Refining Semantics for Multi-stage Programming

Rui Ge
University of British Columbia, Canada
rge@cs.ubc.ca

Ronald Garcia
University of British Columba, Canada
rxg@cs.ubc.ca

Abstract

The multi-stage programming paradigm supports runtime code generation and execution. Though powerful, its potential is impeded by the lack of static analysis support. Van Horn and Might proposed a general-purpose approach to systematically develop static analyses by transforming an environmental abstract machine, which evolves a control string, an environment and a continuation as a program evaluates. To the best of our knowledge, no such semantics exists for a multi-stage language like MetaML.

We develop and prove correct an environmental abstract machine semantics for MetaML by gradually refining the reference substitutional structural operational semantics. Highlights of our approach include leveraging explicit substitutions to bridge the gap between substitutional and environmental semantics, and devising meta-environments to model the complexities of variable bindings in multi-stage environmental semantics.

CCS Concepts • **Theory of computation → Operational semantics**; **Abstract machines**;

Keywords multi-stage programming, abstract machine semantics, explicit substitutions, meta-environments

ACM Reference Format:
Rui Ge and Ronald Garcia. 2017. Refining Semantics for Multi-stage Programming. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3136040.3136047

1 Introduction

Multi-stage programming is a programming paradigm that supports runtime code generation and execution [20]. It helps programmers leverage partial evaluation and program specialisation techniques [8] to optimise the time and space consumption of evaluating programs [17]. Though powerful, its potential has been impeded by the lack of development aids such as code refactoring tools and stage-aware optimisers, for which we believe one key culprit is the lack of static analysis support. Our long-term goal is to design sound and decidable static analyses for a multi-stage language.

Recently, Van Horn and Might [26] proposed a general-purpose approach to systematically develop static analyses by transforming a language's formal semantics, which we believe is applicable to multi-stage programming. This approach requires an environmental abstract machine, which evolves a control string, an environment and a continuation as a program evaluates. Using environments to archive variable bindings allows us to trace function calls in the machine's history. To the best of our knowledge, no such semantics exists for a multi-stage language.

In this paper, we study MetaML [23], a functional multi-stage language that extends ML. We develop and prove correct an environmental abstract machine for MetaML by gradually refining the reference substitutional structural operational semantics. Our contributions are as follows:

(1) Refining substitutional semantics to environmental semantics is not straightforward for multi-stage languages. To bridge the gap between substitutional and environmental semantics, we leverage the concept of *explicit substitutions* [1, 3, 15], which internalises how substitutions percolate through a term at the semantical level. We systematically replace the meta-language substitutions in MetaML with explicit substitutions, resulting in *Explicit MetaML* (Section 3).

(2) To more closely resemble the behaviour of environmental semantics, we refine how to represent functions and applications, i.e., what forms of lambda abstractions are applicable functions and how variable bindings are updated by applications, resulting in *Suspended MetaML* (Section 4).

(3) MetaML allows a variable to be bound to an open term. Thus a variable can be associated with a chain of variable bindings. We devise *meta-environments*, i.e., sequences of environments, to capture these possibly sophisticated variable bindings. We refine Suspended MetaML to *Environmental MetaML* by systematically replacing explicit substitutions with meta-environments (Section 5). For Environmental MetaML, we present its structural operational semantics, reduction semantics and abstract machine semantics.

(4) To evince the correctness of refining semantics, we explain how to rigorously model fresh variables (Section 6.1), uncover a global invariant on variable bindings (Section 6.2), and leverage three proof strategies to establish the equivalences of semantics (Section 6.3).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GPCE'17, October 23–24, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5524-7/17/10…$15.00
https://doi.org/10.1145/3136040.3136047

This development is depicted in the diagram below. It cannot be presented in full in this paper for lack of space. Interested readers may consult the first author's thesis [7].

$$\begin{tikzcd}
\text{MetaML} \arrow[rr, dashed, "\text{Long Term Goal}"] \arrow[d, "\text{Sec. 3}"] \arrow[rrd, Leftrightarrow, "\text{Sec. 6}", "\text{Thm. 5.4}"'] & & \text{Static Analysis} \\
\text{Explicit MetaML} \arrow[r, "\text{Sec. 4}"] & \text{Suspended MetaML} \arrow[r, "\text{Sec. 5}"] & \text{Environmental MetaML} \arrow[u, dashed, "{[26]}"]
\end{tikzcd}$$

2 Formal Semantics of MetaML

We first review the syntax of MetaML based on [24].

2.1 Syntax

The terms of MetaML are defined as follows:

$$t \in \textsc{Term} \quad x, y, z, u \in \textsc{Var} \quad n, i, j \in \mathbb{N}$$
$$t \quad ::= \quad x \mid t\ t \mid \lambda x.t \mid \langle t \rangle \mid {\sim} t \mid !t \mid n \mid t + t$$

MetaML uses staging annotations to explicitly control the evaluation order of terms.

A code operation $\langle t \rangle$ indicates delaying a computation t. If a code operation evaluates to itself, it is a code value. For example, $\langle 3 + 7 \rangle$ evaluates to $\langle 3 + 7 \rangle$.

A run operation $!t$ indicates executing a delayed computation t. For example, $!\langle 3 + 7 \rangle$ evaluates to 10.

A splice operation ${\sim}t$ indicates splicing a delayed computation t into the context of a delayed computation. For example, $\langle {\sim}\langle 3 + 7 \rangle * {\sim}\langle 3 + 7 \rangle \rangle$ evaluates to $\langle (3 + 7) * (3 + 7) \rangle$.

As explained in [24], code, run and splice are analogous to LISP's back-quote, eval and comma respectively.

To explicitly regulate under what circumstances run and splice can eliminate code brackets, we introduce the concept of levels. The level of a term is the difference of the numbers of surrounding brackets and surrounding splices. For example, 3 in $\langle {\sim}\ \langle 3 + 7 \rangle * 10 \rangle$ is at level $2 - 1 = 1$. We use levels to finely distinguish subclasses of terms:

$$t^i \in \textsc{Term}^i$$
$$\begin{array}{lcl}
t^0 & ::= & x \mid t^0\ t^0 \mid \lambda x.t^0 \mid \langle t^1 \rangle \mid !t^0 \mid n \mid t^0 + t^0 \\
t^{i+1} & ::= & x \mid t^{i+1}\ t^{i+1} \mid \lambda x.t^{i+1} \mid \langle t^{i+2} \rangle \mid {\sim}t^i \mid !t^{i+1} \\
 & & \mid n \mid t^{i+1} + t^{i+1}
\end{array}$$

A value at level i is a level-i term that represents a result of computation at its indicated level.

$$v^i \in \textsc{Value}^i$$
$$\begin{array}{lcl}
v^0 & ::= & \lambda x.t^0 \mid \langle v^1 \rangle \mid n \\
v^1 & ::= & x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid !v^1 \mid n \mid v^1 + v^1 \\
v^{i+2} & ::= & x \mid v^{i+2}\ v^{i+2} \mid \lambda x.v^{i+2} \mid \langle v^{i+3} \rangle \mid {\sim}v^{i+1} \mid !v^{i+2} \\
 & & \mid n \mid v^{i+2} + v^{i+2}
\end{array}$$

The denotable terms are terms that are substitutable for variables in the semantics. In particular, variables are denotable to represent renaming.

$$w \in \textsc{Denot}$$
$$w \quad ::= \quad x \mid v^0$$

Properties. We elaborate a few observations from [19] about the syntax. Being a cornerstone for establishing the equivalences of semantics, these observations shall be carefully preserved throughout the process of refining semantics.

First, the sets $\textsc{Term}^i$ contain strictly more terms as the level i increases. For example, the term ${\sim}x$ can have any level higher than 0 and the term ${\sim}{\sim}x$ can have any level higher than 1.

Second, every term has a level. To make a conventional single-stage program multi-stage, we can experiment annotating the program with various combinations of staging annotations as long as the program remains at level 0.

Third, there is a one-to-one correspondence between the subclasses of terms and the subclasses of values. On one hand, a value at some non-zero level is a term at the next lower level. This justifies the semantics of run: if we have a level-i value $\langle t \rangle$, then t is a level-$(i + 1)$ value but not necessarily a level-i value. Removing the brackets makes t a level-i term and allows it to reduce at level i. Although running happens only at level 0, this uniformity makes it clear that we can reason seamlessly about multi-stage programs at levels higher than 0 and 1 in the same way that we reason about two-stage programs. One the other hand, a term at some level is a value at the next higher level. We can always delay a computation by putting it into brackets.

Fourth, a level-i value represents the result of a computation at level i. Hence a level-i value must be a level-i term. This corresponds to our intuition and later formalisation of level-indexed evaluation.

2.2 Substitutional Structural Operational Semantics

A substitutional big-step semantics for MetaML is presented in [24]. Since we derive a small-step abstract machine semantics, it is natural to start from an equivalent substitutional structural operational semantics as our reference semantics[1].

The structural operational semantics comprises a family of level-indexed transition relations $t_1^i \longrightarrow^i t_2^i$ in Figure 1.

Not every term has a corresponding transition rule. There is no (ref-0) that reduces a variable at level 0, analogous to the fact that evaluating a free variable in a single-stage language is disallowed. There is no (splice-0) because a splice operation is always at some level higher than 0. There are no (lambda-0), (ref-(i+1)) and (num-i) because values are irreducible.

The (app-0) rule appeals to substitution. Though standard, we present its definition because it contributes to the process of refining semantics, especially to deriving Explicit MetaML. In case (4), $FV(t)$ denotes the free variables of the term t. The environmental abstract machine that we are heading towards should correctly manage names without appealing to alpha-equivalence. We make the same design choice here.

[1]The first author's thesis [7] has proved the correctness of the reference semantics with respect to the substitutional big-step semantics in [24].

