Multi-Stage Programming: Axiomatization and Type Safety

extended abstract

Walid Taha & Zine-El-Abidine Benaissa & Tim Sheard

January 14, 1998

Abstract

Multi-staged programming provides a new paradigm for constructing efficient solutions to complex problems. Techniques such as program generation, multi-level partial evaluation, and run-time code generation respond to the need for general purpose solutions which do not pay run-time interpretive overheads. This paper provides a foundation for the formal analysis of one such system.

We introduce a multi-stage language and present its axiomatic, reduction, and natural semantics. Our axiomatic semantics is an extension of the call-by-value λ-calculus with staging constructs. We demonstrate the soundness of the axiomatic semantics with respect to the natural semantics. We show that staged-languages can "go Wrong" in new ways, and devise a type system that screens out such programs. Finally, we present a proof of the soundness of this type system with respect to the reduction semantics, and show how to extend this result to the natural semantics.

1 Introduction

Recently, there has been significant interest in various forms of multi-stage computation, including program generation [3, 26], multi-level partial evaluation [11, 12], and run-time code generation [1, 5, 4, 8, 9, 13, 15, 16, 22]. Such techniques combine both the software engineering advantages of general purpose systems and the efficiency of specialized ones.

Because such systems execute generated code *never inspected by human eyes* it is important to use formal analysis to guarantee properties of this generated code. We would like to guarantee statically that a program generator synthesizes only programs with properties such as: type-correctness, global references only to names in scope, and local names which do not inadvertently hide global references.

In previous work [25], we introduced a multi-stage programming language called MetaML. In that work we introduced four staging annotations to control the order of evaluation of terms. We argued that staged programs are an important mechanism for constructing general purpose systems with the efficiency of specialized ones, and addressed engineering issues necessary to make such systems usable by programmers. We introduced an operational semantics and a type system to screen out bad programs, but we were unable to prove the soundness of the type system.

Further investigation revealed important subtleties that were not previously apparent to us. In this paper, we report on work rectifying some of the practical limitations of our previous work. In contrast to our earlier work that focused on implementations and problem solving using multi-staged programs, this paper reports on a more abstract treatment of MetaML's foundations. The key results reported in this paper are as follows:

1. An axiomatic semantics and a reduction semantics for a core sub-language of MetaML.

2. A characterization of the additional ways in which a staged program can "go Wrong".

3. A type system to screen out such programs.

4. A soundness proof for the type system with respect to the reduction semantics using the syntactic approach to type-soundness of Wright and Felliesen [27].

5. A natural semantics that chooses the order in which rules are applied.

6. The soundness of the axiomatic semantics with respect to the natural semantics.

These results form a strong, tightly-woven foundation which gives us both a better understanding of MetaML, and more confidence in the well-foundedness of the multi-stage paradigm. The axiomatic semantics provides us with an equational theory for formally reasoning about the equivalence of MetaML programs, and the reduction semantics is an abstract characterization of the notion of staged computation. The natural semantics provides us with a deterministic strategy for implementing multi-stage computation. The soundness of the axiomatic semantics with respect to the natural semantics formally demonstrates that results based on the reductions semantics are also applicable to our implementation. Finally, formally proving the soundness of the type system with respect to the reduction semantics ensures to us that well-typed programs are well-behaved.

1.1 What are Staged Programs All About?

In staging a program, the user has control over the order of evaluation of terms. This is done by using staging annotations. In MetaML the staging annotations are Brackets `<>`, Escape `~` and `run`. An expression <e> defers the computation of e; ~e splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and `run` e evaluates e to obtain a deferred expression, and then evaluates this deferred expression. It is important to note that ~e is only legal within lexically enclosing Brackets. To illustrate, consider the script of a small MetaML session below:

```
-| val pair = (3+4,<3+4>);
val pair = (7,<3+4>) : (int  * <int>)

-| fun f (x,y) = < 8 - ~y >;
val f = fn : ('a  * <int>) -> <int>

-| val code = f pair;
val code = <8 - (3+4)> : <int>

-| run code;
val it = 1 : int
```

The first declaration defines a variable `pair`. The first component of the pair is evaluated, but the evaluation of the second component is deferred by the Brackets. Brackets in types such as `<int>` are read "Code of int", and distinguish values such as `<3+4>` from values such as `7`. The second declaration illustrates that code can be abstracted over, and that it can be spliced into a larger piece of code. The third declaration applies the function `f` to `pair` performing the actual splicing. And the last declaration evaluates this deferred piece of code.

To give a brief feel for how MetaML is used to construct larger pieces of code at run-time consider:

```
-| fun mult x n = if n=0 then <1> else < ~x * ~(mult x (n-1)) >;
val mult = fn  : <int> -> int  -> <int>
```

```
-| val cube = <fn y => ~(mult <y> 3)>;
val cube = <fn a => a * (a * (a * 1))> : <int  -> int>

-| fun exponent n = <fn y => ~(mult <y> n)>;
val exponent = fn  : int  -> <int  -> int>
```

The function `mult`, given an integer piece of code `x` and an integer `n`, produces a piece of code that is an `n`-way product of `x`. This can be used to construct the code of a function that performs the `cube` operation, or generalized to a generator for producing an exponentiation function from a given exponent `n`. Note how the looping overhead has been removed from the generated code. This is the purpose of program staging and it can be highly effective as discussed elsewhere [4, 10, 13, 17, 22, 25]. In this paper we move away from how staged languages are used and address their foundations.

