n++} & := \lambda x.s^{n++} \mid s^{n++}\ e^{n++} \mid v^{n++}\ s^{n++} \mid \langle s^{n+++} \rangle \mid \tilde{}\, s^{n+} \mid \mathsf{run}\ s^{n++}.
\end{array}$$

Lemma 6.9.9 (Basic Properties of Classes) $\forall n \in \mathbb{N}$.

1. $V^n, W^n, S^n \subseteq E^n$

2. V^n, W^n, S^n partition E^n.

Proof. All these properties are easy to prove by straightforward induction.

1. We verify the claim for each case separately, by induction on $e \in V^n$, $e \in W^n$, and $e \in S^n$, respectively.
2. We prove that, $e \in E^n$ is in exactly one of the three sets V^n, W^n or S^n. The proof is by induction on the judgement $e \in W^n$. This proof is direct albeit tedious.

□

6.9.2 Left Reduction

The notion of left reduction is intended to capture precisely the reductions performed by the big-step semantics, in a small-step manner. Note that the simplicity of the definition depends on the fact that the partial function being defined is *not* defined on values. That is, we expect that there is no e such that $v^n \overset{n}{\longmapsto} e$.

Definition 6.9.10 (Left Reduction) *Left reduction is a partial function* $\_ \overset{n}{\longmapsto} \_ : E^n \to E^n$ *defined as follows:*

1. $$\frac{}{(\lambda x.v_1^1)\ v_2^1 \overset{0}{\longmapsto} v_1^1[x := v_2^1]}$$

2. $$\frac{}{\mathsf{run}\ \langle v^1 \rangle \overset{0}{\longmapsto} v^1}$$

3. $$\frac{}{\tilde{}\langle v^1 \rangle \overset{1}{\longmapsto} v^1}$$

4. $$\frac{e \overset{n+}{\longmapsto} e'}{\lambda x.e \overset{n+}{\longmapsto} \lambda x.e'}$$

5. $$\frac{e_1 \overset{n}{\longmapsto} e_1'}{e_1\ e_2 \overset{n}{\longmapsto} e_1'\ e_2}$$

6. $$\frac{e_2 \overset{n+}{\longmapsto} e_2'}{v_1^{n+}\ e_2 \overset{n+}{\longmapsto} v_1^{n+}\ e_2'}$$

7. $$\frac{e \overset{n+}{\longmapsto} e'}{\langle e \rangle \overset{n}{\longmapsto} \langle e' \rangle}$$

8. $$\frac{e \overset{n}{\longmapsto} e'}{\tilde{}e \overset{n+}{\longmapsto} \tilde{}e'}$$

9. $$\frac{e \overset{n}{\longmapsto} e'}{\mathsf{run}\ e \overset{n}{\longmapsto} \mathsf{run}\ e'}\ .$$

The following lemma says that the set of workables characterizes exactly the set of terms that can be advanced by left reduction.

Lemma 6.9.11 (Left Reduction and Classes) $\forall n \in \mathbb{N}.$

1. $\forall w \in W^n. (\exists e' \in E^n. w \overset{n}{\longmapsto} e')$
2. $\forall e \in E^n. (\exists e' \in E^n. e \overset{n}{\longmapsto} e') \Longrightarrow e \in W^n$
3. $\forall v \in V^n. \neg(\exists e' \in E^n. v \overset{n}{\longmapsto} e')$
4. $\forall s \in S^n. \neg(\exists e' \in E^n. s \overset{n}{\longmapsto} e')$.

Proof. We only need to prove the first two, and the second two follow. The first one is by straightforward induction on the judgement $e \in W^n$. The second is also by straightforward induction on the derivation $e^n \overset{n}{\longmapsto} e'$. □

Lemma 6.9.12 (Left Reduction and CBN λ-U) $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n.$

$$e_1 \overset{n}{\longmapsto} e_2 \Longrightarrow e_1 \longrightarrow e_2.$$

Proof. By straightforward induction on the first judgement. □

Lemma 6.9.13 (Left Reduction and λ-M) $\forall n \in \mathbb{N}. \forall e \in E^n, v \in V^n.$

$$e \overset{n}{\longmapsto}{}^{*} v \Longleftrightarrow e^n \overset{n}{\hookrightarrow} v^n.$$

Proof. The forward direction is by induction on the length of the derivation, and then over the size of e. The second ordering is not needed in the lambda calculus, but is needed for evaluation at higher levels. The proof proceeds by a case analysis on the first left reduction in the left reduction chain. The backward direction is by straightforward induction on the height of the derivation of $e \overset{n}{\hookrightarrow} v$. □

6.9.3 Parallel Reduction with Complexity

In order to prove the two key lemmas presented in this section, we will need to reason by induction on the "complexity" of parallel reduction. Thus, we will use the following definition of parallel reduction with an associated complexity measure:

Definition 6.9.14 (Parallel Reduction with Complexity) *Parallel reduction* $\_ \overset{M}{\gg} \_ : E \to E$ *defined as follows:*

1. $$\frac{}{x \overset{0}{\gg} x}$$

2. $$\frac{e_1^0 \overset{M}{\gg} e_1' \quad e_2^0 \overset{N}{\gg} e_2'}{(\lambda x.e_1^0)\ e_2^0 \overset{M+\#(x,e_1')N+1}{\gg} e_1'[x := e_2']}$$

3. $$\frac{e_1^0 \overset{M}{\gg} e_1'}{\mathsf{run}\ \langle e_1^0\rangle \overset{M+1}{\gg} e_1'}$$

4. $$\frac{e_1^0 \overset{M}{\gg} e_1'}{\tilde{}\langle e_1^0\rangle \overset{M+1}{\gg} e_1'}$$

5. $$\frac{e_1 \overset{M}{\gg} e_1'}{\lambda x.e_1 \overset{M}{\gg} \lambda x.e_1'}$$

6. $$\frac{e_1 \overset{M}{\gg} e_1'}{\langle e_1\rangle \overset{M}{\gg} \langle e_1'\rangle}$$

7. $$\frac{e_1 \overset{M}{\gg} e_1'}{\tilde{}e_1 \overset{M}{\gg} \tilde{}e_1'}$$

8. $$\frac{e_1 \overset{M}{\gg} e_1'}{\mathsf{run}\ e_1 \overset{M}{\gg} \mathsf{run}\ e_1'}$$

9. $$\frac{e_1 \overset{M}{\gg} e_1' \quad e_2 \overset{N}{\gg} e_2'}{e_1\ e_2 \overset{M+N}{\gg} e_1'\ e_2'},$$

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

There is a sense in which parallel reduction should respect the classes. The following lemma explicates these properties.

Lemma 6.9.15 (Parallel Reduction and Classes) $\forall n \in \mathbb{N}$.

1. $\forall e_1 \in E^n, e_2 \in E.\, e_1 \overset{M}{\gg} e_2 \Longrightarrow e_2 \in E^n$
2. $\forall e \in E^n, v \in V^n.\, v \overset{M}{\gg} e \Longrightarrow e \in V^n$
3. $\forall e \in E^n, s \in S^n.\, s \overset{M}{\gg} e \Longrightarrow e \in S^n$
4. $\forall e \in E^n, w \in W^n.\, e \overset{M}{\gg} w \Longrightarrow e \in W^n$.

Proof. The first part of this lemma is proved by straightforward induction on the height of the reduction derivation. It is then enough to establish the second two parts of this lemma, and then the fourth part follows immediately. The proof of the first two is also by a straightforward induction on the derivations of $e \in V^n$ and $e \in S^n$, respectively. $\square$

Remark 6.9.16 *We have already shown that parallel reduction without complexity is equivalent (in many steps) to normal reduction (in many steps). The same result applies to this annotated definition.*

Lemma 6.9.17 (Substitution for Parallel Reduction with Complexity) $\forall e_4, e_5, e_6, e_7 \in E$, $X, Y \in \mathbb{N}$.

$$e_4 \overset{X}{\gg} e_5 \wedge e_6^0 \overset{Y}{\gg} e_7 \Longrightarrow (\exists Z \in \mathbb{N}.\, e_4[x := e_6^0] \overset{Z}{\gg} e_5[x := e_7] \wedge Z \leq X + \#(x, e_5)Y).$$

Proof. By induction on the height of the derivation $e_4 \overset{X}{\gg} e_5$. (A direct extension of the proof for Lemma 5 on page 137 of Plotkin [68].) □

Lemma 6.9.18 (Transition) $\forall n, X \in \mathbb{N}. \forall w \in W^n, v \in V^n$.

$$w \overset{X}{\gg} v \Longrightarrow (\exists v_2 \in E^n, Y \in \mathbb{N}.\, w \xrightarrow{n}{}^{+} v_2 \overset{Y}{\gg} v) \wedge Y < X.$$

Proof. By induction on the complexity X and then on the size of w. (A direct combination and extension of proofs for Lemmas 6 and 7 of Plotkin [68].) □

Lemma 6.9.19 (Permutation) $\forall n, X \in \mathbb{N}. \forall w_1, w_2 \in W^n, e_1 \in E^n$.

$$w_1 \overset{X}{\gg} w_2 \xrightarrow{n} e_1 \Longrightarrow (\exists e_2 \in E^n.\, w_1 \xrightarrow{n}{}^{+} e_2 \gg e_1).$$

Proof. By induction on the complexity X, and by a case analysis on the last case of the derivation of $w_1 \overset{X}{\gg} w_2$. (A direct extension of Lemmas 8 of the previous reference.) □

Lemma 6.9.20 (Push Back) $\forall X \in \mathbb{N}, w_1, w_2 \in W^0, v_2 \in V^0$.

$$w_1 \overset{X}{\gg} w_2 \xrightarrow{0}{}^{+} v_1 \Longrightarrow (\exists v_2 \in V^0.\, w_1 \xrightarrow{0}{}^{+} v_2 \gg v_1).$$

Proof. The assumption corresponds to a chain of reductions:

$$w_1 \gg w_2 \xrightarrow{0} w_3 \xrightarrow{0} \ldots w_{k-1} \xrightarrow{0} w_k \xrightarrow{0} v_1.$$

Applying Permutation to $w_1 \gg w_2 \xrightarrow{0} w_3$ gives us $(\exists e_{2'} \in E^n.\, w_1 \xrightarrow{0}{}^{+} e_{2'} \gg w_3)$. By the monotonicity of parallel reduction, we know that only a workable can reduce to a workable, that is, $(\exists w_{2'} \in W^n.\, w_1 \xrightarrow{0}{}^{+} w_{2'} \gg w_3)$. Now we have the chain:

$$w_1 \xrightarrow{0}{}^{+} w_{2'} \gg w_3 \xrightarrow{0} \ldots w_{k-1} \xrightarrow{0} w_k \xrightarrow{0} v_1.$$

Repeating this step $k-2$ times we have:

$$w_1 \xrightarrow{0}{}^{+} w_{2'} \xrightarrow{0}{}^{+} w_{3'} \xrightarrow{0}{}^{+} \ldots w_{k-1'} \gg w_k \xrightarrow{0} v_1.$$

Applying Permutation to $w_{k-1'} \gg w_k \xrightarrow{0} v_1$ give us $(\exists e_{k'} \in E^n.\, w_{k-1'} \xrightarrow{0}{}^{+} e_{k'} \gg v_1)$. By the monotonicity of parallel reduction, we know that $e_{k'}$ can only be a value or a workable. If it is a value then we have the chain:

$$w_1 \xrightarrow{0}{}^{+} w_{2'} \xrightarrow{0}{}^{+} w_{3'} \xrightarrow{0}{}^{+} \ldots w_{k-1'} \xrightarrow{0}{}^{+} v_{k'} \gg v_1$$

and we are done. If it is a workable, then applying Transition to $w_{k'} \gg v_1$ gives us $(\exists v_2 \in V^n.\, w_{k'} \xrightarrow{0}{}^{+} v_2 \gg v_1)$. This means that we now have the chain:

$$w_1 \xrightarrow{0}{}^{+} w_{2'} \xrightarrow{0}{}^{+} w_{3'} \xrightarrow{0}{}^{+} \ldots w_{k-1'} \xrightarrow{0}{}^{+} w_{k'} \xrightarrow{0}{}^{+} v_2 \gg v_1$$

and we are done. □

6.9.4 Concluding Remarks

Remark 6.9.21 (Equational Theory) *It is often the case that an equational theory for a language will look very similar to a reduction semantics. We should therefore point out that we expect that all the reductions of λ-U to hold as equalities. Such an equational theory is useful for reasoning about programs in general, and for proving the equivalence of two programs in particular. The formal development and the practical utility of such an equational theory are still largely unexplored.*

Remark 6.9.22 (The Stratification of Expressions) *It is necessary to stratify the set of expressions into expression families. In particular, our notions of reduction are certainly* not *sound if we do not explicitly forbid the application of the big-step semantic function on terms that are manifestly not at the right level. In particular, consider the term $\tilde{}\langle i\rangle \in E^1$. If this term is subjected to the big-step semantic function at level 0, the result is undefined. However, if we optimize this term using the Escape reduction, we get back the term i, for which the big-step semantics* is *defined. As such, the stratification of the expressions is crucial to the correctness of our notions of reduction.*

Remark 6.9.23 (Non-left or "Internal" Reductions) *Many standardization proofs (such as those described by Barendregt [1] and Takahashi [96]) employ "complementary" notions of reduction, such as internal reduction (defined simply as non-head). The development presented in Plotkin (and here), does not require the introduction of such notions. While they do posses interesting properties in our setting (such as the preservation of all classes), they are not needed for the proofs. Machkasova and Turbak [46] also point out that complementary reductions preserve all classes. Using such complementary notions, it may be possible to avoid the use of Plotkin's notion of complexity, although the rest of the proof remains essentially the same. We plan to further investigate this point in future work.*

Remark 6.9.24 (Classes) *Plotkin [68] only names the set of values explicitly. The notions of workables and stuck terms employed in this present work*[5] *helped us adapt Plotkin's technique to MetaML, and in some cases, to shorten the development. For example, we have combined Plotkin's Lemmas 6 and 7 into one (Lemma 6.9.18). We expect that our organization of expressions into values, workables, and stuck terms may also be suitable for applying Plotkin's technique to other programming languages.*

Remark 6.9.25 (Standardization) *We have not found need for a Standardization Theorem, or for an explicit notion of standard reduction. Also, our development has avoided Lemma 9 of Plotkin [68], and the non-trivial lexicographic ordering needed for proving that lemma.*

[5] *Such a classification has also been employed in a work by Hatcliff and Danvy [34], where values and stuck terms are named. At the time of writing these results, we had not found a name for "workables" in the literature.*

Remark 6.9.26 (Soundness of CBV λ-U) *An additional degree of care is needed in the treatment of CBV λ-U. In particular, the notion of value induced by the big-step semantics for a call-by-value lambda language is not the same as the notion of value used in the reduction semantics for call-by-value languages. The latter typically contains variables. This subtle difference will require distinguishing between the two notions throughout the soundness proof.*

Part III

Appraisal and Recommendations

Chapter 7

Discussion and Related Works

The Introduction explained the motivation for the study of MetaML and multi-stage languages. Part I explained the basics of MetaML and how it can be used to develop multi-stage programs. Part II explained the need for the formal study of the semantics of MetaML, and presented the main technical results of our work. This chapter expands on some points that would have elsewhere distracted from the essentials of our argument.

