Closed Types for a Safe Imperative MetaML

C. CALCAGNO, E. MOGGI,[∗]
DISI, Univ. di Genova, Genova, Italy. (*e-mail:* {calcagno,moggi}@disi.unige.it)

T. SHEARD, [†]
Oregon Graduate Institute, Portland, OR. (*e-mail:* sheard@cse.ogi.edu)

Abstract

This paper addresses the issue of safely combining computational effects and multi-stage programming. We propose a type system, which exploits a notion of *closed type*, to check statically that an imperative multi-stage program does not cause run-time errors. Our approach is demonstrated formally for a core language called $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$. This core language safely combines multi-stage constructs and ML-style references, and is a *conservative extension* of $\mathsf{MiniML}_{\mathsf{ref}}$, a simple imperative subset of SML. In previous work, we introduced a *closed type constructor*, which was enough to ensure the safe execution of dynamically generated code in the pure fragment of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$.

1 Introduction

Techniques such as program generation, multi-level partial evaluation, and run-time code generation respond to the need for general purpose programs which do not pay unnecessary run-time overheads. Over the past decade, there have been substantial advances in these techniques, as exemplified by work in partial evaluation, in high-level program generation, and in run-time code generation. Aiming to provide a uniform and principled view of these diverse techniques, *multi-stage programming* (Taha & Sheard, 1997; Taha *et al.*, 1998; Moggi *et al.*, 1999; Benaissa *et al.*, 1999; Taha, 1999; Calcagno *et al.*, 2000; Taha, 2000b) is a novel paradigm for the development of maintainable, higher-performance programs.

The key idea in multi-stage programming is the use of simple *annotations* to allow the programmer to break down a computation into distinct *stages*. Multi-stage languages provide support for building, combining, and executing code at run-time. The prototypical example of a multi-stage programming language is MetaML (MHP, 2000), which provides a type constructor $\langle\_\rangle$ for (potentially) open code. Three staging annotations operate on this type: Brackets $\langle\_\rangle$, Escape $\tilde{}\,\_$ and Run $\mathsf{run}\ \_$. Brackets defer the computation of its argument (constructing code instead);

∗ Research supported by MIUR national project NAPOLI, ESPRIT WG APPSEM, and IST FET GC-project DART.
† Research supported by NSF Grant CDA-9703218, the M.J. Murdock Charitable Trust and the Department of Defense.

Escape splices its code argument into the body of surrounding Brackets (combining code fragments into larger pieces of code); and Run executes its code argument.

A characteristic of multi-stage programming languages is the *need* to "evaluate symbolically under lambda" and to manipulate, at run-time, values with free "dynamic variables". This need arises neither in the call-by-name nor the call-by-value pure lambda calculi, and requires special care when defining both the untyped semantics and the type systems for multi-stage programming language (Taha & Sheard, 1997; Taha *et al.*, 1998; Moggi *et al.*, 1999; Taha, 1999).

The current release of MetaML (MHP, 2000) is a substantial language, supporting most features of SML and a host of novel meta-programming constructs. In the current public release, safety is guaranteed only for programs in the pure fragment. However, we are continually working to ensure type safety for a larger and larger subset of the language. A particularly hard problem is the safety of MetaML's staging constructs in the presence of computational effects.

This paper advocates a simple and effective approach for safely adding computational effects into languages that manipulate open values (in general, and open code in particular). The approach capitalizes on previous work (Moggi *et al.*, 1999; Calcagno *et al.*, 2000; Calcagno & Moggi, 2000), and exploits a notion of **closed type**. The key property of a term e of closed type is that "all free occurrences in e of *dynamic* variables are *dead code*". Not surprisingly, our type system is only a static approximation, thus not every well-typed term satisfying such a dynamic property can be ascribed a closed type.

We demonstrate the approach for ML-style references (Milner *et al.*, 1997), by extending recent studies into the semantics and type systems for multi-level and multi-stage languages. MetaML is designed to be an *extension* of SML (Milner *et al.*, 1997). There are two reasons for this design choice: first, to facilitate MetaML's acceptance among an existing user base, and second, to confine the conceptual challenges for new users to the staging constructs only. To be consistent with this design goal, a type system ensuring safety of well-typed MetaML programs should also be consistent with the operational semantics and type system for SML.

One could get a feeling for meta-programming in $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ by skimming through the imperative power function example of Section 4.1.

1.1 The Scope Extrusion Problem

When adding ML-style references to a multi-stage language like MetaML, it is possible for *dynamic* variables to escape the scope of their binder (Taha & Sheard, 2000). This form of **scope extrusion** is specific to multi-level and multi-stage languages supporting "symbolic evaluation under lambda", and does *not* arise in traditional (call-by-name or call-by-value) programming languages, where evaluation is usually restricted to closed terms. The problem lies in the *run-time interaction* between symbolic evaluation under lambda and references. In a big-step operational semantics for an imperative language the natural rule for symbolic evaluation under lambda would be $\dfrac{\mu, e \overset{n+1}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+1}{\hookrightarrow} \mu', \lambda x.v}$, where μ and μ' are the stores before and after symbolic evaluation. During evaluation of e the bound variable x could be stored in μ', as exemplified by the following MetaML sessions. Example 1.1 exhibits a "nasty" scope extrusion problem using a reference to (potentially open) code of type `int`, while Example 1.2 exhibits a "benign" instance of the problem using a reference to a function of ML type `int -> int`.