2 The λ-R Language

The λ-R language represents the core of MetaML. It has the following syntax:

$$e := \; i \mid x \mid e\,e \mid \lambda x.e \mid \texttt{<}e\texttt{>} \mid \texttt{~}e \mid \texttt{run}\ e$$

which includes the normal constructs of the λ-calculus, integer constants, and the three additional staging constructs.

To define the semantics of Escape, which is dependent on the surrounding context, we choose to explicitly annotate all terms with their level. The *level* of a term is the number of Brackets minus the number of Escapes surrounding that term. We define *level-annotated* terms as follows:

$$\begin{array}{lcl} a^0 & := & i^0 \mid x^0 \mid (a^0\,a^0)^0 \mid (\lambda x.a^0)^0 \mid \texttt{<}a^1\texttt{>}^0 \mid (\texttt{run}\ a^0)^0 \\ a^{n+1} & := & i^{n+1} \mid x^{n+1} \mid (a^{n+1}\,a^{n+1})^{n+1} \mid (\lambda x.a^{n+1})^{n+1} \mid \texttt{<}a^{n+2}\texttt{>}^{n+1} \mid (\texttt{~}a^n)^{n+1} \mid (\texttt{run}\ a^{n+1})^{n+1} \end{array}$$

Note that Escape never appears at level 0 in a level-annotated term. We define a λ-R program as a closed term a^0. Hence, example programs are $(\lambda x.x^0)^0$ and $\texttt{<<}((\lambda x.(x^2\,x^2)^2)^2\,5^2)^2\texttt{>}^1\texttt{>}^0$.

2.1 Values

It is instructive to think of values as the set of terms we consider to be acceptable results from a computation. Values are defined as follows:

$$\begin{array}{lcl} v^0 & := & i^0 \mid x^0 \mid (\lambda x.a^0)^0 \mid \texttt{<}v^1\texttt{>}^0 \\ v^1 & := & i^1 \mid x^1 \mid (v^1\,v^1)^1 \mid (\lambda x.v^1)^1 \mid \texttt{<}v^2\texttt{>}^1 \mid (\texttt{run}\ v^1)^1 \\ v^{n+2} & := & i^{n+2} \mid x^{n+2} \mid (v^{n+2}\,v^{n+2})^{n+2} \mid (\lambda x.v^{n+2})^{n+2} \mid \texttt{<}v^{n+3}\texttt{>}^{n+2} \mid (\texttt{~}v^{n+1})^{n+2} \mid (\texttt{run}\ v^{n+2})^{n+2} \end{array}$$

The set of values for λ-R has three notable points. First, values can be bracketed expressions. This means that computations can return pieces of code representing other programs. Second, values can contain applications such as $(\lambda y.y^1)^1\ (\lambda x.x^1)^1$. Third, there are no level 1 Escapes in values. We take advantage of this important property of values in many proofs and propositions in our present work.

Because each rule in the inductive definition above is an instance of one of the rules given in the inductive definition for level-annotated terms it is easy to show that values are a subset of level-annotated terms.

2.2 Contexts

We generalize the notion of contexts [2] to a notion of *annotated contexts*:

$$
\begin{array}{rcl}
c^0 & := & [\,]^0 \mid (c^0\ a^0)^0 \mid (a^0\ c^0)^0 \mid (\lambda x.c^0)^0 \mid \texttt{<}c^1\texttt{>}^0 \mid (\texttt{run}\ \ c^0)^0 \\
c^{n+1} & := & [\,]^{n+1} \mid (c^{n+1}\ a^{n+1})^{n+1} \mid (a^{n+1}\ c^{n+1})^{n+1} \mid (\lambda x.c^{n+1})^{n+1} \mid \\
 & & \texttt{<}c^{n+2}\texttt{>}^{n+1} \mid (\texttt{\~{}}c^n)^{n+1} \mid (\texttt{run}\ \ c^{n+1})^{n+1}
\end{array}
$$

where [] is a *hole*. When instantiating an annotated context $c^n[\,]^m$ to a term e^m we write $c^n[e^m]$.

2.3 Promotion and Demotion

The axioms of MetaML remove Brackets from level-annotated terms. To maintain the consistency of the level-annotations we need an inductive definition for incrementing and decrementing all annotations on a term. We call these operations *promotion* and *demotion*.

