A Sound Reduction Semantics for Untyped CBN Multi-Stage Computation. Or, the Theory of MetaML is Non-trivial

Extended Abstract

Walid Taha[*]
Department of Computing Science
Chalmers University of Technology and the University of Göteborg
S-412 96 Göteborg, Sweden. Tel: +46-31-772 1003
taha@cs.chalmers.se

ABSTRACT

A multi-stage computation is one involving more than one stage of execution. MetaML is a language for programming multi-stage computations. Previous studies presented big-step semantics, categorical semantics, and sound type systems for MetaML. In this paper, we report on a confluent and sound reduction semantics for untyped call-by name (CBN) MetaML. The reduction semantics can be used to formally justify some optimization performed by a CBN MetaML implementation. The reduction semantics demonstrates that non-trivial equalities hold for object-code, even in the untyped setting. The paper also emphasizes that adding intensional analysis (that is, taking-apart object programs) to MetaML remains an interesting open problem.

1. INTRODUCTION

Recently, there has been significant interest in multi-stage programming languages [5; 6; 7; 8; 9; 10; 11; 14; 21; 26]. A *multi-stage programming language* provides high-level constructs for the construction, combination, and execution of code fragments [31]. MetaML [31; 34] is an SML-like multi-stage programming language that provides three staging constructs called Brackets $\langle \_ \rangle$, Escape $\tilde{}\ \_$, and Run run $\_$. Intuitively, these three constructs are analogous to LISP's back-quote, comma, and eval [2]. There are however two notable differences between LISP's and MetaML's constructs: First, the former work on lists, and the latter work only on α-equivalence classes of representations of programs containing binding constructs. Second, MetaML avoids the need for the use of a *newname* or *gensym* constructs to provide fresh names for bound variables. These differences simplify the formal semantics of MetaML substantially.

Previous studies on the formal semantics of MetaML have focused on the feasibility of static (or dynamic) type systems [23; 30; 33; 34], typically using a big-step semantics for characterising correctness [23; 33], and sometimes using categorical denotational semantics for achieving a better understanding of the notions of type [4]. Types, however, are not prerequisite for MetaML to be useful. In particular, as we will show in this paper, code objects in *untyped* MetaML enjoy strong equational properties that are of interest in their own right.

1.1 Intensional Analysis and MetaML

Part of the strength of the equational properties of MetaML comes from the absence of an explicit mechanism for intensional analysis (or the taking-apart) of code. In other words, even though MetaML can express staged computation in a practically useful way [31; 34], it does not provide the programmer with arbitrary access to the representation of object-code.

Given that intensional analysis is needed for describing any kind of program analysis (such as type checking) or source-to-source transformation (such as Binding-Time Analysis (BTA) [9; 11; 15; 16]) a natural question is then, why does MetaML not have intensional analysis? Before answering this question, it is important to note that many transformations such as BTA can be defined *outside* MetaML, and introduced as a constant. This could be sufficient for many users (and applications): We can certainly construct a representation of a program in MetaML, for example, by typing in:

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

and the MetaML implementation prints [34]:

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

[*]The research reported in this paper was carried out at the Pacific Software Research Center (PacSoft), Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland, OR, USA, and was supported by the USAF Air Materiel Command, contract # F19628-93-C-0069, and NSF Grant IRI-9625462. The paper is based on results presented in the author's PhD dissertation [31]. Author is currently a Post-doctoral Fellow supported by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

```
            : ⟨int → int → int⟩.
```