Example 1.1 (*"Nasty"*)
The following session is statically well-typed in naive extensions of previously proposed type systems for MetaML (Taha *et al.*, 1998; Taha & Sheard, 2000; Moggi *et al.*, 1999):

```
-| val l = ref <1>;
val l = ... : <int> ref
-| val f = <fn x => ~(l:=<x+1>; <2>)>;
val f = <fn x => 2> : <int -> int>
-| val c = !l;
val c = <x+1> : <int>
-| run c;
**system crash**
```

Evaluating the declaration of `f`, `x` goes outside the scope of the binding lambda. This means the value bound to `c` is not typable in the current environment (thus, we have lost Subject Reduction). An attempt to execute `c` results in a crash.

Example 1.2 (*"Benign"*)
The following session gives an example where a variables goes outside the scope of its binder. However, in this case (we claim here and we will prove in what follows), this anomaly cannot be observed in the language.

```
-| fun fst (x,y) = x;
val fst = fn ... : 'a * 'b -> 'a
-| val l = ref (fn y => y+0);
val l = ... : (int -> int) ref
-| val f = <fn x => ~(l:=fn y => fst(y,<x+1>)); <2>)>;
val f = <fn x => 2> : <int -> int>
-| val g = !l;
val g = fn ... : int -> int
```

As in the previous example, evaluating the declaration of `f` makes `x` escape the scope of its binding lambda. Hence, the value bound to `g` is an open term, and therefore untypable. But this example does *not* result in a crash, no matter what the rest of the program does with `g`. The reason is that `x` is **dead code** in the value `fn y => fst(y,<x+1>)` bound to `g`.

1.2 Proposed Solution

Our solution involves two ingredients: the type system exploits *closed types* to reject Example 1.1 and accept Example 1.2; the untyped operational semantics exploits a new binder for a *hygienic* handling of scope extrusion.

Closed types. The type system of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ (Figure 3) rejects Example 1.1 by identifying a subset of types, called closed types, and by restricting the Reference type constructor $\mathsf{ref}\ \_$ to be applied only to closed types. Intuitively, closed types can be characterized by the following property: values of closed type have no free occurrences of *dynamic* variables. Since a code type $\langle t\rangle$ is not considered a closed type, the type $\mathsf{ref}\ \langle t\rangle$ is not well-formed. This seems to preclude references to code altogether. Such a limitation is overcome by the Closed type constructor $[\_]$, first introduced in (Moggi *et al.*, 1999) for typing Run. The $[\_]$ type constructor maps a type t to the closed type $[t]$, intuitively $[t]$ is the biggest closed type included in t. In Example 1.1, we cannot assign to `l` the type `<int> ref`, because such a type is ill-formed, but one can assign to `l` the type `[<int>] ref`. On the other hand, in the second line one can assign to `<x>` the open type `<int>`, but not the closed type `[<int>]`. Therefore, `l:=<x>` fails to type-check. When the characterizing property of closed types is interpreted syntactically, as done in (Calcagno *et al.*, 2000), the resulting type system is safe but too restrictive, in particular it fails to extend the ML type system. In fact, in such a system functional types are not closed, thus the ML type `(int -> int) ref` is considered ill-formed. This paper adopts a more semantic reading of the characterizing property, namely: in a value of closed type all free occurrences of *dynamic* variables are "dead code", i.e. if such occurrences are replaced with `raise Unreachable` (and the rest of the program is well-typed), then the exception `Unreachable` will not be raised (Xi, 1999). The resulting type system accepts Example 1.2.

Hygienic handling of scope extrusion. The operational semantics (Section 3.2) must handle scope extrusion compatibly with a proof of subject reduction, since evaluation of well-typed programs may cause some *benign* scope extrusion. For this purpose we introduce a binder $(x)e$ called *Bind*, which *declares* that the free occurrences of x are dead code in e. In this refined operational semantics, when a value v is stored, all its free variables are declared dead code. For instance, what gets stored in `l` is `(x)<x+1>` in Example 1.1 and `(x)fn y => fst(y,<x+1>)` in Example 1.2. Operationally, $(x)e$ is equivalent to $e[x := \mathsf{fault}]$, where fault is some *faulty term*, e.g. `0 0`. The typing rules for Bind allow the user to declare that "in a term of *closed type* all free occurrences of *dynamic* variables are *dead code*". A posteriori, a *type safety theorem* tells us that the declarations of dead code allowed by the type system are *semantically valid*, i.e. no attempt is made to evaluate a variable replaced by fault, because well-typed programs cannot raise run-time errors.

