Mini-MetaML with References

Cristiano Calcagno[1], Eugenio Moggi[1*], and Walid Taha[2**]

[1] DISI, Univ. di Genova, Genova, Italy
{calcagno,moggi}@disi.unige.it

[2] Department of Computing Sciences
Chalmers University of Technology and the University of Göteborg
S-412 96 Göteborg, Sweden
taha@cs.chalmers.se

Abstract. MetaML extends core SML with multi-stage programming constructs. Previous studies of the formal semantics and type systems for multi-stage programming languages do not explain how computational effects, e.g. state, can be *safely* used in MetaML. At the level of the untyped operational semantics, introducing references can lead to *scope extrusion* of lambda bound variables. Because of scope extrusion, a naive extension of the type systems with a type constructor for references invalidates type safety. The approach we propose here uses the Closed type constructor – introduced elsewhere for typing MetaML's Run – to ensure type safety by allowing only the storage of closed values. We expect this approach to work also with other computational effects.

1 Introduction

Multi-stage programming languages provide constructs for the creation, combination, and execution of code fragments [GJ95,TS97,Tah99]. As such, multi-stage languages provide a concise formalism for writing a wide class of program generators, including the result of binding-time analysis in off-line partial evaluation systems [JSS85,JGS93,GJ91,GJ96]. In the current implementation of MetaML [TS97,Tah99], all features of core SML (higher-order functions, polymorphism, data types, references and exceptions) coexist with the multi-stage programming constructs $\langle e \rangle$, $\tilde{}\,e$ and $\mathsf{run}\ e$ (read *"Brackets, Escape, and Run"*, respectively.) MetaML is meant to be a "conservative extension" of SML, that is, programs in the SML fragment should behave as expected at compile-time as well as at run-time. While the dynamic semantics of MetaML is stable [TBS98,MTBS99], there are still major challenges in static type-checking. In fact, the constructs for multi-stage programming introduce new forms of run-time errors [Tah99], which a naive extension of the SML type-checker is unable to detect

[*] Research partially supported by MURST and ESPRIT WG APPSEM.
[**] Supported by a Postdoctoral Fellowship, funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

statically. Therefore, a significant part of the research effort on MetaML has focused on developing better type systems [TBS98,MTBS99,BMTS99]. However, these studies have concentrated on *pure* higher-order functional prototypes and they have *not* addressed the issue of safely introducing computational effects. This paper proposes a simple approach to bridge this gap.

1.1 MetaML's Multi-Stage Programming Constructs

In MetaML, writing $\langle e \rangle$ defers the computation of e; writing ˜ e splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and writing run e evaluates e to obtain a deferred expression, and then evaluates it. Note that ˜e is only legal within lexically enclosing Brackets. Brackets in types such as `<int>` are read *"code of int"*.

The Power Function in MetaML: The following interactive session illustrates the basics of multi-stage programming in MetaML. We consider the canonical example of the power function, and describe how to define a staged and a specialised version.

```
> fun exp_a n x = if n=0 then <1> else <~x * ~(exp_a (n-1) x)>;
val exp_a = fn : int -> <int> -> <int>
> val exp_cg = fn n => <fn x => ~(exp_a n <x>)>;
val exp_cg = fn : int  -> <int -> int>
> val exp_sc = exp_cg 3;
val exp_sc = <fn x => x %* (x %* (x %* 1))> : <int -> int>
> val exp_sp = run exp_sc;
val exp_sp = fn : int -> int
```

Given an exponent `n` and a code fragment representing a base `x`, the *annotated* power function `exp_a` returns code representing a power. The *code-generator* function `exp_cg` is similar, but takes only an exponent and returns code representing a function that takes a base and returns the power. The *specialised code* fragment `exp_sc` is the specialisation of `exp_cg` for the exponent 3. Finally, `exp_sp` is the power function optimised for the exponent 3. In MetaML the last step is generally carried out in *ad hoc* ways [TS00,BMTS99] because the type system cannot express closedness of code.

1.2 Summary and Contributions

The rest of the paper is structured as follows. Section 2 introduces **Mini-MetaML**, a revision of λ^{BN} [BMTS99] with the *same expressivity* but less verbose, in particular the close-with construct is replaced by a simpler closedness annotation. Section 3 explains why a simple-minded extension of MetaML with references leads to **scope extrusion** of bound variables. This is a problem also in multi-level languages for partial evaluation, which combine symbolic evaluation under lambda with storage of dynamic values in static cells. We propose a **hygienic** rule for evaluation under lambda in the presence of state, which avoids scope extrusion and explicates the source of the difficulty with introducing effects. Section 4 proposes a simple extension, called **Mini-MetaML with References**, which ensures run-time safety of well-typed imperative programs. This extension exploits the **Closed type constructor** $[\_]$, which is already used in λ^{BN} (and Mini-MetaML) for typing the Run construct [MTBS99,BMTS99]. Section 5 discusses related work, especially on partial evaluation for imperative programs and on Closed types (i.e. of the form $[\tau]$).

Notation 1 *We introduce notation and terminology used throughout the paper.*

- **Natural numbers***: we write $m+$ for the successor of m.*
- **Finite sequences***: we write $\{x_i | i \in m\}$ for a sequence of length m.*
- **Parallel substitution***: we write $e[x_i := e_i | i \in m]$ for the simultaneous capture-free substitution of each e_i for the corresponding variable x_i in e.*
- **BNF extension***: we write $+=$ in BNF definitions instead of the usual $::=$ to indicate that we are extending a previous BNF definition. That is, after a definition $a \in \mathsf{A} ::= \mathrm{Def}_1$, we write $a \in \mathsf{A} += \mathrm{Def}_2$ as a shorthand for the new definition $a \in \mathsf{A} ::= \mathrm{Def}_1 \mid \mathrm{Def}_2$.*

2 Mini-MetaML

Mini-MetaML's types $\tau \in T$ are defined as follows:

$$\tau \in \mathsf{T} ::= \mathsf{nat} \mid \tau_1 \to \tau_2 \mid [\tau] \mid \langle \tau \rangle$$