For simplicity, 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. One can, in principle, add a *primitive constant* BTA to MetaML with the following type:

```
-| 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"[1]. Then specialization is achieved by running the two-level program on an input term:

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

yielding, in turn, a specialized program.

But there are certainly applications where good support for intensional analysis is highly desirable. And even though multi-stage programming does not need to concern itself with how code is represented, MetaML itself was developed as a *meta-programming language*, and the long-term goals of the MetaML project have at various times included support for intensional analysis.

To answer the question we raise above: there seems to be a number of fundamental reasons why introducing intensional analysis require care. We know of two: First, it is hard to find reasonable type systems in the presence of intensional analysis. This problem is beyond the scope of this paper and the reader is referred to [31]. Second, as we will illustrate in this paper, adding intensional analysis to MetaML weakens the notion of equivalence to the extent that it makes most interesting program optimizations unsound.

1.2 Organization and Contributions

Section 2 reviews the previously proposed big-step semantics for MetaML [23; 33], and explains why it cannot be used directly as a basis for establishing the soundness of a reduction semantics. Using a notion of **expression family** we define the **fine big-step semantics** that seems *crucial* for the formal development of the untyped language.

Section 3 analyses the **basic problems** associated with finding an appropriate reduction semantics for MetaML. In particular, we show why working on raw MetaML terms is problematic, and why "level-annotated terms" [33] also lead to complications.

Section 4 presents a simple **reduction semantic**s for a subset of MetaML that we call λ-U[2]. The proposed reduction semantics works on terms that have no explicit level annotations, and uses the standard notion of substitution.

[1]The lift function is a secondary annotation that takes a ground value and returns a code fragment containing that value [34]. In this study we consider lift to be secondary because most of its high-level behaviour can be modelled with other constructs.

[2]The letter U is simply the last in the sequence R, S, T, U. The first attempt at a calculus was called λ-R, where R stands for "Run" [33]. 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 M stands for "MetaML".

Section 5 summarizes our main results. The first result is **confluence** of the proposed reduction semantics. Confluence states that the result of any two sequences of reduction can always be reduced to a common term. From the point of view of language design, confluence is often an indicator of the well-behavedness of the proposed notions of reduction. The second (and main) result is the **soundness** of the proposed reduction semantics with respect to the proposed fine big-step semantics. This result has two parts. First, all what can be achieved by the big-step semantics, can be achieved by the reductions. Second, applying the reductions to any sub-term of a program does not change the termination behavior of the big-step semantics. This result establishes that the reduction semantics and the big-step semantics are "essentially equivalent" formulations of the same language. Proof details appear in a technical report [32].

2. BIG-STEP SEMANTICS (CBN λ-M)

A big-step semantics is a partial function from expressions to values (or "answers"). There are a number of reasons why the big-step semantics for MetaML [23; 33] is an instructive model for the formal study of multi-stage computation:

1. By making "evaluation under lambda" explicit, this semantics makes it easy to illustrates how a multi-stage computation often violates one of the basic assumptions of many works on programming language semantics, namely, that we are dealing only with closed terms.

2. By using just the standard notion of substitution (see for example Barendregt [1]), this semantics captures the *essence* of static scoping, and there is no need for using additional machinery for performing renaming at run-time.

In this section, we will review the big-step semantics proposed for MetaML in previous work, and will discuss why using it directly to justify a reduction semantics in the untyped setting is problematic. We then present a refined notion called the *fine* big-step semantics that we use in our formal development.

2.1 Coarse Big-Step Semantics

In previous work, we have presented a big-step semantics for a core subset of untyped MetaML called λ-M [33; 23]. The raw terms for this language are:

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

The big-step semantics for λ-M is specified by a partial function $\_ \overset{n}{\hookrightarrow} \_ : E \to E$. Figure 1 summarizes the coarse CBN big-step semantics for λ-M. Taking n to be 0, we can see that the first two rules correspond to the rules of a CBN lambda calculus. 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

Syntax:

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

Big-Step Rules:

$$\frac{}{\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e}\ \text{Lam} \qquad \frac{\begin{array}{c} e_1 \overset{0}{\hookrightarrow} \lambda x.e \\ e[x := e_2] \overset{0}{\hookrightarrow} e_3 \end{array}}{e_1\ e_2 \overset{0}{\hookrightarrow} e_3}\ \text{App} \qquad \frac{e_1 \overset{0}{\hookrightarrow} \langle e_2 \rangle \quad e_2 \overset{0}{\hookrightarrow} e_3}{\mathsf{run}\ e_1 \overset{0}{\hookrightarrow} e_3}\ \text{Run} \qquad \frac{}{x \overset{n+}{\hookrightarrow} x}\ \text{Var+}$$