1.3 Contributions and Summary

This paper proposes a safe approach for adding multi-stage programming constructs to an imperative programming language using a notion of closed type. We expect this notion to provide a general solution for safely adding multi-stage programming constructs to programming languages with other computational effects, e.g. by allowing only values of closed types to be packaged with exceptions, or exchanged between communicating processes.

Section 2 introduces the imperative language $\mathsf{MiniML}_{\mathsf{ref}}$ namely MiniML (Clement *et al.*, 1986) extended with ML-style references. Section 3 defines the multi-stage extension $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ of $\mathsf{MiniML}_{\mathsf{ref}}$. The multi-stage programming constructs of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ can be described informally in a hypothetical two-level language with levels *obj* and *meta*:

- A Code type constructor

$$\text{Code}\ \frac{\Gamma \vdash_{obj} t \quad \text{object-level type}}{\Gamma \vdash_{meta} \langle t \rangle \quad \text{meta-level type}}$$

  and constructs

$$\text{Brackets}\ \frac{\Gamma \vdash_{obj} e : t \quad \text{object-level program fragment of type } t}{\Gamma \vdash_{meta} \langle e \rangle : \langle t \rangle \quad \text{meta-level representation of } e}$$

$$\text{Escape}\ \frac{\Gamma \vdash_{meta} e : \langle t \rangle}{\Gamma \vdash_{obj}\ \tilde{}e : t}$$

  such that $\tilde{}\langle e \rangle \longrightarrow e$.
  These constructs are borrowed from MetaML (Taha & Sheard, 1997; Taha *et al.*, 1998) and the multi-level language $\lambda^{\bigcirc}$ (Davies, 1996).
- Cross-Stage Persistence

$$\text{CSP}\ \frac{\Gamma \vdash_{meta} e : t}{\Gamma \vdash_{obj} \%e : t}\ t \text{ object- and meta-level type}$$

  allows the inclusion of meta-level computations in object-level programs, a similar construct is available in λ^{BN} (Benaissa *et al.*, 1999), and implicitly also in MetaML (Taha & Sheard, 1997).
- A Closed type constructor.

$$\text{Closed}\ \frac{\Gamma \vdash_{meta} t \quad \text{meta-level type}}{\Gamma \vdash_{meta} [t] \quad \text{meta-level type}}$$

  Intuitively $[t]$ is the subset of t consisting of the e without *unresolved links*, and a construct

$$\text{Run}\ \frac{\Gamma \vdash_{meta} e : [\langle t \rangle]}{\Gamma \vdash_{meta} \mathsf{run}\ e : [t]}\ t \text{ object- and meta-level type}$$

  where $e : [\langle t \rangle]$ means that e represents a *complete* object-level program e', and $\mathsf{run}\ e$ corresponds to the execution of e'.

These constructs are similar to those proposed in (Moggi *et al.*, 1999; Benaissa *et al.*, 1999; Taha & Sheard, 1997; Davies & Pfenning, 1996).

In $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ the meta-level is *reflected* in the object-level, thus one gets an infinite tower of levels (Smith, 1982): 0 is the meta-level, 1 is the reflection of the meta-level (i.e. the object-level), 2 is the reflection of the reflection of the meta-level,. . . .

The main technical results are: **type safety** for $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$, i.e. evaluation of well-typed programs does not cause a run-time error (Section 3.3); $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ is a **conservative extension** of $\mathsf{MiniML}_{\mathsf{ref}}$ with respect to typing and operational semantics (Section 3.4). The usability of $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ for imperative multi-stage programming is exemplified in Section 4. Related work is discussed in Section 5.

Note 1.3 (*Notations and Conventions used throughout the paper*)

- Term equivalence, written $\equiv$, is α-conversion. $\mathrm{FV}(e)$ is the set of variables free in e. If E is a set of terms, then E_0 indicates the set of terms in E without free variables. Substitution of e_1 for x in e_2 (modulo $\equiv$) is written $e_2[x := e_1]$.
- m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} | i < m\}$ of its predecessors.
- $f : A \overset{fin}{\to} B$ means that f is a partial function from A to B with a finite domain, written $dom(f)$. We write $\{a_i : b_i | i \in m\}$ for the partial function mapping a_i to b_i (where the a_i must be different, i.e. $a_i = a_j$ implies $i = j$). We use the following operations on partial functions:
  $\emptyset$ is the everywhere undefined partial function;
  f_1, f_2 denotes the union of two partial functions with disjoint domains;
  $f, a : b$ denotes the extension of f to $a \notin dom(f)$;
  $f\{a = b\}$ denotes the update of f in $a \in dom(f)$.
- Given a declaration of a grammar such as $e := P_1 \mid \ldots \mid P_m$, we write $e+ = P_{m+1} \mid \ldots \mid P_{m+n}$ as a shorthand for $e := P_1 \mid \ldots \mid P_{m+n}$.
- Given a relation R, we write $\not{R}$ for the complement, and R^* for the reflexive transitive closure.