These types classify natural numbers, functions, closed values and (open) code fragments, respectively. Mini-MetaML's terms $e \in E$ are defined as follows:

$$\begin{aligned} e \in \mathsf{E} ::= {} & x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{fix}\, x.e \mid \\ & \mathsf{z} \mid \mathsf{s}\, e \mid (\mathsf{case}\, e\, \mathsf{of}\, \mathsf{z} \to e_1 \mid \mathsf{s}\, x \to e_2) \mid \\ & \mathsf{close}\, e \mid (\mathsf{let\ close}\, x = e_1\, \mathsf{in}\, e_2) \mid \\ & \langle e \rangle \mid \tilde{}e \mid \mathsf{up}\, e \mid \mathsf{run}\, e \end{aligned}$$

In addition to the term constructors of Mini-ML [CDDK86], Mini-MetaML provides close and let-close, for introducing and eliminating closedness annotations; Bracket and Escape, for building and combining open code; up, for cross-stage persistence; and run, for executing closed code.

Remark 2 *We should emphasise that the closedness annotations* close *and* let-close *have no run-time cost. In fact, implementations should map* let-close *to* let*-binding, and* close *should be erased.*

2.1 Type System

Typing judgements have the form $\Delta; \Gamma \vdash_n e : \tau$ where $e \in \mathsf{E}$ is a term, $\tau \in \mathsf{T}$ is a type, n is a natural number (the *level* of the term [TS97]), and $\Delta; \Gamma$ is a typing context consisting of two parts, a *closed part* of the form $\Delta \equiv \{x_i\colon \tau_i | i \in m\}$ and an *open part* of the form $\Gamma \equiv \{x_i\colon \tau_i^{n_i} | i \in m\}$. The judgement is read *"term e has type τ at level n in the context $\Delta;\Gamma$"*. Figure 1 gives the typing rules. Mini-MetaML improves on λ^{BN} [BMTS99] in a number of ways:

1. the two-part typing context $\Delta;\Gamma$ (as in $\lambda^{\Box}$ [DP96]) allows to mimic box and let-box of $\lambda^{\Box}$, and to replace the cumbersome close-with of λ^{BN} with close;
2. one has cross-stage persistence at any type [TS97], not only at closed types;
3. there is less syntactic redundancy, since most term-constructs are applicable only at level 0, for example, only up z is well-typed at level 1, and not z;
4. the type of run e is now τ instead of $[\tau]$ (but the two run are interdefinable).

The type system enjoys the following basic properties (see also [BMTS99]).

Lemma 3 (Weakening)

1. *If $\Delta;\Gamma_1,\Gamma_2 \vdash_n e'\colon\tau'$ and x is* fresh, *then $\Delta;\Gamma_1,x\colon\tau^m,\Gamma_2 \vdash_n e'\colon\tau'$*
2. *If $\Delta_1,\Delta_2;\Gamma \vdash_n e'\colon\tau'$ and x is* fresh, *then $\Delta_1,x\colon\tau,\Delta_2;\Gamma \vdash_n e'\colon\tau'$*

Proof. By induction on the derivation of the typing judgement $\Delta;\Gamma_1,\Gamma_2 \vdash_n e'\colon\tau'$ and $\Delta_1,\Delta_2;\Gamma \vdash_n e'\colon\tau'$, respectively.

Lemma 4 (Substitution)

1. *If $\Delta;\Gamma_1,\Gamma_2 \vdash_n e\colon\tau$ and $\Delta;\Gamma_1,x\colon\tau^n,\Gamma_2 \vdash_m e'\colon\tau'$, then $\Delta;\Gamma_1,\Gamma_2 \vdash_m e'[x\colon= e]\colon\tau'$*
2. *If $\Delta_1,\Delta_2;\emptyset \vdash_0 e\colon\tau$ and $\Delta_1,x\colon\tau,\Delta_2;\Gamma \vdash_m e'\colon\tau'$, then $\Delta_1,\Delta_2;\Gamma \vdash_m e'[x\colon= e]\colon\tau'$*

Proof. By induction on the derivation of the typing judgement $\Delta;\Gamma_1,x\colon\tau^n,\Gamma_2 \vdash_m e'\colon\tau'$ and $\Delta_1,x\colon\tau,\Delta_2;\Gamma \vdash_m e'\colon\tau'$, respectively.

2.2 CBV Operational Semantics

The evaluation judgements for Mini-MetaML are of the form $e \overset{n}{\hookrightarrow} v \mid \mathsf{err}$, where $e, v \in \mathsf{E}$ are (open) terms and n is the *level* of the term. We will show that the result of evaluation at level n is a value $v \in \mathsf{V}^n \subset \mathsf{E}$ at level n. The subsets V^n of E are defined by the following BNF:

$$v^0 \in \mathsf{V}^0 ::= \lambda x.e \mid \mathsf{z} \mid \mathsf{s}\ v^0 \mid \mathsf{close}\ v^0 \mid \langle v^1 \rangle$$
$$v^{n+} \in \mathsf{V}^{n+} ::= x \mid \lambda x.v^{n+} \mid v_1^{n+} v_2^{n+} \mid \langle v^{n++} \rangle \mid \mathsf{up}\ v^n$$
$$v^{n++} \in \mathsf{V}^{n++} + = \tilde{\ } v^{n+}$$

Evaluation of run e at level 0 uses *demotion* [TBS98] of a value v at level 1 (written $v\!\downarrow$) to an expression at level 0. The partial function $e\!\downarrow$ is defined by induction on the structure of $e \in \mathsf{E}$ as follows:

$$\begin{aligned} x\!\downarrow &\overset{\Delta}{=} x \\ (\lambda x.e)\!\downarrow &\overset{\Delta}{=} \lambda x.e\!\downarrow \\ (e_1 e_2)\!\downarrow &\overset{\Delta}{=} e_1\!\downarrow e_2\!\downarrow \\ \langle e \rangle\!\downarrow &\overset{\Delta}{=} \langle e\!\downarrow \rangle \\ (\tilde{\ } e)\!\downarrow &\overset{\Delta}{=} \tilde{\ }(e\!\downarrow) \\ (\mathsf{up}\ e)\!\downarrow &\overset{\Delta}{=} e[x\colon= \mathsf{up}\ x | x \in \mathrm{FV}(e)] \end{aligned}$$