The first three sections parallel the organization of the dissertation. The Introduction section reviews the motivation for studying manual staging, and explains how this dissertation allows us to formalize the concept of a stage, which was an informal notion in the Introduction. The Part I section reviews the current state of MetaML, and presents an explanation of why lambda-abstraction is not enough for staging. In this section, we also discuss the practical problem of cross-stage portability that having cross-stage persistence creates. The Part II section discusses the related works multi-level specialization and multi-level languages, and positions our work in this context. A final section reviews snapshots from the history of quasi-quotation in formal logic, LISP, and Prolog.

7.1 Introduction

7.1.1 Why Manual Staging?

Given that partial evaluation performs staging automatically, it is reasonable to ask why manual staging is of interest. There is a number of reasons why manual staging is both interesting and desirable:

Foundational: As we have seen in this dissertation, the subtlety of the semantics of annotated programs warrants studying them in relative isolation, and without the added complexity of other partial evaluation issues such as BTA.

Pedagogical: Explaining the concept of staging to programmers is a challenge. For example, it is sometimes hard for new users to understand the workings of partial evaluation systems [38]. New users often lack a good mental model of how partial evaluation systems work. Furthermore, new users are often uncertain:

- What is the output of a binding-time analysis?
- What are the annotations? How are they expressed?
- What do they really mean?

The answers to these questions are crucial to the effective use of partial evaluation. Although BTA is an involved process that requires special expertise, the annotations it produces are relatively simple and easy to understand. Our observation is that programmers can understand the annotated output of BTA, without actually knowing how BTA works. Having a programming language with explicit staging annotations would help users of partial evaluation understand more of the issues involved in staged computation, and, hopefully, reduce the steep learning curve currently associated with using a partial evaluator effectively [40].

Pragmatic (Performance): Whenever performance is an issue, control of evaluation order is important. BTA optimizes the evaluation order given the time of arrival of inputs, but sometimes it is just easier to say what is wanted, rather than to force a BTA to discover it [39]. Automatic analyses such as BTA are necessarily incomplete, and can only approximate the knowledge of the programmer. By using explicit annotations, the programmer can exploit his full knowledge of the program domain. In a language with manual staging, having explicit annotations can offer the programmer a well designed back-door for dealing with instances when the automatic analysis reaches its limits.

Pragmatic (Termination and Effects): Annotations can alter termination behavior in two ways: 1) specialization of an annotated program can fail to terminate, and 2) the generated program itself might have termination behavior differing from that of the original program [40]. While such termination questions are the subject of active investigation in partial evaluation, programming with explicit annotation gives the user complete control over (and responsibility for) termination behavior in a staged system. For example, any recursive program can be annotated with staging annotations in two fundamentally different ways. Consider the power function. The first way of annotating it is the one which we have discussed in this dissertation:

```
fun exp' (n,x) = (* : int × ⟨real⟩ → ⟨real⟩ *)
```

```
   if n = 0 then ⟨1.0⟩
            else if even n then sqr' (exp' (n div 2,x))
                          else ⟨x * ~(exp' (n - 1,x))⟩.
```

The second way of annotating it is as follows:

```
 fun exp' (n,x) = (* : int × ⟨real⟩ → ⟨real⟩ *)
   ⟨if n = 0 then 1.0
             else if even n then ~(sqr' (exp' (n div 2,x)))
                           else x * ~(exp' (n - 1,x))⟩.
```

Intuitively, all we have done is "factored-out" the Brackets from the Branches of the if-statement to one Bracket around the whole if-statement. This function is perfectly well-typed, but the annotations have just created a non-terminating function out of a function that was always terminating (at least for powers of 0 or more). When applied, this function simply makes repeated calls to itself, constructing bigger and bigger code fragments. In partial evaluation, this problem is known as *infinite unfolding*, and partial evaluation systems must take precautions to avoid it. In MetaML, the fact that there are such anomalous annotations is not a problem, because the programmer specifies explicitly where the annotations go. In particular, whereas with partial evaluation an automatic analysis (BTA) can alter the termination behavior of the program, with multi-stage programming the *programmer* is the one who has both control over, and responsibility for, the correctness of the termination behavior of the annotated program.

7.1.2 The Notion of a Stage

In the introduction, we gave the intuitive explanation for a stage. After presenting the semantics for MetaML, we can now provide a more formal definition. We define (the trace of) *a stage* as *the derivation tree generated by the invocation of the derivation* run $e \overset{0}{\hookrightarrow} v$. (See the Run rule in Chapter 5.) Note that while the notion of a level is defined with respect to *syntax*, the notion of a stage is defined with respect to *a trace of an operational semantics*. Although quite intuitive, this distinction was not always clear to us, especially that there does not seem to be any comparable definition in the literature with respect to an operational semantics.

The levels of the subterms of a program and the stages involved in the execution of the program can be unrelated. A program ⟨1+run ⟨4+2⟩⟩ has expressions at levels 0, 1, and 2. If we define the "level of a program" as the maximum level of any of its subexpressions, then this is a 2-level program. The evaluation of this expression (which just involves rebuilding it), involves no derivations run $e \overset{0}{\hookrightarrow} v$. On the other hand, the evaluation of slightly modified 2-level program run ⟨1+run ⟨4+2⟩⟩ involves two stages.

To further illustrate the distinction between levels and stages, let us define the *number of stages* of a program as the number of times the derivation run $e \overset{0}{\hookrightarrow} v$ is used in its evaluation[1]. Consider:

(fn x ⇒ if P then x else lift(run x)) ⟨1+2⟩.

where P is an arbitrary problem (in other words, a possibly non-terminating program). The number of stages in this program is not statically decidable. Furthermore, we cannot say, in general, which occurrence of Run will be ultimately responsible for triggering the computation of the addition in expression ⟨1+2⟩.

Recognizing this mismatch was a useful step towards finding a type-system for MetaML, which employs the static notion of level to approximate the dynamic notion of stage.

7.1.3 Code Cannot be Added to SML as a Datatype

The simple interpreter for MetaML discussed in Chapter 4 uses a datatype to implement the code type constructor. An interesting question is whether we can define some similar datatype in SML *and* then use the constructors of this datatype in place MetaML's Brackets and Escapes. If this were possible, then we could either make the datatype used in the interpreter available to the object language, or, we can avoid the need for having to implement a full interpreter for MetaML altogether. Unfortunately, there is a number of reasons why MetaML's code type constructor cannot be added to SML as a datatype.

To explain these reasons, assume that such a datatype exists and has some declaration of the form:

```
datatype 'a code = Int of ...
                 | Var of ...
                 | Lam of ...
                 | App of ...
                 | ...
```

Essentially every single variant of such a datatype would contradict some basic assumptions about datatype constructors. To see this, recall that any SML datatype construct has the following type:

$$\mathsf{Constructor} : \mathsf{t}['\mathsf{a}_\mathsf{i}] \rightarrow {'\mathsf{a}_\mathsf{i}}\ \mathsf{T}$$

where $\mathsf{t}['\mathsf{a}_\mathsf{i}]$ stands for a type term that is closed except for the variables $'\mathsf{a}_1,..,'\mathsf{a}_\mathsf{i}$. For any SML datatype, we also get a deconstructor with the following type:

[1] This is an upper bound on what one may wish to define as the number of *sequential* stages in a multi-stage computation. For example, elsewhere we have defined the number of stages based on a data-flow view of computation [93]. The definition given here is simplistic, but is sufficient for illustrating our point.

$$\mathsf{Deconstructor} : {}'\mathsf{a_i}\,\mathsf{T} \to \mathsf{t}[{}'\mathsf{a_i}]$$

Integers Now consider the case of integers. The integer variant EI should have the type

$$\mathsf{EI} : \mathsf{int} \to \mathsf{int\,code}$$

The return type is not polymorphic enough: We wish to define a datatype $'\mathsf{a\ code}$ and the return type does not cover the whole datatype. This problem is more clear when we consider the type of the deconstructor:

$$\mathsf{DeEI} : \mathsf{int\,code} \to \mathsf{int}$$

This deconstructor is only well-typed for $\mathsf{int\,code}$ values. This simple fact means that we cannot express a polymorphic identity function for $'\mathsf{a\,code}$ that works by taking apart a code fragment and putting it back together.

Variables The variable variant EV can be expected to have the type

$$\mathsf{EV} : \mathsf{t\,var} \to \mathsf{t\,code}$$

Again, we run into a problem similar to the one above, because the target type is not completely polymorphic. Furthermore, we will need to introduce an explicit notion of variables in the form of another type constructor $'\mathsf{a\ var}$. It is not obvious whether such a type constructor can be introduced in the form of a datatype. It is not even clear that such a type constructor can be introduced in a consistent manner.

Lambda-abstraction The lambda-abstraction variant EL can be expected to have the type

$$\mathsf{EL} : {}'\mathsf{a\,var} \times ({}'\mathsf{b\,code}) \to ({}'\mathsf{a} \to {}'\mathsf{b})\,\mathsf{code}$$

Again, we run into the problem of the target type being not fully covered. In addition, the first occurrence of $'\mathsf{b}$ is more complex than it appears. In particular, it should be possible that the second argument to EL be an open expression, where the variable bound by the first argument can occur free. Thus, it is very likely that the type $'\mathsf{b}\,\mathsf{code}$ would be insufficient for describing such a fragment, as the fact that it is a $'\mathsf{b}\,\mathsf{code}$ is conditioned by (at least) the fact that the free variable bound by this lambda-abstraction has the type $'\mathsf{a}$. It is not clear how this can be accomplished without introducing additional substantial machinery into our meta-language (SML).

Application Finally, the application variant EA can be expected to have the type

$$\mathsf{EA} : ('\mathsf{a} \to {'\mathsf{b}})\,\mathsf{code} \times {'\mathsf{a}}\,\mathsf{code} \to {'\mathsf{b}}\,\mathsf{code}$$

Here, we do not run into the same problem as above: the target type covers the whole domain. But there is still a problem: $'\mathsf{a}$ is a free type variable in the type of the co-domain, and does not appear in the type of the domain. It may be possible to view $'\mathsf{a}$ as an existentially quantified type, but it is not obvious how this would complicate the treatment of this datatype.

7.2 Part I: The Practice of Multi-Stage Programming

Sheard developed the original design and implementation of MetaML, a language combining a host of desirable language features, including:

- Staging annotations
- Static typing
- Hindley-Milner polymorphism
- Type inference.

The primary goal of the design is to provide a language well-suited for writing program generators. Two implementations of MetaML have been developed. The first was developed by Sheard between 1994 and 1996. This interpreter implemented a pure CBV functional language with polymorphic type inference, and support for recursive functions, SML-style datatypes, and the four staging constructs studied in this dissertation. The first implementation was based largely on an implementation of CRML. (See Section 7.2.3.) The development of the second implementation by Sheard, Taha, Benaissa and Pasalic started in 1996 and continues until today. This interpreter aims at incorporating full SML and extending it with the four staging constructs. The highlights of MetaML are:

- **Cross-stage persistence.** The ability to use variables from any past stage is crucial to writing staged programs in the manner to which programmers are accustomed. Cross-stage persistence provides a solution to hygienic macros in a typed language, that is, macros that bind identifiers in the environment of definition, which are not "captured" in the environment of use.
- **Multi-stage aware type system.** The type checker reports staging errors as well as type errors. We have found the interactive type system to be very useful during staging.
- **Display of code.** When debugging, it is important for users to be able to read the code produced by their multi-stage programs. Supporting this MetaML feature requires a display mechanism (pretty-printer) for values of type code.
- **Display of constants.** The origin of a cross-stage persistent constant can be hard to identify. The named `%_` tags provide an approximation of where these constants came from. While these tags can sometimes be misleading, they are often quite useful.
- **The connection between $\langle \mathsf{A} \rangle \to \langle \mathsf{B} \rangle$ and $\langle \mathsf{A} \to \mathsf{B} \rangle$.** Having the two mediating functions back and forth reduces the number of annotations needed to stage programs.
- **Lift.** The Lift annotation makes it possible to force computation in an early stage and Lift this value into a program to be incorporated at a later stage. While it may seem that cross-stage persistence makes Lift unnecessary, Lift helps producing code that is easier to understand, because constants become explicit.
- **Safe β and rebuilding optimizations.** These optimizations improve the generated code, and often make it more readable.

7.2.1 Why Lambda-Abstraction is not Enough for Multi-Stage Programming

It may appear that staging requires only "delay" and "force" operations (see for example Okasaki or Wadler *et al.* [65, 98],) which can be implemented by lambda-abstraction and application, respectively. While this may be true for certain domains, there are two capabilities that are needed for multi-stage programming and are not provided by "delay" and "force":

1. A delayed computation must maintain an intensional representation so that users can inspect the code produced by their generators, and so that it can be either printed or compiled. In a compiled implementation, lambda-abstractions lose their high-level intensional representation, and it becomes harder to inspect or print lambda-abstractions at run-time.
2. More fundamentally, code generators often need to perform "evaluation under lambda". Evaluation under lambda is necessary for almost any staged application that performs some kind of unfolding, and is used in functions such as back. It is not clear how the effect of Escape

(under lambda) can be imitated in the CBV λ-calculus without extending it with additional constructs.

To further explain the second point, we will show an example of the result of encoding of the operational semantics of MetaML in SML/NJ.

A Schema for Encoding MetaML in a CBV Language with Effects The essential ingredients of a program that requires more than abstraction and application for staging are Brackets, dynamic (non-level 0) abstractions, and Escapes. Lambda-abstraction over unit can be used to encode Brackets, and application to unit to encode Run. However, Escape is considerably more difficult to encode. In particular, the expression inside an Escape has to be executed *before* the surrounding delayed computation is constructed. Implementing such an encoding is difficult when variables introduced inside the delayed expression occur in the Escaped expression, as in terms such as ⟨fn x ⇒ ˜(f ⟨x⟩)⟩.