2 A Language with References: $\mathsf{MiniML}_{\mathsf{ref}}$

We describe the syntax, type system, and operational semantics of $\mathsf{MiniML}_{\mathsf{ref}}$, an extension of MiniML (Clement *et al.*, 1986) with ML-style references, and sketches the main steps in the proof of *weak soundness* (Wright & Felleisen, 1994; Harper & Stone, 1997), i.e. "well-typed programs cannot go wrong". The definitions are quite standard, but they are needed for formalizing and proving that $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ is a conservative extension of $\mathsf{MiniML}_{\mathsf{ref}}$ (see Section 3.4). This section is not essential to understanding Section 3, but comparing the two suggests how to define the multi-stage extension of a different programming language, and clarifies the overheads involved in proving type safety.

The set of $\mathsf{MiniML}_{\mathsf{ref}}$ terms is parametric in an infinite set of variables $x \in \mathsf{X}$ and

$$\frac{}{\Sigma;\Gamma \vdash x : t}\ \Gamma(x)=t \qquad \frac{}{\Sigma;\Gamma \vdash l : t}\ \Sigma(l)=t \qquad \frac{\Sigma;\Gamma, x : t_1 \vdash e : t_2}{\Sigma;\Gamma \vdash \lambda x.e : t_1 \to t_2}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : t_1 \to t_2 \quad \Sigma;\Gamma \vdash e_2 : t_1}{\Sigma;\Gamma \vdash e_1 e_2 : t_2} \qquad \frac{\Sigma;\Gamma, x : t \vdash e : t}{\Sigma;\Gamma \vdash \mathsf{fix}\ x.e : t} \qquad \frac{}{\Sigma;\Gamma \vdash \mathsf{z} : \mathsf{nat}}$$

$$\frac{\Sigma;\Gamma \vdash e : \mathsf{nat}}{\Sigma;\Gamma \vdash \mathsf{s}\ e : \mathsf{nat}} \qquad \frac{\Sigma;\Gamma \vdash e : \mathsf{nat} \quad \Sigma;\Gamma \vdash e_1 : t \quad \Sigma;\Gamma, x : \mathsf{nat} \vdash e_2 : t}{\Sigma;\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) : t}$$

$$\frac{\Sigma;\Gamma \vdash e : t}{\Sigma;\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ t} \qquad \frac{\Sigma;\Gamma \vdash e : \mathsf{ref}\ t}{\Sigma;\Gamma \vdash !e : t} \qquad \frac{\Sigma;\Gamma \vdash e_1 : \mathsf{ref}\ t \quad \Sigma;\Gamma \vdash e_2 : t}{\Sigma;\Gamma \vdash e_1 := e_2 : \mathsf{ref}\ t}$$

Fig. 1. Type System for $\mathsf{MiniML_{ref}}$. Σ signature for locations, Γ type assignment.

an infinite set of locations $l \in \mathsf{L}$

$$\begin{array}{rl} e \in \mathsf{E} ::= & x \mid \lambda x.e \mid e_1 e_2 \mid \mathsf{fix}\ x.e \mid \\ & \mathsf{z} \mid \mathsf{s}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \mid \\ & \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid l \end{array}$$

The first two lines list the MiniML terms: variables, abstraction, application, fixpoint for recursive definitions, zero, successor, and case-analysis on natural numbers. The third line lists the three SML operations on references, and constants l for locations. Locations are not allowed in user-written programs, but they are instrumental to the operational semantics of $\mathsf{MiniML_{ref}}$.

2.1 Type System

Figure 1 gives the type system of $\mathsf{MiniML_{ref}}$. The typing judgment has the form $\Sigma;\Gamma \vdash e : t$, read *"e has type t under the assignment* $\Sigma;\Gamma$*"*, where

- t is a *type*, i.e. $t \in \mathsf{T} ::= \ \mathsf{nat} \mid \mathsf{ref}\ t \mid t_1 \to t_2$
- $\Sigma : \mathsf{L} \overset{fin}{\to} \mathsf{T}$ is a *signature* (for locations only), written $\{l_i : \mathsf{ref}\ t_i | i \in m\}$.
- $\Gamma : \mathsf{X} \overset{fin}{\to} \mathsf{T}$ is a type assignment, written $\{x_i : t_i | i \in m\}$.

The type system enjoys the following basic properties:

Lemma 2.1 (*Weakening*)
1. $\Sigma;\Gamma \vdash e : t_2$ and x *fresh* imply $\Sigma;\Gamma, x : t_1 \vdash e : t_2$
2. $\Sigma;\Gamma \vdash e : t_2$ and l *fresh* imply $\Sigma, l : \mathsf{ref}\ t_1;\Gamma \vdash e : t_2$

Lemma 2.2 (*Substitution*)
$\Sigma;\Gamma \vdash e_1 : t_1$ and $\Sigma;\Gamma, x : t_1 \vdash e_2 : t_2$ imply $\Sigma;\Gamma \vdash e_2[x := e_1] : t_2$