Term : $\Delta;\Gamma\vdash_n e\colon\tau$

$$(x)\ \frac{}{\Delta;\Gamma\vdash_0 x\colon\tau}\ x\colon\tau\in\Delta \qquad (x)\ \frac{}{\Delta;\Gamma\vdash_n x\colon\tau}\ x\colon\tau^n\in\Gamma$$

$$([\_]\mathrm{I})\ \frac{\Delta;\emptyset\vdash_0 e\colon\tau}{\Delta;\Gamma\vdash_0 \mathsf{close}\ e\colon[\tau]} \qquad ([\_]\mathrm{E})\ \frac{\Delta;\Gamma\vdash_0 e_1\colon[\tau_1]\quad \Delta,x\colon\tau_1;\Gamma\vdash_n e_2\colon\tau_2}{\Delta;\Gamma\vdash_n (\mathsf{let\ close}\ x=e_1\ \mathsf{in}\ e_2)\colon\tau_2}$$

$$(\to\mathrm{I})\ \frac{\Delta;\Gamma,x\colon\tau_1^n\vdash_n e\colon\tau_2}{\Delta;\Gamma\vdash_n \lambda x.e\colon\tau_1\to\tau_2} \qquad (\to\mathrm{E})\ \frac{\begin{array}{c}\Delta;\Gamma\vdash_n e_1\colon\tau_1\to\tau_2\\ \Delta;\Gamma\vdash_n e_2\colon\tau_1\end{array}}{\Delta;\Gamma\vdash_n e_1e_2\colon\tau_2}$$

$$(\mathrm{fix})\ \frac{\Delta;\Gamma,x\colon\tau^0\vdash_0 e\colon\tau}{\Delta;\Gamma\vdash_0 \mathsf{fix}\ x.e\colon\tau}$$

$$\frac{}{\Delta;\Gamma\vdash_0 \mathsf{z}\colon\mathsf{nat}} \qquad \frac{\Delta;\Gamma\vdash_0 e\colon\mathsf{nat}}{\Delta;\Gamma\vdash_0 \mathsf{s}\ e\colon\mathsf{nat}} \qquad \frac{\begin{array}{c}\Delta;\Gamma\vdash_0 e\colon\mathsf{nat}\\ \Delta;\Gamma\vdash_0 e_1\colon\tau\\ \Delta;\Gamma,x\colon\mathsf{nat}^0\vdash_0 e_2\colon\tau\end{array}}{\Delta;\Gamma\vdash_0 (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z}\to e_1\mid \mathsf{s}\ x\to e_2)\colon\tau}$$

$$(\langle\_\rangle\mathrm{I})\ \frac{\Delta;\Gamma\vdash_{n+} e\colon\tau}{\Delta;\Gamma\vdash_n \langle e\rangle\colon\langle\tau\rangle} \qquad (\langle\_\rangle\mathrm{E})\ \frac{\Delta;\Gamma\vdash_n e\colon\langle\tau\rangle}{\Delta;\Gamma\vdash_{n+}\ \tilde{}e\colon\tau}$$

$$(\mathrm{up})\ \frac{\Delta;\Gamma\vdash_n e\colon\tau}{\Delta;\Gamma\vdash_{n+} \mathsf{up}\ e\colon\tau} \qquad (\mathrm{run})\ \frac{\Delta;\Gamma\vdash_0 e\colon[\langle\tau\rangle]}{\Delta;\Gamma\vdash_0 \mathsf{run}\ e\colon\tau}$$

Fig. 1. Type assignment rules for Mini-MetaML

and $e\!\downarrow$ is undefined for all other term-constructors. The evaluation rules are given in Figure 2. The following result establishes basic desiderata regarding the operational semantics:

Proposition 5 (Values) *$e \overset{n}{\hookrightarrow} v \not\equiv \mathsf{err}$ implies $v\in \mathsf{V}^n$ and $\mathrm{FV}(v)\subseteq\mathrm{FV}(e)$.*

Proof. By induction on the derivation of the evaluation judgement $e \overset{n}{\hookrightarrow} v$. One can ignore error generation and propagation rules, because we assume $v\not\equiv\mathsf{err}$.

By analogy with λ^{BN} [BMTS99] we can prove that evaluation of well-typed expression does not lead to run-time errors.

Lemma 6 (Demotion) *$\emptyset;\Gamma^+\vdash_{n+} v\colon\tau$ and $v\in\mathsf{V}^{n+}$ imply $\emptyset;\Gamma\vdash_n v\!\downarrow\colon\tau$.*

Lemma 7 (Closedness) *$\emptyset;\Gamma^+\vdash_0 v\colon[\tau]$ and $v\in\mathsf{V}^0$ imply $\mathrm{FV}(v)=\emptyset$.*

Theorem 8 (Type Safety). *$\emptyset;\Gamma^+\vdash_n e\colon\tau$ and $e\overset{n}{\hookrightarrow} d$ imply that there exists $v\in\mathsf{V}^n$ s.t. $d\equiv v$ and $\emptyset;\Gamma^+\vdash_n v\colon\tau$.*

Proof. By induction on the derivation of the evaluation judgement $e\overset{n}{\hookrightarrow} d$, using the Demotion and Closedness lemmas.

2.3 The Power Function in Mini-MetaML

We reconsider the power function example in Mini-MetaML. To ease the comparison, we use the following top-level definitions as syntactic sugar for Mini-MetaML (defined in terms of closedness annotations):

- `val x = e; p` stands for (let close $x =$ close e in p)
- a top-level definition by pattern-matching is reduced to one of the form `val f = e; p` in the usual way (that is, using the case and fix constructs).
- `fn close x => e` stands for $\lambda x.$(let close $x = x$ in e), i.e. it defines a function on a Closed type.