$$\frac{e_1 \overset{n+}{\hookrightarrow} e_3 \quad e_2 \overset{n+}{\hookrightarrow} e_4}{e_1\ e_2 \overset{n+}{\hookrightarrow} e_3\ e_4}\ \text{App+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\lambda x.e_1 \overset{n+}{\hookrightarrow} \lambda x.e_2}\ \text{Lam+} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\langle e_1 \rangle \overset{n}{\hookrightarrow} \langle e_2 \rangle}\ \text{Brk} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} e_2}{\mathsf{run}\ e_1 \overset{n+}{\hookrightarrow} \mathsf{run}\ e_2}\ \text{Run+}$$

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

Figure 1: The Coarse CBN Big-Step Semantics for λ-M

"evaluating at levels higher than 0", is intended to eliminate level 1 Escapes. Rebuilding is performed by traversing the expression while correctly keeping track of level. Thus rebuilding simply traverses a term until a level 1 Escape is encountered, at which point normal (level 0) evaluation function is invoked in the Esc rule. The Escaped expression must yield a Bracketed expression, and then the expression itself is returned.

2.1.1 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 $\langle x \rangle$. While it is common that such a semantics is restricted *a posteriori* to closed terms (for example, in Plotkin [28]), there was nothing in our development that necessitates this restriction.

2.2 The Problem with the Coarse Function

The notions of reduction that we wish to propose are *not* sound under the coarse big-step semantics function described above. In particular, if we do not explicitly forbid the application of the coarse big-step semantic function on terms that are manifestly "not of the right level", finding suitable notions of reduction seems hard. For example, consider the term $\tilde{}\langle \lambda x.x \rangle \in E$. If this term is subjected to the coarse big-step semantic function at level 0, the result is undefined. At the same time, if we pragmatically optimize this term to $\lambda x.x$, the big-step semantics *is* defined. Applying a reduction to a sub-term of a program should *not* change its termination behavior. In particular, an optimization that takes an "undefined" term and makes it "defined" is not sound. In the next section, we will introduce a *finer* notion of big-step semantics that will avoid this kind of problem.

2.3 Fine Big-Step Semantics

To define the fine 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 (that is, should not occur at level 0). Thus, we introduce expression families. An *expression family* is a collection of sets $E^0, E^1, E^2, \ldots$ indexed by the natural numbers. The expression family for terms is defined as follows:

$$\begin{array}{lll} e^0 & \in E^0 & := \ x \mid \lambda x.e^0 \mid e^0\ e^0 \mid \langle e^1 \rangle \mid \mathsf{run}\ e^0 \\ e^{n+1} & \in E^{n+1} & := \ x \mid \lambda x.e^{n+1} \mid e^{n+1}\ e^{n+1} \mid \\ & & \quad \langle e^{n+2} \rangle \mid \tilde{}e^n \mid \mathsf{run}\ e^{n+1}. \end{array}$$

Remark 1 (Notation). *The above presentation of expression families is "essentially BNF" in that it defines a set of terms by simple induction. In more standard notation, the set is defined by induction on the height of a set membership judgment $e \in E^n$ defined by induction over e:*

$$\frac{}{x \in E^n} \qquad \frac{e \in E^n}{\lambda x.e \in E^n} \qquad \frac{e_1, e_2 \in E^n}{e_1\ e_2 \in E^n}$$

$$\frac{e \in E^{n+1}}{\langle e \rangle \in E^n} \qquad \frac{e \in E^n}{\tilde{}e \in E^{n+1}} \qquad \frac{e \in E^n}{\mathsf{run}\ e \in E^n}\ .$$

The BNF-like notation is especially convenient for defining the sets of workable *and* stuck *terms introduced in the technical report [32].*

This stratification of the expression is *crucial* to the correctness of the reduction semantics that we propose in this paper.

Note that we have *not* changed the syntax at all, rather, we have imposed a finer *classification* on terms. For example, the following results states that every element E^n of the expression family is a subset of the set of terms E:

Lemma 2 (Basic Properties of Expression Families). $\forall n \in \mathbb{N}$.

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

Note also that we *do not* annotate terms explicitly with any additional information: The terms of the language are exactly the same as before, and we are simply building a finer classification of the same terms.

While we do not depend on the following property in the formal development, it is instructive to note that it can be easily proved with the help of the previous lemma:

Lemma 3 (Classification).

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

2.3.1 Values

Values are a subset of terms. The defining characteristic of this subset is that a value (an element of the subset) represents the result of a computation (see Lemma 5). To inductively define values, we must once again use expression families, rather than just one set:

$$\begin{array}{llll} v^0 & \in V^0 & := & \lambda x.e^0 \mid \langle v^1 \rangle \\ v^1 & \in V^1 & := & x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1 \\ v^{n+2} & \in V^{n+2} & := & x \mid v^{n+2}\ v^{n+2} \mid \lambda x.v^{n+2} \mid \\ & & & \langle v^{n+3} \rangle \mid \tilde{\ } v^{n+1} \mid \mathsf{run}\ v^{n+2}. \end{array}$$