$$\boxed{\longrightarrow^i \subseteq \textsc{Term}^i \times \textsc{Term}^i}$$

$$\frac{t_1^{i+1} \longrightarrow^{i+1} t_2^{i+1}}{\lambda x.t_1^{i+1} \longrightarrow^{i+1} \lambda x.t_2^{i+1}} \text{ (lambda-(i+1))}$$

$$\frac{t_{11}^i \longrightarrow^i t_{12}^i}{t_{11}^i\ t_2^i \longrightarrow^i t_{12}^i\ t_2^i} \text{ (appL-i)} \qquad \frac{t_{21}^i \longrightarrow^i t_{22}^i}{v_1^i\ t_{21}^i \longrightarrow^i v_1^i\ t_{22}^i} \text{ (appR-i)}$$

$$\frac{}{(\lambda x.t^0)\ v^0 \longrightarrow^0 t^0[v^0/x]} \text{ (app-0)} \qquad \frac{}{!\langle v^1\rangle \longrightarrow^0 v^1} \text{ (run-0)}$$

$$\frac{t_1^i \longrightarrow^i t_2^i}{!t_1^i \longrightarrow^i !t_2^i} \text{ (run-i)} \qquad \frac{t_1^{i+1} \longrightarrow^{i+1} t_2^{i+1}}{\langle t_1^{i+1}\rangle \longrightarrow^i \langle t_2^{i+1}\rangle} \text{ (code-i)}$$

$$\frac{}{\sim\langle v^1\rangle \longrightarrow^1 v^1} \text{ (splice-1)} \qquad \frac{t_1^i \longrightarrow^i t_2^i}{\sim t_1^i \longrightarrow^{i+1} \sim t_2^i} \text{ (splice-(i+1))}$$

$$\frac{t_{11}^i \longrightarrow^i t_{12}^i}{t_{11}^i + t_2^i \longrightarrow^i t_{12}^i + t_2^i} \text{ (plusL-i)}$$

$$\frac{t_{21}^i \longrightarrow^i t_{22}^i}{v_1^i + t_{21}^i \longrightarrow^i v_1^i + t_{22}^i} \text{ (plusR-i)} \qquad \frac{n = n_1 + n_2}{n_1 + n_2 \longrightarrow^0 n} \text{ (plus-0)}$$

$$\boxed{\cdot[\cdot/\cdot] \subseteq (\textsc{Term} \times \textsc{Denotable} \times \textsc{Var}) \times \textsc{Term}}$$

$$\begin{array}{rcll}
x[w/x] & = & w & (1) \\
x_1[w/x_2] & = & x_1 \text{ where } x_1 \not\equiv x_2 & (2) \\
(t_1\ t_2)[w/x] & = & (t_1[w/x])\ (t_2[w/x]) & (3) \\
(\lambda x_1.t_0)[w/x_2] & = & \lambda x_3.t_0[x_3/x_1][w/x_2] & \\
 & & \text{where } x_3 \notin FV(\lambda x_1.t_0) & \\
 & & \cup FV(w) \cup \{x_2\} & (4) \\
\langle t_0\rangle[w/x] & = & \langle t_0[w/x]\rangle & (5) \\
(!t_0)[w/x] & = & !t_0[w/x] & (6) \\
(\sim t_0)[w/x] & = & \sim t_0[w/x] & (7) \\
n[w/x] & = & n & (8) \\
(t_1 + t_2)[w/x] & = & (t_1[w/x]) + (t_2[w/x]) & (9)
\end{array}$$

Figure 1. Substitutional Structural Operational Semantics of MetaML

We define an evaluator that takes a program as its input and provides an answer as its output. Programs Prgm are closed level-0 terms. Answers Ans are observable results of evaluation.

$$eval_{\text{Sub}} : \textsc{Prgm} \rightharpoonup \textsc{Ans}$$

$$eval_{\text{Sub}}(t) = \begin{cases} \texttt{func} & \text{if } t \longrightarrow^{0*} \lambda x.t'^0 \\ \texttt{code} & \text{if } t \longrightarrow^{0*} \langle v^1\rangle \\ n & \text{if } t \longrightarrow^{0*} n \end{cases}$$

Let $\longrightarrow^{i*}$ denote the reflexive-transitive closure of $\longrightarrow^i$.

Example 2.1. To illustrate evaluation, consider the classic puzzle program $!\langle \lambda y.\sim((\lambda x.\langle x\rangle)\ (\lambda x.\langle y\rangle))\ 0\rangle\ 5$ from [25]. Evaluating it yields code because:

$$\begin{array}{lll}
 & !\langle \lambda y.\sim(\underline{(\lambda x.\langle x\rangle)\ (\lambda x.\langle y\rangle))}\ 0\rangle\ 5 & \text{(app-0)} \\
\longrightarrow^0 & !\langle \lambda y.\underline{\sim\langle \lambda x.\langle y\rangle\rangle}\ 0\rangle\ 5 & \text{(splice-1)} \\
\longrightarrow^0 & \underline{!\langle \lambda y.(\lambda x.\langle y\rangle)\ 0\rangle}\ 5 & \text{(run-0)} \\
\longrightarrow^0 & (\lambda y.\underline{(\lambda x.\langle y\rangle)\ 0})\ 5 & \text{(app-0)} \\
\longrightarrow^0 & \underline{(\lambda y.\langle y\rangle)\ 5} & \text{(app-0)} \\
\longrightarrow^0 & \langle 5\rangle &
\end{array}$$

where the redex of each step is underlined, accompanied by the name of the reduction rule on the right.

3 Explicit MetaML

Refining substitutional semantics to environmental semantics is challenging for multi-stage languages. Substitutional semantics model the process of performing substitutions implicitly at the meta-language level. For example, the (app-0) rule in Figure 1 reduces an application $(\lambda x.t^0)\ v^0$ in one step to the *result* of replacing each free occurrence of the variable x in the term t^0 by the value v, denoted by $t^0[v^0/x]$. In contrast, environmental semantics keep track of variable bindings and replace variable references on demand. Unlike single-stage languages such as ISWIM [11, 12], MetaML programs manipulate open code yet still respect lexical scoping, causing complication in modelling variable bindings through environments.

To bridge the gap between substitutional and environmental semantics, we leverage the concept of explicit substitutions [1, 3, 15], which internalises how substitutions percolate through a term at the semantical level. As a result, the (app-0) rule becomes

$$\frac{}{(\lambda x.t^0)\ v^0 \longrightarrow^0 t^0[x := v^0]} \text{ (app-0)}$$

where $[x := v^0]$ denotes an explicit substitution. An explicit substitution $[x := w]$ may take several steps to percolate through a term t, depending on how complex the term t is. We call the resulting dialect Explicit MetaML.

3.1 Syntax

We treat MetaML terms as *source terms* in Explicit MetaML. We define *runtime terms*, values and denotable terms for Explicit MetaML as follows:

$$\begin{array}{lcl}
t^i \in \textsc{RTerm}^i & v^i \in \textsc{Value}^i & w \in \textsc{Denot} \\
t^0 & := & \ldots \mid \underline{\lambda} x.t^0 \mid t^0[x := w] \\
t^{i+1} & := & \ldots \mid \underline{\lambda} x.t^0 \mid t^{i+1}[x := w] \\
v^0 & := & \ldots \mid \cancel{\lambda x.t^0}\ \underline{\lambda} x.t^0 \\
v^{i+1} & := & \ldots \mid \underline{\lambda} x.t^0 \\
w & := & \ldots
\end{array}$$

We use "…" to represent the same expressions as in MetaML's syntax, and "$\cancel{e_{\text{Sub}}}\ e_{\text{Exp}}$" to denote that the expression e_{Sub} in MetaML's syntax is replaced by the expression e_{Exp} in Explicit MetaML.

Like MetaML, Explicit MetaML ensures that a value at one level must remain a value at any higher level. To preserve this property in the presence of explicit substitutions, we distinguish between level-0 source lambda abstractions $\lambda x.t^0$ and level-0 evaluated lambda abstractions $\underline{\lambda} x.t^0$. We demonstrate the significance of this distinction after presenting the semantics. For brevity, we call $\lambda x.v^{i+1}$ a level-$(i+1)$ lambda value and call $\underline{\lambda} x.t^0$ an underlined lambda abstraction or a level-0 lambda value.

$$\boxed{\longrightarrow^{i} \subseteq \text{RTerm}^i \times \text{RTerm}^i}$$

$$\cdots \quad \frac{}{\lambda x.t^0 \longrightarrow^0 \underline{\lambda} x.t^0}\ \text{(lambda-0)}$$

$$\frac{}{(\underline{\lambda} x.t^0)\ v^0 \longrightarrow^0 t^0[x := v^0]}\ \text{(app-0)} \qquad \frac{t_1^i \longrightarrow^{xi} t_2^i}{t_1^i \longrightarrow^{i} t_2^i}\ \text{(inj-subst)}$$

$$\boxed{\longrightarrow^{xi} \subseteq \text{RTerm}^i \times \text{RTerm}^i}$$

$$\frac{}{x[x := w] \longrightarrow^{xi} w}\ \text{(var-eq-subst)}$$

$$\frac{}{x_1[x_2 := w] \longrightarrow^{xi} x_1}\ \text{where } x_1 \not\equiv x_2\ \text{(var-df-subst)}$$

$$\frac{}{n[x := w] \longrightarrow^{xi} n}\ \text{(num-subst)}$$

$$\frac{}{(t_1^i + t_2^i)[x := w] \longrightarrow^{xi} (t_1^i[x := w]) + (t_2^i[x := w])}\ \text{(plus-subset)}$$

$$\frac{}{(t_1^i\ t_2^i)[x := w] \longrightarrow^{xi} (t_1^i[x := w])\ (t_2^i[x := w])}\ \text{(app-subst)}$$

$$\frac{x_3 \notin FV(\lambda x_1.t^i) \cup FV(w) \cup \{x_2\}}{(\lambda x_1.t^i)[x_2 := w] \longrightarrow^{xi} \lambda x_3.t^i[x_1 := x_3][x_2 := w]}\ \text{(lam-subst)}$$

$$\frac{x_3 \notin FV(\underline{\lambda} x_1.t^0) \cup FV(w) \cup \{x_2\}}{(\underline{\lambda} x_1.t^0)[x_2 := w] \longrightarrow^{xi} \underline{\lambda} x_3.t^0[x_1 := x_3][x_2 := w]}\ \text{(lamu-subst)}$$

$$\frac{}{\langle t^{i+1}\rangle[x := w] \longrightarrow^{xi} \langle t^{i+1}[x := w]\rangle}\ \text{(code-subst)}$$

$$\frac{}{(!t^i)[x := w] \longrightarrow^{xi}\ !t^i[x := w]}\ \text{(run-subst)}$$

$$\frac{}{(\sim t^i)[x := w] \longrightarrow^{x(i+1)} \sim t^i[x := w]}\ \text{(splice-subst)}$$

$$\frac{t_1^i[x_1 := w_1] \longrightarrow^{xi} t_2^i}{t_1^i[x_1 := w_1][x_2 := w_2] \longrightarrow^{xi} t_2^i[x_2 := w_2]}\ \text{(subst-subst)}$$

Figure 2. Structural Operational Semantics of Explicit MetaML

3.2 Structural Operational Semantics

Figure 2 presents the semantics of Explicit MetaML. We use $\cdots$ to represent the same rules as in the previous semantics.

Most transition rules are the same as MetaML. The (lambda-0) rule introduces level-0 values. There is no equivalent of the (lambda-(i+1)) rule for underlined lambda abstractions because an underlined lambda abstraction is a value at any level. The (app-0) rule replaces the meta-language substitution $[v^0/x]$ with the explicit substitution $[x := v^0]$. The (inj-subst) rule says every substitution step counts as a computation step.

The substitution transition relations $\longrightarrow^{xi}$ describe how explicit substitutions interact with other terms in the language. Each rule except (lamu-subst) and (subst-subst) corresponds to an equation of MetaML's implicit substitution $\cdot[\cdot/\cdot]$ from Figure 1. The (subst-subst) rule implies that only a substitution transition may happen underneath an explicit substitution. The (lamu-subst) rule accommodates the newly devised underlined lambda abstractions and is analogous to the (lam-subst) rule.

Example 3.1. To see the need for underlined lambda abstractions, consider the program $(\lambda x_1.\langle x_1\rangle)\ ((\lambda x_2.(\lambda x_3.x_2))\ w)$ where $x_2 \not\equiv x_3$.