$$
\begin{array}{rcl@{\qquad}rcl}
 & \textbf{Promotion} & & & \textbf{Demotion} & \\
x^n\uparrow & = & x^{n+1} & x^{n+1}\downarrow & = & x^n \\
(a_1\ a_2)^n\uparrow & = & (a_1\uparrow\ \ a_2\uparrow)^{n+1} & (a_1\ a_2)^{n+1}\downarrow & = & (a_1\downarrow\ \ a_2\downarrow)^n \\
(\lambda x.a)^n\uparrow & = & (\lambda x.a\uparrow)^{n+1} & (\lambda x.a)^{n+1}\downarrow & = & (\lambda x.a\downarrow)^n \\
\texttt{<}a\texttt{>}^n\uparrow & = & \texttt{<}a\uparrow\texttt{>}^{n+1} & \texttt{<}a\texttt{>}^{n+1}\downarrow & = & \texttt{<}a\downarrow\texttt{>}^n \\
(\texttt{\~{}}a)^{n+1}\uparrow & = & (\texttt{\~{}}a\uparrow)^{n+2} & (\texttt{\~{}}a)^{n+2}\downarrow & = & (\texttt{\~{}}a\downarrow)^{n+1} \\
(\texttt{run}\ \ a)^n\uparrow & = & (\texttt{run}\ \ a\uparrow)^{n+1} & (\texttt{run}\ \ a)^{n+1}\downarrow & = & (\texttt{run}\ \ a\downarrow)^n \\
i^n\uparrow & = & i^{n+1} & i^{n+1}\downarrow & = & i^n
\end{array}
$$

Promotion is a total function over level-annotated terms and is defined by a simple inductive definition. Demotion is a partial function over level-annotated terms. Demotion is undefined on terms Escaped at level 1, and on level 0 terms in general.

An important property of demotion is that while it is partial over level-annotated terms it is total over values. Proof of this is a simple induction on the structure of values.

2.4 Substitution

The definition of substitution is standard for the most part. In this paper we are concerned only with the substitution of values for variables. When the level of a value is different from the level of the term in which it is being substituted, promotion (or demotion, whichever is appropriate) is used to correct the level of the subterm.

$$
\begin{array}{rcll}
i^n[x^n := v^n] & = & i^n & \\
x^n[x^n := v^n] & = & v^n & \\
y^n[x^n := v^n] & = & y^n & x \neq y \\
(a_1\ a_2)^n[x^n := v^n] & = & ((a_1[x^n := v^n])\ (a_2[x^n := v^n]))^n & \\
(\lambda x.a_1)^n[x^n := v^n] & = & (\lambda x.a_1)^n & \\
(\lambda y.a_1)^n[x^n := v^n] & = & (\lambda y'.(a_1[y^n := y'^n][x^n := v^n]))^n \quad y' \notin FV(v^n),\ y' \notin FV(a_1) & x \neq y \\
\texttt{<}a_1\texttt{>}^n[x^n := v^n] & = & \texttt{<}a_1[x^{n+1} := v^n\uparrow]\texttt{>}^n & \\
(\texttt{\~{}}a_1)^{n+1}[x^{n+1} := v^{n+1}] & = & (\texttt{\~{}}(a_1[x^n := v^{n+1}\downarrow]))^{n+1} & \\
(\texttt{run}\ \ a_1)^n[x^n := v^n] & = & (\texttt{run}\ \ (a_1[x := v^n]))^n &
\end{array}
$$

This function is total because both promotion and demotion are total over values. A richer notion of demotion is need to perform substitution of a variable by any expression. This generalization is beyond the scope of this paper.

2.5 Axiomatization and Reduction Semantics of λ-R

The axiomatic semantics describes an equivalence between two level-annotated terms. Axioms can be thought of as pattern-based equivalence rules, and are applicable in a context-independent way to any subterm that they match. The three axioms we will introduce can each be given a natural orientation or direction, reducing "bigger" terms to "smaller" terms. This provides a reduction semantics.

Axiomatic	Reduction
$((\lambda x.e^n)^n v^n)^n \;=\; e^n[x := v^n]$	$((\lambda x.e^n)^n v^n)^n \;\xrightarrow{\beta}\; e^0[x := v^n]$
$(\texttt{run}\ \texttt{<}v^{n+1}\texttt{>}^n)^n \;=\; v^{n+1}\downarrow$	$(\texttt{run}\ \texttt{<}v^{n+1}\texttt{>}^n)^n \;\xrightarrow{run}\; v^{n+1}\downarrow$
$(\texttt{\~{}}\texttt{<}e^{n+1}\texttt{>}^n)^{n+1} \;=\; e^{n+1}$	$(\texttt{\~{}}\texttt{<}e^{n+1}\texttt{>}^n)^{n+1} \;\xrightarrow{esc}\; e^{n+1}$

We write λ-R $\vdash M = N$ when $M = N$ is provable by the above axioms and the classical inference rules of an equational theory, and we write $\xrightarrow{*}$ for the reflexive, transitive, context closure of $\longrightarrow$.

Theorem 1 (Confluence). *The reduction semantics is confluent.*

Proof. Using a notion of parallel reduction and a Strip Lemma, following closely the development in [2, pages 277–283]. $\square$

Corollary 2 (Church-Rosser). *The axiomatic semantics is Church-Rosser.*

3 Faulty Terms

Under the reduction semantics, when a term has been sufficiently reduced, we would like such a term to be a *value*, but this is not always the case. If no rules apply, and the term is not a value, we say that such a term is *stuck* [27]. There are four contexts in which such terms can arise:

1. A non-λ value in a function position in an application (at level 0). This is the familiar form of undesirable behavior arising whenever the pure λ-calculus is extended with constants. For example, $(\texttt{<}5^1\texttt{>}^0\ 3^0)^0$ is stuck because $\texttt{<}5^1\texttt{>}^0$ is a piece of code, not a λ-abstraction. This term is not a value and contains no redex.