One way to imitate this behavior uses two non-pure SML features. References can be used to simulate evaluation under lambda, and exceptions to simulate the creation of uninitialized reference cells. Consider the following sequence of MetaML declarations:

```
fun G f = ⟨fn x ⇒ ˜(f ⟨x⟩)⟩
val pc = G (fn xc ⇒ ⟨(˜xc,˜xc)⟩)
val p5 = (run pc) 5.
```

The corresponding imitation in SML would be:

```
exception not_yet_defined
val undefined = (fn () ⇒ (raise not_yet_defined))
fun G f =
    let val xh = ref undefined
        val xc = fn () ⇒ !xh ()
        val nc = f xc
    in
       fn () ⇒ fn x ⇒ (xh:=(fn () ⇒ x);nc ())
    end;
val pc = G (fn xc ⇒ fn () ⇒ (xc(),xc()))
val p5 = (pc ()) 5.
```

In this translation, values of type ⟨'a⟩ are encoded by delayed computations of type () → 'a. We begin by assigning a lifted undefined value to undefined. Now we are ready to write the analog of the function G. Given a function f, the function G first creates an uninitialized reference cell xh. This reference cell corresponds to the occurrences of x in the application f ⟨x⟩ in the MetaML definition of G. Intuitively, the fact that xh is uninitialized corresponds to the fact that x will not yet be bound to a fixed value when the application f ⟨x⟩ is to be performed. This facility is very important in MetaML, as it allows us to unfold functions like f on "dummy" variables like x. The expression fn () ⇒ !xh () is a delayed lookup of xh. This delayed computation corresponds to the Brackets surrounding x in the expression f ⟨x⟩. Now, we simply perform the application of the function f to this delayed construction. It is important to note here that we are applying f as it is passed to the function G, before we know what value x is bound to. Finally, the body of the function G returns a delayed lambda-abstraction, which first assigns a delayed version of x to xh, and then simply includes an applied ("Escaped") version of nc in the body of this abstraction.

The transliteration illustrates the advantage of using MetaML rather than trying to encode multi-stage programs using lambda-abstractions, references, and exceptions. The MetaML version is shorter, more concise, looks like the unstaged version, and is easier to understand.

One might consider an implementation of MetaML based on this approach, hidden under some syntactic sugar to alleviate the disadvantages listed above. The lambda-delay method has the advantage of being a machine-independent manipulation of lambda terms. Unfortunately it fails to meet the intensional representation criterion, and also incurs some overhead not (necessarily) incurred in the MetaML version. In particular, the last assignment to the reference xh is delayed, and must be repeated every time the function returned by G is used. The same happens with the application ("Escaping") of nc. Neither of these expenses would be incurred by the MetaML version of G. Intuitively, these operations are being used to connect the meta-level variable x to its corresponding object-level xh. In MetaML, these overheads would be incurred exactly once during the evaluation of run pc as opposed to every time the function resulting from pc () is applied.

7.2.2 Cross-Stage Portability

Cross-stage persistence is a novel feature of MetaML that did not – to our knowledge – exist in any previous proposals for run-time code generation. This language feature seems highly desirable in run-time code generation systems, where there is generally little interest in inspecting a source level representation of programs. But for high-level program generation, cross-stage persistence comes at a price: Some parts of generated code fragment may not be printable. For example, let us consider the following simple SML/NJ session:

```
- 40+2;
```

```
val it = 42 : int
- fn x ⇒ x;
val it = fn :  'a →  'a.
```

The result of evaluating the first variable is printed back as 42, but not the result of the second. Because SML/NJ is a compiled implementation, the result of evaluating fn x ⇒ x is a structure containing some machine code. This structure is not printed back because it is machine-dependent, and is considered implementation detail. But independently of whether this structure should be printed or not, the source-level representation of our function is generally not maintained after compilation. The lack of high-level representations of values at run-time is the reason why "inlining" cross-stage persistent variables is generally not possible. For example, in the following MetaML session:

```
|- (fn y ⇒ ⟨y⟩) (fn x ⇒ x);
val it = ⟨%y⟩ : ⟨ 'a →  'a⟩.
```

we cannot return ⟨fn x ⇒ x⟩, because the source-level representation of fn x ⇒ x is simply lost at the point when the application is performed.

Loss of printability poses a practical problem if the first stage of a multi-stage computation is performed on one computer, and the second on another. In this case, we need to "port" the local environment from the first machine to the second. Since arbitrary objects, such as functions and closures, can be bound in this local environment, this embedding can cause portability problems. Currently, MetaML assumes that the computing environment does not change between stages, or more generally, that we are computing in an *integrated system*. Thus, current MetaML implementations lack *cross-platform portability*, but we believe that this limitation can be recovered through pickling and unpickling techniques.

7.2.3 Linguistic Reflection and Related MetaML Research

"Linguistic reflection is defined as the ability of a program to generate new program fragments and to integrate these into its own execution [89]." MetaML is a descendent of CRML [79, 80, 37], which in turn was greatly influenced by TRPL [77, 78]. All three of these languages support linguistic reflection. Both CRML and TRPL were two-stage languages that allowed users to provide compile-time functions (much like macros) which directed the compiler to perform compile-time reductions. Both emphasized the use of computations over representations of a program's datatype definitions. By generating functions from datatype definitions, it was possible to create specific instances of generic functions such as equality functions, pretty printers, and parsers [78]. This facility provided

an abstraction mechanism not available in traditional languages. MetaML improves upon these languages by adding hygienic variables, generalizing the number of stages, and emphasizing the soundness of its type system.

Sheard and Nelson investigate a two-stage language for the purpose of program generation [82]. The base language was statically typed, and dependent types were used to generate a wider class of programs than is possible by MetaML restricted to two stages. Sheard, Shields and Peyton-Jones [83] investigate a dynamic type system for multi-staged programs where some type obligations of staged computations can be put off till run-time.

7.3 Part II: The Theory of Multi-Stage Programming

7.3.1 Multi-Level Specialization

Glück and Jørgensen [29] introduced the idea of multi-level BTA (MBTA) as an efficient and effective alternative to multiple self-application. A multi-level language based on Scheme is used for the presentation. MetaML has fewer primitives than this language, and our focus is more on program generation issues rather than those of BTA. All intermediate results in their work are printable, that is, have an intensional representation. In MetaML, cross-stage persistence allows us to have intermediate results between stages that contain constants for which no intensional representation is available.

A second work by Glück and Jørgensen [30] demonstrates that MBTA can be done with efficiency comparable to that of two-level BTA. Their MBTA is implemented using constraint-solving techniques. The MBTA is type-based, but the underlying language is dynamically typed.

Glück and Jørgensen also study partial evaluation in the generalized context where inputs can arrive at an arbitrary number of times rather than just specialization-time and run-time in the context of a flow-chart language called S-Graph-n [31]. This language can be viewed as a dynamically typed multi-level programming language. S-Graph-n was not designed for human use (as a programming language), but rather, for being producing automatically by program generators. One of the contributions of this dissertation is emphasising that programmers can write useful multi-stage programs directly in an appropriate programming language (such as MetaML), and that while an automatic analyses such as BTA and MBTA can be very useful, they are not, strictly speaking, necessary for multi-stage programming.

Hatcliff and Glück study the issues involved in the implementation of a language like S-Graph-n [35]. The syntax of S-Graph-n explicitly captures all the information necessary for specifying the staging of a computation: each construct is annotated with a number indicating the stage during which it is to be executed, and all variables are annotated with a number indicating the stage of their availability. The annotations of this language were one of our motivations for studying level-annotations in MetaML. (See λ-T of Appendix A.) One notable difference is that the explicit level annotations of λ-T reflect "intended" usage-time as opposed to availability time. Availability in our formalisms has generally been reflected at the level of the type system, and in the typing environment. S-Graph-n is dynamically typed, and the syntax and formal semantics of the language are sizable. Programming directly in S-Graph-n would require the user to annotate every construct and variable with stage annotations, and ensuring the consistency of the annotations is the user's responsibility. These explicit annotations are not necessarily a serious drawback, as the language was intended primarily as an internal language for program transformation systems. However, we believe that further simplifying this language could make verifying the correctness of such program transformation systems easier. Finally, Hatcliff and Glück have also identified *language-independence* of the internal representation of "code" as an important characteristic of any multi-stage language.

Glück, Hatcliff and Jørgensen continue the study of S-Graph-n, focusing on issue of generalization of data in multi-level transformation systems (such as self-applicable partial evaluation) [28]. This work advocates S-Graph-n as an appropriate representation for meta-system hierarchies. In essence, a meta-system hierarchy is a sequence of meta-programs where each meta-program is manipulating the next program in the sequence. Roughly speaking, generalization (more precisely, finding the most specific generalization) is the process of finding the most precise characterization of an expression in terms its position in the hierarchy. The work identifies and addresses two fundamental problems that arise in when considering such hierarchies, namely the *space consumption problem* and the *invariance problem*. The space consumption problem arises due to the possibility of encoding object-programs multiple times in such a hierarchy. The space consumption problem cannot be seen in our work, because we have either used level-annotated terms (λ-T of Appendix A), which are similar in spirit to the S-Graph-n solution, or abolished the quest for distinguishing unencoded and encoded terms (λ-U of Chapter 6). The invariance problem arises when a program transformation is not invariant under the encoding operation. In MetaML, the invariance problem is roughly comparable to a transformation that works on a level 0 λ-T term, but cannot continue to work on the same term when it is promoted. Note that such problem *cannot* arise with λ-U terms, as unencoded and encoded terms are syntactically indistinguishable. The work on S-Graph-n shows how the two problems described above can be avoided using the multi-level data structures of S-Graph-n [35]. This feature of S-Graph-n is highly desirable for the success of multi-level transformations because generalization of data should be precise regardless of the number of levels involved in the multi-level transformation.

Because avoiding the use of level-annotated terms (as in λ-U) can simplify the language, it remains an interesting and open question weather abolishing the distinction between unencoded and encoded terms can also be applied to S-Graph-n. Furthermore, because the technical development of the notion of generalizations has some similarities with the problems with substitution that arose in the context of MetaML, it is reasonable to expect to reap more benefits if level annotations can be avoided in S-Graph-n.

7.3.2 Type Systems for Open and Closed Code

Typed languages for manipulating code fragments have typically had either a type constructor for open code [32, 22, 94], or a type constructor for closed code [62, 23, 102]. Languages with open code types are useful in the study of partial evaluation. Typically, they provide constructs for building and combining code fragments with free variables, but do not allow for executing such fragments. Being able to construct open fragments enables the user to force computations "under a lambda". Executing code fragments in such languages is hard because code can contain "not-yet-bound identifiers". In contrast, languages with closed code types have been advocated as models for run-time (machine) code generation. Typically, they provide constructs for building and executing code fragments, but do not allow for forcing computations "under a lambda".

In what follows, we review these languages in more detail.

7.3.3 Nielson & Nielson and Gomard & Jones

Nielson and Nielson pioneered the investigation of multi-level languages with their work on two-level functional languages [58, 62, 59, 60]. They have developed an extensive theory for the denotational semantics of two-level languages, including their use as a framework for abstract interpretation [61]. Their framework allows for a "B-level" language, where B is an arbitrary, possibly partially-ordered set. Recently, Nielson and Nielson have also proposed an algebraic framework for the specification of multi-level type systems [63, 64].

Gomard and Jones [32] proposed a statically-typed two-level language to explain the workings of a partial evaluator for the untyped λ-calculus. This language is the basis for many BTAs. It allows the treatment of expressions containing free variables. Our treatment of object-level variables in the implementation semantics is inspired by their work.

7.3.4 Multi-Level Languages and Logical Modalities

In our research, we have emphasized the pragmatic importance of being able to combine cross-stage persistence, "evaluation under lambda" (or "symbolic computation"), and being able to execute code. In this section, we review some of the basic features of two important statically-typed multi-level languages that are closely related to our work.

Davies and Pfenning present a statically-typed multi-stage language $\lambda^{\Box}$, motivated by constructive modal logic [23]. They show that there is a Curry-Howard isomorphism between $\lambda^{\Box}$ and the modal logic S4. They also show that $\lambda^{\Box}$ type system is equivalent to the binding-time analysis of Nielson and Nielson. The language provides a closed code type constructor $\Box$ that is closely related to the Closed type of λ^{BN}. The language has two constructs, box and let-box, which correspond roughly to Close and Open, respectively.

Davies extends the Curry-Howard isomorphism to a relation between linear temporal logic and the type system for a multi-level language [22]. He presents a language $\lambda^{\bigcirc}$ that allows staged expressions to contain free variables. The language provides an open code type constructor $\bigcirc$ that corresponds closely to MetaML's code type. The $\lambda^{\bigcirc}$ has two constructs next and prev that correspond closely to MetaML's Brackets and Escape, respectively.

In collaboration with Moggi, Benaissa, and Sheard, the present author presents *AIM* (An Idealized MetaML) [55], which extends MetaML with an analog of the Box type of $\lambda^{\Box}$ yielding a more expressive language, yet has a simpler typing judgment than MetaML. We can embed all three languages into *AIM* [55][2]. We view λ^{BN} as a cut-down version of *AIM* that we expect to be sufficiently expressive for the purposes of multi-stage programming.

The Closed type constructor of λ^{BN} is essentially a strict version of the Box type constructor of *AIM*. The present author proposed that the laziness of the Box type constructor of *AIM*, and of the $\Box$ type constructor of $\lambda^{\Box}$, can be dropped. The motivation for this proposal is as follows: In *AIM*, the Box construct delays its argument. To the programmer, this means that there are two "code" types in *AIM*, one for closed code, and one for open code. Having two different code types can cause some confusion from the point of view of multi-stage programming, because manipulating values of type $[\langle A\rangle]$ would be read as "closed code of open code of A". Not only is this reading cumbersome, it makes it (unnecessarily) harder for the programmer to reason about *when* computations are performed. For these reasons, in λ^{BN} we make the pragmatic decision that Closed should not

[2] This claim is presented as a theorem together with a proof [55], but we have identified some shortcomings in statement of the theorem and its proof. Correcting the embedding of MetaML and $\lambda^{\bigcirc}$ is simple, but the embedding of $\lambda^{\Box}$ is more subtle. Nevertheless, we still expect this embedding to hold.

delay its argument, and so, types such as $[\langle A\rangle]$ can be read as simply "closed code of A". In other words, we propose to use the Necessity modality only for asserting closedness, and not for *delaying* evaluation.