Intuitively, level 0 values are what we get as a result of evaluating a term at level 0, and level $n+1$ values are what we get from rebuilding a term at level $n+1$. 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 $n+1$ can contain applications such as $\langle(\lambda y.y)\ (\lambda x.x)\rangle$, reflecting the fact that computations at these levels *can be deferred*[3]. 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 $\langle \tilde{\ }(f\ x)\rangle$. Note, however, that we can have a level n Escape in a level n value, for $n \geq 2$.

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

Lemma 4 (Untyped Strong Value Reflection).

$$\forall n \in \mathbb{N}. V^{n+1} = E^n.$$

The lemma has two parts: One saying that every element in a set of (code) values is also an element of the "previous" set of expressions, and the other, saying the converse. 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 Strong Value Reflection, it is easy to verify that if the big-step semantics at level n ($e \overset{n}{\hookrightarrow} v$) returns an expression, this expression is a value $v \in V^n$ at level n :

Lemma 5 (Basic Properties of Big-Step Semantics). $\forall n \in \mathbb{N}$.

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

Noting also that $V^0 \subseteq E^0$ and $V^{n+1} = E^n \subseteq E^{n+1}$, the previous lemma implies *level-preservation*, in the sense that: $\forall n \in \mathbb{N}. \forall e_1 \in E^n. \forall e_2 \in E$.

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

Remark 6 (Fine Big-Step Function). *In previous works on the semantics of MetaML, the term "big-step semantics" referred to the coarse function. For the purposes of this paper, we will only be concerned with the fine big-step semantic function* $\_ \overset{n}{\hookrightarrow} \_ : E^n \to E^n$*, as it is more closely related to the reduction semantics that we are proposing. We will also use the term "big-step semantics" to refer only to the fine-big step semantic function.*

3. WHAT'S NOT A METAML REDUCTION

A reduction semantics can be viewed as a set of directed rewrite rules. Such a semantics can be used to define a notion of equality between terms: Two terms are equal if they can be reduced to a common term. In our experience, it has also been the case that studying such a semantics has helped us in developing the first type system for MetaML [33]. It is therefore reasonable to expect that having a simple reduction semantics for a given language can be helpful in developing new type systems for MetaML. For example, an important property of a type system is that typability should remain invariant under reductions ("Subject Reduction"). A simple reduction semantics helps language designers quickly eliminate inappropriate type systems, hopefully leading to a better understanding of the design space for such type systems.

In this section, we will analyse the problem of defining a suitable reduction semantics of MetaML. We begin by reviewing the basic notions of coherence and confluence, and then study why defining a reduction semantics on raw MetaML terms is not suitable. We then analyse why using the notion of "level-annotated terms" introduces additional complications.

3.1 Coherence and Confluence

Two important concepts central to this section are coherence and confluence (For confluence, see Barendregt [1]). A reduction semantics 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

[3]A reviewer asked "can" or "must"? The big-step semantic says "must". However, the proposed reduction semantics brings the interesting news, and says "can". The reduction semantics has no mechanism for enforcing that a term is *not* reduced (or evaluated). Thus, the main difference between application at level 0 and application at higher levels is that the former *must* be reduced, while the latter *do not need to* be reduced. This observation comes directly from the definition of values. The fact that the proposed reductions allow us to interpret higher-level applications as "can be deferred" as opposed to "must be deferred" is part of the power of this semantics. In particular, higher level applications *does not need to be deferred* if the effect of performing them cannot be observed by the outside world. This is useful, for example, in allowing an implementation of MetaML to optimize object code, and improve the quality of the code generated by a multi-stage system.

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. Conversely, we will say that a semantics is incoherent if it there is one expression that it can reduce to two different ground values. Clearly, 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 reduction semantics is a way of ensuring coherence. Conversely, if we lose coherence, we lose confluence.

3.2 A First Attempt

A direct attempt at extending the set of expressions and values of basic CBN lambda calculus to incorporate the staging constructs of MetaML yields the following two rules in addition to the β rule:

$$\begin{array}{rcl} (\lambda x.e_1)\,e_2 & \longrightarrow_\beta & e_1[x := e_2] \\ \tilde{}\langle e\rangle & \longrightarrow_E & e \\ \mathsf{run}\,\langle e\rangle & \longrightarrow_R & e. \end{array}$$

There are several reasons why this naive approach is unsatisfactory. In the rest of the section, we will explain the problems with this approach, and explore the space of possible improvements to this semantics.

3.3 Intensional Analysis Conflicts with β

Directed, deterministic 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. Unfortunately, there is a conflict between supporting intensional analysis and allowing the β rule on object-code. To illustrate this undesirable interaction, let us assume a minimal extension to λ-M with a hypothetical construct IsApp that tests a piece of code to see if it is an application. Big-step evaluation would then justify the implementation behaving as follows:

```
-| IsApp ⟨(fn x ⇒ x) (fn y ⇒ y)⟩;
val it = true : bool.
```

Allowing β reduction on object-code 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 is desirable, because it eliminates a function call in the object-program and reduces its size to half. But such an "optimization" would have a devastating effect on the semantics of this hypothetical extension of MetaML. In particular, reduction followed by big-step evaluation would then also justify the implementation behaving 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 reduction semantics in the presence of β reduction at higher levels, *and* deterministic intensional analysis.

3.4 Level-Annotated Terms

The difficulty with adding intensional analysis suggests that distinguishing between level 0 ("meta-") terms and higher level ("object-") terms might be useful.

This idea has already been used in a previous attempt at a reduction semantics for MetaML [33]. In order to control the applicability of the β rule 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 sub-term) in the terms themselves, so as to give us finer control over where different reductions are applicable.

Unfortunately, that reductions semantics contains a subtle flaw[4]. It suffers two main shortcomings:

- Working with level annotations requires introducing auxiliary notions of "promotion", "demotion", and the use of a non-standard notion of substitution, all in order to correctly maintain the level-annotations during the execution of a program.
- In certain instances, the left hand side of a reduction is defined, but the right hand side is not. This subtle flaw was partly a result of the fact that the non-standard notion of substitution (needed to maintain correct level annotations) was *not* a total function.

Level-annotated terms also induce an expression family $E^0, E^1, E^2, \ldots$ where each annotated term lives:

$$\begin{array}{lll} e^0 \in E^0 & := & x^{\mathbf{0}} \mid \lambda x.e^0 \mid e^0e^0 \mid \langle e^1\rangle \mid \mathsf{run}\ e^0 \\ e^{n+1} \in E^{n+1} & := & x^{\mathbf{n+1}} \mid \lambda x.e^{n+1} \mid e^{n+1}e^{n+1} \mid \\ & & \langle e^{n+2}\rangle \mid \tilde{}e^n \mid \mathsf{run}\ e^{n+1} \\ v^0 \in V^0 & := & \lambda x.e^0 \mid \langle v^1\rangle \\ v^1 \in V^1 & := & x^{\mathbf{1}} \mid \lambda x.v^1 \mid v^1v^1 \mid \langle v^2\rangle \mid \mathsf{run}\ v^1 \\ v^{n+2} \in V^{n+2} & := & x^{\mathbf{n+2}} \mid \lambda x.v^{n+2} \mid v^{n+2}v^{n+2} \mid \\ & & \langle v^{n+3}\rangle \mid \tilde{}v^{n+1} \mid \mathsf{run}\ v^{n+2}. \end{array}$$

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.

The key difference between level-annotated terms and raw terms is therefore only in the "leaves", namely, the variables[5]. In level-annotated terms, variables *explicitly* carry around a (term representing a) natural number that corresponding to their level. In other words, where as E^2 is the name of a particular subset of some universal set E, the term $x^{\mathbf{2}}$ is a *pair* consisting of the name of variable x and a representation of a natural number 2. Note also that $x^{\mathbf{2}}$

[4]Surprisingly, the type system is still sound. See [23] for the soundness result.

[5]The original definition of level-annotated terms [33] had every construct carrying level annotations. There is a one-to-one corresponds between that definition and the simpler definition we use here.

does not mean "a name drawn from a set Var^2 dedicated for names of variables at level 0": throughout the formal development, there is exactly one set of variable names, and this is the set used for all variables. For all other constructs, we can simply infer the (unique) level of the whole term by looking at the sub-term. For Brackets and Escapes, the obvious "correction" to levels is performed.

3.4.1 Escapes Conflict with β

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:

$$\langle (\mathsf{fn}\ \mathsf{x} \Rightarrow \tilde{}\mathsf{x}^0)\ \tilde{}\langle 4^1 \rangle \rangle.$$

The level of the whole term is 0. If we allow the β rule at higher levels, this term can be reduced to:

$$\langle \tilde{}\,\tilde{}\langle 4^1 \rangle \rangle.$$

This result contains two nested Escapes. Thus, the level of the whole term can no longer be 0. The outer Escape corresponds to the outer 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.

3.4.2 Substitution Conflicts with β

One possibility for avoiding the problem above is to limit β to level 0 terms:

$$(\lambda x.e_1^0)\, e_2^0 \quad \longrightarrow_\beta \quad e_1^0[x :! = e_2^0].$$

At first, this approach is appealing because it makes the extension of MetaML with code inspection operations less problematic. But note that a *non-standard notion of substitution* must be used in $e_1^0[x :! = e_2^0]$. To illustrate, consider the following term:

$$(\mathsf{fn}\ \mathsf{x} \Rightarrow \langle \mathsf{x}^1 \rangle)\ (\mathsf{fn}\ \mathsf{y} \Rightarrow (\mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{x}^0)\ 5^0).$$

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:

$$\langle \mathsf{fn}\ \mathsf{y} \Rightarrow (\mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{x}^1)\ 5^1 \rangle.$$

The level annotations need to be adjusted after substitution (See [33].) 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

$$(\mathsf{fn}\ \mathsf{x} \Rightarrow \langle \mathsf{x}^1 \rangle)\ (\mathsf{fn}\ \mathsf{y} \Rightarrow 5^0).$$

and then we can still go back and perform the outermost application to get:

$$\langle \mathsf{fn}\ \mathsf{y} \Rightarrow 5^1 \rangle.$$

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 proposed notions of reduction, despite the fact that we have sacrificed β reductions at higher-levels. Intuitively, the example above illustrates that *cross-stage persistence* [34], that is, the possibility of binding a variable level and using it at a higher level, 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 $\mathsf{fn}\ \mathsf{x} \Rightarrow \langle \mathsf{x} \rangle$ to a function causes all the redices in the body of that function to be frozen.

3.4.3 Summary of Problems with Level Annotations

To conclude this section, we emphasize that the seemingly minor difference between level-annotated terms and expression families, namely, the level annotations carried on the leafs of terms, has a profound effect on the formal development. In particular, the substitution operation becomes much more complicated because of the need to change the level annotations inside a term, depending on how the substitution operation affects its level. Also, we lose the simpler reflection property, which means that we have to introduce a new partial function called "demotion" in order to correct the level of the sub-term after a run reduction is performed.

