

# A Sound Reduction Semantics for untyped CBN Multi-Stage Computation. Or, the Theory of MetaML is Non-trivial (Preliminary Report)

Walid Taha\*

Pacific Software Research Center (PacSoft)  
Department of Computer Science and Engineering  
Oregon Graduate Institute of Science & Technology  
Portland, OR, USA  
taha@cs.chalmers.se

**Abstract.** MetaML is a programming language that provides a good formalism for expressing multi-stage computation. Defining a reduction semantics for MetaML has been an illusive challenge. This paper presents a sound and strikingly simple reduction semantics for CBN MetaML that avoids the complexities and pitfalls of the previously proposed notion of “level-annotated terms”. The novelty of this reduction semantics is that it admits computation on object (or “future stage”) programs and is therefore “non-trivial” in the sense of Wand. The reduction semantics also provides a formal basis for equational reasoning about untyped CBN MetaML programs, and formal justification for some optimisations considered desirable in MetaML implementations.

We begin by presenting a *fine* big-step semantics for CBN MetaML that refines previous presentations of this semantics. This refinement is crucial for the soundness of our reductions. We define a simple notion of observational equivalence for open, untyped terms based on the fine big-step semantics, where we only observe termination in level 0 contexts. Using standard techniques, we show that this reduction semantics is confluent and that it preserves observational equivalence.

We point out that, even at the level of untyped terms, adding deterministic intensional analysis to MetaML can lead to incoherence.

## 1 Introduction

Binding-time analysis (BTA) [11, 10] can be viewed as a program transformation which, given a program and information about when the inputs to this program will become available, generates a multi-stage program. A *multi-stage program* is one involving more than one distinct stage of executing. The word “multi-stage” itself seems to have been first introduced by Jones *et al.* [10]. Multi-stage programming languages provide programmers with a formalism to express staging explicitly. MetaML [24, 22] is a multi-stage, SML-like programming language that provides three high-level staging constructs called Brackets  $\langle \_ \rangle$ , Escape  $\sim \_$ , and Run  $\text{run } \_$ . Intuitively, these three constructs are analogous to LISP’s back-quote, comma, and eval, that allow constructing, combining, and executing object-programs<sup>1</sup>. For example, we can construct a representation of a program in MetaML. Let us consider a simple MetaML session. We type in:

---

\* The research reported in this paper 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 [22]. Author is currently a Post-doctoral Fellow at the Department of Computing Science, Chalmers University of Technology and the University of Göteborg, S-412 96 Göteborg, Sweden. Tel: +46-31-772 1003. The manuscript was prepared while the author was supported by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

<sup>1</sup> Two significant differences between LISP’s and MetaML’s constructs are worth pointing out: First, the former work on lists, and the latter work only on parse trees for programs. Second, MetaML avoids the need for the use of a *newname* or *gensym* constructs to provide fresh names for bound variables.

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

and the (CBV) MetaML implementation prints:

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

If the program  $p$  is fed to a partial evaluator, it must first go through BTA. At an implementation level, BTA can be viewed as a source-to-source transformation. Typically, BTA is given a specification of which inputs are static and which inputs are dynamic. For simplicity, let us assume that we are only interested in programs that take two curried arguments, and the first one is static, and the second one is dynamic. Although our MetaML implementation does not provide such a function today, one can, in principle, add a *constant* BTA to MetaML with the following type<sup>2</sup>:

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

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

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

yielding the “annotated program”. The lift function is a secondary annotation that takes a ground value and returns a code fragment containing that value [24].

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

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

yielding, in turn, the specialized program.

In the illustration above, we have taken the view that the output of BTA is simply a two-stage annotated program. This view was first suggested in the works of Nielson and Nielson [19] and by Gomard and Jones [6], when two-level languages were introduced. Glück *et al.* [4, 5, 3, 9] generalized two-stage off-line partial evaluation to multi-level off-line partial evaluation and introduced multi-level languages.

## 1.1 Contributions

In previous work, we have presented a big-step semantics for untyped MetaML [23, 16]. A big-step semantics is a partial function from expressions to values (or “answers”). Another important kind of semantics is the reduction semantics. A reduction semantics is a set of directed rewrite rules. Intuitively, the rewrite rules capture the “notions of reduction” in MetaML. 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 [23]. It is reasonable to expect that having a simple reduction semantics for a given language can be helpful in developing type systems for that language. 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.

A previous attempt was also made to develop a reduction semantics for untyped MetaML [23]. This attempt had two shortcomings:

---

<sup>2</sup> BTA cannot be expressed using Brackets, Escape, and Run. In particular, an analysis such as BTA requires intensional analysis. While we will argue in this paper that adding intensional analysis to MetaML is non-trivial (see Section 3), this does *not* mean that adding a BTA constant would be problematic.

- It used a notion of “level-annotated terms” where each term carried a natural number to indicate its level. 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. Thus subtle flaw was partly a result of the fact that non-standard notion of substitution needed to maintain level annotations was not a total function.

In this paper, we show that these problems can be completely avoided, and that untyped CBN MetaML has a sound reduction semantics. We present

- **A simple reduction semantics for CBN MetaML:** The semantics works on terms that have no explicit level annotations, and uses the standard notion of substitution. Instead, we show that it is sufficient to structure the sets of expressions and values as *expressions families*.
- **Confluence for the reduction semantics:** This result is an indicator of the well-behavedness of our notions of reduction. It states that the result of any two (possibly different) sequences of reduction can always be reduced to a common term.
- **Soundness of the reduction 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 reductions semantics and the big-step semantics are “essentially equivalent” formulations of the same language.

## 1.2 Organization

Section 2 presents (both the coarse and) the fine big-step semantics for CBN  $\lambda$ -M. Section 3 explains the problem of finding an appropriate reduction semantics for MetaML. Section 4 presents a reduction semantics for a subset of MetaML that we call  $\lambda$ -U<sup>3</sup>. Section 5 gives a summary of our technical results, namely, confluence and soundness.

Appendix A presents the details of the proofs reported in the paper, and Appendix B gives an informal introduction to the notions of big-step and reduction semantics in the context of a CBN lambda calculus.

## 2 Big-Step Semantics for CBN $\lambda$ -M

Because “evaluation under lambda” is explicit in the big-step semantics of MetaML, it is a good, instructive model of multi-stage computation. This semantics also illustrates how MetaML violates one of the basic assumptions of many works on programming language semantics, namely, that we are dealing only with closed terms. Furthermore, by simply using 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.

The raw terms of  $\lambda$ -M are:

$$e \in E := x \mid \lambda x.e \mid ee \mid \langle e \rangle \mid \sim e \mid \text{run } e.$$

The CBN big-step semantics for  $\lambda$ -M is specified by a partial function  $\_ \overset{n}{\rightarrow} \_ : E^n \rightarrow E^n$ . We proceed by reviewing the *coarse* function  $\_ \overset{n}{\rightarrow} \_ : E \rightarrow E$ , and then show that we can restrict the type of this function to arrive at the *fine* function  $\_ \overset{n}{\rightarrow} \_ : E^n \rightarrow E^n$ .

Figure 1 summarizes the coarse CBN big-step semantics for  $\lambda$ -M [16]. 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

<sup>3</sup> The letter U is simply the last in the sequence R, S, T, U. Our first attempt at a calculus was called  $\lambda$ -R, where R stood for “Run” [23]. We call this reduction semantics  $\lambda$ -U to avoid asserting *a priori* that it is equivalent to the big-step semantics ( $\lambda$ -M). The letter M stood for “MetaML”.