Another difference is that *AIM* was a "superset" of the three languages that we had studied (that is, $\lambda^{\bigcirc}$, $\lambda^{\Box}$, and MetaML), while λ^{BN} is *not*. On the one hand, we expect that $\lambda^{\bigcirc}$ can be embedded into the *open fragment* of *AIM*, and $\lambda^{\Box}$ into the *closed fragment* [55]. Formal support for this claim would establish a strong relation between the closed code and open code types of *AIM* and the Necessity and Next modalities of modal and temporal logic. The embedding of $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ in *AIM* can be turned into an embedding in λ^{BN}. (The embedding of $\lambda^{\Box}$ needs to be modified to take into account the fact that Closed in λ^{BN} is strict, that is, it does not delay the evaluation of its argument.) On the other hand, the embedding of MetaML in *AIM* *cannot* be adapted for the following two reasons:

1. λ^{BN} does not have full cross-stage persistence. Not having cross-stage persistence simplifies the categorical model. At the same time, from the pragmatic point of view, cross-stage persistence for closed types is expected to suffice,
2. λ^{BN} does not have Run. We were not able to find a general categorical interpretation for this construct, though it is possible to interpret Run in a particular, concrete model [5]. At the same time, the pragmatic need for Run disappears in the presence of $\mathsf{safe\_run}$, which does have a natural categorical interpretation.

To our knowledge, our work is the first successful attempt to define a sound type system combining Brackets, Escape and a safe way for executing code in the same language. We have achieved this combination first in the context of MetaML, then in the more expressive context of of *AIM*, and eventually in λ^{BN}.

7.3.5 An Overview of Multi-level Languages

Figure 7.1 is a summary of the distinguishing characteristics of some of the languages discussed here. For Stages, "2" mean it is a two-level language, and "+" means multi-level. For static typing, "1" means only first level is checked. Reflection refers to the presence of a Run-like construct. Persistence refers to the presence of cross-state persistence. Portability refers to the printability of all code generated at run-time.

Facility	**Example**	Nielson & Nielson [62]	Gomard & Jones [32]	Glück & Jørgensen [29]	Hatcliff & Glück [35]	$\lambda^{\Box}$ [23]	$\lambda^{\bigcirc}$ [22]	MetaML, *AIM*, λ^{BN}
Stages	$\langle\lambda$x.x$\rangle$	2	2	+	+	+	+	+
Static Typing		Y	1	N	N	Y	Y	Y
Reflection	run or eval	N	N	N	N	Y	N	Y
Persistence	λf.$\langle\lambda$x.f x$\rangle$	N	N	N	N	N	N	Y
Portability		Y	Y	Y	Y	Y	Y	N

Fig. 7.1. Comparative Feature Set

7.3.6 A Closer Look at $\lambda^{\Box}$, $\lambda^{\bigcirc}$ and *AIM*

To clarify the relation between MetaML and the other multi-level languages, we take a closer look at the syntax, type system and semantics of $\lambda^{\Box}$, $\lambda^{\bigcirc}$, and *AIM*. We adopt the following unified notation for types:

$$\tau \in T := b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle \mid [\tau]$$

In order, the productions correspond to base types, functions, (open) code fragments, and closed code fragments.

The first language, $\lambda^{\Box}$, features function and closed code types. Typing judgments have the form $\Delta;\Gamma \vdash e : \tau$, where $\Delta, \Gamma \in D := [] \mid x : \tau, \Delta$. The syntax for $\lambda^{\Box}$ is as follows:

$$e \in E := c \mid x \mid \lambda x.e \mid e\, e \mid \mathsf{box}\ e \mid \mathsf{let\ box}\ x = e\ \mathsf{in}\ e$$

The type system of $\lambda^{\Box}$ is:

$$\Delta;\Gamma \vdash c : \tau_c \qquad \Delta;\Gamma \vdash x : \tau \ \text{ if } \tau = \Delta(x) \text{ or } \Gamma(x)$$

$$\frac{\Delta;\Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta;\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Delta;\emptyset \vdash e : \tau}{\Delta;\Gamma \vdash \mathsf{box}\ e : [\tau]}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash e_2 : \tau_1}{\Delta;\Gamma \vdash e_1 e_2 : \tau_2}$$

$$\frac{\Delta;\Gamma \vdash e_1 : [\tau_1] \quad \Delta, x : \tau_1;\Gamma \vdash e_2 : \tau_2}{\Delta;\Gamma \vdash \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2}\,.$$

While there are some superficial differences, this type system is essentially the same as that of λ^{BN} without the (open) code type.

$\lambda^{\bigcirc}$, MetaML and *AIM* feature function and open code types. Typing judgments have the form $\Gamma \vdash^n e : \tau$, where $\Gamma \in D := [] \mid x : \tau^n, \Delta$ and n is a natural number called the *level* of the term.

The syntax for $\lambda^{\bigcirc}$ is as follows:

$$e \in E := c \mid x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \tilde{}e.$$

The type system for $\lambda^{\bigcirc}$ is:

$$\Gamma \vdash^n c : \tau_c \qquad \Gamma \vdash^n x : \tau \text{ if } \tau^n = \Gamma(x)$$

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

$$\frac{\Gamma \vdash^n e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash^n e_2 : \tau_1}{\Gamma \vdash^n e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash^{n+} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+} \tilde{}e : \tau}$$

This type system is very similar to that of MetaML, but without cross-stage persistence and without the Run construct. In other words, the type system of MetaML can be achieved by the addition of the following two rules:

$$\Gamma \vdash^n x : \tau \text{ if } \tau^m = \Gamma(x) \text{ and } m \leq n$$

$$\frac{\Gamma^+ \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \mathsf{run}\ e : \tau}$$

MetaML uses a more relaxed type rule for variables than $\lambda^{\bigcirc}$, in that variables can be bound at a level lower than the level where they are used. This relaxation is cross-stage persistence. Furthermore, MetaML extends the syntax of $\lambda^{\bigcirc}$ with $\mathsf{run}\ e$.

AIM [55] extends MetaML with an analog of the Box type of $\lambda^{\Box}$, yielding a more expressive language, and yet has a simpler typing judgment than MetaML. The syntax of *AIM* extends that of MetaML as follows:

$$\begin{aligned} e \in E := {} & c \mid x \mid \lambda x.e \mid e\ e \mid \langle e \rangle \mid \tilde{}e \mid \\ & \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \\ & \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{unbox}\ e. \end{aligned}$$

Run-with generalizes Run of MetaML, in that it allows the use of additional variables x_i in the body of e if they satisfy certain typing requirements.

AIM is essentially the union of all the languages described above, with the addition of Run-with construct, generalizing the Run construct of MetaML, and with a reformulation of the rules for Closed type:

$$\frac{\Gamma \vdash^n e_i : [\tau_i] \quad \Gamma^+, \{x_i : [\tau_i]^n | i \in m\} \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \textsf{run}\ e\ \textsf{with}\ x_i = e_i : \tau}$$

$$\frac{\Gamma \vdash^n e_i : [\tau_i] \quad \{x_i : [\tau_i]^0 | i \in m\} \vdash^0 e : \tau}{\Gamma \vdash^n \textsf{box}\ e\ \textsf{with}\ x_i = e_i : [\tau]}$$

$$\frac{\Gamma \vdash^n e : [\tau]}{\Gamma \vdash^n \textsf{unbox}\ e : \tau}$$

The operational semantics of *AIM* and its sub-languages is:

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x := v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad \frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\tilde{}e \overset{1}{\hookrightarrow} v}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i}{\textsf{box}\ e\ \textsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \textsf{box}\ e[x_i := v_i]} \qquad \frac{e \overset{0}{\hookrightarrow} \textsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} v}{\textsf{unbox}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i := v_i] \overset{0}{\hookrightarrow} \langle v' \rangle \quad v_0' \overset{0}{\hookrightarrow} v}{\textsf{run}\ e\ \textsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} v}$$

$$\frac{e \overset{n+}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x} \qquad \frac{}{c \overset{n+}{\hookrightarrow} c} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+}{\hookrightarrow} v_1\ v_2} \qquad \frac{e_i \overset{n+}{\hookrightarrow} v_i}{\textsf{box}\ e\ \textsf{with}\ x_i = e_i \overset{n+}{\hookrightarrow} \textsf{box}\ e\ \textsf{with}\ x_i = v_i}$$

$$\frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\tilde{}e \overset{n++}{\hookrightarrow} \tilde{}v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\textsf{unbox}\ e \overset{n+}{\hookrightarrow} \textsf{unbox}\ v} \qquad \frac{e_i \overset{n+}{\hookrightarrow} v_i \quad e \overset{n+}{\hookrightarrow} v}{\textsf{run}\ e\ \textsf{with}\ x_i = e_i \overset{n+}{\hookrightarrow} \textsf{run}\ v\ \textsf{with}\ x_i = v_i}$$

7.3.7 Categorical Semantics of Two- and Multi-Level Languages

Moggi advocates a category-theoretic approach to two-level languages and uses indexed categories to develop models for two languages similar to $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ [53], stressing the formal analogies with a categorical account of phase distinction and module languages. He also points out that the use of stateful functions such as *gensym* or *newname* in the semantics makes their use for formal reasoning hard. The big-step semantics presented in this dissertation avoids the use of a *gensym*. He also points out that two-level languages generally have not been presented along with an equational calculus. Our reduction semantics has eliminated this problem for MetaML, and to our knowledge, is the first correct presentation of a multi-stage language using a reduction semantics[3].

Moggi *et al.* [6] presents a careful categorical analysis of the interactions between the two logical modalities studied by Davies and Pfenning [23, 22] and computational monads [52]. This work builds on a previous study [5] in the the categorical semantics of multi-level languages, which has greatly influenced the design of *AIM* and λ^{BN}. In particular, this study of the categorical semantics

[3] An earlier attempt to devise such a reduction semantics [91] is flawed. It was based on level-annotated terms, and therefore suffers from the complications that are addressed by λ-T.

was highly instrumental in achieving a semantically sound integration of the two logical modalities in λ^{BN}.

7.3.8 Level-Annotations

Level annotations have been used, for example, by Russell to address the paradox he pointed out to Frege[4] [101], by Quine in his system New Foundations for logic [70], by Cardelli in his type system with phase distinction [11], by Danvy and Malmkjær in a study of the reflective tower [19], and by Glück and collaborators [29, 30] in their multi-level programming languages.

In this dissertation, our experience was that level-annotations are useful as part of a predicate classifying terms at various levels, whereas annotating the subterms themselves with levels is instructive but not necessarily practical for direct reasoning about programs at the source level. Danvy and Malmkjær seem to have had a similar experience in their study of the reflective tower.

Reduction Semantics and Equational Theories for Multi-Level Languages Muller has studied the reduction semantics of quote and eval in the context of LISP [56]. Muller observed that his formulation of these constructs breaks confluence. The reason for this seems to be that his calculus distinguishes between s-expressions and representations of s-expressions. Muller proposes a closedness restriction in the notion of reduction for eval and shows that this restores confluence.

Muller has also studied the reduction semantics of the λ-calculus extended with representations of λ terms, and with a notion of β reduction on these representations [57]. Muller observed that this calculus lacks confluence, and uses a type system to restore confluence.

In both of Muller's studies, the language can express taking object-code apart (intensional analysis).

Wand has studied the equational theory for LISP meta-programming construct fexpr and found that "the theory of fexprs is trivial" in the sense that the β-rule (or "semantic equality") is not valid on fexprs [99]. Wand, however, predicted that there are other meta-programming systems with a more interesting equational theory. As evidenced by CBN λ-U, MetaML is an example of such a system.

7.4 On the History of Quotation

Formal logic is a well-developed discipline from which programming languages research inherits many techniques. It is therefore illuminating to review one of the foundational works in formal

[4] As Russell is often viewed as the father of type theory, we can also view his notion of level as primordial to todays notion of types.

logic that is closely related to the development presented in this dissertation, and to review the work done in the context of LISP on migrating this work into the programming languages arena.

7.4.1 Quasi-Quotes. Or, Quine's "Corners"

Quasi-quotes are a formal notation developed by the logician Willard van Orman Quine to emphasize some semantics subtleties involved in the construction of logical formulae. Quine introduced quasi-quotes to formal logic around 1940 as a way of distinguishing between the meaning denoted by some syntax, and the syntax itself [71]. His motivations seems to lie primarily in the fact that variables where used in three semantically distinct ways.

In this section we present two excerpts from Quine's original writings that are largely self-explanatory.

The Greek Letter Convention We begin with Quine's description of the state-of-the-art for dealing with object-programs at that time, namely *the Greek letter convention.* In *Essay V: New Foundations for Mathematical Logic* [71, Page 83] Quine writes:

> *"In stating the definitions, Greek letters 'α', 'β', 'γ', 'ϕ', 'ψ', 'χ', and 'ω' will be used to refer to expressions. The letters 'ϕ', ψ', 'χ', and 'ω' will refer to any formulas, and 'α', 'β', and 'γ' will refer to variables. When they are embedded among signs belonging to the logical language itself, the whole is to refer to the expression formed by so embedding the expressions referred to by those Greek letters. Thus, '$(\phi \mid \psi)$' will refer to the formula which is formed by putting the formulas ϕ and ψ, whatever they may be, in the respective blanks of '$(\quad\mid\quad)$'. The expression '$(\phi \mid \psi)$' itself is not a formula, but a noun describing a formula; it is short for the description 'the formula formed by writing a left parenthesis, followed by the formula ϕ, followed by a stroke, followed by the formula ψ, followed by a right parenthesis', etc. Such use of Greek letter has no place in the language under discussion, but provides a means of discussing that language."*

To rephrase, α, β, and γ range over *object-language variable names*, and ϕ, ψ, χ, and ω range over expressions. Note the use of single-quotes '_' that was the standard at the time for talking about object-terms.