2. A variable appears at a level lower than the level at which it was bound. This is the key, distinguishing form of undesirable behavior in multi-stage computation [25]. For example: $\texttt{<}(\lambda x.\texttt{\~{}}(x^0)^1)^1\texttt{>}^0$ is stuck since x is used at level 0 but bound at level 1.

3. A non-Bracket value is the argument to Run. For example: $(\texttt{run}\ 7^0)^0$ is stuck since 7^0 is an integer and not a piece of code.

4. A non-Bracket value is the argument to Escape. For example: $\texttt{<}(4^1 + \texttt{\~{}}(7^0)^1)^1\texttt{>}^0$

We wish to consider as *faulty*, terms in the form above. We will show that if a term is typable, then it is not faulty, and neither can it reduce to a faulty term. We formalize this notion in the next sections.

We can now present the following formal specification for the set of faulty terms F:

1. $c[((\texttt{<}e^{n+1}\texttt{>})^n\ e')^n] \in F$ Non-λ terms in an application like: $(5^0\ 3^0)^0$ and $(\texttt{<}5^2\texttt{>}^1\ 3^1)^1$
   $c[(i^n\ e')^n] \in F$

2. $c[(\lambda x.c'[x^n])^m] \in F$ where $m > n$. Variables at too low a level like: $\texttt{<}(\lambda x.\texttt{\~{}}(x^0)^1)^1\texttt{>}^0$

3. $c[(\texttt{run}\ (\lambda x.e)^n)^n] \in F$ Non-Bracket in Run like: $(\texttt{run}\ (\lambda x.x)^0)^0$ and $(\texttt{run}\ 4^3)^3$
   $c[(\texttt{run}\ i^n)^n] \in F$

4. $c[(\texttt{\~{}}(\lambda x.e)^n)^{n+1}] \in F$ Non-Bracket in Escape like: $\texttt{<}(4^1 + \texttt{\~{}}((\lambda x.x)^0)^1)^1\texttt{>}^0$ and $\texttt{<}(4^3 + \texttt{\~{}}(5^2)^3)^3\texttt{>}^2$
   $c[(\texttt{\~{}}(i^n))^{n+1}] \in F$

The success of our specification of faulty expressions depends on whether they help us characterize the behavior of our reduction semantics. The following lemma is an example of such a characterization, and is needed for our proof of type soundness.

Lemma 3 (Uniform Evaluation). *Let e^n be a closed term. If e^n is not faulty then either it is a value or it contains a redex.*

Proof: By induction on the structure of e^n.

4 Type System

The main obstacle to defining a sound type system for our language is the interaction between Run and Escape. While this is problematic, it adds significantly to the expressiveness of a staged language [23], so it is worthwhile overcoming the difficulty. The problem is that Escape allows Run to appear inside a Bracketed λ-abstraction, and it is possible for Run to "drop" that λ-bound variable to a level lower than the level at which it is bound. The following example illustrates the phenomenon:

$$\mathtt{<}(\lambda x.(\texttt{\~{}}(\mathtt{run}\ \mathtt{<}x^1\mathtt{>}^0)^0)^1)^1\mathtt{>}^0 \to (\lambda x.(\texttt{\~{}}x^0)^1)^1$$

To avoid this problem, for each λ-abstraction we need to count the number of surrounding Runs for each occurrence of its bound variable (here x^1) in its body. We use this count to check that there are enough Brackets around each formal parameter to execute all surrounding Runs without leading to a faulty term.

The type system for λ-R is defined by a judgment $\Delta \vdash e^n : \tau, m$, where e^n is our well-typed expression, τ is the type of the expression, m is the number of the surrounding Run annotations of e^n and Δ is the environment assigning types to term variables.

Syntax

types	τ	$::=$	$\tau \to \tau \mid \mathtt{<}\tau\mathtt{>} \mid \mathtt{int}$
type assignments	Δ	$::=$	$x \mapsto (\tau, j)^i ; \Delta \mid \{\}$
judgments	J	$::=$	$\Delta \vdash t : \tau, m$

Type System

$$\frac{\Delta(x) = (\tau, j)^i \quad i + m \leq n + j}{\Delta \vdash x^n : \tau, m}\ \text{Var} \qquad \frac{}{\Delta \vdash i^n : \mathtt{int}, m}\ \text{Int}$$

$$\frac{\Delta \vdash e^n : \mathtt{<}\tau\mathtt{>}, m+1}{\Delta \vdash (\mathtt{run}\ e^n)^n : \tau, m}\ \text{Run}$$

$$\frac{\Delta \vdash e^{n+1} : \tau, m}{\Delta \vdash \mathtt{<}e^{n+1}\mathtt{>}^n : \mathtt{<}\tau\mathtt{>}, m}\ \text{Bra} \qquad \frac{\Delta \vdash e^n : \mathtt{<}\tau\mathtt{>}, m}{\Delta \vdash (\texttt{\~{}}e^n)^{n+1} : \tau, m}\ \text{Esc}$$

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

The type system employs a number of mechanisms to reject terms that either are, or can reduce to faulty terms. The **App** rule has the standard role, and rejects non-functions applied to arguments.