Syntax:

$$e \in E := x \mid e e \mid \lambda x.e \mid \langle e \rangle \mid \sim e \mid \text{run } e$$

Big-Step Rules:

$$\begin{array}{c}
\frac{e_1 \xrightarrow{0} \langle e_2 \rangle \quad e_2 \xrightarrow{0} e_3}{\text{run } e_1 \xrightarrow{0} e_3} \text{ Run} \quad \frac{}{\lambda x.e \xrightarrow{0} \lambda x.e} \text{ Lam} \quad \frac{e_1 \xrightarrow{0} \lambda x.e \quad e[x := e_2] \xrightarrow{0} e_3}{e_1 e_2 \xrightarrow{0} e_3} \text{ App} \\
\frac{e_1 \xrightarrow{n+} e_2}{\lambda x.e_1 \xrightarrow{n+} \lambda x.e_2} \text{ Lam+} \quad \frac{e_1 \xrightarrow{n+} e_2}{\langle e_1 \rangle \xrightarrow{n} \langle e_2 \rangle} \text{ Brk} \quad \frac{e_1 \xrightarrow{n+} e_3 \quad e_2 \xrightarrow{n+} e_4}{e_1 e_2 \xrightarrow{n+} e_3 e_4} \text{ App+} \\
\frac{e_1 \xrightarrow{n+} e_2}{\sim e_1 \xrightarrow{n++} \sim e_2} \text{ Esc++} \quad \frac{e_1 \xrightarrow{0} \langle e_2 \rangle}{\sim e_1 \xrightarrow{1} e_2} \text{ Esc} \quad \frac{e_1 \xrightarrow{n+} e_2}{\text{run } e_1 \xrightarrow{n+} \text{run } e_2} \text{ Run+}
\end{array}$$

**Fig. 1.** The Coarse CBN Big-Step Semantics for  $\lambda$ -M

level 0 says that an expression is Run by first evaluating it to get a Bracketed expression, and then evaluating the Bracketed expression. The rule for Brackets at level 0 says that they are evaluated by rebuilding the expression they surround at level 1: *Rebuilding*, or “evaluating at levels higher than 0”, is intended to eliminate level 1 Escapes. Rebuilding is performed by traversing expressions while correctly keeping track of level. Thus rebuilding simply traverses a term until a level 1 Escape is encountered, at which point 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. Next we establish some basic but important properties of the coarse big-step semantics.

## 2.1 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  $\sim e$  does not interest us because Escapes should not occur at top level. Thus, we introduce *expression families*<sup>4</sup>:

$$\begin{aligned}
E^0 &\in E^0 := x \mid \lambda x.e^0 \mid e^0 e^0 \mid \langle e^1 \rangle \mid \text{run } e^0 \\
E^{n+} &\in E^{n+} := x \mid \lambda x.e^{n+} \mid e^{n+} e^{n+} \mid \langle e^{n++} \rangle \mid \sim e^n \mid \text{run } e^{n+}.
\end{aligned}$$

**Lemma 1 (Basic Properties of Expressions Families).**  $\forall n \in \mathbb{N}$ .

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

Values are a subset of terms that denote the results of computations. Again, because of the relative nature of Brackets and Escapes, we must use expression families for values, indexed by the level of the term, rather than just one set. Thus, values are defined as follows:

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

<sup>4</sup> This presentation of the sets of expressions and values is “essentially BNF” in that it defines a set of terms by simple induction. Technically, this set is defined by induction on the height of a set membership judgment  $e \in E^n$  defined by induction over  $e$ . This notation is especially convenient for defining the sets of *workable* and *stuck* terms presented in the Appendix.

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

The following lemma establishes a simple yet important property of  $\lambda$ -M:

**Lemma 2 (Strong Value Reflection for Untyped Terms).**  $\forall n \in \mathbb{N}$ .

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

The lemma has two parts: One saying that every element of in a set of (code) values is also an element of a set of expressions, and the other, saying the converse. 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  $n$  ( $e \xrightarrow{n} v$ ) returns an expression, this expression is a value  $v \in V^n$  at level  $n$  :

**Lemma 3 (Basic Properties of Big-Step Semantics).**  $\forall n \in \mathbb{N}$ .

1.  $V^n \subseteq V^{n+}$ ,
2.  $\forall e, e' \in E^n. e \xrightarrow{n} e' \implies e' \in V^n$ .

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

$$e_1 \xrightarrow{n} e_2 \implies e_2 \in E^n.$$

*Remark 1 (Fine Big-Step Function).* In the rest of the paper, we will only be concerned with the fine big-step semantic function  $\_ \xrightarrow{n} \_ : E^n \rightarrow E^n$  for  $\lambda$ -M. We will also refer to it simply as the big-step semantics.

**The Closedness Assumption Violated** The semantics is standard in its structure, but note it has the unusual feature that it manipulates *open* terms. In particular, rebuilding goes “under lambda” in the rule Lam+, and Escape at level 1 re-invokes evaluation during rebuilding. Thus, even though a closed term such as  $\langle\lambda x.\sim\langle x\rangle\rangle$  evaluates to  $\langle\lambda x.x\rangle$  (that is  $\langle\lambda x.\sim\langle x\rangle\rangle \xrightarrow{0} \langle\lambda x.x\rangle$ ) the derivation of this evaluation involves the sub-derivation  $\langle x\rangle \xrightarrow{0} \langle x\rangle$  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 [21]), there is nothing that necessitates this restriction.

### 3 The Problem of MetaML Reductions

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 set of expressions and values:

$$e \in E := x \mid \lambda x.e \mid ee \mid \langle e \rangle \mid \sim e \mid \text{run } e,$$

and we add the following two rules to the  $\beta$  rule:

$$\begin{array}{l} \sim\langle e \rangle \longrightarrow_E e \\ \text{run } \langle e \rangle \longrightarrow_R e. \end{array}$$

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

### 3.1 Intensional Analysis Conflicts with $\beta$ on Raw MetaML Terms

Our first observation is that there is a conflict between the  $\beta$  rule and supporting intensional analysis. Support for intensional analysis means adding constructs to MetaML that would allow a program to inspect a piece of code, and possibly change its execution based on either the structure or content of that piece of code. This conflict is an example of a high-level insight that resulted from studying the formal semantics of MetaML. In particular, MetaML was developed as a *meta-programming language*, and while multi-stage programming does not need to concern itself with how the code type is represented, the long-term goals of the MetaML project have at one time included support for intensional analysis. The idea is that intensional analysis could be used, for example, to allow programmers to write their own optimizers for code.

It turns out that such intensional analysis is in direct contention with allowing the  $\beta$ -rule on object-code (that is, at levels higher than 0). To illustrate the interaction between the  $\beta$  rule and intensional analysis, assume that we have a minimal extension to core MetaML that tests a piece of code to see if it is an application. This extension can be achieved using a simple hypothetical construct with the following semantics:

```
-| lsApp <<(fn x => x) (fn y => y)>>;
val it = true : bool.
```

Allowing  $\beta$  on object-code then means that  $\langle\langle\text{fn } x \Rightarrow x \rangle\rangle \langle\langle\text{fn } y \Rightarrow y \rangle\rangle$  can be replaced by  $\langle\langle\text{fn } y \Rightarrow y \rangle\rangle$ . Such a reduction could be performed by an optimizing compiler, and could be justifiable, because it eliminates a function call in the object-program. But such an “optimization” would have a devastating effect on the semantics of MetaML. In particular, it would also allow our language to behave as follows:

```
-| lsApp <<(fn x => x) (fn y => y)>>;
val it = false : bool.
```

When the reduction is performed, the argument to `lsApp` is no longer an application, but simply the lambda term  $\langle\langle\text{fn } y \Rightarrow y \rangle\rangle$ . In other words, allowing both intensional analysis and object-program optimization implies that we can get the result `false` just as well as we can get the result `true`. This example illustrates a problem of coherence of MetaML’s semantics with the presence of  $\beta$  reduction at higher levels, and code inspection. While this issue is what first drew our attention to the care needed in specifying what equalities should hold in MetaML, there are more subtle concerns that are of direct relevance to multi-stage programming, even in the absence of intensional analysis.

### 3.2 Level-Annotated MetaML Terms and Expression Families

In order to control the applicability of the  $\beta$  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.