The Problem and The Solution In the next chapter, *Essay VI: Logic and The Reification of Universals* [71, Page 111], Quine contrasts the semantic differences between the usage of variables in the expression

$$(\exists\, \alpha)(\phi \ \vee\ \psi) \tag{7.1}$$

and the expression

$$p \;\Rightarrow\; p \wedge p \tag{7.2}$$

where p is a particular name in an object-language, say propositional logic, that is under discussion. Quine explains[5]:

> *" ... 'ϕ' contrasts with 'p' in two basic ways. First, 'ϕ' is a variable, taking sentences as values; 'p', construed schematically, is not a variable (in the value-taking sense) at all. Second, 'ϕ' is grammatically substantival, occupying the place of names of sentences; 'p' is grammatically sentential, occupying the place of sentences.*
> *This latter contrast is dangerously obscured by the usage 7.1*[6]*, which shows the Greek letters 'ϕ' and 'ψ' in sentential rather than substantival positions. But this usage would be nonsense except for the special and artificial convention of Essay V (p. 83)*[7] *concerning the embedding of Greek letters among signs of the logical language. According to that convention, the usage 7.1 is shorthand for the unmisleading substantive:*
>
> > *the result of putting the variable α and the sentences ϕ and ψ in the respective blanks of '$(\exists\ \ )(\ \ \vee\ \ )$'.*
>
> *Here the Greek letters clearly occur in noun positions (referring* to *a variable and to two statements), and the whole is a noun in turn. In some of my writings, for example [69]*[8]*, I have insisted on fitting the misleading usage 7.1 with a safety device in the form of a modified type of quotation marks, thus:*
>
> $$\ulcorner(\exists\,\alpha)(\phi\ \vee\ \psi)\urcorner$$
>
> *These marks rightly suggest that the whole is, like an ordinary quotation, a substantive which refers* to *an expression; also they conspicuously isolate those portions of text in which the combined use of Greek letters and logical signs is to be oddly construed. In most of the literature, however, these quasi-quotation marks are omitted. The usage of most logicians who take care to preserve the semantic distinctions at all is that exemplified by Essay V (though commonly with German or boldface Latin letters instead of Greek)."*

Today, Quine's quasi-quotes are a standard tool for distinguishing between object-language terms and meta-language terms. (See for example [1, 16, 51].)

[5] Footnotes in the following quotation will point out when a reference is numbered with respect to this dissertation (*meta-level*) or to Quine's book (*object-level*). Dwelling on this concrete example of problems that arise when we want to be formal about the semantics of multi-level expressions seemed appropriate for this dissertation.

[6] *Meta-level equation number.*

[7] *Object-level page number.*

[8] *Meta-level citation number.*

7.4.2 LISP's Back-Quote and Comma

Like many good ideas from mathematics and logic – such as types and the λ-calculus – that were reincarnated in one form or the other in programming languages, so were Quine's quasi-quotes. *Back-quote* and *comma* appeared in LISP when Timothy Hart introduced macros to the language [88]. To illustrate the behavior of back-quote, comma, and eval, consider the following simple Scheme [73] session:

```
> (define f (lambda (x) `(+ ,x ,x)))
> (f 1)
(+ 1 1)
> (eval (f 1))
2
```

Steele and Gabriel write [88]:

> *"The back-quote syntax was particularly powerful when nested. This occurred primarily within macro-defining macros; because such were coded primarily by wizards, the ability to write and interpret nested back-quote expressions was soon surrounded by a certain mystique. Alan Bawden of MIT acquired a particular reputation as back-quote-meister in the early days of Lisp Machine."*

And even though the advent of the back-quote and the macro mechanisms was a *"leap in expressive power"*, there where some problems:

> *"Macros are fraught with the same kinds of scoping problems and accidental name capture that had accompanied special variables. The problem in Lisp macros, from the time of Hart in 1963 to the mid-1980's, is that a macro call expands into an expressions that is composed of symbols that have no attached semantics."*

Thus, while back-quote had some of the "spirit" of quasi-quote, it did not quite capture the semantics spirit of Quine's construct: While quasi-quote was invented to clarify binding issues, back-quote suffered from a variety of semantic problems. These problems may have been partly due to the fact that back-quote seems to have been primarily a construct for conveniently constructing lists, and the fact that these lists can also happen to represent programs was probably somewhat secondary. It is also worth noting that the ease with which programs can be manipulate in LISP encourages meta-programming. However, it should be noted that this encoding does not provide any automatic renaming support needed for a correct treatment of bound variables in meta-programming systems.

The Scheme community continued the development of these features. *"Nearly everyone agreed that macro facilities were invaluable in principle and in practice but looked down upon each particular instance as a sort of shameful family secret. If only The Right Thing could be found!"* [88]. Clinger and Rees clarified the problem of renaming in macros using an analogy with reductions in the λ-calculus [13]. In particular, they explained why there is a need for renaming at run-time, and why static renaming is not enough. In the λ-calculus, run-time corresponds to β-reductions, and renaming corresponds to α-renamings. In the λ-calculus, both kinds of reductions must be interleaved because β reductions can involve duplication and then re-combination (thus potential conflict) of code. Hygienic macros were later developed by Kohlbecker, Friedman, Felleisen, and Duba [44] and provided a means for defining and using macros without needing to worry about accidental name capture. Bawden [3] gives a detailed historical review of the history of quasi-quotations in LISP.

A Comparison of MetaML and LISP MetaML's Brackets, Escape, and Run are analogous to LISP's back-quote, comma, and eval constructs: Brackets are similar to back-quote, Escape is similar to comma, and Run is similar to eval under the empty environment. But the analogy is not perfect. LISP does not ensure that variables (atoms) occurring in a back-quoted expressions are bound according to the rules of static scoping. For example `‘(plus 3 5)` does *not* bind `plus` in the scope where the term is *produced*. We view addressing this problem as an important feature of MetaML. We also view MetaML's semantics as a concise formalization of the semantics for LISP's three constructs[9], but with static scoping. This view is similar in spirit to Smith's semantically motivated LISP [85, 86]. Finally, whereas LISP is dynamically typed, MetaML is statically typed.

MetaML's annotations can also be viewed as providing a simple but statically typed macro-expansion system. But it is also important to note that the annotations do not allow the definition of new language constructs or binding mechanisms, as is often expected from macro-expansion systems.

It is worth noting that Hart also used *gensym* to explicitly avoid capture issues in his original report on macros [88]. Today, this trick is still closely associated with the use of back-quote in Scheme or LISP. As we have pointed out in Chapter 4, managing renaming in this manner can be especially difficult in the presence of functional values. Thus, alleviating the need for such stateful construct in a meta-programming language is an invaluable benefit of MetaML.

Finally, we should point out that back-quote and comma are themselves macros in LISP. This state of affairs leads to two more concerns. First, they have non-trivial formal semantics (about

[9] When used in the restricted context of program generation.

two pages of LISP code). Second, because of the way they expand at parse-time, they can lead to a representation overhead exponential in the number of levels in a multi-level program [30]. MetaML avoids both problems by a direct treatment of Bracket and Escape as language constructs.

Remark on Meta-Programming in Prolog Just as the similarity between the representation of programs and the representation of data makes meta-programming in LISP attractive, it also makes meta-programming in Prolog attractive [90, 47]. Prolog too supports intensional analysis over code, in that it *"allows the programmer to examine and alter the program (the clauses that are used to satisfy his goals). This is particularly straightforward, because a clause can be seen as just an ordinary Prolog structure. Therefore Prolog provides built-in predicates to allow the programmer to:*

- *Construct a structure representing a clause in the database,*
- *Add a clause, represented by a given structure, to the database,*
- *Remove a clause, represented by a given structure, from the database."* [14]

The application of the results of our experience with MetaML to Prolog are currently unexplored. But at least on the surface, there seem to be some common themes. For example, Prolog, data can be "run" using the call predicate. But there again, problems with variables being "available at the right time" show up:

> *"[...] most Prolog implementations relax the restriction we have imposed on logic programs, that the goals in the body of a clause must be non-variable terms. The* meta-variable facility *allows a variable to appear as a goal in a conjunctive goal or in the body of the clause. During the computation, by the time it is called, the variable must be instantiated to a term. It will then be treated as usual. If the variable is not instantiated when it comes to be called, an error is reported."* [90]

It will be interesting to see if these problems arising in meta-programming in a Prolog can be addressed in a way similar to the way we have addressed some of these basic meta-programming questions for a functional language.

Chapter 8

Conclusion

The single biggest problem in communication
is the illusion that it has taken place.

George Bernard Shaw

In this chapter, we review and appraise our findings and the extent to which they support our thesis. We outline future work, and pose some open questions that we believe are of significance to the research community.

8.1 Review

This dissertation reports on the results of scrutinizing the design and implementation of the multi-stage programming language MetaML. Our approach to studying MetaML has been to formalize its semantics and type system, and the properties we expect them to enjoy. In doing so, we have identified a variety of subtleties related to multi-stage programming, and provided solutions to a number of them. Our results include various forms of formal semantics for MetaML, in addition to a sound type system. Our study has also resulted in a proposal for a refined type system, and a refined view of the process of developing program generators using MetaML.

In Chapters 2 and 3, we saw how MetaML is a promising framework for meta-programming in general, and for staging in particular. We saw how staging allows us to distribute the cost of a computation over more than one stage of computation. This goal was achieved in the case of the term-rewriting example, and in the case of other, more concise examples such as the power and member functions. Our experience with programming in MetaML helped us recognize multi-stage computation as a simple, natural, and clear computational phenomenon:

– It is simple because it requires only three annotations to explain.

– It is natural because it appears to be distinct from translation, another important computational phenomenon that we have described in introduction to this dissertation. It is also natural because it applies, at least in principle, to both CBV and CBN evaluation strategies.
– It is clear because it can be easily described and demonstrated with mathematical rigor.

But while we emphasize the promise that multi-stage programming exhibits, we also caution that, at least in its current incarnation, multi-stage programming has some limitations. Most notably, the term-rewriting example needed to be rewritten in CPS in order to allow for better staging. At this point, the question of whether the CPS transformation can be avoided or not is still open. (See Section 8.4.)

In Chapter 4, we saw that a direct approach to extending a simple interpreter for a CBV lambda calculus to a multi-level language such as MetaML can be difficult. On one hand, this difficulty is surprising, because these simple interpreters can be extended to incorporate powerful and semantically complex features such as side effects and exceptions. On the other hand, such difficulty is not surprising, because the essential "trick" behind these interpreters is that they are reifiers: *Reifiers* take object-level concepts and map them directly to meta-level concepts. For example, integers are mapped to integers, lambda-abstractions are mapped to lambda-abstractions, applications are mapped to applications, conditionals to conditionals, and recursion to recursion. Multi-level features of object-programs, such as Brackets, Escapes, and Run, cannot be interpreted naturally in this "reifier" style because this style can only work when the feature we wish to interpret already has a counterpart in the meta-language. This prerequisite is not satisfied for multi-level features.

In Chapter 5, we presented a deterministic semantics that is more suitable for formally specifying the semantics of MetaML. We showed how a formal type-safety property can be established for a subset of MetaML that we called λ-M. We then pointed out an expressivity limitation in the basic type system for MetaML, and presented a proposal that avoids this problem. We argued that this proposal for an extended MetaML is a better match for a multi-stage programming method. However, our current practical experience with this proposed language is still limited.

In Chapter 6, we studied the reduction semantics for a subset of MetaML that we called λ-U, and saw that it is hard to limit β reduction to level 0 in MetaML, that is, it is hard to stop semantic equality at the meta-level from "leaking" into the object-level. The alternative interpretation of our observations is that it is more natural to allow semantic equality at all levels in MetaML. The essential reason is that the level of a raw MetaML terms is not "local" information that can be determined just by looking at the term. Rather, the level of a term is determined from the context. Because of substitution (in general) and cross-stage persistence (in particular), we are forced to allow β to "leak" into higher levels. This "leakage" of the β-rule could be interpreted as a desirable phenomenon because it would allow implementations of MetaML to perform a wider range of semantics-preserving optimizations on programs. If we accept this interpretation and therefore accept allowing β at all levels, we need to be careful about introducing intensional analysis. In particular, the direct (deterministic) way of introducing intensional analysis on code would lead to an incoherent reduction semantics and equational theories.

8.2 Summary of Results

We summarize the key results as follows:

1. Programming: By staging a number of programs in MetaML, we illustrated the potential of multi-stage programming, and pointed out some weaknesses of the approach. Staging in MetaML is a promising and intuitive process. The main weakness we identified is that the term-rewriting example had to be rewritten in CPS so that better staging was achieved.
2. Implementation: We identified a variety of anomalies in the implementation, and suggested corrections to many of them. The main example presented in this dissertation was the flawed treatment of hidden free-variables, to which we proposed a solution using the idea of a cover.
3. Semantics: In order to understand how we can ensure the safety of generated programs, we have studied the formal semantics of MetaML in many styles. The two formulations presented in this dissertation were the reduction and big-step semantics. The reduction semantics and the proofs of its confluence and soundness (in the CBN case) are novel, independent technical results.
4. Type System: We identified various ways in which "multi-stage programs go *Wrong*," and developed examples that demonstrated that some type systems were too weak or too strong for MetaML.
5. Meta-Programming: We identified coherence (and in turn, confluence) as a property that is violated by trying to have semantic equality (β) and intensional (structural) analysis on code in MetaML.

8.3 Appraisal of Results

Our thesis is that MetaML is a well-designed language that is useful in developing meta-programs and program generators. We broke down the thesis into three main hypotheses:

H1. *MetaML is a useful medium for meta-programming.* In Chapter 2, we have explained the idea of staging, and how it is easy to see that it can reduce cost in the abstract setting of the CBV and CBN lambda calculus. We have given small example applications in Chapter 3 and a more substantial example in Chapter 4.

H2. *MetaML can be placed on a standard, formal foundation whereby staging annotations are viewed as language constructs amenable to the formal techniques of programming languages.* In Chapters 5 we presented a big-step semantics for a core subset of MetaML, together with a type system. In Chapter 6 we presented a reduction semantics for this core subset and proved it confluent and sound in the CBN case.

H3. *MetaML in particular, and multi-level languages in general, can be improved both in their design and implementation by what we have learned while building MetaML's formal foundations.* In Chapter 4 we have identified subtle and previously unknown problems that can riddle implementations of multi-stage languages. Identifying many of these problems was a direct consequence of studying the formal semantics of MetaML. In Chapter 5, we have proposed adding an explicit type constructor for expressing closedness in MetaML, and argued for its utility in supporting multi-stage programming in MetaML. In Chapter 6, we explained the potentially negative effects of adding deterministic intensional analysis in a direct manner to MetaML.

8.3.1 Limitations Future Works

The work presented in this dissertation can be improved in a number of ways:

1. Formal relations should be established between the various formulations of the semantics. The reduction semantics should be also shown to respect the big-step semantics in the CBV case, as we have done for the CBN case.
2. The implementation should be validated with respect to one of the abstract formulations of the semantics. A good candidate seems to be the big-step semantics, because it is the closest to the implementation.
3. The proposed extension to MetaML should be incorporated into the implementation, and this implementation should be used to develop more substantial examples than the one we have presented in this dissertation. It will be important to understand the practical implications of the extensions that we have proposed, especially from the programmer's point of view.
4. The subject reduction and type-safety results should be extended to a Hindley-Milner polymorphic type system, and the decidability of the type system in the light of the current extensions should be verified.