The **Escape** and **Run** rules require that their operand must have type Code. This means terms such as `run` 5 and $\mathtt{<}\lambda x.\texttt{\~{}}5\mathtt{>}$ are rejected. But while this restriction in the **Escape** and **Run** rules rejects faulty terms, it is not enough to reject all terms that can be reduced to faulty terms. The first example of such a term is $\mathtt{<}\lambda x.\texttt{\~{}}(\mathtt{run}\ \mathtt{<}x\mathtt{>})\mathtt{>}$ which would be typable if we use only the restrictions discussed above, but reduces to the term $\mathtt{<}\lambda x.\texttt{\~{}}x\mathtt{>}$ which would not be typable. The second examples involves an application $(\lambda f.{<}\lambda x.\tilde{}(f{<}x{>}){>})(\lambda x.\mathtt{run}\ x)$ which would also be typable, but reduces to ${<}\lambda x.\tilde{}x{>}$. To reject such terms we need the **Var** rule.

The **Var** rule is instrumented with the condition $i + m \leq n + j$. Here i is the number of Bracket's surrounding the λ-abstraction where the variable was bound, m is the number of Runs surrounding this occurence of the variable, n is the number of Brackets surrounding this occurence of the variable, and j is the number of Runs surrounding the λ-abstraction where it was bound. This ensures that every variable has more Brackets than Runs surrounding it.

In previous work, we have attempted to avoid these two kinds of problems using two distinct mechanisms: First, the argument of Run cannot contain free variables, and second, we prohibit the λ-abstraction of Run. We used unbound polymorphic type variable names in a scheme similar to that devised by Launchbury and Peyton Jones for ensuring the safety of state in Haskel [14]. It turns out that not allowing any free variables is too strong, and that using polymorphism was too weak. It is better to simply take account of the number of surrounding occurrences of Run in the **Var** rule. This way we ensure that if Run is ever in a λ-abstraction, it can only strip away Brackets that are explicitly apparent in that λ-abstraction.

5 Type Soundness of the Reduction Semantics

The type soundness proof closely follows the subject reduction proofs of Wright and Felliesen [27]. Once the reduction semantics and type system have been defined, the syntactic type soundness proof proceeds as follows:

1. Show that reduction in the standard reduction semantics preserves typing. This is called *subject reduction.*

2. Show that faulty terms are not typable.

If programs are well-typed, then the two results above can be used as follows: By (1), evaluation of a well-typed program will only produce well-typed terms. By Lemma 3, every such term is either faulty, or a value, or contains a redex. The first case is impossible by (2). Thus the program either reduces to a well-typed value or it diverges.

5.1 Subject Reduction

The Subject Reduction Lemma states that a well-typed term remains well-typed under reduction. The proof relies on the Demotion, Promotion and Substitution Type Preservation Lemmas. First we need to introduce two operations on the environment assigning types to term variables:

$$\Delta \uparrow_{(q,p)} (x) = (\tau, j+q)^{i+p} \text{ iff } \Delta(x) = (\tau, j)^i$$
$$\Delta \downarrow_{(q,p)} (x) = (\tau, j)^i \text{ iff } \Delta(x) = (\tau, j+q)^{i+p}$$

These two operations map environments to environments. They are needed in the Promotion and Demotion Lemmas. They provide an environment necessary to derive a valid judgement for a promoted or demoted well-typed value. Notice that we have the following two properties:

$$(\Delta \uparrow_{(q,p)}) \uparrow_{(i,j)} = \Delta \uparrow_{(q+i,p+j)} \quad \text{and} \quad (\Delta \uparrow_{(q+i,p+j)}) \downarrow_{(i,j)} = \Delta \uparrow_{(q,p)}$$

We write$v \uparrow^p$ and $v \downarrow^p$, respectively, for an abbreviation of p applications of $\uparrow$ and $\downarrow$ to v. Note that this operation is different from $\uparrow_{(q,p)}$ and $\downarrow_{(q,p)}$ which is a function on environments assigning types to term variables.

Lemma 4 (Demotion). *If $q \leq p$ and $\Delta_2 \downarrow_{(q,p)}$ is defined and $\Delta_1 \cup \Delta_2 \vdash v^{n+p} : \tau, m+q$ then $\Delta_1 \cup (\Delta_2 \downarrow_{(q,p)}) \vdash v^{n+p} \downarrow^p : \tau, m$.*

Proof. By induction on the structure of v^{n+p}. We develop only the variable case $v^{n+p} = x^{n+p}$. There are only two possible sub-cases, which are:

$$\frac{\Delta_1(x) = (\tau, j)^i \quad i+m+q \leq n+j+p}{(\Delta_1 \cup \Delta_2) \vdash x^{n+p} : \tau, m+q} \text{(Var)}$$

By hypothesis $q \leq p$ implies $m+i \leq n+j$. Hence $(\Delta_1 \cup (\Delta_2 \downarrow_{(q,p)})) \vdash v^{n+p} \downarrow^p : \tau, m$.

$$\frac{\Delta_2(x) = (\tau, j+q)^{i+p} \quad i+m+2q \leq n+j+2p}{(\Delta_1 \cup \Delta_2) \vdash x^{n+p} : \tau, m+q} \text{(Var)}$$

Similar to the above sub-case. □

Lemma 5 (Promotion). *Let $q \leq p$. If $\Delta \vdash v^n : \tau, m$ then $\Delta_1 \cup (\Delta_2 \uparrow_{(q,p)}) \vdash v^n \uparrow^p : \tau, m+q$.*