If we had not introduced the underlined lambda abstraction, its reduction would be:

$$\begin{aligned}
& (\lambda x_1.\langle x_1\rangle)\ ((\lambda x_2.(\lambda x_3.x_2))\ w) \\
\longrightarrow^{0*}\ & (\lambda x_1.\langle x_1\rangle)\ \boxed{\lambda x_4.x_2[x_3 := x_4][x_2 := w]} \\
& \text{where } x_4 \notin FV(\lambda x_3.x_2) \cup FV(w) \cup \{x_2\} \\
\longrightarrow^{0}\ & \langle x_1\rangle[x_1 := (\lambda x_4.x_2[x_3 := x_4][x_2 := w])] \\
\longrightarrow^{0*}\ & \langle \boxed{\lambda x_4.x_2[x_3 := x_4][x_2 := w]} \rangle \\
\longrightarrow^{0*}\ & \langle \lambda x_4.w\rangle
\end{aligned}$$

At first, the boxed lambda abstraction is a level-0 value. However, after substituting x_1, the boxed lambda abstraction reappears at level 1. We can apply the (lambda-(i+1)) rule to evaluate the body of the lambda abstraction at level 1. This violates the property that a value at one level must remain a value at any higher level, and eventually leads to an incorrect evaluation result. To avoid this, we make each level-0 lambda abstraction single-step to its underlined counterpart, which is a value at all levels. Thus by Explicit MetaML, we have:

$$\begin{aligned}
& (\lambda x_1.\langle x_1\rangle)\ ((\lambda x_2.(\lambda x_3.x_2))\ w) \\
\longrightarrow^{0*}\ & (\underline{\lambda} x_1.\langle x_1\rangle)\ \boxed{\underline{\lambda} x_4.x_2[x_3 := x_4][x_2 := w]} \\
& \text{where } x_4 \notin FV(\lambda x_3.x_2) \cup FV(w) \cup \{x_2\} \\
\longrightarrow^{0}\ & \langle x_1\rangle[x_1 := (\underline{\lambda} x_4.x_2[x_3 := x_4][x_2 := w])] \\
\longrightarrow^{0*}\ & \langle \boxed{\underline{\lambda} x_4.x_2[x_3 := x_4][x_2 := w]} \rangle
\end{aligned}$$

We define an evaluator in terms of Explicit MetaML:

$$eval_{\text{Exp}} \ :\ \text{Prgm} \rightharpoonup \text{Ans}$$
$$eval_{\text{Exp}}(t) = \begin{cases} \texttt{func} & \text{if } t \longrightarrow^{0*} \underline{\lambda} x.t'^0 \\ \cdots \end{cases}$$

This evaluator is equivalent to the one defined in terms of substitutional structural operational semantics of MetaML.

Proposition 3.2. $eval_{\text{Sub}} = eval_{\text{Exp}}$.

Proof. See Section 6.3 for proof strategies for this proposition and the succeeding four propositions. □

4 Suspended MetaML

To more closely resemble the behaviour of environmental semantics, we first reconsider how to represent functions. We refine what forms of lambda abstractions are functions as opposed to code. Conventional environmental semantics represent functions as closures. As a comparison, Explicit MetaML represents functions as underlined lambda abstractions. Motivated by the concept of closures [11], we delay explicit substitutions that surround any level-0 lambda value until the lambda abstraction is applied.

We then reconsider how to represent applications. We refine how variable bindings are updated by applications. In conventional environmental semantics, when a closure is applied, the environment of the closure is extended with a new variable binding. Any previous binding for the variable is overridden. For semantics with explicit substitutions, to perform an application in an environmental manner, we promote the substitution for the lambda bound variable to the front and override any existing explicit substitution for that variable. That is, a level-0 application $(\underline{\lambda} x.t^0)\overline{[x_i := w_i]}\ v^0$ makes a single-step transition to $t^0[x := v^0]\overline{[x_i := w_i]}$. We call the resulting dialect Suspended MetaML.

4.1 Syntax

Starting from the syntax of Explicit MetaML, we define runtime terms, values and denotable terms for Suspended MetaML as follows:

$$
\begin{array}{lll}
\multicolumn{3}{c}{t^i \in \textsc{RTerm}^i \quad v^i \in \textsc{Value}^i \quad w \in \textsc{Denot}} \\
t^0 & := & \ldots \mid \xcancel{t^0[x := w]}\ t^0\overline{[x := w]} \\
t^{i+1} & := & \ldots \mid \xcancel{t^{i+1}[x := w]}\ t^{i+1}\overline{[x := w]} \mid \hat{\lambda} x.t^{i+1} \\
v^i & := & \ldots \mid \xcancel{\underline{\lambda} x.t^0}\ (\underline{\lambda} x.t^0)\overline{[x := w]} \\
w & := & \ldots
\end{array}
$$

In Explicit MetaML, a level-0 lambda value $\underline{\lambda} x.t^0$ is a value at any level. Since Suspended MetaML no longer pushes explicit substitutions into an underlined lambda abstraction, a level-0 lambda value surrounded by explicit substitutions, $(\underline{\lambda} x.t^0)\overline{[x := w]}$, is a value at any level. Another significant difference in Suspended MetaML is the introduction of the hatted lambda abstraction $\hat{\lambda} x.t^{i+1}$, which accommodates changes in semantics to perform applications in an environmental manner. We explain its necessity in more detail after presenting the semantics.

4.2 Structural Operational Semantics

Figure 3 presents the semantics of Suspended MetaML. The (app-0) rule is refined from the following rule:

$$
\frac{x_0 \notin FV(\underline{\lambda} x.t^0) \cup \bigcup_i (FV(w_i) \cup \{x_i\})}{(\underline{\lambda} x.t^0)\overline{[x_i := w_i]}\ v^0 \longrightarrow^0 t^0[x := x_0]\overline{[x_i := w_i]}[x_0 := v^0]}
$$

The above rule is semantically correct. However, we want to eliminate renaming in any level-0 application because conventional environmental semantics do not rename when performing an application.

Observe that if $FV(v^0) \cap (\bigcup_i \{x_i\}) = \emptyset$, we can promote the substitution $[x_0 := v^0]$ to the front of the explicit substitutions $\overline{[x_i := w_i]}$. Then we can eliminate renaming the lambda bound variable x by combining two explicit substitutions $[x := x_0]$ and $[x_0 := v^0]$ to one $[x := v^0]$. We can prove that the condition $FV(v^0) \cap (\bigcup_i \{x_i\}) = \emptyset$ is always satisfied when the (app-0) rule applies during evaluation of a program. This fact is discussed and justified in Section 6.2.

$$
\boxed{\longrightarrow^i \subseteq \textsc{RTerm}^i \times \textsc{RTerm}^i}
$$

$$
\cdots \quad \frac{}{(\lambda x.t^0)\overline{[x_j := w_j]} \longrightarrow^0 (\underline{\lambda} x.t^0)\overline{[x_j := w_j]}}\ \text{(lambda-0)}
$$

$$
\frac{t^{i+1} \notin \textsc{Value}^{i+1} \quad x_N \text{ is fresh}}{\lambda x.t^{i+1} \longrightarrow^{i+1} \hat{\lambda} x_N.t^{i+1}[x := x_N]}\ \text{(lambda-(i+1)-t)}
$$

$$
\frac{}{\hat{\lambda} x.v^{i+1} \longrightarrow^{i+1} \lambda x.v^{i+1}}\ \text{(lambda-(i+1)-v)}
$$

$$
\frac{t_1^{i+1} \longrightarrow^{i+1} t_2^{i+1}}{\hat{\lambda} x.t_1^{i+1} \longrightarrow^{i+1} \hat{\lambda} x.t_2^{i+1}}\ \text{(lambda-(i+1)-r)}
$$

$$
\frac{}{(\underline{\lambda} x.t^0)\overline{[x_i := w_i]}\ v^0 \longrightarrow^0 t^0[x := v^0]\overline{[x_i := w_i]}}\ \text{(app-0)}
$$

$$
\boxed{\longrightarrow^{\mathrm{x}i} \subseteq \textsc{RTerm}^i \times \textsc{RTerm}^i}
$$

$$
\cdots \quad \text{No (lamu-subst)}
$$

$$
\frac{x_N \text{ is fresh}}{\begin{array}{l}(\lambda x_1.t^{i+1})[x_2 := w] \longrightarrow^{\mathrm{x}(i+1)} \\ \qquad \lambda x_N.t^{i+1}[x_1 := x_N][x_2 := w]\end{array}}\ \text{(lam-subst)}
$$

Figure 3. Structural Operational Semantics of Suspended MetaML

Since we want to ultimately develop an environmental operational semantics for MetaML, the (app-0) rule promotes the substitution for the underlined lambda bound variable to the front, superseding any existing explicit substitution for that variable. This behaviour resembles the updating of an environment in a conventional environmental semantics.

Suspended MetaML replaces Explicit MetaML's (lambda-(i+1)) rule with three (lambda-(i+1)) rules. To evaluate a level-$(i + 1)$ lambda abstraction, if its body is not a level-$(i + 1)$ value, we first apply the (lambda-(i+1)-t) rule to rename the lambda bound variable to a fresh variable and replace λ by $\hat{\lambda}$ to indicate such a renaming is in process. To prevent unnecessary renaming, it is important to check whether the lambda abstraction is already a value before renaming the lambda bound variable. Checking this is a deep syntactic operation in Suspended MetaML but becomes a shallow check in our abstract machine. Then we repeatedly apply the (lambda-(i+1)-r) rule to reduce its body to a value. Finally we apply the (lambda-(i+1)-v) rule to change $\hat{\lambda}$ back to λ. Suspended MetaML forces this renaming to make the (app-0) rule sound with respect to Explicit MetaML.

The (lam-subst) rule only concerns levels higher than 0 because the level-0 case is covered by the (lambda-0) rule. There is no (lamu-subst) rule because the concerned term is a value at any level.

The freshness condition "x_N is fresh" means that the variable x_N does not appear in the current term being evaluated nor in the surrounding scope. A rigorous explanation of the freshness condition is presented in Section 6.1.

Example 4.1. To see the need for hatted lambda abstractions, consider the program $!\langle \lambda y.{\sim}(((\lambda y.(\lambda x.x))\ n)\ \langle y \rangle)\rangle$.

If we had not introduced the hatted lambda abstraction, its transitions would be:

$$\begin{array}{ll}
 & !\langle \lambda y.{\sim}(((\lambda y.(\lambda x.x))\ n)\ \langle y\rangle)\rangle \\
\longrightarrow^{0*} & !\langle \lambda y.{\sim}(\boxed{(\underline{\lambda} x.x)[y := n]\ \langle y\rangle})\rangle \\
\longrightarrow^{0} & !\langle \lambda y.{\sim}(\boxed{x[x := \langle y\rangle][y := n]})\rangle \\
\longrightarrow^{0*} & !\langle \lambda y.{\sim}\langle n\rangle\rangle \\
\longrightarrow^{0*} & \lambda y.n
\end{array}$$

The boxes highlight a critical mistake during evaluation. For the boxed application $(\underline{\lambda} x.x)[y := n]\ \langle y\rangle$, the condition $FV(\langle y\rangle) \cap \{y\} = \emptyset$ is not satisfied. The free variable of the operand $\langle y\rangle$, i.e., the variable y, clashes with an existing substitution $[y := n]$ on the abstraction. Notice that the (locally) free variable y is bound by a lambda in the surrounding context. Generally, for a level-0 application, the operand can have free variables only as a result of performing an evaluation under lambdas at levels higher than 0. To avoid such variable clashes, whenever we go under a lambda during evaluation at a level higher than 0, we need to rename the lambda bound variable to a fresh variable:

$$\begin{array}{ll}
 & !\langle \lambda y.{\sim}(((\lambda y.(\lambda x.x))\ n)\ \langle y\rangle)\rangle \\
\longrightarrow^{0} & !\langle \hat{\lambda} z.({\sim}(((\lambda y.(\lambda x.x))\ n)\ \langle y\rangle))[y := z]\rangle \\
 & \text{where } z \notin \{x, y\} \\
\longrightarrow^{0*} & !\langle \hat{\lambda} z.{\sim}(\boxed{(\underline{\lambda} x.x)[y := n][y := z]\ \langle z\rangle})\rangle \\
\longrightarrow^{0} & !\langle \hat{\lambda} z.{\sim}(x[x := \langle z\rangle][y := n][y := z])\rangle \\
\longrightarrow^{0*} & !\langle \hat{\lambda} z.{\sim}\langle z\rangle\rangle \\
\longrightarrow^{0*} & \underline{\lambda} z.z
\end{array}$$

For this boxed application, the condition $FV(\langle z\rangle) \cap \{y\} = \emptyset$ is satisfied.