Level-annotated terms induce an infinite family of sets  $E^0, E^1, E^2, \dots$  where each annotated term lives. The *family* of level-annotated expressions and values is defined as follows:

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

The key difference between level-annotated terms and raw terms is in the “leaves”, namely, the variables<sup>5</sup>. In level-annotated terms, variables explicitly carry around a natural number that represents

<sup>5</sup> The original definition of level-annotated terms [23] had every construct carrying level annotations. There is a one-to-one corresponds between that definition and the simpler definition we use here.

their level. For all other constructs, we can simply infer the level of the whole term by looking at the sub-term. For Brackets and Escapes, the obvious “correction” to levels is performed.

Note that whenever we “go inside” a Bracket or an Escape, the index of the expression set is changed in accordance with the way the level changes when we “go inside” a Bracket or an Escape.

### 3.3 Escapes Conflict with $\beta$ on Annotated MetaML terms

There is a problematic interaction between the  $\beta$  rule at higher levels and Escape. In particular,  $\beta$  does not preserve the syntactic categories of level annotated terms. Consider the following term:

$$\langle\langle \text{fn } x \Rightarrow \sim x^0 \rangle \sim \langle 4^1 \rangle\rangle.$$

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

$$\langle \sim \sim \langle 4^1 \rangle \rangle.$$

This result contains two nested Escapes. Thus, the level of the whole term can no longer be 0. The outer Escape corresponds to the Bracket, but what about the inner Escape? Originally, it corresponded to the same Bracket, but after the  $\beta$  reduction, what we get is an expression that cannot be read in the same manner as the original term.

### 3.4 Substitution Conflicts with $\beta$ on Level 0 Annotated Terms

One possibility for avoiding the problem above is to limit  $\beta$  to level 0 terms:

$$(\lambda x. e_1^0) e_2^0 \longrightarrow_{\beta} e_1^0[x := e_2^0].$$

At first, this approach is appealing because it makes the extension of MetaML with code inspection operations less problematic. But consider the following term:

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

There are two possible  $\beta$  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:

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

The level annotations need to be adjusted after substitution (See [23].) But first note that there are no  $\beta$  reductions at level 0 left in this term. If we do the second application first, we get

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

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

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

Again, in the presence of code inspection, this example illustrates an incoherence problem. But even in the absence of code inspection, we still lose the *confluence* of our reductions, despite the fact that we have sacrificed  $\beta$  reductions at higher-levels. Intuitively, the example above illustrates that *cross-stage persistence* [24], 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  $\beta$  to level 0 in a consistent (that is, confluent) way. In the example above, applying the lift-like term  $\text{fn } x \Rightarrow \langle x \rangle$  to a function causes all the redices in the body of that function to be frozen.

## 4 Reduction Semantics for CBN $\lambda$ -U

The syntax of  $\lambda$ -U consists of the set of *raw* expressions and values defined as follows:

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

The essential subtlety in this definition is in the last production in the set of values: Inside the Brackets of a code value, what is needed is simply *an expression of level 0*.

**Definition 1 (CBN  $\lambda$ -U).** *The CBN notions of reduction of  $\lambda$ -U are simply:*

$$\begin{aligned} (\lambda x.e_1^0) e_2^0 &\longrightarrow_{\beta_V} e_1^0[x := e_2^0] \\ \sim \langle e^0 \rangle &\longrightarrow_{E_V} e^0 \\ \text{run } \langle e^0 \rangle &\longrightarrow_{R_V} e^0. \end{aligned}$$

Just like the rules for a CBN lambda calculus, these rules are intended to be applied in any context. This calculus allows us to apply the  $\beta$  rule to any expression that *looks like* a level 0 application. By restricting the body of the lambda term and its argument to be in  $E^0$ , the  $\lambda$ -U language avoids the conflict between Escapes and  $\beta$  that we discussed earlier on, because level 0 terms are free of top-level Escapes.

*Remark 2 (The Coarse Big-Step Semantics is not Enough).* It is necessary to stratify the set of expressions into expression families. In particular, our notions of reduction are certainly *not* sound if we do not explicitly forbid the application of the coarse big-step semantic function on terms that are manifestly not at the right level. In particular, consider the term  $\sim \langle i \rangle \in E^1$ . If this term is subjected to the coarse big-step semantic function at level 0, the result is undefined. However, if we “optimize” this term using the Escape reduction of  $\lambda$ -U, we get back the term  $i$ , for which the big-step semantics *is* defined. Applying a reduction to a sub-term of a program should *not* change its termination behavior. As such, the stratification of the expressions is *crucial* to the correctness of our notions of reduction.

## 5 Summary of Technical Results

This section presents the statement and gives an overview of our two main technical results on CBN  $\lambda$ -U, namely, confluence and soundness with respect to the fine big-step semantics of CBN  $\lambda$ -M. The details of the two results are presented in Appendix A.

### 5.1 Confluence

Establishing the confluence property in the presence of the  $\beta$  rule can be involved, largely because substitution can duplicate redices, and establishing that these redices are not affected by substitution can be non-trivial. Barendregt presents a number of different ways for proving confluence, and discusses their relative merits [1]. Recently, Takahashi has produced a concise yet highly rigorous technique for proving confluence, and demonstrated its application in a variety of settings, including proving some subtle properties of reduction systems such as standardization [26]. The basic idea that Takahashi promotes is the use of an explicit notion of a parallel reduction. While the idea goes back to the original and classic (yet unpublished) works of Tait and Martin-Löf, Takahashi emphasizes that the rather verbose notion of residuals (see Barendregt [1], for example,) can be completely avoided.

The CBN reductions of  $\lambda$ -U do not introduce any notable complications to the proof, and it is as simple, concise, and rigorous as the one presented by Takahashi. Our proof follows closely the development in the introduction to Takahashi’s paper.

**Theorem 1 (CBN  $\lambda$ -U is Confluent).**  $\forall e_1, e, e_2 \in E.$

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

## 5.2 Soundness of CBN $\lambda$ -U under Fine CBN $\lambda$ -M

CBN  $\lambda$ -U reductions preserve observational equivalence, where our notion of observation is simply the termination behavior of the level 0  $\lambda$ -M big-step evaluation. A reduction semantics for a lambda calculus is generally not “equal” to a big-step semantics. For example, the reduction semantics for the lambda calculus can do “reductions under lambda”, and the big-step semantics generally does not [13, 28].

**Definition 2 (Level 0 Termination).**  $\forall e \in E^0$ .

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

**Definition 3 (Observational Equivalence).** We define  $\approx_n \in E^n \times E^n$  as follows:  $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$ .

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

*Remark 3.* The definition says that two terms are observationally equivalent exactly when they can be interchanged in every level 0 term without affecting the level 0 termination behavior of the term.

**Notation 2** We will drop the  $U$  subscript from  $\longrightarrow_U$  in the rest of presentation.

**Theorem 3 (CBN  $\lambda$ -U Reduction is Sound under  $\lambda$ -M Big-Steps).**  $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n$ .

$$e_1 \longrightarrow e_2 \implies e_1 \approx_n e_2.$$

## 6 Related Works

The author’s dissertation reports on the results of scrutinizing the design and implementation of the multi-stage programming language MetaML. Our approach to studying MetaML has been to formalize its semantics and type system, and the properties we expect them to enjoy. In doing so, we have identified a variety of subtleties related to multi-stage programming, and provided solutions to a number of them. The results presented in this paper are based directly on Chapters 5 and 6 of the author’s dissertation.

Muller [17] studies the reduction semantics of `quote` and `eval` in the context of LISP. 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 [18] also studies the reduction semantics of the lambda-calculus extended with representations of lambda terms, and with a notion of  $\beta$  reduction on these representations. Muller observes that this calculus lacks confluence, and uses a type system to restore confluence.

In both of Muller’s studies, the language can express taking object-code apart (intensional analysis).