In other words, identifiers declared at top-level go in the closed part Δ of a Mini-MetaML typing context $\Delta; \Gamma$. We also assume that the function times $*$: nat $\rightarrow$ nat $\rightarrow$ nat and the constant $1 =$ s z: nat are already in the environment.

```
> fun exp_a z x = <up 1>
    | exp_a (s n) x = <up * ~x ~(exp_a n x)>;
val exp_a = fn : nat -> <nat> -> <nat>
> val exp_cg = fn n => <fn x => ~(exp_a n <x>)>;
val exp_cg = fn : nat  -> <nat -> nat>
> val exp_sc = exp_cg 3;
val exp_sc = <fn x => %* x (%* x (%* x %1))> : <nat -> nat>
> val exp_sp = run (close exp_sc);
val exp_sp = fn : nat -> nat
> val exp_spg = fn close n => run (close (exp_cg n));
val exp_spg = fn : [nat] -> nat -> nat
(* the following is another definition of exp_sp using exp_spg *)
> val exp_sp = exp_spg (close 3);
val exp_sp = fn : nat -> nat
```

The first three steps are very similar to those in the MetaML example, apart from the explicit use of up for cross-stage persistence and the pre-fixed notation for the $*$ operation. The definition of exp_sp is unproblematic in Mini-MetaML, while it is problematic for the MetaML type-checker. Finally, the definition of the *specialised program generator* exp_spg makes essential use of Closed types, and cannot be reproduced in MetaML.

The steps above can also be performed in λ^{BN} [BMTS99] but are more verbose due to the close-with construct. In fact, an expression e in the scope of n top-level definitions `val x_j=e_j` with $j \in n$ corresponds to the Mini-MetaML term

$$(\mathsf{let}\ \{\mathsf{close}\ x_j = \mathsf{close}\ e_j | j \in n\}\ \mathsf{in}\ e)$$

while in λ^{BN} in general it corresponds to a term

$$(\mathsf{let}\ \{x_j = (\mathsf{close}\ e_j[x_i := \mathsf{open}\ x_i | i \in j]\ \mathsf{with}\ \{x_i = x_i | i \in j\}) | j \in n\}\ \mathsf{in}\ e)$$

of size *quadratic* in n because x_i may occur free in e_j when $i < j$.

3 The Problem of Adding References to MetaML

The difficulty in adding references to a multi-stage language like MetaML is that "dynamically bound" variables can go out of the scope of their binder [TS00]. This nasty behaviour is due to the interaction between evaluation under lambda (which entails manipulating open terms) and storage of arbitrary values (including open ones), and is exemplified by the following interactive session:

```
> val a = ref <1>;
val a = ... : ref <int>
> val b = <fn x => ~(a:=<x>; <2>)>;
val b = <fn x => 2> : <int -> int>
> val c = !a;
val c = <x> : <int>
> run c;
system crash!
```

Here, the variable `x` goes outside the scope of the binding lambda, even though the first three lines in the sessions are well-typed. Since `c` is not within the scope of a dynamic lambda, it seems plausible to expect that it should be bound to closed code, but if we run it we get a run-time error. (As remarked in the power function example, in MetaML the last step is generally carried out in *ad hoc* ways because the type system cannot express closedness of code).

This problem of scope extrusion does *not* arise in traditional (CBV or CBN) functional languages, as evaluation can be defined on closed terms only.

Analysis: The problem can be traced back to the level of the operational semantics for MetaML [TBS98,MTBS99,BMTS99] (and for $\lambda^{\bigcirc}$ [Dav96] too). In fact, the rule for symbolic evaluation under lambda in the pure language has the form:

$$(\text{Lam+})\ \frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v}$$

We can identify a technical problem with the naive approach to extending the operational semantics of a multi-stage language with references: In the presence of a store μ the following rule may seem natural

$$(\text{Naive RefLam+})\ \frac{\mu, e \overset{n+}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+}{\hookrightarrow} \mu', \lambda x.v}$$

In the example above, when we symbolically evaluate `fn x => ~(a:=<x>; <2>)`, we apply the following instance of the rule (Naive RefLam+)

$$\frac{\dfrac{\ldots}{l = \langle 1\rangle, \tilde{}\,(l\!:= \langle x\rangle; \langle 2\rangle) \overset{1}{\hookrightarrow} l = \langle x\rangle, 2}}{l = \langle 1\rangle, \lambda x.\tilde{}\,(l\!:= \langle x\rangle; \langle 2\rangle) \overset{1}{\hookrightarrow} l = \langle x\rangle, \lambda x.2}$$

where l is the location bound to the identifier a. This leads to an awkward situation, namely the bound variable x has gone out of scope! In fact, it is not possible to distinguish (in a consistent manner) between the x stored *outside* the scope of the lambda and the x *bound* by the lambda.

Solution (Part I): Hygienic Specification of Operational Semantics. First, we propose the following rule for evaluation under lambda:

$$(\text{RefLam+})\ \frac{\mu, e \overset{n+}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+}{\hookrightarrow} \mu'[x := \mathsf{fault}], \lambda x.v}$$

where fault is a new constant of our untyped language. This formulation avoids scope extrusion (which is mathematically unpleasant), yet it makes explicit that any free occurrence of x in the store μ' is a potential source of run-time errors. Now, the derivation above becomes:

$$\frac{\dfrac{\dots}{l = \langle 1\rangle, \tilde{}(l := \langle x\rangle; \langle 2\rangle) \overset{n+}{\hookrightarrow} l = \langle x\rangle, 2}}{l = \langle 1\rangle, \lambda x.\tilde{}(l := \langle x\rangle; \langle 2\rangle) \overset{n+}{\hookrightarrow} l = \langle \mathsf{fault}\rangle, \lambda x.2}$$

Solution (Part II): Static Fault Prevention. A simple way to prevent the introduction of fault in the store is to allow only the storage of closed values. This restriction is not excessive, since it is implicit in conventional (single-stage) programming languages, where all values *are* closed. Furthermore, expressing closedness in our type system can be done in a natural manner, using the Closed type constructor.