We define an evaluator in terms of Suspended MetaML:

$$\begin{array}{l}
eval_{\text{Sus}} \ :\ \textsc{Prgm} \rightharpoonup \textsc{Ans} \\
eval_{\text{Sus}}(t) = \begin{cases} \mathtt{func} & \text{if } t \longrightarrow^{0*} (\underline{\lambda} x.t'^{0})\overline{[x_i := w_i]} \\ \ldots \end{cases}
\end{array}$$

This evaluator is equivalent to the one defined using the substitutional structural operational semantics of MetaML.

Proposition 4.2. $eval_{\text{Sub}} = eval_{\text{Sus}}$.

5 Environmental MetaML

Suspended MetaML is peculiar in the sense that the top-level structure of a lambda value surrounded by zero or more explicit substitutions is not immediately recognisable. For example, the term $(\underline{\lambda} x.t^0)\overline{[x_i := w_i]}$ truly represents $((\ldots(((\underline{\lambda} x.t^0)[x_1 := w_1])[x_2 := w_2])\ldots)[x_i := w_i])\ldots$. To ensure that its top-level structure is a lambda value, we must dive down through the cascaded explicit substitutions to search for a lambda abstraction.

For a term surrounded by explicit substitutions, $t\overline{[x_i := w_i]}$, if every denotable term w_i is closed, it is a natural step to replace the explicit substitutions with a corresponding environment. However, MetaML allows substituting an open term for a variable. The term of some inner explicit substitution in a cascade may have free variables that are bound by outer substitutions. For example, in Suspended MetaML, we have

$$\begin{array}{ll}
 & !\langle \lambda y.({\sim}((\lambda x.\langle x\rangle)\ (\lambda z.y))\ 0)\rangle\ 5 \\
\longrightarrow^{0*} & \langle \boxed{y[z := 0][y := u][y := u][u := 5]} \rangle
\end{array}$$

where $u \notin \{x, y, z\}$. In the boxed term, the variable y is bound to the variable u, which in turn is bound to 5. Can we represent the explicit substitutions that surround a term by an environment? The answer is negative unfortunately. If the boxed term was represented by pairing the variable y with the environment $\rho = \{(z, 0), (y, u), (u, 5)\}$, looking up the variable y in the environment ρ would return the variable u solely, which would no longer be bound to 5.

We systematically replace cascaded explicit substitutions with meta-environments instead of environments. A meta-environment[2] is a finite sequence of environments, among which the free variables of one environment are bound by the next environment. Evaluating a variable with a meta-environment returns the lookup result of the variable in the first environment paired with the remaining environments of the meta-environment. For example, the boxed term is represented by pairing the variable y with the meta-environment $(\rho_1; \rho_2)$ where $\rho_1 = \{(z, 0), (y, u)\}$ and $\rho_2 = \{(u, 5)\}$. One small-step of transition returns the variable u paired with the meta-environment ρ_2 because $\rho_1(y) = u$. Another step returns 5 because $\rho_2(u) = 5$. We call the resulting dialect Environmental MetaML.

5.1 Syntax

Starting from the syntax of Suspended MetaML, we define runtime terms, values, denotable terms and configurations for Environmental MetaML as follows.

$$t^i \in \textsc{RTerm}^i \quad v^i \in \textsc{Value}^i \quad w \in \textsc{Denot}$$

$$c^i \in \textsc{Conf}^i \quad \rho \in \textsc{Env} = \textsc{Var} \overset{\text{fin}}{\longrightarrow} \textsc{Denot}$$

$$\begin{array}{lcl}
t^0 & := & \ldots \mid \cancel{\underline{\lambda} x.t^0} \mid \cancel{t^0\overline{[x := w]}} \mid \llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis \\
t^{i+1} & := & \ldots \mid \cancel{\underline{\lambda} x.t^0} \mid \cancel{t^{i+1}\overline{[x := w]}} \mid \cancel{\hat{\lambda} x.t^{i+1}} \mid \llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis \\
v^i & := & \ldots \mid \cancel{(\underline{\lambda} x.t^0)\overline{[x := w]}}\ \llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis \\
w & := & \ldots \\
c^0 & := & v^0 \mid c^0\ c^0 \mid \lambda x.c^0 \mid \LEFTcircle\ t^0,\ \rho^*\ \RIGHTcircle \\
 & & \mid \langle c^1\rangle \mid !c^0 \mid c^0 + c^0 \\
c^{i+1} & := & v^{i+1} \mid c^{i+1}\ c^{i+1} \mid \lambda x.c^{i+1} \mid \LEFTcircle\ t^{i+1},\ \rho^*\ \RIGHTcircle \\
 & & \mid \langle c^{i+2}\rangle \mid {\sim}c^i \mid !c^{i+1} \mid c^{i+1} + c^{i+1}
\end{array}$$

An environment $\rho \in \textsc{Env}$ is a total function from the set of variables to the set of denotable terms with the restriction that only a finite number of variables do not map to themselves. An environment is denoted by listing all non-identity

[2]The concept of meta-environments is analogous to meta-continuations [4] in the context of delimited control.

mappings $\{\overline{(x_i, w_i)}\}$. The identity environment is denoted by $\emptyset$.

A meta-environment $\rho^* \in \textsc{Env}^*$ is a finite sequence of environments. An empty meta-environment is denoted by ϵ.

A term paired with a meta-environment, $\text{◖}\ t^i,\ \rho^*\ \text{◗}$ or $\llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis$, is called a *closure*. A level-0 lambda abstraction paired with a meta-environment, $\llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis$, is called a *closure value*. A closure that is not a value, $\text{◖}\ t^i,\ \rho^*\ \text{◗}$, is called a *non-value closure*. A closure makes its top-level structure immediately evident. For example, it is immediately recognisable that the top-level structure of the closure $\llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis$ is a level-0 lambda abstraction $\lambda x.t^0$ without having to dive into a cascade of explicit substitutions.

In a conventional single-stage programming language, for a closure consisting of a term and an environment, the term must be closed by the environment. Variable bindings in closures are more complicated in MetaML because an environment may bind a variable to an open term. As for a closure $\text{◖}\ t,\ (\rho_1; \rho_2; \ldots)\ \text{◗}$, the free variables of the term t are bound by the first environment ρ_1, the free variables of the closure $\text{◖}\ t,\ \rho_1\ \text{◗}$ must be bound by the second environment ρ_2 and so on. In the end, the closure $\text{◖}\ t,\ (\rho_1; \rho_2; \ldots)\ \text{◗}$ may even still have free variables, which are bound by its surrounding context.

The design choice of closures and closure values resembles the original interpreter of MetaML [25]. To ensure that any variable that has been eliminated by some substitution or renaming does not escape from the scope of the substitution or renaming, the interpreter uses a delayed environment called a cover. A cover works like a normal environment on non-function terms. If a cover encounters a function, the substitutions of the cover are delayed and are only performed on the result of calling the function. Analogously, in Environmental MetaML, environments on a level-0 lambda abstraction are delayed, as modelled by closure values. These environments work like normal environments on the result of applying the level-0 lambda abstraction.

$$\boxed{\longrightarrow^i \subseteq \textsc{Conf}^i \times \textsc{Conf}^i}$$

$$\frac{c_1^{i+1} \longrightarrow^{i+1} c_2^{i+1}}{\lambda x.c_1^{i+1} \longrightarrow^{i+1} \lambda x.c_2^{i+1}}\ \text{(lambda-(i+1))} \qquad \frac{c_{11}^i \longrightarrow^i c_{12}^i}{c_{11}^i\ c_2^i \longrightarrow^i c_{12}^i\ c_2^i}\ \text{(appL-i)}$$

$$\frac{c_{21}^i \longrightarrow^i c_{22}^i}{v_1^i\ c_{21}^i \longrightarrow^i v_1^i\ c_{22}^i}\ \text{(appR-i)}$$

$$\frac{}{\llparenthesis\ \lambda x.t^0,\ (\rho;\rho^*)\ \rrparenthesis\ v^0 \longrightarrow^0 \text{◖}\ t^0,\ (\rho[x \mapsto v^0];\rho^*)\ \text{◗}}\ \text{(app-0)}$$

$$\frac{c_1^i \longrightarrow^i c_2^i}{!c_1^i \longrightarrow^i\ !c_2^i}\ \text{(run-i)} \qquad \frac{}{!\langle v^1 \rangle \longrightarrow^0 \text{◖}\ v^1,\ (\emptyset;\epsilon)\ \text{◗}}\ \text{(run-0)}$$

$$\frac{c_1^{i+1} \longrightarrow^{i+1} c_2^{i+1}}{\langle c_1^{i+1} \rangle \longrightarrow^i \langle c_2^{i+1} \rangle}\ \text{(code-i)} \qquad \frac{c_1^i \longrightarrow^i c_2^i}{\sim c_1^i \longrightarrow^{i+1} \sim c_2^i}\ \text{(splice-(i+1))}$$

$$\frac{}{\sim\langle v^1 \rangle \longrightarrow^1 v^1}\ \text{(splice-1)} \qquad \frac{c_{11}^i \longrightarrow^i c_{12}^i}{c_{11}^i + c_2^i \longrightarrow^i c_{12}^i + c_2^i}\ \text{(plusL-i)}$$

$$\frac{c_{21}^i \longrightarrow^i c_{22}^i}{v_1^i + c_{21}^i \longrightarrow^i v_1^i + c_{22}^i}\ \text{(plusR-i)} \qquad \frac{n = n_1 + n_2}{n_1 + n_2 \longrightarrow^0 n}\ \text{(plus-0)}$$

$$\frac{}{\text{◖}\ \lambda x.t^0,\ \rho^*\ \text{◗} \longrightarrow^0 \llparenthesis\ \lambda x.t^0,\ \rho^*\ \rrparenthesis}\ \text{(lam-0-env)}$$

$$\frac{x_N \text{ is fresh}}{\text{◖}\ \lambda x.t^{i+1},\ (\rho;\rho^*)\ \text{◗} \longrightarrow^{i+1} \lambda x_N.\text{◖}\ t^{i+1},\ (\rho[x \mapsto x_N];\rho^*)\ \text{◗}}\ \text{(lam-(i+1)-env)}$$

$$\frac{}{\text{◖}\ \llparenthesis\ \lambda x.t,\ \rho_1^*\ \rrparenthesis,\ \rho_2^*\ \text{◗} \longrightarrow^i \llparenthesis\ \lambda x.t,\ (\rho_1^*;\rho_2^*)\ \rrparenthesis}\ \text{(clov-env)}$$

$$\frac{}{\text{◖}\ w,\ \epsilon\ \text{◗} \longrightarrow^i w}\ \text{(den-env)}$$

$$\frac{}{\text{◖}\ x,\ (\rho;\rho^*)\ \text{◗} \longrightarrow^i \text{◖}\ \rho(x),\ \rho^*\ \text{◗}}\ \text{(var-env)}$$

$$\frac{}{\text{◖}\ n,\ \rho^*\ \text{◗} \longrightarrow^i n}\ \text{(num-env)}$$

$$\frac{}{\text{◖}\ t_1\ t_2,\ \rho^*\ \text{◗} \longrightarrow^i \text{◖}\ t_1,\ \rho^*\ \text{◗}\text{◖}\ t_2,\ \rho^*\ \text{◗}}\ \text{(app-env)}$$

$$\frac{}{\text{◖}\ \langle t^{i+1} \rangle,\ \rho^*\ \text{◗} \longrightarrow^i \langle\text{◖}\ t^{i+1},\ \rho^*\ \text{◗}\rangle}\ \text{(code-env)}$$

$$\frac{}{\text{◖}\ !t^i,\ \rho^*\ \text{◗} \longrightarrow^i !\text{◖}\ t^i,\ \rho^*\ \text{◗}}\ \text{(run-env)}$$

$$\frac{}{\text{◖}\ \sim t^i,\ \rho^*\ \text{◗} \longrightarrow^{i+1} \sim\text{◖}\ t^i,\ \rho^*\ \text{◗}}\ \text{(splice-env)}$$

$$\frac{}{\text{◖}\ t_1 + t_2,\ \rho^*\ \text{◗} \longrightarrow^i \text{◖}\ t_1,\ \rho^*\ \text{◗} + \text{◖}\ t_2,\ \rho^*\ \text{◗}}\ \text{(plus-env)}$$