Wand [27] has studied the equational theory for LISP meta-programming construct `fexpr` and found that “The Theory of `fexprs` is Trivial” in the sense that the  $\beta$ -rule (or “semantic equality”) is not valid on `fexprs`. Wand predicted that there are other meta-programming languages with a more interesting “non-trivial” equational theory. As we have demonstrated in this paper, MetaML is an example of such a language.

Moggi [14] points out that two-level languages generally have not been presented along with an equational calculus. Our reduction semantics has eliminated this problem for CBN MetaML, and to our knowledge, is the first correct presentation of a multi-stage language using a reduction semantics. Unfortunately, an earlier attempt to devise such a reduction semantics [23] is flawed (although the type system presented in that work is correct [16]). The earlier attempt was based on level-annotated terms, and therefore suffers from the complications discussed in Section 3.3.

Bawden [2] gives a detailed historical review of the history of quasi-quotations in LISP.

## 7 Future Work: CBV $\lambda$ -M and $\lambda$ -U

Corresponding results on the confluence and soundness of CBV  $\lambda$ -U have not been established yet. We chose to start with CBN for two reasons:

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 ( $\lambda$ -M) is only in the evaluation rule for application at level 0. For CBV, this rule becomes

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_2 \quad e[x := v_2] \overset{0}{\hookrightarrow} v}{e_1 e_2 \overset{0}{\hookrightarrow} v} \text{ App-CBV.}$$

The difference between CBV and the CBN reduction semantics for MetaML ( $\lambda$ -U) is also only in the rule for application:

$$(\lambda x.e) v \longrightarrow_{\beta_v} e[x := v],$$

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  $\lambda$ -U. In particular, the notion of value induced by the big-step semantics for a call-by-value lambda language is not the same as the notion of value used in the reduction semantics for call-by-value languages. The latter typically contains variables. This subtle difference will require distinguishing between the two notions throughout the soundness proof.

## 8 Conclusions

In this paper, we presented big-step semantics ( $\lambda$ -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 ( $\lambda$ -U) that we have shown to be confluent and sound with respect to our notion of observational equivalence based on the level 0 termination behavior 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  $\beta$  reduction to level 0 in MetaML, that is, it is hard to stop semantic equality at the meta-level from “leaking” into the object-level. The alternative interpretation of our observations is that it is more natural to allow semantic equality at all levels in MetaML. The essential reason is that the level of a raw MetaML terms is not “local” information that can be determined just by looking at the term. Rather, the level of a term is determined from the context. Because of substitution (in general) and cross-stage persistence (in particular), we are forced to allow  $\beta$  to “leak” into higher levels. This “leakage” of the  $\beta$ -rule could be interpreted as a desirable phenomenon because it would allow implementations of MetaML to perform a wider range of semantics-preserving optimizations on programs. If we accept this interpretation and therefore accept allowing  $\beta$  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.

## 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. Quasiquote 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. 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.
4. 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.
5. GLÜCK, R., AND JØRGENSEN, J. An automatic program generator for multi-level specialization. *LISP and Symbolic Computation* 10, 2 (1997), 113–158.
6. GOMARD, C. K., AND JONES, N. D. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming* 1, 1 (Jan. 1991), 21–69.
7. GUNTER, C. A. *Semantics of Programming Languages*. MIT Press, 1992.
8. HATCLIFF, J., AND DANVY, O. Thunks and the  $\lambda$ -calculus. *Journal of Functional Programming* 7, 3 (May 1997), 303–319.
9. 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.
10. JONES, N. D., GOMARD, C. K., AND SESTOFT, P. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
11. 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.
12. MACHKASOVA, E., AND TURBAK, F. A. A calculus for link-time compilation (extended abstract). (Unpublished manuscript.) By way of Franklyn A. Turbak (fturbak@wellesley.edu), June 1999.
13. MITCHELL, J. C. *Foundations for Programming Languages*. MIT Press, Cambridge, 1996.
14. MOGGI, E. A categorical account of two-level languages. In *Mathematics Foundations of Program Semantics* (1997), Elsevier Science.
15. MOGGI, E., TAHA, W., BENAÏSSA, Z., AND SHEARD, T. An idealized MetaML: Simpler, and more expressive (includes proofs). Tech. Rep. CSE-98-017, OGI, Oct. 1998. Available from [20].
16. MOGGI, E., TAHA, W., BENAÏSSA, Z., AND SHEARD, T. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)* (1999), vol. 1576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 193–207. An extended version appears in [15].
17. 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.
18. 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.
19. NIELSON, F., AND NIELSON, H. R. *Two-Level Functional Languages*. No. 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, 1992.
20. 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.
21. PLOTKIN, G. D. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science* 1 (1975), 125–159.
22. 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).
23. TAHA, W., BENAÏSSA, 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.
24. 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 [25].
25. TAHA, W., AND SHEARD, T. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1-2 (In Press).
26. TAKAHASHI, M. Parallel reductions in  $\lambda$ -calculus. *Information and Computation* 118, 1 (Apr. 1995), 120–127.
27. WAND, M. The theory of fexprs is trivial. *Lisp and Symbolic Computation* 10 (1998), 189–199.
28. WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing series. MIT Press, Feb. 1993.

## A Proofs and Auxiliary Definitions

### A.1 Basic Properties of Big-Step Semantics

*Proof (Lemma 1).* All parts of this lemma are proven by easy inductions.

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

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

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

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

*Proof (Lemma 2).* By simple induction.  $\square$

*Proof (Lemma 3).* Part 1 is by a simple induction on the derivation of  $v \in V^n$  to prove that:

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

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

### A.2 Proof of Confluence

**Definition 4 (Context).** A Context is an expression with exactly one hole  $\square$ .

$$C \in \mathbb{C} := \square \mid \lambda x. C \mid C e \mid e C \mid \langle C \rangle \mid \sim C \mid \text{run } C.$$

We write  $C[e]$  for the expression resulting from replacing (“filling”) the hole  $\square$  in the context  $C$  with the expression  $e$ .

**Lemma 4 (Basic Property of Contexts).**  $\forall C \in \mathbb{C}. \forall e \in E.$

$$C[e] \in E.$$

*Proof.* By an induction on the derivation of  $C \in \mathbb{C}$ .  $\square$

*Remark 4.* Filling a hole in a context can involve variable capture, in the sense that given  $C \equiv \lambda x. \square$ ,  $C[x] \equiv \lambda x. x$ , and the binding occurrence of  $x$  in  $C$  is not renamed.

**Definition 5 (Parallel Reduction).** The parallel reduction relation  $\_ \gg \_ \subseteq E \times E$  is defined as follows:

$$\frac{}{x \gg x} \quad \frac{e_1 \gg e_2}{\lambda x. e_1 \gg \lambda x. e_2} \quad \frac{e_1 \gg e_3 \quad e_2 \gg e_4}{e_1 e_2 \gg e_3 e_4} \quad \frac{e_1^0 \gg e_3 \quad e_2^0 \gg e_4}{(\lambda x. e_1^0) e_2^0 \gg e_3[x := e_4^0]} \\ \frac{e_1 \gg e_2}{\langle e_1 \rangle \gg \langle e_2 \rangle} \quad \frac{e_1 \gg e_2}{\sim e_1 \gg \sim e_2} \quad \frac{e_1^0 \gg e_2^0}{\sim \langle e_1^0 \rangle \gg \langle e_2^0 \rangle} \quad \frac{e_1 \gg e_2}{\text{run } e_1 \gg \text{run } e_2} \quad \frac{e_1^0 \gg e_2^0}{\text{run } \langle e_1^0 \rangle \gg \langle e_2^0 \rangle}.$$

*Remark 5 (Idempotence).* It is easy to see that  $e \gg e$ .

**Lemma 5 (Parallel Reduction Properties).**  $\forall e_1 \in E.$

1.  $\forall e_2 \in E. e_1 \longrightarrow e_2 \implies e_1 \gg e_2$
2.  $\forall e_2 \in E. e_1 \gg e_2 \implies e_1 \longrightarrow^* e_2$

3.  $\forall e_3 \in E. e_2, e_4 \in E^0. e_1 \gg e_3, e_2^0 \gg e_4^0 \implies e_1[y := e_2^0] \gg e_3[y := e_4^0]$ .