2.2 Operational Semantics

We give a small-step operational semantics in the style advocated in (Wright & Felleisen, 1994). The semantics is given by a relation $\longmapsto\ \subset (\mathsf{S}\times\mathsf{E})\times((\mathsf{S}\times\mathsf{E})+\{\mathsf{err}\})$ defined in terms of a reduction $\longrightarrow \subset (\mathsf{S} \times \mathsf{Red}) \times ((\mathsf{S} \times \mathsf{E}) + \{\mathsf{err}\})$ and evaluation contexts $E \in \mathsf{EC}$ (see Figure 2). The special term err is analogous to the *Wrong* value in Milner-style untyped denotational semantics (Milner, 1978). The semantic categories involved in the definition of $\longmapsto$ are:

$$\begin{array}{rcl}
\mu, (\lambda x.e)v_2 & \longrightarrow & \mu, e[x := v_2] \\
\mu, \mathsf{fix}\, x.e & \longrightarrow & \mu, e[x := \mathsf{fix}\, x.e] \\
\mu, \mathsf{case\ z\ of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\, x \to e_2) & \longrightarrow & \mu, e_1 \\
\mu, \mathsf{case\ s}\ v\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\, x \to e_2) & \longrightarrow & \mu, e_2[x := v] \\
\mu, \mathsf{ref}\ v & \longrightarrow & (\mu, l : v), l \quad \text{with } l \notin dom(\mu) \\
\mu, !l & \longrightarrow & \mu, v \quad \text{if } v = \mu(l) \\
\mu, l := v & \longrightarrow & \mu\{l = v\}, l \quad \text{if } l \in dom(\mu) \\
\mu, r & \longrightarrow & \mathsf{err} \quad \text{otherwise}
\end{array}$$

$$\frac{\mu, r \longrightarrow \mu', e'}{\mu, E[r] \longmapsto \mu', E[e']}\ E \in \mathsf{EC} \qquad \frac{\mu, r \longrightarrow \mathsf{err}}{\mu, E[r] \longmapsto \mathsf{err}}\ E \in \mathsf{EC}$$

Fig. 2. Reduction for $\mathsf{MiniML}_{\mathsf{ref}}$. $E \in \mathsf{EC}$ Evaluation Context.

- values $v \in \mathsf{V} \subset \mathsf{E} ::= \lambda x.e \mid \mathsf{z} \mid \mathsf{s}\, v \mid l$
- stores $\mu \in \mathsf{S} \overset{\Delta}{=} \mathsf{L} \overset{fin}{\to} \mathsf{V}$.
- evaluation contexts $E \in \mathsf{EC}$, i.e.

$$\begin{array}{rl}
E \in \mathsf{EC} ::= & \square \mid E\, e_2 \mid v_1 E \mid \\
& \mathsf{s}\, E \mid \mathsf{case}\ E\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\, x \to e_2) \mid \\
& \mathsf{ref}\ E \mid !E \mid E := e_2 \mid v_1 := E
\end{array}$$

- redexes

$$\begin{array}{rl}
r \in \mathsf{Red} \subset \mathsf{E} ::= & x \mid v_1 v_2 \mid \mathsf{fix}\, x.e \mid \\
& \mathsf{case}\ v\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\, x \to e_2) \mid \\
& \mathsf{ref}\ v \mid !v \mid v_1 := v_2
\end{array}$$

The relations $\longrightarrow$ and $\longmapsto$ enjoy the following progress properties:

- in a configuration (μ, r) with $r \in \mathsf{Red}$, the relation $\longrightarrow$ can
  — either perform a computation step, yielding a configuration μ', e';
  — or report a run-time error err.
- in a configuration (μ, e) with $e \notin \mathsf{V}$, the relation $\longmapsto$ can
  — either perform a computation step, yielding a configuration μ', e';
  — or report a run-time error err.

Usually in $\mathsf{MiniML}_{\mathsf{ref}}$, one is interested only in execution of *complete programs*, i.e. $e \in \mathsf{E}_0$ (with no occurrences of l), starting from the empty store. We will establish that from such configurations $\longmapsto$ will only reach *closed configurations*, i.e. configurations in $\mathsf{S}_0 \times \mathsf{E}_0$. These properties are stated formally in the following Lemmas.

Lemma 2.3 (*Progress for* $\longrightarrow$)
If $(\mu, r) \in \mathsf{S} \times \mathsf{Red}$, then there exists d such that $\mu, r \longrightarrow d$.
If $\mu, r \longrightarrow \mu', e'$, then $dom(\mu) \subseteq dom(\mu')$ and $\mathrm{FV}(\mu', e') \subseteq \mathrm{FV}(\mu, r)$.

Lemma 2.4 (*Unique Decomposition*)

If $e \in \mathsf{E}$, then

- either $e \in \mathsf{V}$
- or exist (unique) $E \in \mathsf{EC}$ and $r \in \mathsf{Red}$ such that $e \equiv E[r]$.

Proof
By induction on $e \in \mathsf{E}$. $\Box$

Lemma 2.5 (*Progress for* $\longmapsto$)
If $(\mu, e) \in \mathsf{S} \times \mathsf{E}$, then either $e \in \mathsf{V}$ or there exists d such that $\mu, e \longmapsto d$.
If $\mu, e \longmapsto \mu', e'$, then $dom(\mu) \subseteq dom(\mu')$ and $\mathrm{FV}(\mu', e') \subseteq \mathrm{FV}(\mu, e)$.