4 Mini-MetaML with References

This section describes how Mini-MetaML can be *safely* extended with updateable references. The first step is to extend the types:

$$\tau \in \mathsf{T}+ = \mathsf{ref}\ \tau$$

where $\mathsf{ref}\ \tau$ is the type of references to a *closed* value of type τ, or simply a value of type $[\tau]$. The terms of the language are extended as follows

$$e \in \mathsf{E}+ = l \mid \mathsf{ref}_\tau\ e \mid \mathsf{get}\ e \mid \mathsf{set}(e_1, e_2) \mid \mathsf{fault}$$

where $l \in \mathsf{Loc}$ is a constant for a location, and the other operations are the usual ones for manipulating the store. We require programs to specify the type of a new location, so that we don't need an instrumented semantics (as done for instance in [MP99]) to prove Type Safety.

4.1 Type System

Since we have introduced constants for locations, we must also specify the types of such constants in the typing judgement. For this purpose, we use a signature Σ, which can be *confused* with contexts of the form Δ. For this language the only signatures of interest have the form $\Sigma \equiv \{l_i{:}\,\mathsf{ref}\ \tau_i | i \in m\}$. The type judgement now has the general form $\Sigma, \Delta; \Gamma \vdash_n e'{:}\,\tau$, and the extension to the type system is presented in Figure 3. Note that with cross-stage persistence, we need to type these operations at level 0 only. By analogy with Mini-MetaML, we establish the following basic properties.

Lemma 9 (Weakening)

1. *If $\Sigma, \Delta; \Gamma_1, \Gamma_2 \vdash_n e'{:}\,\tau'$ and x is* fresh, *then $\Sigma, \Delta; \Gamma_1, x{:}\,\tau^m, \Gamma_2 \vdash_n e'{:}\,\tau'$*
2. *If $\Sigma, \Delta_1, \Delta_2; \Gamma \vdash_n e'{:}\,\tau'$ and x is* fresh, *then $\Sigma, \Delta_1, x{:}\,\tau, \Delta_2; \Gamma \vdash_n e'{:}\,\tau'$*
3. *If $\Sigma_1, \Sigma_2, \Delta; \Gamma \vdash_n e'{:}\,\tau'$ and l is* fresh, *then $\Sigma_1, l{:}\,\mathsf{ref}\ \tau, \Sigma_2, \Delta; \Gamma \vdash_n e'{:}\,\tau'$*

Lemma 10 (Substitution)

1. *If $\Sigma, \Delta; \Gamma_1, \Gamma_2 \vdash_n e{:}\,\tau$ and $\Sigma, \Delta; \Gamma_1, x{:}\,\tau^n, \Gamma_2 \vdash_m e'{:}\,\tau'$, then $\Sigma, \Delta; \Gamma_1, \Gamma_2 \vdash_m e'[x{:}= e]{:}\,\tau'$*
2. *If $\Sigma, \Delta_1, \Delta_2; \emptyset \vdash_0 e{:}\,\tau$ and $\Sigma, \Delta_1, x{:}\,\tau, \Delta_2; \Gamma \vdash_m e'{:}\,\tau'$, then $\Sigma, \Delta_1, \Delta_2; \Gamma \vdash_m e'[x{:}= e]{:}\,\tau'$*

4.2 CBV Operational Semantics

The evaluation judgements for the operational semantics with references are of the form $\mu, e \overset{n}{\hookrightarrow} (\mu', v) \mid \mathsf{err}$, where $\mu, \mu' \in \mathsf{S} \overset{\Delta}{=} \mathsf{Loc} \overset{fin}{\to} (\mathsf{V}^0 \times \mathsf{T})$ are *annotated* stores. The type annotation is set when a new location is created, but is never updated. When a run-time error occurs, the resulting store μ' is lost. Figure 4 gives the full operational semantics. We will show that the result of evaluation at level n is a value $v \in \mathsf{V}^n \subset \mathsf{E}$ at level n, where the definition of value at level n is extended as follows

$$v^0 \in V^0 + = l$$

$$v^{n+} \in V^{n+} + = \mathsf{fault}$$

The definition of $e\!\downarrow$ is also extended, namely it is the identity on fault

$$\mathsf{fault}\!\downarrow \overset{\Delta}{=} \mathsf{fault}$$

and it is undefined on terms of the form $\mathsf{ref}_\tau\ e$, $\mathsf{get}\ e$ and $\mathsf{set}(e_1, e_2)$.

Notation 11 *We write $\Sigma \models \mu$ iff for any $l \in \mathsf{Loc}$*

– *$\mu(l) = (v, \tau)$ implies $\Sigma; \emptyset \vdash_0 v{:}\,\tau$*

We write Σ_μ for the signature determined by μ, i.e. $\{l{:}\,\mathsf{ref}\ \tau | \mu(l) = (\_, \tau)\}$.

By analogy with Mini-MetaML, we can establish the following results.

Proposition 12 (Values) *$\mu, e \overset{n}{\hookrightarrow} \mu', v$ implies $v \in \mathsf{V}^n$ and $\Sigma_\mu \subseteq \Sigma_{\mu'}$ and $\mathrm{FV}(\mu', v) \subseteq \mathrm{FV}(\mu, e)$.*

Proof. By induction on the derivation of the evaluation judgement $\mu, e \overset{n}{\hookrightarrow} \mu', v$.