5. The relationship between multi-stage programming partial evaluation concepts such as self-application and the Futamura projections [40] has not been addressed in this dissertation, and remains as future work.

8.4 Open Problems and Promising Research Directions

The work presented in this dissertation has directed our attention to many important questions relating to multi-stage computation in general, and MetaML in particular. Here is the list of questions that we see as significant:

1. *Can we make the closedness annotations implicit in the terms of the programming language?* In particular, the term language of λ^{BN} is verbose, and has lost some of the simplicity of core MetaML, which had only three constructs for staging. In λ^{BN}, asserting that a code fragment is closed (using $[\_]$) has become part of the responsibilities of the programmer. It remains an open question whether there exists a statically typed language with the expressivity of λ^{BN} yet needs only three additional constructs to realize staging.
2. *Can we avoid the need for rewriting programs in CPS before staging?* The term-rewriting example in Chapter 3 shows that further optimizations on the generated code can be useful. It may be possible, through the use of an η_v-like reduction, to produce optimal results without having the user rewrite the program in CPS. Bondorf has studied improving binding times without resorting to explicit CPS conversion [9]. The work of Sabry and Wadler suggests that the use of Moggi's λ_c may also be relevant to dealing with this problem [76].
3. *Can MetaML be compiled using the traditional functional programming language compilation techniques?* Compiling MetaML requires staging its semantics, and will reveal new design options that need to be pursued and understood. Model-theoretic interpretation functions seem to have provided a good starting point for such research in the past, and we believe that this will also be the case for MetaML.

   Furthermore, by exposing the Safe-run constant to programmers, they will eventually have new demands for controlling the behavior of this new operation, and will therefore pose more interesting questions for the developers of compilers supporting this construct. Safe-run is a way for the user to "appeal to the higher power of the compiler developer and to avoid meddling with the complex and fragile implementation details," and the user will necessary be picky in specifying what he is asking for: A light-weight compilation phase that produces machine code quickly, or a heavy-weight phase that takes longer but produces more efficient code? Compiling for memory or for speed?

4. *Is there a modular way for supporting a heterogeneous meta-programming language based on MetaML?* By focusing on the staging aspect of meta-programming, this dissertation presented a number of concrete solutions to fundamental problems in staging. However, as we mentioned in the introduction, another important application for meta-programming is translation. We believe that our work can be extended in a fairly natural way to allow the manipulation of multiple different object-languages at the same time. In particular, we can index the code type by the name of a language, for which there is an associated syntax, and special typing rules. Such a scheme is implicit in the meta-language used by Jones (see for example [40]) for describing partial evaluation.

   We believe that such a framework may even be appropriate for dealing run-time code generation. Run-time code generation is concerned with the efficient generation of specialized machine code at run-time. While a compiled implementation of MetaML can provide a version of Safe-run that generates machine code, it is implicitly assumed to be making calls to a compiler, which is generally not considered fast enough for run-time code generation. If machine code is considered as a typed language, we believe that it can be incorporated into a Heterogeneous MetaML setting. With the ongoing research efforts on typed assembly languages, this possibility is not far-fetched.

5. *Are there practical, sound type systems for expressing tag-less reifiers?* In Chapter 4, we explained the subtleties in implementing a language like MetaML in the context of an interpreter that we have described as "essentially a reifier". Such interpreters can be of great utility if they are staged, because they then are total maps from object-programs to meta-programs. If the meta-language has a compiled implementation, reifiers provide us with a very simple means for implementing compilers for many languages that are "similar" to (in the sense that they are syntactically sugared subsets of) the meta-language. However, one problem remains: The interpreters presented in Chapter 4 do not map object-programs to meta-programs, but rather to a datatype carrying meta-programs. The tags in this datatype introduce a level of run-time overhead that is not necessary if the object-program is well-typed. While it is currently possible to express a tag-less reifier in untyped MetaML, it is not typable in MetaML's type system. The problem of finding an appropriate type system that would allow us to express tag-less reifiers remains open.

6. *Are there practical, theoretically sound approaches for allowing the programer to specify optimizations on MetaML code?* We have pointed out there there is a conflict between allowing optimizations on object-code and intensional analysis. But there is no obvious problem with

associating *domain-specific optimization* to specific instances of the code datatype. The semantics of such a datatype would then be defined modulo the optimizations, and the optimizations themselves can be applied by the compiler non-deterministically. Furthermore, with practical, statically verifiable methods for establishing confluence of a set of rewrite rules, it may be possible to further simplify the semantics of such a *domain-specific code type*.

Appendix A

A Calculus with Level Annotations

Celery, raw
Develops the jaw,
But celery, stewed,
Is more quietly chewed.

Celery, Ogden Nash

This appendix presents an alternative approach to the reduction semantics of MetaML. We had explored this approach, but did not develop it fully, as it involves a substantially larger calculus than the one presented in Chapter 6. This appendix also explains in more detail how working towards a Subject Reduction lemma as our guide helped us arrive at the type system for MetaML.

A.1 The λ-T Language

To avoid the problems discussed at the beginning of Chapter 6, we introduce Bubbles $\boxed{\_}$ into our language at levels higher than 0 as a means of explicitly controlling the elimination of Escapes. Furthermore, we build on the simple observation that the set of expressions and the set of values for the λ language can be defined as follows:

$$e \in E := v \mid x \mid e\, e$$
$$v \in V := i \mid \lambda x.e$$

The mutual recursion in the definition is not strictly necessary — as we have seen in Chapter 6, we can define expressions first, then values — but we have also found that the compactness of this definition can pay-off when we extend a core language λ with more syntactic constructs. The resulting multi-level calculus, called λ-T, has the following syntax:

$$\begin{array}{lll} e^0 & \in E^0 & := v \mid x^0 \mid e^0\, e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+} & \in E^{n+} & := \boxed{e^n} \mid i^{n+} \mid x^{n+} \mid e^{n+}\, e^{n+} \mid \lambda x.e^{n+} \mid \langle e^{n++} \rangle \mid \,\tilde{}\, e^n \mid \mathsf{run}\ e^{n+} \\ v & \in V & := i^0 \mid \lambda x.e^0 \mid \langle \boxed{e^0} \rangle \end{array}$$

Values are either level 0 integers, lambda terms, or code fragments. Note that we have further specified that the code fragments must be Bubbles. The primary role of Bubbles is to ensure that there are no level 1 Escapes in a code value. It is also important to note that we have now refined our notion of level to take Bubbles into account. The level of a term is the number of surrounding Brackets, less surrounding Escapes *and Bubbles*. Intuitively, keeping track of this refined notion of levels in the terms will allow us to deduce the exact state of rebuilding that the term has reached. (We illustrate this point with a concrete example in Section A.1.5.)

Remark A.1.1 (Notation) *We will simply write e, v, E, V whenever the index is clear from the context.*

A.1.1 Notions of Reduction. Or, Calculating with Bubbles

Now we have enough structure in the terms to direct reduction in a sensible manner, and without need for contextual information. The three basic notions of reduction for the λ-T language are:

$$\begin{array}{rcl} (\lambda x.e_1^0)\, e_2^0 & \longrightarrow_{\beta_T} & e_1^0[x ::= e_2^0] \\ \tilde{}\langle \boxed{e^0} \rangle & \longrightarrow_{E_T} & \boxed{e^0} \\ \mathsf{run}\ \langle \boxed{e^0} \rangle & \longrightarrow_{R_T} & \boxed{e^0} \end{array}$$

The β_T rule is essentially β restricted to level 0, but with an extra precaution taken in the definition of substitution to preserve level-annotatedness when the variable x appears at a level higher than 0 in the body of the lambda term. In particular $e_1[x ::= e_2]$ denotes a *special* notion of substitution[1]:

[1] In the treatment of this chapter, we use Barendregt's convention for free and bound variables [1]. In essence, this convention states that, for any set of terms used in a proof or a definition, all bound variables are chosen to be different from the free variables.

$$\begin{array}{rll}
i^n[x ::= e] & = i^n & \\
x^0[x ::= e] & = e & \\
x^{n+}[x ::= e] & = (x^n[x ::= e])^+ & \\
y^n[x ::= e] & = y^n & x \neq y \\
(e_1\ e_2)^n[x ::= e] & = e_1[x ::= e]\ e_2[x ::= e] & \\
(\lambda y.e_1)[x ::= e] & = (\lambda z.(e_1[z/y][x ::= e])) & \\
& \quad z \notin FV(e, e_1) & x \neq y \\
\langle e_1 \rangle[x ::= e] & = \langle e_1[x ::= e] \rangle & \\
\tilde{\ }e_1[x ::= e] & = \tilde{\ }(e_1[x ::= e]) & \\
(\mathsf{run}\ e_1)[x ::= e] & = \mathsf{run}\ (e_1[x ::= e]) & \\
\boxed{a}[x ::= e] & = \boxed{a[x ::= e]} &
\end{array}$$

and where Promotion $\_^+ : E \to E$ is a total function inductively as follows:

$$\begin{array}{rl}
(i^n)^+ & = i^{n+} \\
(x^n)^+ & = x^{n+} \\
(e_1\ e_2)^+ & = e_1{}^+\ e_2{}^+ \\
(\lambda x.e)^+ & = \lambda x.e^+ \\
\langle e \rangle^+ & = \langle e^+ \rangle \\
(\tilde{\ }e)^+ & = \tilde{\ }(e^+) \\
(\mathsf{run}\ e)^+ & = (\mathsf{run}\ e^+) \\
\boxed{e}^+ & = \boxed{e^+}
\end{array}$$

Thus the only non-standard feature of this definition of substitution is the case of variables at levels higher than 0. This case arises exactly when a cross-stage persistent variable is being eliminated. For example, the notion of level-annotatedness still allows terms such as $\lambda x.\langle x \rangle$. We chose to use the non-standard notion of substitution rather than using more sophisticated well-formedness conditions. In particular, the latter approach would require us to make cross-stage persistence explicit in the language rather than implicit, and is likely to complicate the formal treatment rather than to simplify it.

A.1.2 Bubble Reductions

Now we come to the main feature of the λ-T calculus, namely, the use of a set of reduction rules that mimic the behavior of the "rebuilding" functions. These reduction rules start at the leaves of a delayed term, and begin propagating a Bubble upwards.

$$\begin{array}{rcl}
i^{n+} & \longrightarrow_{B1_T} & \boxed{i^n} \\
x^{n+} & \longrightarrow_{B2_T} & \boxed{x^n} \\
\boxed{e_1^n}\ \boxed{e_2^n} & \longrightarrow_{B3_T} & \boxed{e_1^n\ e_2^n} \\
\lambda x.\boxed{e^n} & \longrightarrow_{B4_T} & \boxed{\lambda x.e^n} \\
\langle\boxed{e^{n+}}\rangle & \longrightarrow_{B5_T} & \boxed{\langle e^{n+}\rangle} \\
\text{\textasciitilde}\boxed{e^n} & \longrightarrow_{B6_T} & \boxed{\text{\textasciitilde} e^n} \\
\mathsf{run}\ \boxed{e^n} & \longrightarrow_{B7_T} & \boxed{\mathsf{run}\ e^n}
\end{array}$$

Intuitively, a Bubble around a term will assert that it is "free of top-level escapes". The key concept is that a delayed term free of top-level escapes can be treated as a normal program from the previous level. Note further that Bubble reductions "generate" a Bubble surrounding the whole term either from a level annotation (in the case of integers and variables) or from Bubbles surrounding all subterms. If we can make a term generate enough surrounding Bubbles so that what is left of the term reaches level 0, β at level 0 becomes applicable to what is left of the term.

A.1.3 Deriving a Type Rule for Bubble and Run

The question now is whether we can synthesis the type system for the language described above, and whether there is any systematic way of approaching this question.

Starting from the type system for Brackets and Escapes, we will explain how one can arrive at the extra rules for Bubble and Run by analysing the reduction rules. The type system is a judgment $\Gamma \vdash^n e : \tau$ where $e \in E^n$, $\tau \in T$ is a type, and $\Gamma \in D$ is an environment, and where types and environments are defined as follows:

$$\begin{aligned}
\tau \in T &:= b \mid \tau_1 \to \tau_2 \mid \langle\tau\rangle \\
\Gamma \in D &:= [] \mid x : \tau^i, \Gamma
\end{aligned}$$

The unusual thing about this definition is the use of an integer i in the bindings of environments. We will explain how the Bubble reductions motivate this particular generalization. We will also need two simple operations on environments, namely Γ^+ and Γ^-, which increment and decrement (respectively) indices on all bindings in Γ.

The two extra rules that are needed are as follows:

$$\frac{\Gamma^- \vdash^n e : \tau}{\Gamma \vdash^{n+} \boxed{e} : \tau}\text{Bubble} \qquad \frac{\Gamma^+ \vdash^n e : \langle\tau\rangle}{\Gamma \vdash^n \mathsf{run}\ e : \tau}\text{Run}$$

Trying to prove Subject Reduction for the λ-T language provides us with concrete motivation for these two typing rules. A Subject Reduction lemma says that every reduction preserves typability.

In other words, knowing *only* that the left-hand side of a reduction is typable, we must be able to show that the right-hand side of the reduction is also typable. In what follows, we will explain how this "provability" property helped us in synthesising the two new type rules.