Proof
By Unique Decomposition, Progress for $\longrightarrow$, and the fact that in an evaluation context the hole $\Box$ cannot be within the scope of a binder. $\Box$

Comparison with Wright and Felleisen. The *reduction semantics* of (Wright & Felleisen, 1994; Harper & Stone, 1997) models a run-time error by a *stuck* configuration, i.e. a (μ, e) such that $e \notin \mathsf{V}$ and $\mu, e \not\longmapsto$, instead of a reduction to err. For $\mathsf{MiniML}_{\mathsf{ref}}$ the two formulations are equivalent, but a more direct definition of run-time errors is preferable in the following cases: when one wants to distinguish among different run-time errors, as suggested in (Cardelli, 1997); when a configuration can progress both in err and in another configuration (this can happen in a parallel while language). Moreover, modeling a run-time error as a transition to err allows one to prove progress properties of the operational semantics independently from typing assumptions, such as those in (Harper & Stone, 1997).

2.3 Type Safety

We define well-formedness of closed stores in the obvious way:

- $\Sigma \models \mu \quad \overset{\Delta}{\Longleftrightarrow} \quad \begin{cases} dom(\Sigma) = dom(\mu) \text{ and,} \\ \mu(l) = v \wedge \Sigma(l) = \mathsf{ref}\ t \implies \Sigma; \emptyset \vdash v : t \end{cases}$

The following Lemmas summarize the main steps for proving weak soundness. These Lemmas are about closed configurations.

Lemma 2.6 (*Safety for* $\longrightarrow$)
If $\Sigma \models \mu$ and $\Sigma; \emptyset \vdash r : t$ with $r \in \mathsf{Red}$, then

- $\mu, r \not\longrightarrow \mathsf{err}$
- $\mu, r \longrightarrow \mu', e' \implies$ there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \models \mu'$ and $\Sigma'; \emptyset \vdash e' : t$

Proof
By case analysis on $r \in \mathsf{Red}$, using Substitution. $\Box$

Lemma 2.7 (*Replacement for Evaluation Contexts*)
If $\Sigma; \emptyset \vdash E[e] : t$ with $E \in \mathsf{EC}$, then there exists t_1 such that

- $\Sigma; \emptyset \vdash e : t_1$
- $\Sigma'; \emptyset \vdash e' : t_1 \implies \Sigma'; \emptyset \vdash E[e'] : t$ for any $\Sigma' \supseteq \Sigma$ and e'

Proof
By induction on the structure of $E \in \mathsf{EC}$, using Weakening and the fact that in an evaluation context the hole $\square$ cannot be within the scope of a binder. $\square$

Lemma 2.8 (*Safety for* $\longmapsto$)
If $\Sigma \models \mu$ and $\Sigma; \emptyset \vdash e : t$, then

- $\mu, e \not\longmapsto \mathsf{err}$
- $\mu, e \longmapsto \mu', e' \implies$ there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \models \mu'$ and $\Sigma'; \emptyset \vdash e' : t$

Proof
By Unique Decomposition, Replacement, and Safety for $\longrightarrow$. $\square$

Theorem 2.9 (*Weak Soundness*)
If $\emptyset; \emptyset \vdash e : t$, then $(\emptyset, e) \not\longmapsto^* \mathsf{err}$.

Proof
$(\emptyset, e) \longmapsto{}^m d$ implies $d \not\equiv \mathsf{err}$, by induction on m, using Safety for $\longmapsto$. $\square$

3 A Multi-Stage Language with References: $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$

We describe the syntax, type system, and operational semantics of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$, a multi-stage extension of $\mathsf{MiniML}_{\mathsf{ref}}$, and establishes *weak soundness* (Wright & Felleisen, 1994), i.e. "well-typed programs cannot go wrong".

The set of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ terms is an extension of the set of $\mathsf{MiniML}_{\mathsf{ref}}$ terms

$$e \in \mathsf{E} \;\; += \;\; \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e \mid \%e \mid \mathsf{let_c}\ x = e_1\ \mathsf{in}\ e_2 \mid (x)e$$

The new constructs are: the three multi-stage constructs of MetaML, Brackets $\langle e \rangle$, Escape $\tilde{}e$ and Run $\mathsf{run}\ e$; Cross-Stage Persistence $\%e$; Letc-binding $\mathsf{let_c}\ x = e_1\ \mathsf{in}\ e_2$ for terms of closed type; and Bind $(x)e$ for handling scope extrusion. In well-typed user programs one could always eliminate Bind, by replacing $(x)e$ with $e[x := \bot]$, where $\bot = \mathsf{fix}\ x.x$. However, Bind could be reintroduced during evaluation.