Proof. By induction on v^n. □

Lemma 6 (Substitution). *If $j \leq m$ and $\Delta_1 \cup (x \mapsto (\tau', j)^i; \Delta_2) \vdash e^n : \tau, m$ and $\Delta_1 \vdash v^i : \tau', j$ then one of the following three judgments holds.*

1. $\Delta_1 \vdash e^n[x^n := v^i \uparrow^{n-i}] : \tau, m$ *if* $n > i$.
2. $\Delta_1 \vdash e^n[x^n := v^i \downarrow^{i-n}] : \tau, m$ *if* $n < i$
3. $\Delta_1 \vdash e^n[x^n := v^n] : \tau, m$, *otherwise*

Proof. By induction on the structure e^n. If $e^n = x^n$ then we have:

$$\frac{\Delta(x) = (\tau, j)^i \quad m+i \leq n+j}{\Delta_1 \cup (x \mapsto (\tau, j)^i; \Delta_2) \vdash x^n : \tau, m}$$

- If $n < i$ and by the hypothesis $j \leq m$ then $m+i > n+j$. Hence $\Delta_1 \cup (x \mapsto (\tau, j)^i; \Delta_2) \vdash x^n : \tau, m$ cannot be typable.
- if $n > i$ then $m - j < n - i$ and the Promotion Lemma 5 applies.
- $i = n$ and by hypothesis $j \leq m$ and $m+i \leq n+j$ then $j = m$. Then, $\Delta_1 \vdash e^n[x^n := v^n] : \tau, m$.

□

Corollary 7 (β Rule). *If $\Delta \vdash ((\lambda x.e^n)^n \, v^n)^n : \tau, m$ then $\Delta \vdash e^n[x^n := v^n] : \tau, m$.*

Lemma 8 (Escape Rule). *If $\Delta \vdash (\texttt{\~{}<}e^{n+1}\texttt{>}^n)^n : \tau, m$ then $\Delta \vdash e^n : \tau, m$.*

Proof. Straightforward from the type system. □

Lemma 9 (Run Rule). *If $\Delta \vdash (\texttt{run <}v^{n+1}\texttt{>}^n)^n : \tau, m$ then $\Delta \vdash v^1 \downarrow : \tau, m$.*

Proof. If $\Delta \vdash (\texttt{run <}v^{n+1}\texttt{>}^n)^n : \tau, m$ then $\Delta \vdash v^{n+1} : \tau, m+1$ is valid. By Demotion Lemma 4, $\Delta \vdash v^{n+1} \downarrow : \tau, m$ is valid. □

Proposition 10. *If $\Delta \vdash e_1^n : \tau, m$ and $e_1^n \to e_2^n$ then $\Delta \vdash e_2^n : \tau, m$.*

Proof. By induction on the structure of e_1^n. If the rewrite is at the root then use Lemmas 8 and 9, and Corollary 7. If e_1^n contains a redex then apply induction hypothesis. □

Proposition 11 (Subject Reduction). *If $\Delta \vdash e_1^n : \tau, m$ and $e_1^n \xrightarrow{*} e_2^n$ then $\Sigma \, \Delta \vdash e_2^n : \tau, m$.*

Proof. By induction on the length of the derivation. □

5.2 Faulty Terms

Lemma 12 (Faulty Terms are Not Typable). *If $e \in F$ then there is no Δ, t, a such that $\Delta \vdash e : t, a$.*

Proof. By case analysis over the structure of e. Let $e = c_1[(\lambda x.c_2[x^n])^i]$ such that $n < i$, that is, $i = n + k_1 + 1$. Assume that $\Delta \vdash e : \tau, m$. This implies that $x \mapsto (\tau', j)^i \Delta' \vdash x^n : \tau', p$. This means that $i + p \leq n + j$. Because $p = j + k_2$ then $j \leq p$. This implies that $n + k + 1 + 1 + j + k_2 \leq n + j$ which is impossible. The other cases are straight-forward. □

6 Natural Semantics

In previous work, we defined core MetaML by a natural semantics [25]. While this style of presentation is closer to the implementation of MetaML than the reduction semantics presented in this paper, it is more complex. We have found that it was easier to prove type soundness first with respect to the reduction semantics, and then to extend this result to the natural semantics.

In this paper, we present a more concise natural semantics for MetaML than the one we have presented in previous work [25]:

$$\frac{}{(\lambda x.e^0)^0 \hookrightarrow (\lambda x.e^0)^0} \qquad \frac{{e_1}^0 \hookrightarrow (\lambda x.e^0)^0 \quad {e_2}^0 \hookrightarrow {v_1}^0 \quad (e^0[x := v_1^0]) \hookrightarrow v_2^0}{(e_1^0\ e_2^0)^0 \hookrightarrow {v_2}^0} \qquad \frac{e^0 \hookrightarrow \texttt{<}v^1\texttt{>}^0}{\texttt{\~{}}(e^0)^1 \hookrightarrow v^1}$$