Remark 13 *If we did not include type information in the store, we should have written $dom(\mu) \subseteq dom(\mu')$ instead of $\Sigma_\mu \subseteq \Sigma_{\mu'}$.*

Lemma 14 (Demotion) *$\Sigma; \Gamma^+ \vdash_{n+} v : \tau$ and $v \in \mathsf{V}^{n+}$ imply $\Sigma; \Gamma \vdash_n v\!\downarrow : \tau$.*

Lemma 15 (Closedness) *$\Sigma; \Gamma^+ \vdash_0 v : [\tau]$ and $v \in \mathsf{V}^0$ imply $\mathrm{FV}(v) = \emptyset$.*

Theorem 16 (Type Safety). *$\Sigma_\mu \models \mu$ and $\Sigma_\mu; \Gamma^+ \vdash_n e : \tau$ and $\mu, e \overset{n}{\hookrightarrow} d$ imply that there exist $\mu' \in \mathsf{S}$ and $v \in \mathsf{V}^n$ s.t. $d \equiv (\mu', v)$ and $\Sigma_{\mu'} \models \mu'$ and $\Sigma_{\mu'}; \Gamma^+ \vdash_n v : \tau$.*

Proof. By induction on the derivation of the evaluation judgement $\mu, e \overset{n}{\hookrightarrow} d$.

5 Related Work

Closed Types and Language Design: Closed types in combination with (open) code types are introduced in [MTBS99,BMTS99,Tah99] to provide a natural typing for Run and at the same time support the techniques typical of partial evaluation, in particular symbolic evaluation under lambda. It is pleasing that no additional type infra-structure is needed to accommodate references (and we conjecture that the same is true when adding other computational effects).

We have found it convenient to split the typing context in two parts, as in $\lambda^\Box$ of [DP96]. $\lambda^\Box$ does not allow open code nor symbolic evaluation under lambda, therefore the addition of reference types is straightforward, because in the dynamic semantics values (and expressions) are always closed. On the other hand, $\lambda^\bigcirc$ of [Dav96] allows open code and symbolic evaluation under lambda (but has no construct for running code), therefore a naive addition of references leads to the same problem of scope extrusion we have exemplified in Section 3.

Closed Types and Monadic Semantics of MetaML: The solution proposed in this paper for typing multi-stage imperative programs, i.e. to allow only storage of closed values, seems to provide a general recipe for adding other computational effects to MetaML *safely*. For instance, in the case of exceptions, only closed values should be used in packets; in the case of communications, only closed values should be transmitted on channels. The monadic λ^{BN}-model in Example 4.3 of [BMTS99] suggests already such a restriction. In particular, the monad on the *open universe $\mathcal{D}$* is defined as point-wise extension of a monad on the *closed universe $\mathcal{C}$*. On the other hand, the monadic λ^{BN}-model does not provide a complete picture. In particular, Example 4.3 imposes other requirements on the monad on $\mathcal{C}$, that do not seem to have an *operational* justification.

Binding-Time Analysis of Imperative Programs: Thiemann and Dussart [TD] describe an off-line partial evaluator for a higher-order language with first-class references, which uses a two-level language with regions (see also [Thi97]). In comparison to their work, other off-line partial evaluators for imperative languages either adopt a more conservative binding-time analysis, or do not address the combination of higher-order and first class-references. The two-level language of [TD] allows storing dynamic values in static cells, but the type and effect system prohibits operating on static cells within the scope of a dynamic lambda (unless these cells belong to a region local to the body of the dynamic lambda). While both the type system of [TD] and ours ensure that no run-time error (like scope extrusion) can occur, they provide incomparable extensions. Hatcliff and Danvy [HD97] propose a partial evaluator for a computational metalanguage, and they formalise existing techniques in a uniform framework by abstracting from dynamic computational effects. However, this partial evaluator does not allow interesting computational effects at specialisation time.

Goal of Type System: The goal of the type systems that we have proposed for MetaML [TBS98,MTBS99] has been to ensure run-time safety. An alternative approach, taken by Shields, Sheard, and Peyton-Jones [SSJ98], gives up static type checking/inference in favour of *staged* type checking/inference. In their approach, some type information has to be kept at run-time, and the programmer has to cope with the possibility of type errors being detected at later stages, as programs still need to pass the type checker (at run-time) before being executed.

Hughes has proposed a new paradigm called type specialisation [Hug98], which has also been applied to imperative programs [DHT97]. There, the type systems for the two-level languages used in type specialisation is not intended to give many guarantees. For example, a well-formed two-level program may fail to specialise; rather, the type information is intended (and is essential) for *driving* the specialisation of terms. For these reasons it seems more appropriate to compare type specialisation with staged type inference, rather than type systems for static checking (such as the one proposed in the present paper).

6 Conclusions and Future Work

Elsewhere we have shown that closedness annotations are important for typing the Run construct. In this paper we argue that they are useful also in developing type systems for multi-stage programming languages with first-class references. More generally, we conjecture that closedness annotations are useful for multi-stage programming in the presence of computational effects, and we are currently testing this conjecture on specific examples. A related issue is whether multi-stage programming can make good use of computational effects, or whether it can just co-exist *peacefully* with them.

Another line of research is the use of closedness annotations for partial evaluation. Ideally, one would like to improve on [TD] by considering a two-level language with closedness and region annotations. Here the main difficulty could be in the binding-time analysis for inferring such annotations.

References

[BMTS99] Zine El-Abidine Benaissa, Eugenio Moggi, Walid Taha, and Tim Sheard. Logical modalities and multi-stage programming. In *Federated Logic Conference (FLoC) Satellite Workshop on Intuitionistic Modal Logics and Applications (IMLA)*, July 1999. To appear.

[CDDK86] D. Clement, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 13–27. ACM, ACM, August 1986.

[Dav96] Rowan Davies. A temporal-logic approach to binding-time analysis. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.

[DHT97] D. Dussart, J. Hughes, and P. Thiemann. Type specialization for imperative languages. In *International Conference on Functional Programming (ICFP'97), Amsterdam, The Netherlands, June 1997*, pages 204–216. New York: ACM, 1997.

[DP96] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 258–270, St. Petersburg Beach, January 1996.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions for program specialization. In S. D. Swierstra and M. Hermenegildo, editors, *Programming Languages: Implementations, Logics and Programs (PLILP'95)*, volume 982 of *Lecture Notes in Computer Science*, pages 259–278. Springer-Verlag, 1995.

[GJ96] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[HD97] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, October 1997.

[Hug98] John Hughes. Type specialization. *ACM Computing Surveys*, 30(3es), September 1998.