*Proof.* The first is proved by induction on the context of the redex, and the second and third by induction on  $e_1$ .  $\square$

*Remark 6.* From 1 and 2 above we can see that that  $\gg^* = \longrightarrow^*$ .

The Church-Rosser theorem [1] for  $\longrightarrow$  follows from Takahashi's property [26]. The statement of Takahashi's property uses the following notion.

**Definition 6 (Star Reduction).** *The star reduction function  $_* : E \rightarrow E$  is defined as follows:*

$$\begin{aligned} x^* &\equiv x \\ (\lambda x. e_1)^* &\equiv \lambda x. e_1^* \\ (e_1 e_2)^* &\equiv e_1^* e_2^* \text{ if } e_1 e_2 \not\equiv (\lambda x. e_3^0) e_4^0 \\ ((\lambda x. e_1^0) e_2^0)^* &\equiv (e_1^0)^* [x := (e_2^0)^*] \\ \langle e_1 \rangle^* &\equiv \langle e_1^* \rangle \\ (\sim e_1)^* &\equiv \sim (e_1^*) \text{ if } \sim e_1 \not\equiv \sim \langle e_2^0 \rangle \\ \sim \langle e_1^0 \rangle^* &\equiv (e_1^0)^* \\ (\text{run } e_1)^* &\equiv \text{run } (e_1^*) \text{ if } \text{run } e_1 \not\equiv \text{run } \langle e_2^0 \rangle \\ (\text{run } \langle e_1^0 \rangle)^* &\equiv (e_1^0)^*. \end{aligned}$$

*Remark 7.* By a simple induction on  $e$ , we can see that  $e \gg e^*$ .

**Theorem 4 (Takahashi's Property).**  $\forall e_1, e_2 \in E$ .

$$e_1 \gg e_2 \implies e_2 \gg e_1^*.$$

*Proof.* By induction on  $e_1$ .  $\square$

The following two results then follow in sequence:

**Notation 5 (Relation Composition)** *For any two relations  $\oplus$  and  $\otimes$ , we write  $a \oplus b \otimes c$  as a shorthand for  $(a \oplus b) \wedge (b \otimes c)$ .*

**Lemma 6 (Parallel Reduction is Diamond).**  $\forall e_1, e, e_2 \in E$ .

$$e_1 \ll e \gg e_2 \implies (\exists e' \in E. e_1 \gg e' \ll e_2).$$

*Proof.* Take  $e' = e^*$  and use Takahashi's property.  $\square$

### Main Confluence Result

*Proof (Theorem 1).* Follows directly from the above lemma.  $\square$

### A.3 Proof of Soundness of Reduction Semantics

*Proof (Theorem 3).* By the definition of  $\approx_n$ , to prove our goal

$$\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n. e_1 \longrightarrow e_2 \implies e_1 \approx_n e_2$$

is to prove

$$\forall n \in \mathbb{N}, C \in \mathbb{C}. \forall e_1, e_2 \in E^n. e_1 \longrightarrow e_2 \wedge C[e_1], C[e_2] \in E^0 \implies (C[e_1] \Downarrow \iff C[e_2] \Downarrow).$$

Noting that by the compatibility of  $\longrightarrow$ , we know that  $\forall n \in \mathbb{N}, C \in \mathbb{C}. \forall e_1, e_2 \in E^n. e_1 \longrightarrow e_2 \implies C[e_1] \longrightarrow C[e_2]$ , it is sufficient to prove a stronger statement:

$$\forall n \in \mathbb{N}, C \in \mathbb{C}. \forall e_1, e_2 \in E^n. C[e_1] \longrightarrow C[e_2] \wedge C[e_1], C[e_2] \in E^0 \implies (C[e_1] \Downarrow \iff C[e_2] \Downarrow).$$

Noting further that  $\forall n \in \mathbb{N}, C \in \mathbb{C}. \forall a, b \in E^n. a \equiv C[b] \in E^0 \implies a \in E^0$ , it is sufficient to prove an even stronger statement:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies (e_1 \Downarrow \iff e_2 \Downarrow).$$

This goal can be broken down into two parts:

S1

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies (e_1 \Downarrow \implies e_2 \Downarrow),$$

and

S2

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies (e_2 \Downarrow \implies e_1 \Downarrow).$$

Let us consider S1. By definition of termination, it says:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v)).$$

We will show that big-step evaluation is included in reduction (Lemma 7). Thus, to prove S2 it is enough to prove:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0. e_1 \longrightarrow^* v) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v)).$$

Confluence (Theorem 1) tell us that any two reduction paths are joinable, so we can weaken our goal as follows:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0, e_3 \in E. e_1 \longrightarrow^* v \longrightarrow^* e_3 \wedge e_2 \longrightarrow^* e_3) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v))$$

We will show (Lemmas 13 and Remark 6) that any reduction that starts from a value can only lead to a value (at the same level). Thus we can weaken further:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v, v_3 \in V^0. e_1 \longrightarrow^* v \longrightarrow^* v_3 \wedge e_2 \longrightarrow^* v_3) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v))$$

In other words, we already know that  $e_2$  reduces to a value, and the question is really whether it *evaluates* to a value. Formally:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v_3 \in V^0. e_2 \longrightarrow^* v_3) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v)).$$

In fact, the original assumption is no longer necessary, and we will prove:

T1

$$\forall e_2 \in E^0. ((\exists v_3 \in V^0. e_2 \longrightarrow^* v_3) \implies (\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v)).$$

Now consider S2. By definition of termination, it says:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0. e_2 \overset{0}{\hookrightarrow} v) \implies (\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v)).$$

Again, by the inclusion of evaluation in reduction, we can weaken:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0. e_2 \longrightarrow^* v) \implies (\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v)).$$

Given the first assumption in this statement we can also say:

$$\forall e_1, e_2 \in E^0. e_1 \longrightarrow e_2 \implies ((\exists v \in V^0. e_1 \longrightarrow^* v) \implies (\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v)),$$

and we no longer need the assumption as it is sufficient to show:

T2

$$\forall e_1 \in E^0. ((\exists v \in V^0. e_1 \longrightarrow^* v) \implies (\exists v \in V^0. e_1 \overset{0}{\hookrightarrow} v)).$$

But note that T1 and T2 are identical goals. They state:

T

$$\forall e \in E^0. ((\exists v \in V^0. e \longrightarrow^* v) \implies (\exists v \in V^0. e \overset{0}{\hookrightarrow} v)).$$

This statement is a direct consequence of Lemma 8.  $\square$

It is easy to show that  $e^0 \overset{0}{\hookrightarrow} v \implies e^0 \longrightarrow^* v$ , as it follows directly from the following result:

**Lemma 7 (CBN  $\lambda$ -M is in CBN  $\lambda$ -U).**  $\forall n \in \mathbb{N}. \forall e \in E^n, v \in V^n.$

$$e \overset{n}{\hookrightarrow} v \implies e \longrightarrow^* v.$$

*Proof.* By a straightforward induction on the height of the judgement  $e \overset{n}{\hookrightarrow} v$ .  $\square$

What is harder to show is the “converse”, that is, that  $e^0 \longrightarrow^* v \implies (\exists v' \in V^0. e^0 \overset{0}{\hookrightarrow} v')$ . It is a consequence of the following stronger result:

**Lemma 8 (CBN  $\lambda$ -U is in CBN  $\lambda$ -M).**  $\forall e \in E^0, v_1 \in V^0.$

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in V^0. e \overset{0}{\hookrightarrow} v_3 \longrightarrow^* v_1).$$

*Proof.* We arrive at this result by an adaptation of Plotkin’s proof for a similar result for the CBV and CBN lambda calculi [21]. The main steps in the development are:

1. We strengthen our goal to become:

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in V^0. e \overset{0}{\hookrightarrow} v_3 \longrightarrow^* v_1).$$

2. We define a *left reduction function*  $\overset{n}{\mapsto}$  (Definition 8) such that (Lemma 12):  $\forall e \in E^0, v \in V^0.$

$$e \overset{0}{\mapsto}^* v \iff e \overset{0}{\hookrightarrow} v$$

and  $\forall e_1, e_2 \in E^0. e_1 \overset{0}{\mapsto} e_2 \implies e_1 \longrightarrow e_2$  (Lemma 11). Thus, big-step evaluation (or simply evaluation) is exactly a chain of left reductions that ends in a value.