4. REDUCTION SEMANTICS (CBN λ-U)

We are now ready to present the proposed reduction semantics for MetaML, called λ-U. The syntax of λ-U consists of *exactly* the same expression family introduce for λ-M. Taking advantage of Strong Reflection, we can simplify the definition of values and present the following equivalent but more compact definition:

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

The definition of values explicates two essential subtleties: First, that *a code value at level 0* carries *an expression of level 0.* Second, higher level values are simply the expressions of the previous level.

Definition 7 (CBN λ-U Reductions). *The CBN notions of reduction of λ-U are simply:*

$$\begin{array}{rcl} (\lambda x.e_1^0)\, e_2^0 & \longrightarrow_{\beta_U} & e_1^0[x := e_2^0] \\ \tilde{}\langle e^0 \rangle & \longrightarrow_{E_U} & e^0 \\ \mathsf{run}\ \langle e^0 \rangle & \longrightarrow_{R_U} & e^0. \end{array}$$

Just like the rules for a CBN or CBV lambda calculus, these rules are intended to be applied in *any* context, independently of what the context is, or the *level* imposed by the context. The reduction relation is therefore defined is follows:

Definition 8 (Reduction Relation). *The CBN λ-U reduction relation $\longrightarrow \subseteq E \times E$ as the compatible extension of rewriting a term using any of the three λ-U notions of reduction. We write $\longrightarrow^*$ for its reflexive, transitive closure.*

Note in particular that the reduction relation allows us to apply the β rule to any expression that *looks like* a level 0 application. So, in contrast to the big-step semantics, the reductions are *independent* of the level of the term. By simply restricting the body of the lambda term and its argument to be in E^0, λ-U reductions avoid the problems that riddled the level-annotated term approach, namely, the non-standard notion of substitution, and the conflict between Escapes and β. The essential idea seems to be defining the reductions only using expressions in E^0, which are free of top-level Escapes[6]

Note also that restricting the body of the lambda term in the β_U rule to be a member of E^0 makes evaluation under lambda explicit. This restriction is no different from the value restriction on the argument of an application in Plotkin's β_v, which forces the evaluation of the argument before it can be passed to (substituted in the body of) the function. Thus, we can read the expression-0 restrictions on the body of the function and on the argument in the β_U rule as explicitly forcing the body and the argument to be"proper expressions" before they can be reduced as an application.

5. SUMMARY OF TECHNICAL RESULTS

This section gives an overview of the two main technical results on CBN λ-U, namely, confluence and soundness with respect to the fine big-step semantics of CBN λ-M. The details of the two results are presented in the technical report [32].

5.1 Confluence

Establishing the confluence of a reduction semantics 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]. A recently paper by Takahashi promotes 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 [36]. Takahashi promotes the use of an explicit notion of a *parallel reduction* and a notion of *star reduction* that avoids the need for "residuals". The idea of parallel reduction goes back to the original and classic (yet unpublished) works of Tait and Martin-Löf and is introduced early on in Barendregt's book [1, Section 3.2]. Parallel reduction is a relation between terms that allows multiple reductions to be performed at the same time. The essential feature of this notion is that it is indifferent to the number of times a redix occurs in a term. Star reduction is a function defined directly by induction on the structure of terms, and greatly simplifies establishing the confluence of parallel reduction. The corresponding notion to star reduction in Barendregt is *complete development* [1, Section 11.2]. This notion is not used explicitly in the first (an probably simplest) confluence proof, rather, is postponed to Section 11.2 and defined using the more involved notion of residuals.

Our proof of confluence follows closely the development in the introduction to Takahashi's paper, and the CBN reductions of λ-U do not introduce any notable complications to the proof of confluence. Our proof is as simple, concise, and rigorous as the one presented by Takahashi.

Theorem 9 (CBN λ-U is Confluent). $\forall e_1, e, e_2 \in E.$

$$e_1 \longleftarrow^* e \longrightarrow^* e_2 \Longrightarrow (\exists e' \in E. e_1 \longrightarrow^* e' \longleftarrow^* e_2).$$

5.2 Soundness

Establishing the soundness of a reduction semantics with respect to a big-step semantics involves two parts: First, one must show that what can be achieved by big-step evaluation can also be achieved by reduction. This part is typically easy to prove, and proceeds by induction over the height of the big-step evaluation derivation. Second, one must show that what can be achieved by reduction can, "in some sense", also be achieved by big-step evaluation. The basic problem here is that reductions can be applied anywhere in a term (non-deterministically), and big-step evaluation performs only certain reductions (deterministically). In other words, the reduction relation typically reduces *more* redices than big-step evaluation. For example, the reduction semantics for the lambda calculus can do "reductions under lambda", and the big-step semantics generally does not [19; 38]. At the same time, there is typically a formal sense, called *observational equivalence* [19; 38], in which these extra reductions are irrelevant. This state of matters generally makes establishing the second half of soundness more involved.

Our proofs have followed closely Plotkin's proofs for the soundness of the CBV and CBN lambda-calculi [28]. While carrying out these proofs has been largely uneventful, we have accumulated a number of hints that may benefit others interested in pursuing similar formal development. These can be found in the technical report [32].