[JGS93] Neil D. Jones, Carsten K Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[JSS85] Neil D. Jones, Peter Sestoft, and Harald Sondergraard. An experiment in partial evaluation: The generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *Lecture Notes in Computer Science*, pages 124–140. Springer-Verlag, 1985.

[MP99] E. Moggi and F. Palumbo. Monadic encapsulation of effects: a revised approach. In *HOOTS 3rd International Workshop*, volume 26 of *Electronic Notes in Theoretical Computer Science*, 1999.

[MTBS98] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, OGI, October 1998. Available from [Ore].

[MTBS99] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive. In *European Symposium on Programming (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999. An extended version appears in [MTBS98].

[Ore] 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.

[SSJ98] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing through staged type inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 289–302, January 1998.

[Tah99] Walid Taha. *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).

[TBS98] Walid Taha, Zine-El-Abidine Benaissa, and Tim Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th International Colloquium on Automata, Languages, and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 918–929, Aalborg, July 1998.

[TD] Peter Thiemann and Dirk Dussart. Partial evaluation for higher-order languages with state. Available online from http://www.informatik.uni-freiburg.de/~thiemann/papers/index.html.

[Thi97] P. Thiemann. Correctness of a region-based binding-time analysis. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference, Pittsburgh, Pennsylvania, March 1997*, page 26. Carnegie Mellon University, Elsevier, 1997.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of the ACM-SIGPLAN Symposium on Partial Evaluation and semantic based program manipulations PEPM'97, Amsterdam*, pages 203–217. ACM, 1997. An extended and revised version appears in [TS00].

[TS99] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Department of Computer Science, Oregon Graduate Institute, January 1999. Extended version of [TS97]. Available from [Ore].

[TS00] Walid Taha and Tim Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000. In Press. Revised version of [TS99].

Terms

$$e \in \mathsf{E} ::= x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{fix}\ x.e \mid \mathsf{z} \mid \mathsf{s}\ e \mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \mid$$
$$\mathsf{close}\ e \mid (\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \mid \langle e \rangle \mid {}^{\sim}e \mid \mathsf{up}\ e \mid \mathsf{run}\ e$$

Normal Evaluation

$$\lambda x.e \overset{0}{\hookrightarrow} \lambda x.e \qquad \frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v \quad e[x := v] \overset{0}{\hookrightarrow} v'}{e_1 e_2 \overset{0}{\hookrightarrow} v'} \qquad \frac{e[x := \mathsf{fix}\ x.e] \overset{0}{\hookrightarrow} v}{\mathsf{fix}\ x.e \overset{0}{\hookrightarrow} v}$$

$$\mathsf{z} \overset{0}{\hookrightarrow} \mathsf{z} \qquad \frac{e \overset{0}{\hookrightarrow} v}{\mathsf{s}\ e \overset{0}{\hookrightarrow} \mathsf{s}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{z} \quad e_1 \overset{0}{\hookrightarrow} v}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} v}$$

$$\frac{e \overset{0}{\hookrightarrow} \mathsf{s}\ v \quad e_2[x := v] \overset{0}{\hookrightarrow} v'}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} v'} \qquad \frac{e \overset{0}{\hookrightarrow} v}{\mathsf{close}\ e \overset{0}{\hookrightarrow} \mathsf{close}\ v}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} \mathsf{close}\ v \quad e_2[x := v] \overset{n}{\hookrightarrow} v'}{(\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \overset{n}{\hookrightarrow} v'} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v \rangle \quad v{\downarrow} \overset{0}{\hookrightarrow} v'}{\mathsf{run}\ e \overset{0}{\hookrightarrow} v'} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{{}^{\sim}e \overset{1}{\hookrightarrow} v}$$

Symbolic Evaluation

$$x \overset{n+}{\hookrightarrow} x \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v} \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{e_1 e_2 \overset{n+}{\hookrightarrow} v_1 v_2} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{{}^{\sim}e \overset{n++}{\hookrightarrow} {}^{\sim}v}$$

$$\frac{e \overset{n}{\hookrightarrow} v}{\mathsf{up}\ e \overset{n+}{\hookrightarrow} \mathsf{up}\ v}$$

Error Generation

$$x \overset{0}{\hookrightarrow} \mathsf{err} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \not\equiv \lambda x.e}{e_1 e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{0}{\hookrightarrow} v \not\equiv \mathsf{z} \mid \mathsf{s}\ e'}{(\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{e_1 \overset{0}{\hookrightarrow} v \not\equiv \mathsf{close}\ e}{(\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{e \overset{0}{\hookrightarrow} v \not\equiv \mathsf{close}\ \langle e' \rangle \text{ or } e'{\downarrow} \text{ undefined}}{\mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{e \overset{0}{\hookrightarrow} v \not\equiv \langle e' \rangle}{{}^{\sim}e \overset{1}{\hookrightarrow} \mathsf{err}}$$

In addition there are SML-like error propagation rules, that are not spelled out.

Fig. 2. Operational semantics for Mini-MetaML

Term : $\Sigma, \Delta; \Gamma \vdash_n e : \tau$

$$(l)\ \frac{}{\Sigma, \Delta; \Gamma \vdash_0 l : \mathsf{ref}\ \tau}\ l : \mathsf{ref}\ \tau \in \Sigma \qquad \frac{\Sigma, \Delta; \Gamma \vdash_0 e : [\tau]}{\Sigma, \Delta; \Gamma \vdash_0 \mathsf{ref}_\tau\ e : \mathsf{ref}\ \tau}$$

$$\frac{\Sigma, \Delta; \Gamma \vdash_0 e : \mathsf{ref}\ \tau}{\Sigma, \Delta; \Gamma \vdash_0 \mathsf{get}\ e : [\tau]} \qquad \frac{\Sigma, \Delta; \Gamma \vdash_0 e_1 : \mathsf{ref}\ \tau \quad \Sigma, \Delta; \Gamma \vdash_0 e_2 : [\tau]}{\Sigma, \Delta; \Gamma \vdash_0 \mathsf{set}(e_1, e_2) : \mathsf{ref}\ \tau}$$