Figure 4. Environmental MetaML: Structural Operational Semantics

5.2 Structural Operational Semantics

Figure 4 presents the structural operational semantics of Environmental MetaML. Unlike our previous dialects of MetaML, Environmental MetaML reduces configurations rather than (runtime) terms. The (app-0) rule models how to perform an application as updating the meta-environment, which completes the unfinished job of its counterpart in Suspended MetaML. The (run-0) rule executes a code value at level 0. The value v^1 must be paired with the initial meta-environment $(\emptyset;\epsilon)$ in order to make transitions at level 0. The (-env) rules specify how to evaluate a closure. The (lam-0-env) rule turns a closure into a closure value. The (lam-(i+1)-env) rule combines Suspended MetaML's (lambda-(i+1)-t) and (lam-subst) rules. To evaluate a lambda abstraction at a level higher than 0, before diving into its body, we must rename the lambda bound variable to a fresh variable that does not occur in the current configuration nor in its surrounding scope. The (clov-env) rule concatenates two meta-environments. The (den-env) rule discards the empty meta-environment. The other (-env) rules correspond to the reduction rules of Suspended MetaML's substitution transition relations. Unlike Explicit MetaML and Suspended MetaML, Environmental MetaML does not separate closure transitions from other transitions because making transitions at the term position of a closure is disallowed.

We define an evaluator in terms of the structural operational semantics of Environmental MetaML. To evaluate a program t, we first pair it with the initial meta-environment $(\emptyset;\epsilon)$ to make it a configuration, and then pass it to the semantics:

$$eval_{\text{EnvSOS}} \;:\; \textsc{Prgm} \rightharpoonup \textsc{Ans}$$

$$eval_{\text{EnvSOS}}(t) = \begin{cases} \texttt{func} & \text{if } init_c(t) \longrightarrow^{0*} ⦇\, \lambda x.t'^{0},\ \rho^* \,⦈ \\ \texttt{code} & \text{if } init_c(t) \longrightarrow^{0*} \langle v^1 \rangle \\ n & \text{if } init_c(t) \longrightarrow^{0*} n \end{cases}$$

where $init_c(t) = \text{◖}\, t,\ (\emptyset;\epsilon)\, \text{◗}$. This evaluator is equivalent to the one defined using substitutional structural operational semantics of MetaML.

Proposition 5.1. $eval_{\text{Sub}} = eval_{\text{EnvSOS}}$.

5.3 Reduction Semantics

Since a reduction semantics can be viewed as a concise representation of an abstract machine, we first develop a reduction semantics for Environmental MetaML.

We define evaluation contexts to regulate where reduction may happen. An evaluation context $E^{i\multimap j}$ has an inner level i and an outer level j. The inner level i specifies the level of configurations that can fill the hole of the context. The outer level j is the level of the configuration produced by the context when the hole is filled.

The hole $\square^{i\multimap i}$ evaluation context can be filled by and will produce a level-i configuration. $E^{i\multimap j}[c^i]$ is a level-j configuration constructed by filling the hole of the evaluation context $E^{i\multimap j}$ with a level-i configuration c^i. The levels i and j in an evaluation context $E^{i\multimap j}$ can be related in any of the following three ways: (1) $i > j$. For example, a level-0 configuration $!\langle \lambda x.x \rangle$ can be represented as $(\square[!\square][\langle\square\rangle][\lambda x.\square])^{1\multimap 0}[x]$. (2) $i = j$. For example, a level-0 term $⦇\,(\lambda x.x),\ \ (\emptyset;\epsilon)\,⦈\ \ 7$ can be represented as $(\square[⦇\,(\lambda x.x),\ \ (\emptyset;\epsilon)\,⦈\ \square])^{0\multimap 0}[7]$. (3) $i < j$. For example, a level-1 term $\sim(⦇\,(\lambda x.x),\ (\emptyset;\epsilon)\,⦈\ \langle \lambda y.y \rangle)$ can be represented as $(\square[\sim\square])^{0\multimap 1}[⦇\,(\lambda x.x),\ (\emptyset;\epsilon)\,⦈\ \langle \lambda y.y\rangle]$.

The definition of evaluation contexts is motivated by the structural rules of the structural operational semantics shown in Figure 4. For example, the (code-i) rule says that a code operation at level i can be evaluated by reducing its operand at level $i+1$. Since an evaluation context determines where reduction may happen, we may replace the (code-i) rule by an evaluation context $(\square[\langle\square\rangle])^{(i+1)\multimap i}$ and allow any level-$(i+1)$ reduction to happen at the hole of the context.

Figure 5 presents the reduction semantics. Each notion of reduction $\mathcal{R}^i$ corresponds to one reduction rule of the structural operational semantics shown in Figure 4. Only a few are presented due to space restrictions. The reduction relations $\longmapsto^i$ respect performing any notion of reduction $\mathcal{R}^j$ in an evaluation context $E^{j\multimap i}$.

We define an evaluator in terms of this reduction semantics, which we call $eval_{\text{EnvRed}}$. Its definition is analogous to that of the evaluator $eval_{\text{EnvSOS}}$. In fact, these two evaluators are equivalent.

$$\boxed{\mathcal{R}^i \ \subseteq\ \textsc{Conf}^i \times \textsc{Conf}^i}$$

$$\vdots$$

$$\text{◖}\, \lambda x.t^0,\ \rho^* \,\text{◗} \quad \mathcal{R}^0 \quad ⦇\, \lambda x.t^0,\ \rho^* \,⦈ \qquad \text{(conf-lam-0)}$$

$$\text{◖}\, \lambda x.t^{i+1},\ (\rho;\rho^*) \,\text{◗} \quad \mathcal{R}^{i+1} \quad \lambda x_N.\text{◖}\, t^{i+1},\ (\rho[x \mapsto x_N];\rho^*) \,\text{◗} \quad \text{where } x_N \text{ is fresh} \qquad \text{(conf-lam-(i+1))}$$

$$\text{◖}\, ⦇\, \lambda x.t,\ \rho_1^* \,⦈,\ \rho_2^* \,\text{◗} \quad \mathcal{R}^i \quad ⦇\, \lambda x.t,\ (\rho_1^*;\rho_2^*) \,⦈ \qquad \text{(conf-clov-i)}$$

$$\boxed{\longmapsto^i \ \subseteq\ \textsc{Conf}^i \times \textsc{Conf}^i}$$

$$\frac{t_1^j \ \mathcal{R}^j\ t_2^j}{E^{j\multimap i}[t_1^j] \longmapsto^i E^{j\multimap i}[t_2^j]}$$

Figure 5. Environmental MetaML: Reduction Semantics

Proposition 5.2. $eval_{\text{EnvSOS}} = eval_{\text{EnvRed}}$.

5.4 Abstract Machine Semantics (MCEK Machine)

An abstract machine can be viewed as a concrete representation of the reduction semantics as it encodes a systematic strategy to break a configuration into an evaluation context and a redex. We develop an abstract machine semantics for Environmental MetaML based on its reduction semantics.

An abstract machine models computation through state transitions. A machine state comprises a level, a continuation (i.e., an evaluation context) and a control string (i.e., a configuration):

$$\begin{array}{l} S \in \textsc{State},\ c^i \in \textsc{Conf}^i,\ v^i \in \textsc{Value}^i,\ E^{i\multimap j} \in \textsc{Ecxt}^{i\multimap j} \\ S \ \ ::=\ \ \langle i,\ E^{i\multimap 0},\ c^i\rangle_{\text{r}} \mid \langle i,\ E^{i\multimap 0},\ c^i\rangle_{\text{f}} \\ \qquad\quad \mid \langle i,\ E^{i\multimap 0},\ v^i\rangle_{\text{b}} \mid v^0 \end{array}$$

The machine operates in four modes. Value mode v^0 represents that the level-0 value v^0 is the result of executing the machine. Reduce mode $\langle i,\ E^{i\multimap 0},\ c^i\rangle_{\text{r}}$ signifies that a proper notion of reduction can be applied to the redex c^i. Focus mode $\langle i,\ E^{i\multimap 0},\ c^i\rangle_{\text{f}}$ indicates searching downward into the configuration c^i for a redex to reduce. Build mode $\langle i,\ E^{i\multimap 0},\ v^i\rangle_{\text{b}}$ returns the value v^i to the current evaluation context $E^{i\multimap 0}$. A machine state $\langle i,\ E^{i\multimap 0},\ c^i\rangle$ unloads to the configuration $E^{i\multimap 0}[c^i]$ regardless of its mode. We lay out the abstract machine semantics of Environmental MetaML in Figure 6. We call it the MCEK machine, where M stands for multi-stage, C stands for control, E stands for environment, and K stands for continuation.