In what follows, we present our definition of observational equivalence (the termination behavior of the level 0 λ-M big-step evaluation) and state the soundness result, namely, that CBN λ-U reductions preserve observational equivalence.

Definition 10 (Level 0 Termination). $\forall e \in E^0.$

$$e \Downarrow \triangleq (\exists v \in V^0. e \overset{0}{\hookrightarrow} v).$$

To define the notion of observational equivalence, we need to introduce the notion the formal notion of a context:

Definition 11 (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.

Remark 12. *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 13 (Observational Equivalence). *We define $\approx_n \in E^n \times E^n$ as follows: $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$.*

$$e_1 \approx_n e_2 \triangleq \; \forall C \in \mathbb{C}. \; C[e_1], C[e_2] \in E^0 \Longrightarrow (C[e_1]\Downarrow \iff C[e_2]\Downarrow).$$

[6]A reviewer suggested a "free variable analogy". Top-level Escapes can be viewed as "free" (or "unbound") Escapes. The key use of E^0 can then be viewed as simply the exclusion of expressions with "free" Escapes.

REMARK 14. *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.*

THEOREM 15 (CBN λ-U IS SOUND UNDER λ-M). $\forall n \in \mathbb{N}.\forall e_1, e_2 \in E^n$.

$$e_1 \longrightarrow e_2 \Longrightarrow e_1 \approx_n e_2.$$

6. RELATED WORKS

The author's dissertation promotes the formal study of multi-stage programming, and reports on the results of scrutinizing the design and implementation of the multi-stage programming language MetaML. The focus is on MetaML's operational semantics, two type systems, and formalizing 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. The results presented in this paper are based directly on Chapters 5 and 6.

Davies and Pfenning propose two typed frameworks for staged computation, $\lambda^{\Box}$ [6] and $\lambda^{\bigcirc}$ [5]. The latter is more directly related to MetaML because it allows evaluation under lambda (and therefore computation with open code). Davies uses a notion of expression family in the typed development of $\lambda^{\bigcirc}$ to classify values, but it is not used to impose restrictions in the notions of reduction. Thus, the reductions suggested for $\lambda^{\bigcirc}$ are *not* sound for the untyped MiniML$^{\bigcirc}$ big-step semantics (which has fixed points), even when the β-rule is restricted to β_v and when the terms are well-typed. This observation does not conflict with any of the formal claims or results that Davies reports. In the typed setting, these reductions can only be sound if effects are introduced using Moggi's notion of a computational monad [20] (see for example Hatcliff and Danvy [13] and Lawall and Thiemann [17] for studies in this direction). It is not clear from the categorical analysis of multi-stage computation [3] whether or not staging and monads can coexist fruitfully, and this question remains an open and interesting one.

There has also been a number of related studies in the contexts of LISP and the untyped lambda calculus. Muller [24] studies the reduction semantics of LISP's quote and eval. Muller observes 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 [25] also studies the reduction semantics of the lambda-calculus extended with representations of lambda terms, and with a notion of β reduction on these representations. Muller observes that this calculus lacks confluence, and uses a type system to restore confluence. In both Muller's studies, the language can express intensional analysis. Wand [37] studies the equational theory for LISP meta-programming construct fexpr and finds that *"the theory of* fexpr*s is trivial"* in that the only meaning-preserving equality is *alpha* conversion. In the context of a study on the expressive power of programming languages using a notion of "abstraction contexts", Mitchell also remarks that *"[in LISP with* fexpr*s,] syntactically different LISP expressions are not observationally equivalent"* [18]. These observations imply that β is not valid in a language with fexprs. Wand predicts that there are other meta-programming languages with a more interesting "non-trivial" equational theory. As λ-U demonstrates, MetaML is one such language.

Moggi [21] points out that two-level languages generally have not been presented along with an equational calculus. Lawall and Thiemann give such a presentation in the context of a study on sound specialization in the presence of effects [17]. Our proposed reduction semantics has eliminated this problem for CBN MetaML, and to our knowledge, is the first correct presentation of *a typed or untyped multi-stage* programming language using a reduction semantics.

7. FUTURE WORK

Corresponding results on the confluence and soundness of CBV λ-U have not been established yet. There were two reasons for pursuing these results first in the CBN setting:

1. To avoid developing accidental dependency on a particular strategy (CBV), and
2. To avoid a technical inconvenience relating to the mismatch between the notion of value in the reduction semantics and in the big-step semantics for a CBV language.

The difference between the CBV and the CBN big-step semantics for MetaML (λ-M) is only in the evaluation rule for application at level 0. The difference between CBV and the CBN reduction semantics for MetaML (λ-U) is also only in the rule for application where the argument is restricted to be a CBV value, thus forcing it be evaluated before it is passed to the function. 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 usually *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 development. Furthermore, it seems necessary to add a special reduction rule for reducing an application where the argument is a variable in order to be (strictly speaking) a conservative extension of the CBV lambda-calculus.

We expect the results to hold when naturals are added.

8. CONCLUSIONS

In this paper, we presented a ("fine") big-step semantics (λ-M) for a subset of CBN MetaML and explained why the naive approach to a reduction semantics for MetaML does not work. We presented a reduction semantics (λ-U) that we have shown to be confluent and sound with respect to a notion of observational equivalence based on the level 0 termination behavior of the big-step semantics on open terms.