Fig. 3. Additional type assignment rules for Mini-MetaML with References

Terms

$$\begin{aligned} e \in \mathsf{E} ::= {} & x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{fix}\ x.e \mid \mathsf{z} \mid \mathsf{s}\ e \mid (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \mid \\ & \mathsf{close}\ e \mid (\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \mid \langle e \rangle \mid {\char126}e \mid \mathsf{up}\ e \mid \mathsf{run}\ e \\ & l \mid \mathsf{ref}_\tau\ e \mid \mathsf{get}\ e \mid \mathsf{set}(e_1, e_2) \mid \mathsf{fault} \end{aligned}$$

Normal Evaluation

$$\mu, \lambda x.e \overset{0}{\hookrightarrow} \mu, \lambda x.e \qquad \frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', \lambda x.e \quad \mu', e_2 \overset{0}{\hookrightarrow} \mu'', v \quad \mu'', e[x := v] \overset{0}{\hookrightarrow} \mu''', v'}{\mu, e_1 e_2 \overset{0}{\hookrightarrow} \mu''', v'}$$

$$\frac{\mu, e[x := \mathsf{fix}\ x.e] \overset{0}{\hookrightarrow} \mu', v}{\mu, \mathsf{fix}\ x.e \overset{0}{\hookrightarrow} \mu', v} \qquad \mu, \mathsf{z} \overset{0}{\hookrightarrow} \mu, \mathsf{z} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v}{\mu, \mathsf{s}\ e \overset{0}{\hookrightarrow} \mu', \mathsf{s}\ v}$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \mathsf{z} \quad \mu', e_1 \overset{0}{\hookrightarrow} \mu'', v}{\mu, (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} \mu'', v}$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \mathsf{s}\ v \quad \mu', e_2[x := v] \overset{0}{\hookrightarrow} \mu'', v'}{\mu, (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} \mu'', v'} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v}{\mu, \mathsf{close}\ e \overset{0}{\hookrightarrow} \mu', \mathsf{close}\ v}$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', \mathsf{close}\ v \quad \mu', e_2[x := v] \overset{n}{\hookrightarrow} \mu'', v'}{\mu, (\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \overset{n}{\hookrightarrow} \mu'', v'}$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \mathsf{close}\ \langle v \rangle \quad \mu', v{\downarrow} \overset{0}{\hookrightarrow} \mu'', v'}{\mu, \mathsf{run}\ e \overset{0}{\hookrightarrow} \mu'', v'} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', \langle v \rangle}{\mu, {\char126}e \overset{1}{\hookrightarrow} \mu', v} \qquad \mu, l \overset{0}{\hookrightarrow} \mu, l$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \mathsf{close}\ v}{\mu, \mathsf{ref}_\tau\ e \overset{0}{\hookrightarrow} \mu'\{l = (v, \tau)\}, l}\ l \notin dom(\mu') \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', l}{\mu, \mathsf{get}\ e \overset{0}{\hookrightarrow} \mu', \mathsf{close}\ v}\ \mu'(l) \equiv (v, \_)$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', l \quad \mu', e_2 \overset{0}{\hookrightarrow} \mu'', \mathsf{close}\ v}{\mu, \mathsf{set}(e_1, e_2) \overset{0}{\hookrightarrow} \mu''\{l = (v, \tau)\}, l}\ \mu'(l) \equiv (\_, \tau)$$

Symbolic Evaluation

$$\mu, x \overset{n+}{\hookrightarrow} \mu, x \qquad \frac{\mu, e \overset{n+}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+}{\hookrightarrow} \mu'[x := \mathsf{fault}], \lambda x.v} \qquad \frac{\mu, e_1 \overset{n+}{\hookrightarrow} \mu', v_1 \quad \mu', e_2 \overset{n+}{\hookrightarrow} \mu'', v_2}{\mu, e_1 e_2 \overset{n+}{\hookrightarrow} \mu'', v_1 v_2}$$

$$\frac{\mu, e \overset{n+}{\hookrightarrow} \mu', v}{\mu, \langle e \rangle \overset{n}{\hookrightarrow} \mu', \langle v \rangle} \qquad \frac{\mu, e \overset{n+}{\hookrightarrow} \mu', v}{\mu, {\char126}e \overset{n++}{\hookrightarrow} \mu', {\char126}v} \qquad \frac{\mu, e \overset{n}{\hookrightarrow} \mu', v}{\mu, \mathsf{up}\ e \overset{n+}{\hookrightarrow} \mu', \mathsf{up}\ v} \qquad \mu, \mathsf{fault} \overset{n+}{\hookrightarrow} \mu, \mathsf{fault}$$

Error Generation

$$\mu, x \overset{0}{\hookrightarrow} \mathsf{err} \qquad \frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv \lambda x.e}{\mu, e_1 e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv \mathsf{z} \mid \mathsf{s}\ e'}{\mu, (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv \mathsf{close}\ e}{\mu, (\mathsf{let\ close}\ x = e_1\ \mathsf{in}\ e_2) \overset{n}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv \mathsf{close}\ \langle e' \rangle\ \text{or}\ e'{\downarrow}\ \text{undefined}}{\mu, \mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv \langle e' \rangle}{\mu, {\char126}e \overset{1}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv \mathsf{close}\ e'}{\mu, \mathsf{ref}_\tau\ e \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv l \in dom(\mu')}{\mu, \mathsf{get}\ e \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv l \in dom(\mu')}{\mu, \mathsf{set}(e_1, e_2) \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', l \in dom(\mu') \quad \mu', e_2 \overset{0}{\hookrightarrow} \mu'', v \not\equiv \mathsf{close}\ e'}{\mu, \mathsf{set}(e_1, e_2) \overset{0}{\hookrightarrow} \mathsf{err}}$$

$$\mu, \mathsf{fault} \overset{0}{\hookrightarrow} \mathsf{err}$$

Fig. 4. Operational semantics for Mini-MetaML with References