We define an evaluator in terms of the MCEK machine. To evaluate a program t, we first build a level-0 machine state in focus mode comprising the empty context $\square$, the program t and the initial meta-environment $(\emptyset;\epsilon)$, and then pass it to

$$\boxed{\longmapsto_{\mathrm{mcek}} \subseteq \textsc{State} \times \textsc{State}}$$

Reduce rules: $\langle i,\ E^{i\multimap 0},\ c^i\rangle_{\mathrm{r}}$

$$\begin{array}{rcll}
\langle 0,\ E,\ \llparenthesis\, \lambda x.t^0,\ (\rho;\rho^*)\,\rrparenthesis\ v^0\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ ◖\, t^0,\ (\rho[x\mapsto v^0];\rho^*)\,◗\rangle_{\mathrm{f}} & \text{(r-app-0)}\\
\langle 0,\ E,\ !\langle v^1\rangle\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ ◖\, v^1,\ (\emptyset;\epsilon)\,◗\rangle_{\mathrm{f}} & \text{(r-run-0)}\\
\langle 1,\ E,\ {\sim}\langle v^1\rangle\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle 1,\ E,\ v^1\rangle_{\mathrm{f}} & \text{(r-splice-1)}\\
\langle 0,\ E,\ n_1+n_2\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ n\rangle_{\mathrm{f}} \text{ where } n = n_1+n_2 & \text{(r-plus-0)}\\
\langle 0,\ E,\ ◖\, \lambda x.t^0,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ \llparenthesis\, \lambda x.t^0,\ \rho^*\,\rrparenthesis\rangle_{\mathrm{f}} & \text{(r-conf-lam-0)}\\
\langle i+1,\ E,\ ◖\, \lambda x.t^{i+1},\ (\rho;\rho^*)\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ \lambda x_N.◖\, t^{i+1},\ (\rho[x\mapsto x_N];\rho^*)\,◗\rangle_{\mathrm{f}} & \\
 & & \text{where } x_N \text{ is fresh} & \text{(r-conf-lam-(i+1))}\\
\langle i,\ E,\ ◖\,\llparenthesis\, \lambda x.t,\ \rho_1^*\,\rrparenthesis,\ \rho_2^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ \llparenthesis\, \lambda x.t,\ (\rho_1^*;\rho_2^*)\,\rrparenthesis\rangle_{\mathrm{f}} & \text{(r-conf-clov-i)}\\
\langle i,\ E,\ ◖\, w,\ \epsilon\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ w\rangle_{\mathrm{f}} & \text{(r-conf-den-i)}\\
\langle i,\ E,\ ◖\, x,\ (\rho;\rho^*)\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ ◖\, \rho(x),\ \rho^*\,◗\rangle_{\mathrm{f}} & \text{(r-conf-var-i)}\\
\langle i,\ E,\ ◖\, n,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ n\rangle_{\mathrm{f}} & \text{(r-conf-num-i)}\\
\langle i,\ E,\ ◖\, t_1\ t_2,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ ◖\, t_1,\ \rho^*\,◗\,◖\, t_2,\ \rho^*\,◗\rangle_{\mathrm{f}} & \text{(r-conf-app-i)}\\
\langle i,\ E,\ ◖\, \langle t^{i+1}\rangle,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ \langle ◖\, t^{i+1},\ \rho^*\,◗\rangle\rangle_{\mathrm{f}} & \text{(r-conf-code-i)}\\
\langle i,\ E,\ ◖\, !t^i,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ !◖\, t^i,\ \rho^*\,◗\rangle_{\mathrm{f}} & \text{(r-conf-run-i)}\\
\langle i+1,\ E,\ ◖\, {\sim}t^i,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ {\sim}◖\, t^i,\ \rho^*\,◗\rangle_{\mathrm{f}} & \text{(r-conf-splice-(i+1))}\\
\langle i,\ E,\ ◖\, t_1+t_2,\ \rho^*\,◗\rangle_{\mathrm{r}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ ◖\, t_1,\ \rho^*\,◗ + ◖\, t_2,\ \rho^*\,◗\rangle_{\mathrm{f}} & \text{(r-conf-plus-i)}
\end{array}$$

Focus rules: $\langle i,\ E^{i\multimap 0},\ c^i\rangle_{\mathrm{f}}$

$$\begin{array}{rcll}
\langle i,\ E,\ ◖\, t,\ \rho^*\,◗\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ ◖\, t,\ \rho^*\,◗\rangle_{\mathrm{r}} & \text{(f-conf-i)}\\
\langle i+1,\ E,\ x\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ x\rangle_{\mathrm{b}} & \text{(f-var-(i+1))}\\
\langle i,\ E,\ c_1\ c_2\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[\square\ c_2],\ c_1\rangle_{\mathrm{f}} & \text{(f-appL-i)}\\
\langle i,\ E,\ \llparenthesis\, \lambda x.t,\ \rho^*\,\rrparenthesis\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ \llparenthesis\, \lambda x.t,\ \rho^*\,\rrparenthesis\rangle_{\mathrm{b}} & \text{(f-lambda-0)}\\
\langle i+1,\ E,\ \lambda x.c\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E[\lambda x.\square],\ c\rangle_{\mathrm{f}} & \text{(f-lambda-(i+1))}\\
\langle i,\ E,\ \langle c\rangle\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E[\langle\square\rangle],\ c\rangle_{\mathrm{f}} & \text{(f-code-i)}\\
\langle i+1,\ E,\ {\sim}c\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[{\sim}\square],\ c\rangle_{\mathrm{f}} & \text{(f-splice-(i+1))}\\
\langle i,\ E,\ !c\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[!\square],\ c\rangle_{\mathrm{f}} & \text{(f-run-i)}\\
\langle i,\ E,\ n\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ n\rangle_{\mathrm{b}} & \text{(f-num-i)}\\
\langle i,\ E,\ c_1+c_2\rangle_{\mathrm{f}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[\square+c_2],\ c_1\rangle_{\mathrm{f}} & \text{(f-plusL-i)}
\end{array}$$

Build rules: $\langle i,\ E^{i\multimap 0},\ v^i\rangle_{\mathrm{b}}$

$$\begin{array}{rcll}
\langle 0,\ \square,\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & v & \text{(b-value-0)}\\
\langle i,\ E[\square\ c_2],\ v_1\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[v_1\ \square],\ c_2\rangle_{\mathrm{f}} & \text{(b-appL-i)}\\
\langle 0,\ E[v_1\ \square],\ v_2\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ v_1\ v_2\rangle_{\mathrm{r}} & \text{(b-appR-0)}\\
\langle i+1,\ E[v_1\ \square],\ v_2\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ v_1\ v_2\rangle_{\mathrm{b}} & \text{(b-appR-(i+1))}\\
\langle i+1,\ E[\lambda x.\square],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ \lambda x.v\rangle_{\mathrm{b}} & \text{(b-lambda-(i+1))}\\
\langle i+1,\ E[\langle\square\rangle],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E,\ \langle v\rangle\rangle_{\mathrm{b}} & \text{(b-code-(i+1))}\\
\langle 0,\ E[{\sim}\square],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle 1,\ E,\ {\sim}v\rangle_{\mathrm{r}} & \text{(b-splice-0)}\\
\langle i+1,\ E[{\sim}\square],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i+2,\ E,\ {\sim}v\rangle_{\mathrm{b}} & \text{(b-splice-(i+1))}\\
\langle 0,\ E[!\square],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ !v\rangle_{\mathrm{r}} & \text{(b-run-0)}\\
\langle i+1,\ E[!\square],\ v\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ !v\rangle_{\mathrm{b}} & \text{(b-run-(i+1))}\\
\langle i,\ E[\square+c_2],\ v_1\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i,\ E[v_1+\square],\ c_2\rangle_{\mathrm{f}} & \text{(b-plusL-i)}\\
\langle 0,\ E[v_1+\square],\ v_2\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle 0,\ E,\ v_1+v_2\rangle_{\mathrm{r}} & \text{(b-plusR-0)}\\
\langle i+1,\ E[v_1+\square],\ v_2\rangle_{\mathrm{b}} & \longmapsto_{\mathrm{mcek}} & \langle i+1,\ E,\ v_1+v_2\rangle_{\mathrm{b}} & \text{(b-plusR-(i+1))}
\end{array}$$

Figure 6. Environmental MetaML: Abstract Machine Semantics

the MCEK machine:

$$eval_{\mathrm{MCEK}}\ :\ \textsc{Prgm} \rightharpoonup \textsc{Ans}$$

$$eval_{\mathrm{MCEK}}(t) = \begin{cases} \mathtt{func} & \text{if } init_{\mathrm{s}}(t) \longmapsto^*_{\mathrm{mcek}} \llparenthesis\, \lambda x.t'^{0},\ \rho^*\,\rrparenthesis \\ \mathtt{code} & \text{if } init_{\mathrm{s}}(t) \longmapsto^*_{\mathrm{mcek}} \langle v^1\rangle \\ n & \text{if } init_{\mathrm{s}}(t) \longmapsto^*_{\mathrm{mcek}} n \end{cases}$$

where $init_{\mathrm{s}}(t) = \langle 0,\ \square,\ ◖\, t,\ (\emptyset;\epsilon)\,◗\rangle_{\mathrm{f}}$. This evaluator is equivalent to the one defined using Environmental MetaML's reduction semantics.

Proposition 5.3. $eval_{\mathrm{EnvRed}} = eval_{\mathrm{MCEK}}$.

The MCEK machine is equivalent to our reference semantics for MetaML.

Theorem 5.4. $eval_{\mathrm{Sub}} = eval_{\mathrm{MCEK}}$.

Proof. By Propositions 5.1, 5.2 and 5.3. □

The MCEK machine has more machine state transitions than strictly necessary. It is structured so that the each transition is determined by considering only one component of the machine state. Felleisen et al. [6] simplified abstract machines by (1) letting the machine exploit information from both the control strings and the evaluation contexts, and (2) fusing determined transitions. Adopting the same approach, we can simplify our MCEK machine analogously. For example, rules (b-appR-0) and (r-app-0) can be merged into one rule:

$$\langle 0,\ E[v_1\ \square],\ v_2\rangle_{\mathrm{b}} \longmapsto_{\mathrm{mcek}} \langle 0,\ E,\ ◖\ t^0,\ (\rho[x \mapsto v^0]; \rho^*)\ ◗\rangle_{\mathrm{f}}.$$

The MCEK machine is compatible with Van Horn and Might's framework [26]. We can augment the MCEK machine with a store and direct any structure of unbounded size through the store. After several transformations, the machine can be abstracted by restricting its address space and allocation strategy. With this primary point of abstraction, we get a sound and decidable control flow analysis, where the precision of the analysis is determined by the structure of the addresses used by the allocator.

6 Correctness

To evince the correctness of our approach, we explain how to rigorously model fresh variables, uncover a critical global invariant on variable bindings, and leverage three proof strategies to establish the equivalences of semantics.

6.1 Fresh Variables

In Suspended MetaML and Environmental MetaML, we informally used the freshness condition "x_N is fresh" to mean that the variable x_N has not occurred in the current term (or configuration) being evaluated nor in its surrounding scope. The freshness condition is intuitive to understand, but we must introduce additional machinery to model it rigorously. Doing so supports our correctness proofs.

6.1.1 Suspended MetaML

To track all variables in the term being evaluated and in its surrounding scope, we add a set of *active variables*, i.e., variables that are not fresh, to the transition relations $\longrightarrow^i$ and $\longrightarrow^{\mathrm{x}i}$ of Suspended MetaML. For all rules, the active variable component is preserved between the premise and the conclusion. Furthermore, we change the (lambda-(i+1)-t) rule:

$$\frac{t^{i+1} \notin \textsc{Value}^{i+1} \quad \cancel{x_N \text{ is fresh}}\ \boxed{x_N \notin \mathcal{X}}}{\boxed{\mathcal{X} \vdash}\ \lambda x.t^{i+1} \longrightarrow^{i+1} \hat{\lambda} x_N.t^{i+1}[x := x_N]}\ \text{(lambda-(i+1)-t)}$$

We make the same change to the (lam-subst) rule.