Let us begin with Bubble. The Bubble reduction for variables is, operationally, aimed at capturing the fact that this variable is free of Escapes, and this rule was our basis for developing the rules for the untyped language. There are many reduction rules for Bubble, each a potential source of (different) insights into what the type rule for Bubble should look like. It is crucial when looking for insights to pick the simplest rule that will suggest the most concrete constraint. In this case, it is the Bubble rule for variables. The most important feature of this rule is that it does not involve Bubbles on its left-hand side. The same is true for the rule for integers, but the rule for integers does not involve any "use" of the environment. Now, let us consider exactly what information is available when we know that the left hand side of the Bubble rule for variables (B-2) is typable. In other words, what do we know when $\Gamma \vdash^{n+} x^{n+} : \tau$ holds? All we know is that $\Gamma(x) = \tau^{n+}$. What we want to prove, in general, is that the left-hand side is typable. More precisely, we want to make sure that it is typable under the same environment, at the same level, and with the same type. In other words, we want to be able to prove that $\Gamma \vdash^{n+} \boxed{x^n} : \tau$. We want to find a *uniform* way of inferring this result. The notion of uniformity here is hard to define, but let us take it to mean "a simple way". More concretely, given the following schema:

$$\frac{? \vdash^{n} x : ?}{\Gamma \vdash^{n+} \boxed{x} : \tau} \text{ A New Rule Schema.}$$

Now we consider the following question: What are the simplest transformations (on environments and types) that would let us fill-in the missing parts, that would let us succeed in proving the concrete problem of type preservation for the Bubble reduction for variables? Again, we already have a rule for variables, so, we follow the derivation tree (schema) one level up using the variable rule, to get a simpler equation:

$$\frac{?(x) = ?^{n}}{? \vdash^{n} x : ?} \text{ Var on Schema}$$

Recalling that what we know from the left-hand side of the rule is $\Gamma(x) = \tau^{n+}$, we see that we can fill in the schema $?(x) = ?^n$ as $\Gamma^{-}(x) = \tau^n$. We then propagate this information back to our new rule schema, to get the following concrete Bubble rule:

$$\frac{\Gamma^{-} \vdash^{n} x : \tau}{\Gamma \vdash^{n+} \boxed{x} : \tau} \text{ Concrete Bubble Rule}$$

What we have argued so far is that if this rule holds, subject reduction would hold. But we still do not have a useful rule, because this rule is only for variables. To arrive at a useful rule candidate, we generalize the occurrence of the variable x to an arbitrary expression to get:

$$\frac{\Gamma^{-} \vdash^{n} e : \tau}{\Gamma \vdash^{n+} \boxed{e} : \tau} \text{ Bubble (Tentative)}$$

Again, we can use subject-reduction to test the validity of this more general rule. Indeed, using this rule, we can show that all Bubble reductions preserve typing (note that we cannot do that for Run yet, because we are ignoring the presence of its type rule for now). This progress is promising, but the road to completely justifying this rule still requires ensuring that it preserves substitutivity, and eventually, that we do have type-safety for our language.

With this promising Bubble rule, we can use either of the two rules that involve Run and Bubble, we infer a "uniform schema" for Run in the same way that we have done for Bubble.

A.1.4 Subject Reduction

We will now consolidate the observations made in this section into a formal Subject Reduction lemma. Figure A.1 summarizes the language λ-T that we present and study in this section, with the exception of the lengthy definition of the non-standard notion of substitution $\_[\_ ::= \_]$.

The following lemma tells us that any term typable at one level remains typable at the next level. Furthermore, we can also increase (by one) the level of any subset of the variables in the environment under which the term is typable.

Lemma A.1.2 (Promotion) *If* $\Gamma_1, \Gamma_2 \vdash^{n} e_1 : \tau_1$ *then* $\Gamma_1, \Gamma_2^{+} \vdash^{n+} e_1^{+} : \tau_1$*.*

Lemma A.1.3 (Co-Promotion) *If* $\Gamma_1, \Gamma_2 \vdash^{n} e_1 : \tau_1$ *then* $\Gamma_1, \Gamma_2^{-} \vdash^{n} e_1 : \tau_1$*.*

Lemma A.1.4 (Generalized Substitution) *If* $\Gamma_1^{i}, \Gamma_2^{i}; x : \tau_2^{i} \vdash^{n} e_1 : \tau_1$ *and* $\Gamma_2 \vdash^{0} e_2 : \tau_2$ *then* $\Gamma_1^{i}, \Gamma_2^{i} \vdash^{n} e_1[x ::= e_2] : \tau_1$*.*

Corollary A.1.5 (Substitution) *If* $\Gamma_2; x : \tau_2^{0} \vdash^{0} e_1 : \tau_1$ *and* $\Gamma_2 \vdash^{0} e_2 : \tau_2$ *then* $\Gamma_2 \vdash^{0} e_1[x ::= e_2] : \tau_1$*.*

Theorem A.1.6 (Subject Reduction for CBN λ-T) $\forall n \in \mathbb{N}, \Gamma \in D, \tau \in T. \forall e_1, e_2 \in E^{n}.$

$$\Gamma \vdash^{n} e_1 : \tau \wedge e_1 \longrightarrow e_2 \Longrightarrow \Gamma \vdash^{n} e_2 : \tau.$$

A.1.5 Towards Confluence

The key observation to be made about the new calculus is that the promotion performed in the substitution does not injure confluence: Bubbles allow us to recover confluence. For example, let us reconsider the example from Section 6.6:

Syntax:

$$\begin{array}{lll} e^0 & \in E^0 & := v \mid x^0 \mid e^0\, e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+} & \in E^{n+} & := \boxed{e^n} \mid i^{n+} \mid x^{n+} \mid e^{n+}\, e^{n+} \mid \lambda x.e^{n+} \mid \langle e^{n++} \rangle \mid \tilde{}\, e^n \mid \mathsf{run}\ e^{n+} \\ v & \in V & := i^0 \mid \lambda x.e^0 \mid \langle \boxed{e^0} \rangle \end{array}$$

Reductions:

$$\begin{array}{rcl} (\lambda x.e_1^0)\, e_2^0 & \longrightarrow_{\beta_T} & e_1^0[x ::= e_2^0] \\ \tilde{}\langle \boxed{e^0} \rangle & \longrightarrow_{E_T} & \boxed{e^0} \\ \mathsf{run}\ \langle \boxed{e^0} \rangle & \longrightarrow_{R_T} & \boxed{e^0} \\ i^{n+} & \longrightarrow_{B1_T} & \boxed{i^n} \\ x^{n+} & \longrightarrow_{B2_T} & \boxed{x^n} \\ \boxed{e_1^n}\ \boxed{e_2^n} & \longrightarrow_{B3_T} & \boxed{e_1^n\ e_2^n} \\ \lambda x.\boxed{e^n} & \longrightarrow_{B4_T} & \boxed{\lambda x.e^n} \\ \langle \boxed{e^{n+}} \rangle & \longrightarrow_{B5_T} & \boxed{\langle e^{n+} \rangle} \\ \tilde{}\,\boxed{e^n} & \longrightarrow_{B6_T} & \boxed{\tilde{}\, e^n} \\ \mathsf{run}\ \boxed{e^n} & \longrightarrow_{B7_T} & \boxed{\mathsf{run}\ e^n} \end{array}$$

Types and Type Environments:

$$\begin{array}{l} \tau \in T := b \mid \tau_1 \to \tau_2 \mid \langle \tau \rangle \\ \Gamma \in D := [] \mid x : \tau^i; \Gamma \end{array}$$

Type Rules:

$$\frac{}{\Gamma \vdash^n i : \mathsf{int}}\ \text{Int}$$

$$\frac{\Gamma(x) = \tau^{(n-m)}}{\Gamma \vdash^n x : \tau}\ \text{Var} \qquad \frac{x : \tau'^n; \Gamma \vdash^n e : \tau}{\Gamma \vdash^n \lambda x.e : \tau' \to \tau}\ \text{Lam} \qquad \frac{\Gamma \vdash^n e_2 : \tau' \quad \Gamma \vdash^n e_1 : \tau' \to \tau}{\Gamma \vdash^n e_1\ e_2 : \tau}\ \text{App}$$

$$\frac{\Gamma \vdash^{n+} e : \tau}{\Gamma \vdash^n \langle e \rangle : \langle \tau \rangle}\ \text{Brk} \qquad \frac{\Gamma \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^{n+} \tilde{}\, e : \tau}\ \text{Esc} \qquad \frac{\Gamma^- \vdash^n e : \tau}{\Gamma \vdash^{n+} \boxed{e} : \tau}\ \text{Bubble} \qquad \frac{\Gamma^+ \vdash^n e : \langle \tau \rangle}{\Gamma \vdash^n \mathsf{run}\ e : \tau}\ \text{Run}$$

Fig. A.1. The Reduction Semantics and Type System for λ-T

$$(\mathsf{fn\ x} \Rightarrow \langle \mathsf{x}^1 \rangle)\ (\mathsf{fn\ y} \Rightarrow (\mathsf{fn\ x} \Rightarrow \mathsf{x}^0)\ 5^0).$$

As we saw before, if we perform the outermost redex we get:

$$\langle \mathsf{fn\ y} \Rightarrow (\mathsf{fn\ x} \Rightarrow \mathsf{x}^1)\ 5^1 \rangle.$$

And there are still no β_T redices left. But there *are* Bubble reductions that allow us to first derive:

$$\langle \mathsf{fn\ y} \Rightarrow (\mathsf{fn\ x} \Rightarrow \boxed{\mathsf{x}^0})\ \boxed{5^0} \rangle.$$

then

$$\langle \mathsf{fn\ y} \Rightarrow \boxed{\mathsf{fn\ x} \Rightarrow \mathsf{x}^0}\ \boxed{5^0} \rangle.$$

then

$$\langle \mathsf{fn\ y} \Rightarrow \boxed{(\mathsf{fn\ x} \Rightarrow \mathsf{x}^0)\ 5^0} \rangle.$$

recovering the desired β_T redex which we can perform to get:

$$\langle \mathsf{fn\ y} \Rightarrow \boxed{5^0} \rangle.$$

Furthermore, both reduction sequences can be reduced to the common term:

$$\langle \boxed{\mathsf{fn\ y} \Rightarrow 5^0} \rangle.$$

The following lemma formalizes the claim that we can always recover the level 0 redices in a term even if the term has been promoted during a substitution.

Lemma A.1.7 (Effervescence) $e^+ \rightarrow^* \boxed{e}$

Thus, even though this property was not our goal, we do get a carefully restricted form of β at levels higher than 0 in λ-T.

We expect that this calculus is confluent, but we do not report a formal proof of such a property. We do report a formal proof of confluence for a more concise calculus that we present in the Section 6.7.

A.1.6 Remark on λ-U

Studying the λ-T language has allowed us to illustrate how carefully restricting reductions can yield a promising calculus for MetaML. But λ-T has a large number of rules, and uses a non-standard notion of substitution. Both these features can make it, at least, inconvenient for pencil and paper reasoning. The λ-U language has a smaller set of reduction rules, and uses a standard notion of substitution. Intuitively, λ-U moved the burden of recognizing that a piece of code is free of top level Escapes from the object-language back to the meta-language: λ-T tracks freedom of top level Escapes using Bubbles, and λ-U tracks freedom of top level Escape by testing for membership in the set E^0. As Bubbles were themselves introduced to "fix" level annotations, which themselves were an object-level codification of a meta-level concept, we have gone full circle: Attempting to exploit one meta-level concept: levels, we have identified a more fundamental meta-level concept: freedom of top-level Escapes.

A.1.7 Remark on Level Annotations

It should be noted that while level annotations can be highly illuminating, they suggest that it is appropriate to have the inside of level n Brackets *always* be a member of a set of values of level $n+$. But this is not generally true. In fact, level annotations get in the way of unifying the set of values V^1 with the set of expression E^0, which is an important feature of λ-U.

A.1.8 Remark on Big-Step Semantics, Level Annotations and Bubbles

The determinism of the big-step semantics seems to alleviate the need for explicit level-annotations and the Bubble construct. We presented the big-step semantics for the toy language λ-M that does not have explicit level-annotations on terms. Defining the big-step semantics for a language with level-annotations (such as λ-T) can be done in essentially the same way, but would be more verbose. In particular, we would have to explicitly pass around the level-annotation "baggage".

Defining the big-step semantics for λ-T would require adding a rule that says that rebuilding a Bubble expression is idempotent. We expect that a big-step semantics with explicit Bubbles will have superior performance to one without Bubbles, because the idempotence rule for rebuilding Bubbles would allow us to avoid redundant rebuilding. This semantics is still largely unexplored.

Bibliography

[1] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed. North-Holland, Amsterdam, 1984.

[2] Barendregt, H. P. Lambda calculi with types. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, Oxford, 1991.

[3] Bawden, A. Quasiquotation in LISP. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (San Antonio, Jan. 1999), pp. 88–99. Invited talk.

[4] Benaissa, Z.-E.-A., Briaud, D., Lescanne, P., and Rouyer-Degli, J. $\lambda\nu$, a calculus of explicit substitutions which preserves strong normalisation. *Journal of Functional Programming 6*, 5 (Sept. 1996), 699–722.

[5] Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. A categorical analysis of multi-level languages (extended abstract). Tech. Rep. CSE-98-018, Department of Computer Science, Oregon Graduate Institute, Dec. 1998. Available from [66].

[6] Benaissa, Z. E.-A., Moggi, E., Taha, W., and Sheard, T. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)* (July 1999). In press.

[7] Bondorf, A. A self-applicable partial evaluator for term rewriting systems. In *TAPSOFT '89. Proceedings of the Theory and Practice of Software Development, Barcelona, Spain* (Mar. 1989), Lecture Notes in Computer Science, Springer-Verlag, pp. 81–95.

[8] Bondorf, A. *Self-Applicable Partial Evaluation.* PhD thesis, University of Copenhagen, 1990.

[9] Bondorf, A. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference on Lisp and Functional Programming. San Francisco, California* (June 1992), pp. 1–10.

[10] Cardelli. Type systems. In *The Computer Science and Engineering Handbook.* CRC Press, 1997.

[11] CARDELLI, L. Phase distinctions in type theory. (Unpublished manuscript.) Available online from http://www.luca.demon.co.uk/Bibliography.html, 1988. Last viewed August 1999.

[12] CARDELLI, L. Typeful programming. In *Formal Description of Programming Concepts*, E. J. Neuhold and M. Paul, Eds., IFIP State-of-the-Art Reports. Springer-Verlag, New York, 1991, pp. 431–507.

[13] CLINGER, W., AND REES, J. Macros that work. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)* (Orlando, Jan. 1991), ACM Press, pp. 155–162.

[14] CLOCKSIN, W. F., AND MELLISH, C. S. *Programming in Prolog*. Springer-Verlag, 1981.

[15] CONSEL, C., AND DANVY, O. For a better support of static data flow. In *Functional Programming Languages and Computer Architecture* (Cambridge, Aug. 1991), vol. 523 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 496–519.

[16] CURRY, H. B., AND FEYS, R. *Combinatory Logic, Volume I*. North-Holland, Amsterdam, 1958. Second printing 1968.

[17] DANVY, O. Across the bridge between reflection and partial evaluation. In *Partial Evaluation and Mixed Computation* (1988), D. Bjorner, A. P. Ershov, and N. D. Jones, Eds., North-Holland, pp. 83–116.

[18] DANVY, O. Type-directed partial evaluation. In *ACM Symposium on Principles of Programming Languages* (Florida, Jan. 1996), ACM Press, pp. 242–257.

[19] DANVY, O., AND MALMKJÆR, K. Intensions and extensions in a reflective tower. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming* (1988), ACM Press, pp. 327–341.

[20] DANVY, O., MALMKJAER, K., AND PALSBERG, J. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation 1*, 19 (1995).

[21] DANVY, O., MALMKJAER, K., AND PALSBERG, J. Eta-expansion does the trick. Tech. Rep. RS-95-41, University of Aarhus, Aarhus, Aug. 1995.