$$\frac{{e_1}^0 \hookrightarrow \texttt{<}v_1^1\texttt{>}^0 \quad (v_1^1 \downarrow)^0 \hookrightarrow {v_2}^0}{(\texttt{run}\ e_1^0)^0 \hookrightarrow {v_2}^0} \qquad \frac{{e_1}^{n+1} \hookrightarrow {e_3}^{n+1} \quad {e_2}^{n+1} \hookrightarrow {e_4}^{n+1}}{(e_1^{n+1}\ e_2^{n+1})^{n+1} \hookrightarrow (e_3^{n+1} e_4^{n+1})^{n+1}} \qquad \frac{}{x^{n+1} \hookrightarrow x^{n+1}}$$

$$\frac{{e_1}^{n+1} \hookrightarrow {e_2}^{n+1}}{(\lambda x.e_1^{n+1})^{n+1} \hookrightarrow (\lambda x.e_2^{n+1})^{n+1}} \qquad \frac{{e_1}^{n+1} \hookrightarrow {e_2}^{n+1}}{\texttt{\~{}}(e_1^{n+1})^{n+2} \hookrightarrow \texttt{\~{}}(e_2^{n+1})^{n+2}} \qquad \frac{{e_1}^{n+1} \hookrightarrow {e_2}^{n+1}}{\texttt{<}e_1^{n+1}\texttt{>}^n \hookrightarrow \texttt{<}e_2^{n+1}\texttt{>}^n}$$

$$\frac{{e_1}^{n+1} \hookrightarrow {e_2}^{n+1}}{(\texttt{run}\ e_1^{n+1})^{n+1} \hookrightarrow (\texttt{run}\ e_2^{n+1})^{n+1}} \qquad \frac{}{i^n \hookrightarrow i^n}$$

A key property of this presentation is that it avoids the explicit use of a *gensym* or *newname* function for renaming abstractions at levels greater than zero. This improvement avoids the problems that Moggi points out regarding the use of such stateful functions in defining the semantics of two-level languages [18].

Now we move on to present some fundamental results about the untyped λ-R language, and use these results, in addition to the soundness of the type system with respect to the reduction semantics, to prove the soundness of the type system with respect to the natural semantics.

We say that two terms e_1 and e_2 are *observationally equivalent*, written $e_1 \sim e_2$, if for any context $c[]$ such that both $c[e_1]$ and $c[e_2]$ are closed, then $c[e_1]^0 \hookrightarrow {v_1}^0$ if and only if $c[e_2]^0 \hookrightarrow {v_2}^0$, and $v_1^0 = i^0$ if and only if $v_2^0 = i^0$ when both relations are defined.

Lemma 13. *If $e^n \hookrightarrow v^n$ then $e^n \overset{*}{\longrightarrow} v^n$.*

Proof. By induction on the proof tree for $e^n \hookrightarrow v^n$. □

Lemma 14. *If $e \overset{*}{\longrightarrow} v$ then $e \hookrightarrow v'$.*

Proof. This proof requires a Standardization Theorem along the lines of Plotkin [20], but one extended to deal with Brackets, Escape and Run. We omit the details for the sake of brevity. Please see the technical report for the full details [24]. □

Corollary 15. *There exists a value v such that $\lambda\text{-}R \vdash e = v$ if and only if $e \hookrightarrow v'$.*

Proof. Consequence of Lemmas 14 and 13. $\square$

Theorem 16 (Soundness of Axiomatic Semantics). *If $\lambda\text{-}R \vdash e_1 = e_2$ then $e_1 \sim e_2$.*

Proof. If $e_1 \hookrightarrow v_1$ then by Corollary 15 $\lambda\text{-R} \vdash e_1 = v_1$. Hence, $\lambda\text{-R} \vdash e_2 = v_1$. By Corollary 15, there exists a value v_2 such that $e_2 \hookrightarrow v_2$. By Lemma 13, $\lambda\text{-R} \vdash v_1 = v_2$. Since the axiomatic semantics is Church-Rosser, we have $v_1 \xrightarrow{*} v$ and $v_2 \xrightarrow{*} v$. Thus, $e_1 \sim e_2$ $\square$

We define undesirable behavior in the natural semantics in the classical manner: we introduce a new "value" Wrong, written $\top$, and a set of rules complementing the rules of the natural semantics, and returning $\top$ in all these new cases. We call the combination of these two sets of rules the *augmented* natural semantics, and denote it by $\overset{\top}{\hookrightarrow}$.

Lemma 17. *If $e \overset{\top}{\hookrightarrow} \top$ then $e \xrightarrow{*} f$ and $f \in F$ and $f \neq v$.*

Proof. By induction on the proof tree of the augmented natural semantics $\overset{\top}{\hookrightarrow}$. $\square$

Theorem 18 (Type Soundness). *If $\Delta \vdash e : \tau, m$ and $e \overset{\top}{\hookrightarrow} e'$ then $e' \neq \top$*

Proof. We prove the contrapositive. If $e' = \top$ and $e \overset{\top}{\hookrightarrow} \top$ then by Lemma 17, $e \xrightarrow{*} f$. Hence by type soundness of the reduction semantics, e is not typable. $\square$

7 Related Work

Multi-stage programming techniques have been used in a wide variety of settings, including run-time program generation in ML [17], run-time specialization of C programs [5, 4, 21, 9], and advanced dynamic compilation for C programs [1].