3. Our goal is restated as:

$$e \longrightarrow^* v_1 \implies (\exists v_3 \in V^0. e \overset{0}{\mapsto}^* v_3 \longrightarrow^* v_1).$$

4. For technical reasons, the proofs are simpler if we use a parallel reduction relation  $\gg$  (Definition 9) similar to the one introduced in the last section. Our goal is once again restated as:

$$e \gg^* v_1 \implies (\exists v_3 \in V^0. e \overset{0}{\mapsto}^* v_3 \gg^* v_1).$$

5. The left reduction function induces a very fine classification on terms (Definition 7). In particular, any term  $e \in E^n$  must be exactly one of the following three (Lemma 9):

- (a) a *value*  $e \in V^n$ ,
- (b) a *workable*  $e \in W^n$ , or
- (c) a *stuck*  $e \in S^n$ ,

where membership in each of these three sets is defined inductively over the structure of the term. We write  $v^n, w^n$  and  $s^n$  to refer to a member of one of the three sets above, respectively. Left reduction at level  $n$  is a total function exactly on the members of the set  $W^n$  (Lemma 10). Thus, left reduction is strictly undefined on non-workables, that is, it is undefined on values and on stuck terms. Furthermore, if the result of any parallel reduction is a value, the source must have been either a value or a workable (Lemma 13). We will refer to this property of parallel reduction as *monotonicity*.

6. Using the above classification, we break our goal into two cases, depending on whether the starting point is a value or a workable:

$$\text{G1 } \forall v_1, v \in V^0.$$

$$v \gg^* v_1 \implies (\exists v_3 \in V^0. v = v_3 \gg^* v_1),$$

$$\text{G2 } \forall w \in W^0, v \in V^0.$$

$$w \gg^* v_1 \implies (\exists v_3 \in V^0. w \xrightarrow{0+} v_3 \gg^* v_1).$$

It is obvious that G1 is true. Thus, G2 becomes the current goal.

7. By the monotonicity of parallel reduction, it is clear that all the intermediate terms in the reduction chain  $w^0 \gg^* v_1^0$  are either workables or values. Furthermore, workables and values do not interleave, and there is exactly one transition from workables to values in the chain. Thus, this chain can be visualized as follows:

$$w_1^0 \gg w_2^0 \gg \dots w_{k-1}^0 \gg w_k^0 \gg v^0 \gg^* v_1^0.$$

We prove that the transition  $w_k^0 \gg v^0$  can be replaced by an evaluation (Lemma 15):

$$\text{R1 } \forall w \in W^0, v \in V^0.$$

$$w \gg v \implies (\exists v_2 \in V^0. w \xrightarrow{0+} v_2 \gg v).$$

With this lemma, we know that we can replace the chain above by one where the evaluation involved in going from the last workable to the first value is explicit:

$$w_1^0 \gg w_2^0 \gg \dots w_{k-1}^0 \gg w_k^0 \xrightarrow{0+} v_2^0 \gg^* v_1^0.$$

What is left is then to “push back” this information about the last workable in the chain to the very first workable in the chain. This is achieved by a straightforward iteration (by induction over the number of  $k$  of workables in the chain) of a result that we prove (Lemma 17):

$$\text{R2 } \forall w_1, w_2 \in W^0, v_1 \in V^0.$$

$$w_1 \gg w_2 \xrightarrow{0+} v_1 \implies (\exists v_2 \in V^0. w_1 \xrightarrow{0+} v_2 \gg v_1).$$

With this result, we are able to move the predicate  $\_ \xrightarrow{0+} v_3^0 \gg^* v^0$  all the way back to the first workable in the chain. This step can be visualized as follows. With one application of R2 we have the chain:

$$w_1^0 \gg w_2^0 \gg \dots w_{k-1}^0 \xrightarrow{0+} v_3^0 \gg^* v_1^0,$$

and with  $k - 2$  applications of R2 we have:

$$w_1^0 \xrightarrow{0+} v_{k+1}^0 \gg^* v_1^0,$$

thus completing the proof. □

In the rest of this section, we present the definitions and lemmas mentioned above. It should be noted that proving most of the lemmas mentioned above require generalizing the level from 0 to  $n$ . In the rest of the development, we present the generalized forms, which can be trivially instantiated to the statements mentioned above.

## A Basic Classification of Terms

**Definition 7 (Classes).** We define three judgements on raw (that is, type-free) classes: Values  $V^n$ , Workables  $W^n$ , and Stuck terms  $S^n$ . The four sets are defined as follows:

$$\begin{aligned}
v^0 &\in V^0 &:= \lambda x.e^0 \mid \langle v^1 \rangle \\
v^1 &\in V^1 &:= x \mid \lambda x.v^1 \mid v^1 v^1 \mid \langle v^2 \rangle \mid \text{run } v^1 \\
v^{n++} &\in V^{n++} &:= x \mid \lambda x.v^{n++} \mid v^{n++} v^{n++} \mid \langle v^{n+++} \rangle \mid \sim v^{n+} \mid \text{run } v^{n++} \\
w^0 &\in W^0 &:= (\lambda x.e^0) e^0 \mid w^0 e^0 \mid \text{run } \langle e^0 \rangle \mid \text{run } w^0 \mid \langle w^1 \rangle \\
w^1 &\in W^1 &:= \lambda x.w^1 \mid w^1 e^1 \mid v^1 w^1 \mid \langle w^2 \rangle \mid \sim w^0 \mid \sim \langle e^0 \rangle \\
w^{n++} &\in W^{n++} &:= \lambda x.w^{n++} \mid w^{n++} e^{n++} \mid v^{n++} w^{n++} \mid \langle w^{n+++} \rangle \mid \sim w^{n+} \\
s^0 &\in S^0 &:= x \mid s^0 e^0 \mid \langle v^1 \rangle e^0 \mid \langle s^1 \rangle \mid \text{run } \lambda x.e^0 \mid \text{run } s^0 \\
s^1 &\in S^1 &:= \lambda x.s^1 \mid s^1 e^1 \mid v^1 s^1 \mid \langle s^2 \rangle \mid \sim s^0 \mid \sim \lambda x.e^0 \mid \text{run } s^1 \\
s^{n++} &\in S^{n++} &:= \lambda x.s^{n++} \mid s^{n++} e^{n++} \mid v^{n++} s^{n++} \mid \langle s^{n+++} \rangle \mid \sim s^{n+} \mid \text{run } s^{n++}.
\end{aligned}$$

**Lemma 9 (Basic Properties of Classes).**  $\forall n \in \mathbb{N}$ .

1.  $V^n, W^n, S^n \subseteq E^n$
2.  $V^n, W^n, S^n$  partition  $E^n$ .

*Proof.* All these properties are easy to prove by straightforward induction.

1. We verify the claim for each case separately, by induction on  $e \in V^n$ ,  $e \in W^n$ , and  $e \in S^n$ , respectively.
2. We prove that,  $e \in E^n$  is in exactly one of the three sets  $V^n, W^n$  or  $S^n$ . The proof is by induction on the judgement  $e \in W^n$ . This proof is direct albeit tedious. □

**Left Reduction** The notion of left reduction is intended to capture precisely the reductions performed by the big-step semantics, in a small-step manner. Note that the simplicity of the definition depends on the fact that the partial function being defined is *not* defined on values. That is, we expect that there is no  $e$  such that  $v^n \xrightarrow{n} e$ .