[22] DAVIES, R. A temporal-logic approach to binding-time analysis. In *Proceedings, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science* (New Brunswick, July 1996), IEEE Computer Society Press, pp. 184–195.

[23] DAVIES, R., AND PFENNING, F. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)* (St. Petersburg Beach, Jan. 1996), pp. 258–270.

[24] DERSHOWITZ, N. Computing with rewrite systems. *Information and Control 65* (1985), 122–157.

[25] Di Cosmo, R. *Isomorphisms of Types: from λ-calculus to information retrieval and language design.* Birkhäuser, 1995.

[26] Engler, D. R., Hsieh, W. C., and Kaashoek, M. F. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL '96: The 23$^{\text{rd}}$ ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Jan. 1996), pp. 131–144.

[27] Friedman, D. P., and Wand, M. Reification: Reflection without metaphysics. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Aug. 1984), pp. 348–355.

[28] Glück, R., Hatcliff, J., and Jørgensen, J. Generalization in hierarchies of online program specialization systems. In *Logic-Based Program Synthesis and Transformation* (1999), vol. 1559 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 179–198.

[29] Glück, R., and Jørgensen, J. Efficient multi-level generating extensions for program specialization. In *Programming Languages: Implementations, Logics and Programs (PLILP'95)* (1995), S. D. Swierstra and M. Hermenegildo, Eds., vol. 982 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 259–278.

[30] Glück, R., and Jørgensen, J. Fast binding-time analysis for multi-level specialization. In *Perspectives of System Informatics* (1996), D. Bjørner, M. Broy, and I. V. Pottosin, Eds., vol. 1181 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 261–272.

[31] Glück, R., and Jørgensen, J. An automatic program generator for multi-level specialization. *LISP and Symbolic Computation 10*, 2 (1997), 113–158.

[32] Gomard, C. K., and Jones, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming 1*, 1 (Jan. 1991), 21–69.

[33] Gunter, C. A. *Semantics of Programming Languages.* MIT Press, 1992.

[34] Hatcliff, J., and Danvy, O. Thunks and the λ-calculus. *Journal of Functional Programming 7*, 3 (May 1997), 303–319.

[35] Hatcliff, J., and Glück, R. Reasoning about hierarchies of online specialization systems. In *Partial Evaluation* (1996), vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 161–182.

[36] Hindley, J. R. *Basic Simple Type Theory*, vol. 42 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, Cambridge, 1997.

[37] Hook, J., and Sheard, T. A semantics of compile-time reflection. Tech. Rep. CSE 93-019, Oregon Graduate Institute, 1993. Available from [66].

[38] Jones, N. D. Mix ten years later. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (June 1995), ACM Press, ACM Press, pp. 24–38.

[39] Jones, N. D. What not to do when writing an interpreter for specialisation. In *Partial Evaluation* (1996), vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–237.

[40] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[41] Jones, N. D., Sestoft, P., and Sondergraard, H. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications* (1985), vol. 202 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 124–140.

[42] Jørring, U., and Scherlis, W. L. Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida* (1986), ACM Press, pp. 86–96.

[43] Kieburtz, R. B., McKinney, L., Bell, J., Hook, J., Kotov, A., Lewis, J., Oliva, D., Sheard, T., Smith, I., and Walton, L. A software engineering experiment in software component generation. In *18th International Conference in Software Engineering* (Mar. 1996), pp. 542–553.

[44] Kohlbecker, E. E., Friedman, D. P., Felleisen, M., and Duba, B. Hygienic macro expansion. *ACM Conference on LISP and Functional Programming* (Aug. 1986), 151–161.

[45] Leone, M., and Lee, P. Deferred compilation: The automation of run-time code generation. Tech. Rep. CMU-CS-93-225, Carnegie Mellon University, Dec. 1993.

[46] Machkasova, E., and Turbak, F. A. A calculus for link-time compilation (extended abstract). (Unpublished manuscript.) By way of Franklyn A. Turbak (fturbak@wellesley.edu), June 1999.

[47] Maier, D., and Warren, D. S. *Computing with logic: Logic programming with Prolog.* The Benjamin/Cummings Publishing Company, Inc., 1988.

[48] Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences 17* (1978), 348–375.

[49] Milner, R., Tofte, M., Harper, R., and MacQueen, D. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[50] Mitchell, J. C. *Foundations for Programming Languages.* MIT Press, Cambridge, 1996.

[51] Mogensen, T. Æ. Efficient self-interpretation in lambda calculus. *Functional Programming 2*, 3 (July 1992), 345–364.

[52] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991).

[53] Moggi, E. A categorical account of two-level languages. In *Mathematics Foundations of Programming Semantics* (1997), Elsevier Science.

[54] Moggi, E., Taha, W., Benaissa, Z., and Sheard, T. An idealized MetaML: Simpler, and more expressive (includes proofs). Tech. Rep. CSE-98-017, OGI, Oct. 1998. Available from [66].

[55] Moggi, E., Taha, W., Benaissa, Z., and Sheard, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207. An extended version appears in [54].

[56] Muller, R. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems 14*, 4 (Oct. 1992), 589–616.

[57] Muller, R. A staging calculus and its application to the verification of translators. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Jan. 1994), pp. 389–396.

[58] Nielson, F. Program transformations in a denotational setting. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 359–379.

[59] Nielson, F. Correctness of code generation from a two-level meta-language. In *Proceedings of the European Symposium on Programming (ESOP 86)* (Saarbrücken, Mar. 1986), vol. 213 of *Lecture Notes in Computer Science*, Springer, pp. 30–40.

[60] Nielson, F. Two-level semantics and abstract interpretation. *Theoretical Computer Science 69*, 2 (Dec. 1989), 117–242.

[61] Nielson, F., and Nielson, H. R. Two-level semantics and code generation. *Theoretical Computer Science 56*, 1 (Jan. 1988), 59–133.

[62] Nielson, F., and Nielson, H. R. *Two-Level Functional Languages*. No. 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[63] Nielson, F., and Nielson, H. R. Multi-level lambda-calculi: An algebraic description. In *Partial Evaluation International Seminar, Dagstuhl Castle, Germany, Selected Papers* (1996), vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 338–354.

[64] Nielson, F., and Nielson, H. R. A prescriptive framework for designing multi-level lambda-calculi. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, June 1997), ACM Press, pp. 193–202.

[65] Okasaki, C. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1996.

[66] Oregon Graduate Institute Technical Reports. P.O. Box 91000, Portland, OR 97291-1000,USA. Available online from ftp://cse.ogi.edu/pub/tech-reports/README.html. Last viewed August 1999.

[67] Paulson, L. *ML for the Working Programmer*, second ed. Cambridge University Press, 1992.

[68] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[69] Quine, W. V. O. *Mathematical Logic*, revised ed. Harvard University Press, 1951. First published in 1940 by Norton.

[70] Quine, W. V. O. *Set Theory and Its Logic*. Harvard University Press, Cambridge, 1963.

[71] Quine, W. V. O. *From a Logical Point of View*, second, revised ed. Harper & Row, 1971.

[72] Reade, C. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, Wokingham, 1989.

[73] Rees, J., Clinger, W., Abelson, H., Adams IV, N. I., Bartley, D., Brooks, G., Dybvig, R. K., Friedman, D. P., Halstead, R., Hanson, C., Haynes, C. T., Kohlbecker, E., Oxley, D., Pitman, K. M., Rozas, G. J., Sussman, G. J., and Wand, M. Revised^4 report on the algorithmic language Scheme. Tech. Rep. AI Memo 848b, MIT Press, Nov. 1992.

[74] Reynolds, J. C. Definitional interpreters for higher-order programming languages. In *ACM National Conference* (1972), ACM, pp. 717–740.

[75] Reynolds, J. C. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation 11*, 4 (1998). (Reprinted from the proceedings of the 25th ACM National Conference (1972) [74]).

[76] Sabry, A., and Wadler, P. A reflection on call-by-value. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, May 1996), pp. 13–24.

[77] Sheard, T. A user's guide to TRPL, a compile-time reflective programming language. Tech. Rep. COINS 90-109, Dept. of Computer and Information Science, University of Massachusetts, 1990.

[78] Sheard, T. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct. 1991), 531–557.

[79] Sheard, T. Guide to using CRML, compile-time reflective ML. Available http://www.cse.ogi.edu/~sheard/CRML.html. Last viewed August 1999, Oct. 1993.

[80] Sheard, T. Type parametric programming. Tech. Rep. CSE-93-018, Oregon Graduate Institute, 1993. Available from [66].

[81] Sheard, T. A type-directed, on-line partial evaluator for a polymorphic language. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, June 1997), pp. 22–3.

[82] Sheard, T., and Nelson, N. Type safe abstractions using program generators. Tech. Rep. CSE-95-013, Oregon Graduate Institute, 1995. Available from [66].

[83] Shields, M., Sheard, T., and Jones, S. P. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1998), pp. 289–302.

[84] Smaragdakis, Y., and Batory, D. Distil: A transformation library for data structures. In *USENIX Conference on Domain-Specific Languages* (Oct. 1997), pp. 257–270.

[85] Smith, B. C. *Reflection and Semantics in a Procedural Language.* PhD thesis, Massachusetts Institute of Technology, Jan. 1982.

[86] Smith, B. C. Reflection and semantics in LISP. In *ACM Symposium on Principles of Programming Languages* (Jan. 1984), pp. 23–35.

[87] Smith, D. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering — Special Issue on Formal Methods 16*, 9 (Sept. 1990), 1024–1043.

[88] Steele, Jr., G. L., and Gabriel, R. P. The evolution of LISP. In *Proceedings of the Conference on History of Programming Languages* (New York, Apr. 1993), R. L. Wexelblat, Ed., vol. 28(3) of *ACM Sigplan Notices*, ACM Press, pp. 231–270.

[89] Stemple, D., Stanton, R. B., Sheard, T., Philbrow, P., Morrison, R., Kirby, G. N. C., Fegaras, L., Cooper, R. L., Connor, R. C. H., Atkinson, M. P., and Alagic, S. Type-safe linguistic reflection: A generator technology. Tech. Rep. FIDE/92/49, ESPRIT BRA Project 3070 FIDE, 1992.

[90] Sterling, L., and Shapiro, E. *The Art of Prolog*, second ed. MIT Press, Cambridge, 1994.

[91] Taha, W., Benaissa, Z.-E.-A., and Sheard, T. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming* (Aalborg, July 1998), vol. 1443 of *Lecture Notes in Computer Science*, pp. 918–929.

[92] Taha, W., and Hook, J. The anatomy of a component generation system. In *International Workshop on the Principles of Software Evolution* (Kyoto, Apr. 1998).

[93] Taha, W., and Sheard, T. Facets of multi-stage computation in software architectures. Tech. Rep. CSE-97-010, Oregon Graduate Institute, 1997. Available from [66].

[94] Taha, W., and Sheard, T. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam* (1997), ACM, pp. 203–217. An extended and revised version appears in [95].

[95] Taha, W., and Sheard, T. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science 248*, 1-2 (In Press).

[96] Takahashi, M. Parallel reductions in λ-calculus. *Information and Computation 118*, 1 (Apr. 1995), 120–127.

[97] Veldhuizen, T. L. C++ templates as partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (San Antonio, Jan. 1999), O. Danvy, Ed., University of Aarhus, Dept. of Computer Science, pp. 13–18.

[98] Wadler, P., Taha, W., and MacQueen, D. B. How to add laziness to a strict language withouth even being odd. In *Proceedings of the 1998 ACM Workshop on ML* (Baltimore, Sept. 1998), pp. 24–30.

[99] Wand, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation 10* (1998), 189–199.

[100] Wand, M., and Friedman, D. P. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Meta-Level Architectures and Reflection*, P. Maes and D. Nardi, Eds. Elsevier Science, 1988, pp. 111–134.

[101] Whitehead, A. N., and Russell, B. *Principia Mathematica.* Cambridge University Press, Cambridge, 1925.

[102] Wickline, P., Lee, P., and Pfenning, F. Run-time code generation and Modal-ML. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)* (Montreal, June 1998), pp. 224–235.

[103] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction.* Foundations of Computing series. MIT Press, Feb. 1993.

[104] Wright, A. K., and Felleisen, M. A syntactic approach to type soundness. *Information and Computation 115*, 1 (Nov. 1994), 38–94.

Biographical Note

Walid Taha was born on April 17th, 1972, in Tanta, Egypt. He spent the school year 1985/86 attending Corvallis High-School, Corvallis, Oregon. He completed the Oxford University General Certificate of Education Ordinary level examinations at Kuwait English School, Salwa, Kuwait, in 1987. Walid went off to study what was at first Mechanical Engineering, then Electrical Engineering, then eventually Computer Engineering, at Kuwait University. During his college studies, he interned a number of times at Asea Brown Boveri, Baden, Switzerland, under the supervision of Dr. Karl Imhof. At Kuwait University, Prof. Mansoor Sarwar's Programming Languages course introduced Walid to functional programming. Because of the Arabian Gulf conflict, a one week vacation turned into a school year at the University of Alexandria, and two internships (one with Schlumberger in Alexandria, Egypt, and Linz, Austria, and one with Asea Brown Boveri, Baden, Switzerland). Eventually, Walid meandered back to Kuwait University and finished his senior project on compiling LISP to Pascal, and completed the Bachelors of Science in Computer Science and Engineering in 1993.

After graduation, Walid spent two months at the Göthe Institut, Bremen, Germany, studying German on a scholarship from the German government. He then returned to Kuwait to contemplate the future. When he got tired, he joined the Masters program in Electrical Engineering for one semester, and then worked as a Software Engineer at the Office of the Vice President for Planning Affairs at Kuwait University.

During this year, Prof. Khalid Al-Saqabi of Kuwait University was on sabbatical at some "Oregon Graduate Institute". Prof. Khalid was obviously having too much fun at OGI, and insisted that Walid apply there for the PhD program. And so he did.

At OGI, Prof. David Maier made math look interesting again. Prof. James Hook lured Walid into spending the next four years of his life working on semantics in PacSoft. Walid's advisor, Prof. Tim Sheard, was starting to work on a most intriguing creature called MetaML.

During his Doctoral studies at the Oregon Graduate Institute, Beaverton, Oregon, Walid interned for four months at Lucent Bell Laboratories, Murry Hill, New Jersey, working with Dr. Phil Wadler on implementing Wadler's proposal for lazy ML datatypes.

Currently, Walid is a Post-doctoral Fellow at the Department of Computing Sciences, Chalmers University of Technology, Gothenborg, Sweden.