Note 3.1 (*Derived notation*)
- $(X)e$ is Iterated Bind, i.e. $(x_1)\dots(x_m)e$ with x_i an enumeration of $X \subseteq_{fin} \mathsf{X}$
- $\bullet e$ is the Bind-Closure of e, i.e. $(X)e$ with $X = \mathrm{FV}(e)$.

Remark 3.2 (*Bind as Dead Code annotation*)
Operationally, Bind $(x)e$ is *equivalent* to $e[x := \mathsf{fault}]$, where fault is a closed term which causes a run-time error when evaluated at level 0 and is a value at levels > 0. An example of such a term is $\mathsf{z}\ \mathsf{z}$, but not $\tilde{}\mathsf{z}$. In fact, Bind-bound variables never get substituted during evaluation, because intuitively they have already been substituted with fault; evaluation *at any level* can go under Bind (see the BNF for evaluation contexts E^n_i), because intuitively $(x)e$ is a substitution instance of e. If evaluation of a *complete program* e_0 does not cause a run-time error, then all occurrences of variables that are (or get) bound by Bind must be *dead code* in e_0, i.e. there will be no attempt to evaluate such occurrences (and their residuals) at level 0. The following points anticipate the discussion of Bind in relation to the operational semantics and the type system of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$.

- The operational semantics of Figure 4 uses Bind to prevent scope extrusion when a location is initialized or assigned. In fact, what gets stored is always the Bind-closure $\bullet v^0$ of a value. Therefore, if a free variable in v^0 was in the scope of an enclosing binder, it gets bound by Bind instead of becoming free. In other words, the operational semantics assumes that the free variables in v^0 are dead code, and uses Bind to make the assumption explicit. If the assumption is wrong, then a run-time error will eventually occur. According to the Weak Soundness Theorem 3.11, in the evaluation of a well-typed program no run-time errors can occur, thus the Bind annotation is used correctly.
- The typing rules for Bind in Figure 3 say that "in a term of *closed type* at level n all free variables declared at level $> n$ must be dead code". While Type Safety (see Section 3.3) tells us that the *dead code annotations* allowed by the type system are operationally sound. Notice that, type-theoretically, $(x)e$ is not equivalent to $e[x := \mathsf{fault}]$. In fact, $(x)e$ is typable whenever $e[x := \mathsf{fault}]$ is (typability of $e[x := \mathsf{fault}]$ implies $x \notin \mathrm{FV}(e)$), but the converse fails.

3.1 Type System

Figure 3 gives the type system of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$. A typing judgment has the form $\Sigma; \Delta; \Gamma \vdash_n e : t$, read *"e has type t at level n under the assignment* $\Sigma; \Delta; \Gamma$*"*, where

- t is a *type* and c is a *closed type*, i.e.

$$\begin{aligned} t \in \mathsf{T} &::= \;\; \mathsf{nat} \mid t_1 \to t_2 \mid [t] \mid \mathsf{ref}\ c \mid \langle t \rangle \\ c \in \mathsf{C} &::= \;\; \mathsf{nat} \mid t_1 \to c_2 \mid [t] \mid \mathsf{ref}\ c \end{aligned}$$

- $\Sigma : \mathsf{L} \overset{fin}{\to} \mathsf{C}$ is a *signature* (for locations only), written $\{l_i : \mathsf{ref}\ c_i | i \in m\}$.
- $\Delta : \mathsf{X} \overset{fin}{\to} (\mathsf{C} \times \mathsf{N})$ and $\Gamma : \mathsf{X} \overset{fin}{\to} (\mathsf{T} \times \mathsf{N})$ are type-and-level assignments, written $\{x_i : c_i^{n_i} | i \in m\}$ and $\{x_i : t_i^{n_i} | i \in m\}$ respectively. We use the following operations on type-and-level assignments:
  $\{x_i : t_i^{n_i} | i \in m\}^{op\, n} \overset{\Delta}{=} \{x_i : t_i^{n_i\, op\, n} | i \in m\}$, with op binary operation on N;
  $\{x_i : t_i^{n_i} | i \in m\}^{R\, n} \overset{\Delta}{=} \{x_i : t_i^{n_i} | n_i \, R\, n \wedge i \in m\}$, with R binary relation on N.

The type system enjoys the following basic properties:

Lemma 3.3 (*Weakening*)

1. $\Sigma; \Delta; \Gamma \vdash_n e : t_2$ and x *fresh* imply $\Sigma; \Delta; \Gamma, x : t_1^m \vdash_n e : t_2$
2. $\Sigma; \Delta; \Gamma \vdash_n e : t_2$ and x *fresh* imply $\Sigma; \Delta, x : c_1^m; \Gamma \vdash_n e : t_2$
3. $\Sigma; \Delta; \Gamma \vdash_n e : t_2$ and l *fresh* imply $\Sigma, l : \mathsf{ref}\ c_1; \Delta; \Gamma \vdash_n e : t_2$