**Definition 8 (Left Reduction).** Left reduction is a partial function  $\_ \xrightarrow{n} \_ : E^n \rightarrow E^n$  defined as follows:

$$\frac{\frac{\frac{\lambda x.v_1^1}{e_1 \xrightarrow{n} e'_1} \quad v_2^1 \xrightarrow{0} v_1^1[x := v_2^1]}{e_1 e_2 \xrightarrow{n} e'_1 e_2} \quad \frac{e_2 \xrightarrow{n+} e'_2}{v_1^{n+} e_2 \xrightarrow{n+} v_1^{n+} e'_2}}{\frac{\text{run } \langle v^1 \rangle \xrightarrow{0} v^1 \quad \sim \langle v^1 \rangle \xrightarrow{1} v^1}{e \xrightarrow{n+} e'} \quad \frac{e \xrightarrow{n} e'}{\langle e \rangle \xrightarrow{n} \langle e' \rangle} \quad \frac{e \xrightarrow{n+} e'}{\sim e \xrightarrow{n+} \sim e'} \quad \frac{\frac{e \xrightarrow{n+} e'}{\lambda x.e \xrightarrow{n+} \lambda x.e'} \quad \frac{e \xrightarrow{n} e'}{e \xrightarrow{n} e'}}{\text{run } e \xrightarrow{n} \text{run } e'}}{\_ \xrightarrow{n} \_}$$

The following lemma says that the set of workables characterizes exactly the set of terms that can be advanced by left reduction.

**Lemma 10 (Left Reduction and Classes).**  $\forall n \in \mathbb{N}$ .

1.  $\forall w \in W^n. (\exists e' \in E^n. w \xrightarrow{n} e')$
2.  $\forall e \in E^n. (\exists e' \in E^n. e \xrightarrow{n} e') \implies e \in W^n$
3.  $\forall v \in V^n. \neg(\exists e' \in E^n. v \xrightarrow{n} e')$
4.  $\forall s \in S^n. \neg(\exists e' \in E^n. s \xrightarrow{n} e')$ .

*Proof.* We only need to prove the first two, and the second two follow. The first one is by straightforward induction on the judgement  $e \in W^n$ . The second is also by straightforward induction on the derivation  $e^n \xrightarrow{n} e'$ . □

**Lemma 11 (Left Reduction and CBN  $\lambda$ -U).**  $\forall n \in \mathbb{N}. \forall e_1, e_2 \in E^n.$

$$e_1 \xrightarrow{n} e_2 \implies e_1 \longrightarrow e_2.$$

*Proof.* By straightforward induction on the first judgement.  $\square$

**Lemma 12 (Left Reduction and  $\lambda$ -M).**  $\forall n \in \mathbb{N}. \forall e \in E^n, v \in V^n.$

$$e \xrightarrow{n}^* v \iff e^n \xrightarrow{n} v^n.$$

*Proof.* The forward direction is by induction on the length of the derivation, and then over the size of  $e$ . The second ordering is not needed in the lambda calculus, but is needed for evaluation at higher levels. The proof proceeds by a case analysis on the first left reduction in the left reduction chain. The backward direction is by straightforward induction on the height of the derivation of  $e \xrightarrow{n} v$ .  $\square$

**Parallel Reduction with Complexity** In order to prove the two key lemmas presented in this section, we will need to reason by induction on the “complexity” of parallel reduction. Thus, we will use the following definition of parallel reduction with an associated complexity measure:

**Definition 9 (Parallel Reduction with Complexity).** *Parallel reduction  $\xrightarrow{M} : E \rightarrow E$  defined as follows:*

$$\frac{x \xrightarrow{0} x}{e_1 \xrightarrow{M} e'_1} \quad \frac{e_1^0 \xrightarrow{M} e'_1 \quad e_2^0 \xrightarrow{N} e'_2}{(\lambda x. e_1^0) e_2^0 \xrightarrow{M+\#(x, e'_1)N+1} e'_1[x := e'_2]} \quad \frac{e_1^0 \xrightarrow{M} e'_1}{\text{run } \langle e_1^0 \rangle \xrightarrow{M+1} e'_1} \quad \frac{e_1^0 \xrightarrow{M} e'_1}{\sim \langle e_1^0 \rangle \xrightarrow{M+1} e'_1}$$

$$\frac{e_1 \xrightarrow{M} e'_1}{\lambda x. e_1 \xrightarrow{M} \lambda x. e'_1} \quad \frac{e_1 \xrightarrow{M} e'_1}{\langle e_1 \rangle \xrightarrow{M} \langle e'_1 \rangle} \quad \frac{e_1 \xrightarrow{M} e'_1}{\sim e_1 \xrightarrow{M} \sim e'_1} \quad \frac{e_1 \xrightarrow{M} e'_1}{\text{run } e_1 \xrightarrow{M} \text{run } e'_1} \quad \frac{e_1 \xrightarrow{M} e'_1 \quad e_2 \xrightarrow{N} e'_2}{e_1 e_2 \xrightarrow{M+N} e'_1 e'_2},$$

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

There is a sense in which parallel reduction should respect the classes. The following lemma explicates these properties.

**Lemma 13 (Parallel Reduction and Classes).**  $\forall n \in \mathbb{N}.$

1.  $\forall e_1 \in E^n, e_2 \in E. e_1 \xrightarrow{M} e_2 \implies e_2 \in E^n$
2.  $\forall e \in E^n, v \in V^n. v \xrightarrow{M} e \implies e \in V^n$
3.  $\forall e \in E^n, s \in S^n. s \xrightarrow{M} e \implies e \in S^n$
4.  $\forall e \in E^n, w \in W^n. e \xrightarrow{M} w \implies e \in W^n.$

*Proof.* The first part of this lemma is proved by straightforward induction on the height of the reduction derivation. It is then enough to establish the second two parts of this lemma, and then the fourth part follows immediately. The proof of the first two is also by a straightforward induction on the derivations of  $e \in V^n$  and  $e \in S^n$ , respectively.  $\square$

*Remark 8.* We have already shown that parallel reduction without complexity is equivalent (in many steps) to normal reduction (in many steps). The same result applies to this annotated definition.

**Lemma 14 (Substitution for Parallel Reduction with Complexity).**  $\forall e_4, e_5, e_6, e_7 \in E, X, Y \in \mathbb{N}.$

$$e_4 \xrightarrow{X} e_5 \wedge e_6^0 \xrightarrow{Y} e_7 \implies (\exists Z \in \mathbb{N}. e_4[x := e_6^0] \xrightarrow{Z} e_5[x := e_7] \wedge Z \leq X + \#(x, e_5)Y).$$

*Proof.* By induction on the height of the derivation  $e_4 \xrightarrow{X} e_5$ . (A direct extension of the proof for Lemma 5 on page 137 of Plotkin [21].)  $\square$

**Lemma 15 (Transition).**  $\forall n, X \in \mathbb{N}. \forall w \in W^n, v \in V^n.$

$$w \ggg^X v \implies (\exists v_2 \in E^n, Y \in \mathbb{N}. w \mapsto^{n+} v_2 \ggg^Y v) \wedge Y < X.$$

*Proof.* By induction on the complexity  $X$  and then on the size of  $w$ . (A direct combination and extension of proofs for Lemmas 6 and 7 of Plotkin [21].)  $\square$

**Lemma 16 (Permutation).**  $\forall n, X \in \mathbb{N}. \forall w_1, w_2 \in W^n, e_1 \in E^n.$

$$w_1 \ggg^X w_2 \mapsto^n e_1 \implies (\exists e_2 \in E^n. w_1 \mapsto^{n+} e_2 \ggg e_1).$$

*Proof.* By induction on the complexity  $X$ , and by a case analysis on the last case of the derivation of  $w_1 \ggg^X w_2$ . (A direct extension of Lemmas 8 of the previous reference.)  $\square$