In reviewing the failure of the naive approach to the reduction semantics, we 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. An alternative interpretation of the concrete observations is that, in MetaML, it is more natural to allow semantic equality at all levels. In particular, 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. We see this "leakage" of β as desirable because it allows 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 an unsound equational theory. Finding a better way for introducing intensional analysis to MetaML is an important open problem.

Another important research direction is the formal study of the notion of observational equivalence itself (see for example [12]). An understanding of this notion would pave the way for the investigation of *formal improvement theories* along the lines of those developed by Sands *et al.* (see for example [29].) Such a development is especially appropriate, since MetaML is intended as a framework for staging, which is itself a form of cost improvement.

To conclude, we view the untyped development proposed in this paper as capturing the *simplest* notion of well-formedness needed for having a well-behaved *syntax*. Well-behaved syntax is necessary for a clean and useful separation between the untyped and typed language. Our initial interest was in fact the study of expressive type systems for MetaML. The difficulty of defining the semantics in a type-independent way hindered this program substantially. Thus, we hope that λ-U will be useful for us and for others in the study of richer type systems for multi-stage programming languages.

9. ACKNOWLEDGEMENTS

Without the guidance, support, and enthusiasm of my advisor, Prof. Tim Sheard, this work would have not happened. I would also like to thank James Hook, Neil Jones, John Launchbury, David Maier, and Eugenio Moggi for being a most active and helpful thesis committee. I am especially grateful to Eugenio Moggi for his careful reading of early version of these results, and for being critical and also doubtful of the possibility of relating the reduction and big-step semantics. I also enjoyed discussions with, and received helpful comments from Zino Benaissa, Rowan Davies, Jürgen Gustavsson, Sava Kristic, Amr Sabry, Franklyn Turbak, and Hongwei Xi. Finally, I would like to thank the *seven* anonymous reviewers for their excellent feedback.

10. REFERENCES

[1] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, revised ed., vol. 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, Amsterdam, 1984.

[2] Bawden, A. Quasiquotation in LISP. 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. 88–99. Invited talk.

[3] 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 [27].

[4] 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). To appear.

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

[6] 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.

[7] Glück, R., Hatcliff, J., and Jørgensen, J. Generalization in hierarchies of online program specialization systems. In *Logic-Based Program Synthesis and Transformation* (1999), P. Flener, Ed., vol. 1559 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 179–198.

[8] 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.

[9] 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.

[10] Glück, R., and Jørgensen, J. An automatic program generator for multi-level specialization. *LISP and Symbolic Computation 10*, 2 (1997), 113–158.

[11] Gomard, C. K., and Jones, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming 1*, 1 (Jan. 1991), 21–69.

[12] Gordon, A., and Pitts, A. *Higher Order Operational Techniques in Semantics.* Cambridge Univeristy Press, 1998.

[13] Hatcliff, J., and Danvy, O. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science 7*, 5 (Oct. 1997), 507–541.

[14] Hatcliff, J., and Glück, R. Reasoning about hierarchies of online specialization systems. In *Partial Evaluation* (1996), O. Danvy, R. Glück, and P. Thiemann, Eds., vol. 1110 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 161–182.

[15] Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[16] Jones, N. D., Sestoft, P., and Sondergraard, H. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, J.-P. Jouannaud, Ed., vol. 202 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985, pp. 124–140.

[17] Lawall, J., and Thiemann, P. Sound specialization in the presence of computational effects. In *Theoretical Aspects of Computer Software, Sendai, Japan*, vol. 1281 of *Lecture Notes in Computer Science*. Berlin: Springer-Verlag, 1997, pp. 165–190.

[18] Mitchell, J. C. On abstraction and the expressive power of programming languages. In *Theoretical Aspects of Computer Software* (Sept. 1991), T. Ito and A. R. Meyer, Eds., vol. 526 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 290–310.

[19] Mitchell, J. C. *Foundations for Programming Languages*. MIT Press, Cambridge, 1996.

[20] Moggi, E. Notions of computation and monads. *Information and Computation 93*, 1 (1991).

[21] Moggi, E. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics* (1997), Elsevier Science.

[22] 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 [27].

[23] 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 Scieince*, Springer-Verlag, pp. 193–207. An extended version appears in [22].

[24] 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.

[25] 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.

[26] Nielson, F., and Nielson, H. R. *Two-Level Functional Languages*. No. 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.

[27] 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.

[28] Plotkin, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[29] Sands, D. A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation 5*, 4 (1995).

[30] 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.

[31] Taha, W. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999. Revised October 99. Available from author (taha@cs.chalmers.se).

[32] Taha, W. A sound reduction semantics for untyped cbn multi-stage computation. Or, the theory of MetaML is non-trivial (preliminary report). Tech. Rep. CSE-99-014, OGI, Oct. 1999. Available from [27].

[33] 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.

[34] 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 [35].

[35] Taha, W., and Sheard, T. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science 248*, 1-2 (In Press).

[36] Takahashi, M. Parallel reductions in λ-calculus. *Information and Computation 118*, 1 (Apr. 1995), 120–127.

[37] Wand, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation 10* (1998), 189–199.

[38] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing series. MIT Press, Feb. 1993.