To apply the semantics to a program, the active variable set $\mathcal{X}$ is initialised to contain all variables of the program. A multi-step transition must update the active variable set $\mathcal{X}$ for each underlying single-step:

$$\frac{\textsc{Var}(t^i) \subseteq \mathcal{X}}{\mathcal{X} \vdash t^i \longrightarrow^{i*} t^i}\ \text{(zero)}$$

$$\frac{\mathcal{X} \vdash t_1^i \longrightarrow^i t_2^i \quad \textsc{Var}(t_1^i) \subseteq \mathcal{X} \\ \mathcal{X} \cup \textsc{Var}(t_2^i) \vdash t_2^i \longrightarrow^{i*} t_3^i}{\mathcal{X} \vdash t_1^i \longrightarrow^{i*} t_3^i}\ \text{(more)}$$

6.1.2 Environmental MetaML

We make the same changes as above to the structural operational semantics of Environmental MetaML.

For the reduction semantics, we first change every notion of reduction $c_1^i\ \mathcal{R}^i\ c_2^i$ by adding a component for active variables:

$$\boxed{\mathcal{X} \vdash}\ c_1^i\ \mathcal{R}\ c_2^i$$

We refine the freshness condition "x_N is fresh" of the (conf-lam-(i+1)) rule to "$x_N \notin \mathcal{X}$". Next we add an active variable component to the reduction relations $\longmapsto^i$:

$$\frac{\boxed{\mathcal{X} \vdash}\ c_1^j\ \mathcal{R}^j\ c_2^j}{\boxed{\mathcal{X} \vdash}\ E^{j\multimap i}[c_1^j] \longmapsto^i E^{j\multimap i}[c_2^j]}$$

where the active variable set $\mathcal{X}$ contains the variables that appear in the configuration $E^{j\multimap i}[c_1^j]$ or its surrounding scope.

An MCEK machine state contains all information needed to check the freshness of a variable: the current configuration and its surrounding scope are the control string and the continuation respectively. For the (r-conf-lam-(i+1)) rule, the freshness condition "x_N is fresh" is refined to "$x_N \notin \textsc{Var}(E[◖\ \lambda x.t^{i+1},\ (\rho; \rho^*)\ ◗])$" where the configuration $◖\ \lambda x.t^{i+1},\ (\rho; \rho^*)\ ◗$ is the control string and the evaluation context E is the continuation.

6.2 Well-boundness

MetaML sometimes binds variables to open terms. We uncover a global invariant on variable bindings, which we call *well-boundness*, that justifies reordering explicit substitutions in Suspended MetaML to perform applications in an environmental manner.

6.2.1 Suspended MetaML

For any subterm t_{11}^i of a program t_1^0 during evaluation, we want to ensure that all its free variables are bound in its surrounding scope. Suppose $t_{11}^i \longrightarrow^i t_{12}^i$. The (subst-subst) and (lambda-(i+1)-r) rules preserve the invariant that the free variables of t_1^i must be bound by either explicit substitutions or by hatted lambda bound variables in the surrounding scope. To keep track of the variables bound by these two means, we augment the transition relations $\longrightarrow^i$ and $\longrightarrow^{\mathrm{x}i}$ with two more components $\mathcal{U}$ and $\mathcal{V}$. The new relations have the form $\mathcal{U}; \mathcal{V} \vdash t \longrightarrow^i t'$ and $\mathcal{U}; \mathcal{V} \vdash t \longrightarrow^{\mathrm{x}i} t'$. The variable set $\mathcal{U}$ tracks free variables of the term t (and the term t' as well by a property) that are bound in the surrounding scope by explicit substitutions or hatted lambda bound variables. The variable set $\mathcal{V}$ tracks those bound by hatted lambdas due to their special interest to the well-boundness judgement that is introduced soon.

Almost all rules merely propagate $\mathcal{U}$ and $\mathcal{V}$. For the (lambda-(i+1)-r) rule, we have:

$$\frac{\boxed{\mathcal{U} \cup \{x\}; \mathcal{V} \cup \{x\} \vdash}\ t_1^{i+1} \longrightarrow^{i+1} t_2^{i+1}}{\boxed{\mathcal{U}; \mathcal{V} \vdash}\ \hat{\lambda} x.t_1^{i+1} \longrightarrow^{i+1} \hat{\lambda} x.t_2^{i+1}}\ \text{(lambda-(i+1)-r)}$$

$$\boxed{\vdash \; wb \;\subseteq\; \mathcal{P}(\textsc{Var}) \times \mathcal{P}(\textsc{Var}) \times \textsc{RTerm}}$$

$$\frac{}{\mathcal{U};\mathcal{V} \vdash x\ wb}\ \text{where } x \in \mathcal{U}$$

$$\frac{\mathcal{U};\mathcal{V} \vdash t_1\ wb \quad \mathcal{U};\mathcal{V} \vdash t_2\ wb}{\mathcal{U};\mathcal{V} \vdash t_1\ t_2\ wb}$$

$$\frac{\mathcal{U} \cup \{x\};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash \lambda x.t\ wb}\ \text{where } x \notin \mathcal{V}$$

$$\frac{\mathcal{U} \cup \{x\};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash \underline{\lambda} x.t\ wb}\ \text{where } x \notin \mathcal{V}$$

$$\frac{\mathcal{U} \cup \{x\};\mathcal{V} \cup \{x\} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash \hat{\lambda} x.t\ wb}\ \text{where } x \notin \mathcal{V}$$

$$\frac{\mathcal{U};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash \langle t \rangle\ wb} \qquad \frac{\mathcal{U};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash \sim t\ wb} \qquad \frac{\mathcal{U};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash !t\ wb}$$

$$\frac{}{\mathcal{U};\mathcal{V} \vdash n\ wb}$$

$$\frac{\mathcal{U};\mathcal{V} \vdash t_1\ wb \quad \mathcal{U};\mathcal{V} \vdash t_2\ wb}{\mathcal{U};\mathcal{V} \vdash t_1 + t_2\ wb}$$

$$\frac{\mathcal{U};\mathcal{V} \vdash w\ wb \quad \mathcal{U} \cup \{x\};\mathcal{V} \vdash t\ wb}{\mathcal{U};\mathcal{V} \vdash t[x := w]\ wb}\ \text{where } x \notin \mathcal{V}$$

Figure 7. Well-boundness Judgement for Suspended MetaML

That is, when diving into the body of a hatted lambda abstraction $\hat{\lambda} x.t_1^{i+1}$, we update the variable set $\mathcal{U}$ to record that any free appearance of the variable x in the term t_1^{i+1} must be bound by its surrounding scope, and in particular, the variable set $\mathcal{V}$, to record that such a free variable must be bound by a hatted lambda bound variable.

For the (subst-subst) rule, we have:

$$\frac{\boxed{\mathcal{U} \cup \{x_2\};\mathcal{V} \vdash}\ t_1^i[x_1 := w_1] \longrightarrow^{xi} t_2^i}{\boxed{\mathcal{U};\mathcal{V} \vdash}\ t_1^i[x_1 := w_1][x_2 := w_2] \longrightarrow^{xi} t_2^i[x_2 := w_2]}\ \text{(subst-subst)}$$

That is, when making a substitution transition underneath the outermost explicit substitution $[x_2 := w_2]$, we update the variable set $\mathcal{U}$ to record that any free appearance of the variable x_2 in $t_1^i[x_1 := w_1]$ must be bound by its surrounding scope.

Well-boundness Judgement and Properties. Using the well-boundness judgement in Figure 7, we formalise the global invariant that the free variables of any subterm of a program during evaluation must be bound by either explicit substitutions or by hatted lambda bound variables in the surrounding scope.

Suppose t_{11} is a subterm of a program t_1^0 during evaluation. We immediately have $\emptyset;\emptyset \vdash t_1^0\ wb$. Observe that a sub-derivation of $\emptyset;\emptyset \vdash t_1^0\ wb$ must be the derivation of $\mathcal{U};\mathcal{V} \vdash t_{11}\ wb$ for some variable sets $\mathcal{U}$ and $\mathcal{V}$. The variable set $\mathcal{U}$ tracks all free variables of the subterm t_{11} that are bound in the surrounding scope. We observe the following property that gives an upper bound on the free variables of a well bound term.

Proposition 6.1. *If $\mathcal{U};\mathcal{V} \vdash t\ wb$, then $FV(t) \subseteq \mathcal{U}$.*

The variable set $\mathcal{V}$ in $\mathcal{U};\mathcal{V} \vdash t_{11}\ wb$ tracks the free variables of the subterm t_{11} that are bound by hatted lambdas in the surrounding scope. Suppose t_{11} is an application $(\underline{\lambda} x.t^0)\overline{[x_i := w_i]}\ v^0$ that is reducible by the (app-0) rule. Since the (app-0) rule is not a substitution transition, it cannot be applied under any explicit substitution. Thus the free variables of v^0 must be bound by hatted lambdas in its surrounding context, which are tracked by the variable set $\mathcal{V}$ of the judgement $\mathcal{U};\mathcal{V} \vdash (\underline{\lambda} x.t^0)\overline{[x_i := w_i]}\ v^0\ wb$. By the definition of the well-boundness judgement, we have $x_i \notin \mathcal{V}$. Hence the condition $FV(v^0) \cap (\bigcup_i \{x_i\}) = \emptyset$ is satisfied, which guarantees the correctness of the (app-0) rule.

The well-boundness judgement cooperates well with the multi-step transition relations $\longrightarrow^{i*}$. The second property says the former is preserved by the latter.

Proposition 6.2. *If $\mathcal{U};\mathcal{V} \vdash t_1^i\ wb$, $\textsc{Var}(t_1^i) \subseteq \mathcal{X}$, $\mathcal{V} \subseteq \mathcal{U} \subseteq \mathcal{X}$ and $\mathcal{U};\mathcal{V};\mathcal{X} \vdash t_1^i \longrightarrow^{i*} t_2^i$, then $\mathcal{U};\mathcal{V} \vdash t_2^i\ wb$.*

As a corollary of the above properties, multi-step transitioning preserves the closedness of runtime terms.

6.2.2 Environmental MetaML

Consider the structural operational semantics of Environmental MetaML. As in Suspended MetaML, we track lambda bound variables when diving into the body of a higher level non-value lambda. In contrast to Suspended MetaML, we never make transitions underneath an environment. Thus we only need to track lambda bound variables in the surrounding scope. The new relation has the form $\mathcal{V} \vdash c \longrightarrow^i c'$. It says the free variables of the configuration c (and the configuration c' as well by a property) must be bound in its surrounding scope by lambda bound variables, which are tracked by the variable set $\mathcal{V}$.

Well-boundness Judgement and Properties. The augmented transition rules and the well-boundness judgement for Environmental MetaML are analogous to those for Suspended MetaML. We omit their definition for lack of space. Interested readers may consult the first author's thesis [7].

For the well-boundness judgement $\mathcal{U};\mathcal{V} \vdash c\ wb$, the variable set $\mathcal{U}$ tracks all free variables of the configuration c that are bound in the surrounding scope, and in particular, the variable set $\mathcal{V}$ tracks those bound by higher level non-value lambdas.

Proposition 6.3. *If $\mathcal{U};\mathcal{V} \vdash c\ wb$, then $FV(c) \subseteq \mathcal{U}$.*

Proposition 6.4. *If $\mathcal{V};\mathcal{V} \vdash c_1^i\ wb$, $\textsc{Var}(c_1^i) \subseteq \mathcal{X}$, $\mathcal{V} \subseteq \mathcal{X}$ and $\mathcal{V};\mathcal{X} \vdash c_1^i \longrightarrow^{i*} c_2^i$, then $\mathcal{V};\mathcal{V} \vdash c_2^i\ wb$.*

6.3 Proof Strategies

Our equivalence proofs can be categorised as (1) equating two structural operational semantics, (2) equating a structural operational semantics and a reduction semantics, and (3) equating a reduction semantics and an abstract machine semantics. In the following exposition, for the sake of simplicity, we pretend small-step transitions are defined on terms rather than configurations.