**Lemma 17 (Push Back).**  $\forall X \in \mathbb{N}, w_1, w_2 \in W^0, v_2 \in V^0.$

$$w_1 \ggg^X w_2 \mapsto^{0+} v_1 \implies (\exists v_2 \in V^0. w_1 \mapsto^{0+} v_2 \ggg v_1).$$

*Proof.* The assumption corresponds to a chain of reductions:

$$w_1 \ggg w_2 \mapsto^0 w_3 \mapsto^0 \dots w_{k-1} \mapsto^0 w_k \mapsto^0 v_1.$$

Applying Permutation to  $w_1 \ggg w_2 \mapsto^0 w_3$  gives us  $(\exists e_{2'} \in E^n. w_1 \mapsto^{0+} e_{2'} \ggg w_3)$ . By the monotonicity of parallel reduction, we know that only a workable can reduce to a workable, that is,  $(\exists w_{2'} \in W^n. w_1 \mapsto^{0+} w_{2'} \ggg w_3)$ . Now we have the chain:

$$w_1 \mapsto^{0+} w_{2'} \ggg w_3 \mapsto^0 \dots w_{k-1} \mapsto^0 w_k \mapsto^0 v_1.$$

Repeating this step  $k - 2$  times we have:

$$w_1 \mapsto^{0+} w_{2'} \mapsto^{0+} w_{3'} \mapsto^{0+} \dots w_{k-1'} \ggg w_k \mapsto^0 v_1.$$

Applying Permutation to  $w_{k-1'} \ggg w_k \mapsto^0 v_1$  give us  $(\exists e_{k'} \in E^n. w_{k-1'} \mapsto^{0+} e_{k'} \ggg v_1)$ . By the monotonicity of parallel reduction, we know that  $e_{k'}$  can only be a value or a workable. If it is a value then we have the chain:

$$w_1 \mapsto^{0+} w_{2'} \mapsto^{0+} w_{3'} \mapsto^{0+} \dots w_{k-1'} \mapsto^{0+} v_{k'} \ggg v_1$$

and we are done. If it is a workable, then applying Transition to  $w_{k'} \ggg v_1$  gives us  $(\exists v_2 \in V^n. w_{k'} \mapsto^{0+} v_2 \ggg v_1)$ . This means that we now have the chain:

$$w_1 \mapsto^{0+} w_{2'} \mapsto^{0+} w_{3'} \mapsto^{0+} \dots w_{k-1'} \mapsto^{0+} w_{k'} \mapsto^{0+} v_2 \ggg v_1$$

and we are done.  $\square$

#### A.4 Remarks on Proofs

*Remark 9 (Classes).* Plotkin [21] only names the set of values explicitly. The notions of workables and stuck terms employed in this present work<sup>6</sup> helped us adapt Plotkin's technique to MetaML, and in some cases, to shorten the development. For example, we have combined Plotkin's Lemmas 6 and 7 into one (Lemma 15). We expect that our organization of expressions into values, workables, and stuck terms may also be suitable for applying Plotkin's technique to other programming languages.

<sup>6</sup> Such a classification has also been employed in a work by Hatcliff and Danvy [8], where values and stuck terms are named. At the time of writing these results, we had not found a name for "workables" in the literature.

*Remark 10 (Standardization).* We have not found need for a Standardization Theorem, or for an explicit notion of standard reduction. Also, our development has avoided Lemma 9 of Plotkin [21], and the non-trivial lexicographic ordering needed for proving that lemma.

*Remark 11 (Non-left or “Internal” Reductions).* Many standardization proofs (such as those described by Barendregt [1] and Takahashi [26]) employ “complementary” notions of reduction, such as internal reduction (defined simply as non-head). The development presented in Plotkin (and here), does not require the introduction of such notions. While they do possess interesting properties in our setting (such as the preservation of all classes), they are not needed for the proofs. Machkasova and Turbak [12] also point out that complementary reductions preserve all classes. Using such complementary notions, it may be possible to avoid the use of Plotkin’s notion of complexity, although the rest of the proof remains essentially the same. We plan to further investigate this point in future work.

## B Basics of Big-Step and Reduction Semantics

In this section, we review the notions of big-step and reduction semantics in a simple CBN lambda calculus  $\lambda$ .

### B.1 Big-Step Semantics for $\lambda$

Formalizing a *big-step semantics* (see for example Gunter [7]) allows us to specify the semantics as a function that goes directly from expressions to values. A big-step semantics is a partial function, and therefore resembles an interpreter for our language.

The syntax for the  $\lambda$  language is as follows:

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

The CBN big-step semantics for  $\lambda$  is specified by a partial function  $_ \hookrightarrow _ : E \rightarrow E$ , where  $E$  is the set of  $\lambda$  expressions:

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

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

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

Note that there is no need for an environment that keeps track of bindings of variables: whenever a value is available for a variable, we immediately substitute the value for the variable. This substitution is performed in the rule for applications. It is possible to implement the  $\lambda$  language directly by mimicking the big-step semantics. We should point out, however, that a direct implementation based on this big-step semantics would be somewhat inefficient, as every application would require a traversal of the body of the lambda-abstraction. Most realistic implementations (including the MetaML implementation [24, 22]) do not perform substitution by traversing terms at run-time.

### B.2 Reduction Semantics for $\lambda$

A formal semantics, in general, provides us with a means for going from arbitrary expressions to values, with the provision that certain expressions may not have a corresponding value. An important

conceptual tool for the study of a programming language is a reduction semantics. A *reduction semantics* is a set of rewrite rules that formalize the “notions of reduction” for a given language. Having such a semantics can be useful in developing an equational theory. We will first review how this semantics can be specified for a simple language we call  $\lambda$ .

Recall that the set of expressions and the set of values for the  $\lambda$  language can be defined as follows:

$$\begin{aligned} e \in E &:= i \mid x \mid \lambda x.e \mid ee \\ v \in V &:= i \mid \lambda x.e. \end{aligned}$$

In order, the productions for expressions are for integers, lambda abstractions, and applications. Values for this language are integers and lambda-abstractions.

Intuitively, expressions are “commands” or “computations”, and values are the “answers”, “acceptable results” or simply “expressions that require no further evaluation”. Note that we allow any value to be used as an expression with no computational content. In order to build a mechanism for going from expressions to values, we need to specify a formal rule for eliminating both variables and applications from a program. In a *reduction semantics*, (see for example Barendregt [1]) this elimination process is specified by introducing rewrite rules called “notions of reduction”. The well-known  $\beta$  rule helps us eliminate both applications and variables at the same time:

$$(\lambda x.e_1) e_2 \longrightarrow_{\beta} e_1[x := e_2].$$

This rule says that the application of a lambda-abstraction to an expression can be simplified to the substitution the expression into the body of the lambda-abstraction. The CBN semantics is based on this rule.

Using the  $\beta$  rule, we build a new relation  $\longrightarrow$  (with no subscript) that allows us to perform this rewrite on any subexpressions. More formally, for any two expressions  $C[e]$  and  $C[e']$  which are identical everywhere but in exactly one hole filled with  $e$  and  $e'$ , respectively, we can say:

$$e \longrightarrow_{\beta} e' \implies C[e] \longrightarrow C[e'].$$

When there is more than one rule in our reduction semantics, the left hand side of this condition is the disjunction of the rewrites from  $e$  to  $e'$  using *any* of the rules in our rewrite system. Thus the relation  $\longrightarrow$  holds between any two terms if exactly one of their sub-terms is rewritten using any of the rules in our reduction semantics.

### B.3 Coherence and Confluence

Two important concepts central to this paper 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 could reduce a program  $e$  to either 0 or 1. We say a reduction semantics is *coherent* when any path that leads to a ground value leads to the same ground value. A semantics that lacks coherence is not satisfactory for a deterministic programming language.

Intuitively, knowing that a rewriting system is *confluent* tells us that the reductions can be applied in any order, without affecting the set of results that we can reach by applying more reductions. Thus, confluence of a reduction semantics is a way of ensuring coherence. Conversely, if we lose coherence, we lose confluence.