Nielson and Nielson present a seminal detailed study into a two-level functional programming language [19]. This language was developed for studying code generation. Davies and Pfenning show that a generalization of this language to a multi-level language called $\lambda^{\Box}$ gives rise to a type system very related to a modal logic, and that this type system is equivalent to the binding-time analysis of Nielson and Nielson [7]. Intuitively, $\lambda^{\Box}$ provides a natural framework where LISP's quote and eval can be present in a language. The semantics of our Bracket and Run correspond closely to those of quote and eval, respectively.

Glück and Jørgensen 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 [12]. They also demonstrate that binding-time analysis in a multi-level setting can be done with efficiency comparable to that of two-level binding time analysis. Our notion of level is very similar to that used by Glück and Jørgensen[10, 11].

Davies extended the Curry-Howard isomorphism to a relation between modal logic and the type system for a multi-level language [6]. Intuitively, $\lambda^{\bigcirc}$ provide a good framework for formalizing the presence of quote and quasi-quote in a language. The semantics of our Bracket and Escape correspond closely to those of quote and quasi-quote, respectively. Previous attempts to combine the $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ systems have not been successful [7, 6, 25]. To our knowledge, our work is the first successful attempt to define a sound type system combining Brackets, Escape and Run in the same language.

Moggi advocates a categorical approach to two-level languages, and and uses indexed categories to develop models for two languages similar to $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ [18]. He points out that two-level languages generally have not been presented along with an equational calculus. Our paper has eliminated this problem for MetaML, and to our knowledge, is the first presentation of a multi-level language using axiomatic and reductions semantics.

8 Conclusion

In this paper, we have presented an axiomatic and reduction semantics for a language with three staging constructs: Brackets, Escape, and Run. Arriving at the axiomatic and reduction semantics was of great value to enhancing our understanding of the language. In particular, it helped us to formalize an accurate syntactic characterization of faulty terms for this language. This characterization played a crucial role in leading us to the type system presented here. Finally, it is useful to note that our reduction semantics allows for β-reductions inside Brackets, thus giving us a basis for verifying the soundness of the safe-β optimization that we discussed in previous work [25].

MetaML currently exists as a prototype implementation that we intend to distribute freely on the web. The implementation supports the three programming constructs, higher-order datatypes (with support for Monads), Hindley-Milner polymorphism, recursion, and mutable state. The system has been used for developing a number of small applications, including simply term-rewriting system, monadic staged compilers, and numerous small bench-mark functions.

We are currently investigating the incorporation of an explicit recursion operator and Hindley-Milner polymorphism into the type system presented in this paper.

Acknowledgements: We would like to thank John Matthews and Matt Saffell for comments on a draft of this paper.

References

[1] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 149–159, Philadelphia, Pennsylvania, May 1996.

[2] Henk . P. Barendregt. *The Lambda-Calculus, its syntax and semantics.* Studies in Logic and the Foundation of Mathematics. North-Holland, Amsterdam, 1984. Second edition.

[3] Don Batory and Bart J. Geraci. Composition validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, 1997.

[4] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, Florida, 21–24 January 1996.

[5] Charles Consel, Calton Pu, and Jonathan Walpole. Incremental specialization: The key to high performance, modularity, and portability in operating systems. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 44–46, New York, NY, USA, June 1993. ACM Press.

[6] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11*th *Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[7] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.

[8] Dawson R. Engler. VCODE : A retargetable, extensible, very fast dynamic code generation system. In *Proceedingsof the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 160–170, New York, May 1996. ACM Press.

[9] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynaic code generation. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 131–144, St. Petersburg Beach, Florida, January 1996.

[10] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[11] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[12] Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

[13] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. Annotation-directed run-time specialization in C. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 163–178, Amsterdam, The Netherlands, June 1997.

[14] John Launchbury and Simon L. Peyton-Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–342, December 1995. pldi94.

[15] Peter Lee. *Realistic Compiler Generation*. Foundations of Computing Series. MIT Press, 1989.

[16] Peter Lee and Mark Leone. Optimizing ML with run-time code uen&ation. In *Proceedingsof the ACM SIGPLAN Conference on Programming Language Design and Implemantation*, pages 137–148, New York, May21–24 1996. ACM Press.

[17] Mark Leone and Peter Lee. A declarative approach to run-time code generation. In *Workshop on Compiler Support for System Software (WCSSS)*, February 1996.

[18] Eugenio Moggi. A categorical account of two-level languages. In *MFPS 1997*, 1997.

[19] Flemming Nielson and Hanne Rijs Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[20] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1975.

[21] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. Microlanguages for operating system specialization. In *Proceedings of the SIGPLAN Workshop on Domain-Specific Languages*, Paris, January 1997.

[22] Calton Pu and Jonathan Walpole. A study of dynamic optimization techniques: Lessons and directions in kernel design. Technical Report OGI-CSE-93-007, Oregon Graduate Institute of Science and Technology, 1993.

[23] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages, San Diego, Ca.* ACM Press, jan 1998.

[24] Walid Taha, Zine-el-abidine Benaissa, and Tim Sheard. The essence of staged programming. Technical report, OGI, Portland, OR, December 1997.

[25] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997.

[26] Richard Waldinger and Michael Lowry. AMPHION: Towards kinder, gentler formal methods. In *Proceedings of the 1994 Monterey Workshop on Formal Methods*. U.S. Naval Postgraduate School, September 1994.

[27] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.