Our strategy for equating two structural operational semantics is motivated by the bisimulation proof method [14, 16]. Our strategy for equating a reduction semantics and an abstract machine semantics is extracted from the proof of the equivalence of the CC machine and a substitutional reduction semantics for ISWIM in [6].

Equating Two Structural Operational Semantics. To prove the equivalence of a structural operational semantics for language $\mathcal{A}$ (defined by the transition relation $\longrightarrow_{\mathcal{A}}$) and a structural operational semantics for language $\mathcal{B}$ (defined by the transition relation $\longrightarrow_{\mathcal{B}}$), construct a bisimulation relation $\simeq$ between their terms that has the following properties.

1. For any legal program, its initialisations in two languages are related, i.e., $\forall p \in \textsc{Prgm}$, $init_{\mathcal{A}}(p) \simeq init_{\mathcal{B}}(p)$.
2. Related values are observationally indistinguishable, i.e., if $v_{\mathcal{A}} \simeq v_{\mathcal{B}}$, then $obs_{\mathcal{A}}(v_{\mathcal{A}}) = obs_{\mathcal{B}}(v_{\mathcal{B}})$.
3. Canonisation:
   (1) If $v_{\mathcal{A}} \simeq t_{\mathcal{B}}$, then $t_{\mathcal{B}} \longrightarrow^*_{\mathcal{B}} v_{\mathcal{B}}$ and $v_{\mathcal{A}} \simeq v_{\mathcal{B}}$.
   (2) If $t_{\mathcal{A}} \simeq v_{\mathcal{B}}$, then $t_{\mathcal{A}} \longrightarrow^*_{\mathcal{A}} v_{\mathcal{A}}$ and $v_{\mathcal{A}} \simeq v_{\mathcal{B}}$.
4. Weak Bisimulation:
   (1) If $t_{\mathcal{A}} \simeq t_{\mathcal{B}}$ and $t_{\mathcal{A}} \longrightarrow_{\mathcal{A}} t'_{\mathcal{A}}$, then $t_{\mathcal{B}} \longrightarrow^*_{\mathcal{B}} t'_{\mathcal{B}}$ and $t'_{\mathcal{A}} \simeq t'_{\mathcal{B}}$.
   (2) If $t_{\mathcal{A}} \simeq t_{\mathcal{B}}$ and $t_{\mathcal{B}} \longrightarrow_{\mathcal{B}} t'_{\mathcal{B}}$, then $t_{\mathcal{A}} \longrightarrow^*_{\mathcal{A}} t'_{\mathcal{A}}$ and $t'_{\mathcal{A}} \simeq t'_{\mathcal{B}}$.

Equating Structural Operational Semantics and Reduction Semantics. To prove the equivalence of a structural operational semantics (defined by the transition relation $\longrightarrow$) and a reduction semantics (defined by the reduction relation $\longmapsto$) for the same language, prove the following two propositions.

1. If $t_1 \longrightarrow t_2$, then $t_1 \longmapsto t_2$.
   Lemma: If $t_1 \longrightarrow t_2$, then $E[t_1] \longmapsto E[t_2]$.
2. If $t_1 \longmapsto t_2$, then $t_1 \longrightarrow t_2$.
   Lemma: If $t_1 \longrightarrow t_2$, then $E[t_1] \longrightarrow E[t_2]$.

Equating Reduction Semantics and Abstract Machine Semantics. To prove the equivalence of a reduction semantics (defined by the reduction relation $\longmapsto$) and an abstract machine (defined by the state transition relation $\longmapsto_{\text{abs}}$) of the same language, first define a translator $\mathcal{T}$ to unload machine states to (runtime) terms. Then prove the following two propositions.

1. If $E_0[t_0] = E_1[t_1]$ and $E_1[t_1] \longmapsto E_1[t_2]$ where $t_1 \mathrel{\mathcal{R}} t_2$, then $\langle E_0,\ t_0\rangle_{\text{f}} \longmapsto^*_{\text{abs}} \langle E_0,\ t_0\rangle_{\text{f}}$.
   Lemma: If $t = E_1[t_1]$ and $t_1 \mathrel{\mathcal{R}} t_2$, then $\langle E,\ t\rangle_{\text{f}} \longmapsto^*_{\text{abs}} \langle EE_1,\ t_1\rangle_{\text{f}}$.
2. If $S_1 \longmapsto_{\text{abs}} S_2$, then $\mathcal{T}(S_1) \longmapsto^* \mathcal{T}(S_2)$.

7 Related Work

A variety of operational semantics [7, 9, 10, 13, 19, 21, 23–25] have been developed for variants of MetaML or its extensions, each with some goal in mind. Taha et al. [23] modelled a core subset of MetaML and their call-by-value (CBV) environmental big-step semantics is the first well-known implementation semantics for MetaML. Taha et al. [24] presented a more concise CBV substitutional big-step semantics, which is less implementation-oriented and more suitable for reasoning, but did not establish its equivalence to the existing implementation semantics. We started from this substitutional big-step semantics and made a firm connection to an environmental semantics by stepwise refinement. To justify the optimisation of some MetaML implementation, Taha et al. [19] presented a call-by-name (CBN) substitutional reduction semantics, and established its equivalence to a CBN substitutional big-step semantics. They addressed the known challenges for establishing the equivalence of the CBV substitutional big-step and reduction semantics, but they did not do the equivalence proof. Ge [7] proved their equivalence on the way to deriving a substitutional abstract machine, the MK machine. To soundly add effects, delimited control in particular, to a two-stage language, Kameyama et al. [9] studied how to translate the staging away. Their source language λ^{α}_{1v} is a two-stage restricted variant of the multi-stage language λ^{α} [22]. They translated the source language λ^{α}_{1v} (with a CBV substitutional reduction semantics and without effects) to an unstaged target language, System F, where they used tuples to represent environments in the future stage. Work on deriving abstract machines from interpreters [2] and abstracting interpreters [5] may be applicable to multi-stage programming, but each requires a correct environment-passing interpreter.

8 Conclusion

"You can eat an elephant one bite at a time." Through several intermediate semantics, we systematically refined our reference semantics for MetaML to an environmental abstract machine, the MCEK machine, and proved their equivalence. The MCEK machine shall shed new light on static analysis of multi-stage programming. Particularly, to design sound and decidable static analyses for MetaML, we plan to apply Van Horn and Might's framework [26] to the MCEK machine. That "good theory leads to good tools" [18] is our belief.

Acknowledgments

We would like to thank Felipe Bañados Schwerter, Joshua Dunfield, Ivan Beschastnikh, Jonatan Milewski, Zheng Dong, Minchen Li, and the anonymous referees for their feedback.

References

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1990. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 31–46. https://doi.org/10.1145/96709.96712

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP '03)*. ACM, New York, NY, USA, 8–19. https://doi.org/10.1145/888251.888254

[3] Pierre-Louis Curien. 1985. Categorical combinatory logic. In *Automata, Languages and Programming: 12th Colloquium, ICALP '85*, Wilfried Brauer (Ed.). Lecture Notes in Computer Science, Vol. 194. Springer, Berlin, Heidelberg, 130–139. https://doi.org/10.1007/BFb0015738

[4] Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

[5] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110256

[6] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex* (1st ed.). The MIT Press.

[7] Rui Ge. 2016. *Refining Semantics for Multi-stage Programming*. Master's thesis. University of British Columbia. https://doi.org/10.14288/1.0319338

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[9] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2008. Closing the stage: from staged code to typed closures. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '08)*. ACM, New York, NY, USA, 147–157. https://doi.org/10.1145/1328408.1328430

[10] Yukiyoshi Kameyama, Oleg Kiselyov, and Chung-chieh Shan. 2009. Shifting the stage: staging with delimited control. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '09)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1480945.1480962

[11] Peter J. Landin. 1964. The mechanical evaluation of expressions. *Comput. J.* 6, 4 (1964), 308–320.

[12] Peter J. Landin. 1966. The next 700 programming languages. *Commun. ACM* 9, 3 (1966), 157–166.

[13] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. 1999. An idealized MetaML: simpler, and more expressive. In *Programming Languages and Systems: 8th European Symposium on Programming, ESOP '99*, S. Doaitse Swierstra (Ed.). Lecture Notes in Computer Science, Vol. 1576. Springer, Berlin, Heidelberg, 193–207. https://doi.org/10.1007/3-540-49099-X_13

[14] Damien Pous and Davide Sangiorgi. 2012. Enhancements of the bisimulation proof method. In *Advanced Topics in Bisimulation and Coinduction*, Davide Sangiorgi and Jan Rutten (Eds.). Cambridge Tracts in Theoretical Computer Science, Vol. 52. Cambridge University Press, New York, NY, USA, Chapter 6, 233–289.

[15] Kristoffer Høgsbro Rose. 1996. *Explicit Substitution: Tutorial & Survey*. Technical Report. BRICS, Department of Computer Science, University of Aarhus.

[16] Davide Sangiorgi. 2011. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, New York, NY, USA.

[17] Tim Sheard. 1999. Using MetaML: A staged programming language. In *Advanced Functional Programming: Third International School, AFP'98*, S. Doaitse Swierstra, José N. Oliveira, and Pedro R. Henriques (Eds.). Lecture Notes in Computer Science, Vol. 1608. Springer, Berlin, Heidelberg, 207–239. https://doi.org/10.1007/10704973_5

[18] Tim Sheard. 2001. Accomplishments and research challenges in meta-programming. In *Semantics, Applications, and Implementation of Program Generation: Second International Workshop, SAIG 2001 Proceedings*, Walid Taha (Ed.). Lecture Notes in Computer Science, Vol. 2196. Springer, Berlin, Heidelberg, 2–44. https://doi.org/10.1007/3-540-44806-3_2

[19] Walid Taha. 1999. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trival. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM '00)*. ACM, New York, NY, USA, 34–43. https://doi.org/10.1145/328690.328697

[20] Walid Taha. 2004. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation: International Seminar*, Christian Lengauer, Don Batory, Charles Consel, and Martin Odersky (Eds.). Lecture Notes in Computer Science, Vol. 3016. Springer, Berlin, Heidelberg, 30–50. https://doi.org/10.1007/978-3-540-25935-0_3

[21] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. 1998. Multi-stage programming: axiomatization and type safety. In *Automata, Languages and Programming: 25th International Colloquium, ICALP'98 Proceedings*, Kim G. Larsen, Sven Skyum, and Glynn Winskel (Eds.). Lecture Notes in Computer Science, Vol. 1443. Springer Berlin Heidelberg, Berlin, Heidelberg, 918–929. https://doi.org/10.1007/BFb0055113

[22] Walid Taha and Michael Florentin Nielsen. 2003. Environment classifiers. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/604131.604134

[23] Walid Taha and Tim Sheard. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '97)*. ACM, New York, NY, USA, 203–217. https://doi.org/10.1145/258993.259019

[24] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0 PEPM'97.

[25] Walid Mohamed Taha. 1999. *Multistage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology. https://www.cs.rice.edu/~taha/publications/thesis/thesis.pdf

[26] David Van Horn and Matthew Might. 2012. Systematic abstraction of abstract machines. *Journal of Functional Programming* 22, 4-5 (2012), 705–746. https://doi.org/10.1017/S0956796812000238