Lemma 3.4 (*Substitution*)

$$\text{subst.}\Gamma\ \frac{\Sigma; \Delta; \Gamma \vdash_m e_1 : t_1 \qquad \Sigma; \Delta; \Gamma, x : t_1^m \vdash_n e_2 : t_2}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2} \qquad \text{subst.}\Delta\ \frac{\Sigma; \Delta^{\leq m}; \emptyset \vdash_m e_1 : c_1 \qquad \Sigma; \Delta, x : c_1^m; \Gamma \vdash_n e_2 : t_2}{\Sigma; \Delta; \Gamma \vdash_n e_2[x := e_1] : t_2}$$

Comparison with $\mathsf{MiniML}_{\mathsf{ref}}$. The typing judgments of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ have two additional features:

$$\text{(var)}\ \frac{}{\Sigma;\Delta;\Gamma\vdash_n x:t}\ (\Delta,\Gamma)(x)=t^n \qquad \text{(cst)}\ \frac{}{\Sigma;\Delta;\Gamma\vdash_n l:c}\ \Sigma(l)=c$$

$$\text{(lam)}\ \frac{\Sigma;\Delta;\Gamma,x:t_1^n\vdash_n e:t_2}{\Sigma;\Delta;\Gamma\vdash_n \lambda x.e:t_1\to t_2} \qquad \text{(app)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e_1:t_1\to t_2\quad \Sigma;\Delta;\Gamma\vdash_n e_2:t_1}{\Sigma;\Delta;\Gamma\vdash_n e_1e_2:t_2}$$

$$\text{(fix)}\ \frac{\Sigma;\Delta;\Gamma,x:t^n\vdash_n e:t}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{fix}\,x.e:t} \qquad \text{(zero)}\ \frac{}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{z}:\mathsf{nat}} \qquad \text{(succ)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:\mathsf{nat}}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{s}\,e:\mathsf{nat}}$$

$$\text{(case)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:\mathsf{nat}\quad \Sigma;\Delta;\Gamma\vdash_n e_1:t\quad \Sigma;\Delta;\Gamma,x:\mathsf{nat}^n\vdash_n e_2:t}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{z}\to e_1\mid \mathsf{s}\ x\to e_2):t}$$

$$\text{(ref)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:c}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{ref}\ e:\mathsf{ref}\ c} \qquad \text{(deref)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:\mathsf{ref}\ c}{\Sigma;\Delta;\Gamma\vdash_n\, !e:c}$$

$$\text{(setref)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e_1:\mathsf{ref}\ c\quad \Sigma;\Delta;\Gamma\vdash_n e_2:c}{\Sigma;\Delta;\Gamma\vdash_n e_1:=e_2:\mathsf{ref}\ c}$$

$$\text{(brck)}\ \frac{\Sigma;\Delta;\Gamma\vdash_{n+1} e:t}{\Sigma;\Delta;\Gamma\vdash_n \langle e\rangle:\langle t\rangle} \qquad \text{(esc)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:\langle t\rangle}{\Sigma;\Delta;\Gamma\vdash_{n+1} \tilde{}e:t} \qquad \text{(run)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:[\langle t\rangle]}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{run}\ e:[t]}$$

$$\text{(csp)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:t}{\Sigma;\Delta;\Gamma\vdash_{n+1} \%e:t} \qquad \text{(letc)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e_1:c_1\quad \Sigma;\Delta,x:c_1^n;\Gamma\vdash_n e_2:t_2}{\Sigma;\Delta;\Gamma\vdash_n \mathsf{let_c}\ x=e_1\ \mathsf{in}\ e_2:t_2}$$

$$\text{(bind1)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:t}{\Sigma;\Delta;\Gamma\vdash_n (x)e:t}\ x\text{ fresh} \qquad \text{(bind2)}\ \frac{\Sigma;\Delta;\Gamma,x:t_1^m\vdash_n e:c}{\Sigma;\Delta;\Gamma\vdash_n (x)e:c}\ m>n$$

$$\text{(bind3)}\ \frac{\Sigma;\Delta,x:c_1^m;\Gamma\vdash_n e:c}{\Sigma;\Delta;\Gamma\vdash_n (x)e:c}\ m>n \qquad \text{(closI1)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:c}{\Sigma;\Delta;\Gamma\vdash_n e:[c]}$$

$$\text{(closI2)}\ \frac{\Sigma;\Delta^{\le n};\emptyset\vdash_n e:t}{\Sigma;\Delta;\Gamma\vdash_n e:[t]} \qquad \text{(closE)}\ \frac{\Sigma;\Delta;\Gamma\vdash_n e:[t]}{\Sigma;\Delta;\Gamma\vdash_n e:t}$$

Fig. 3. Type System for $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$. Σ signature for locations, Δ and Γ type-and-level assignments with disjoint domains.

- the level information, typical of multi-level languages like $\lambda^{\bigcirc}$ (Davies, 1996);
- the splitting of type-and-level assignments in two parts (Δ and Γ), borrowed from $\lambda^{\Box}$ (Davies & Pfenning, 1996); The difference between a declaration in Δ and the same declaration in Γ is expressed in the Substitution Lemma 3.4.

The level information is essential to express the typing rules for the Code type constructor, while the splitting is essential to express the typing rules for the Closed type constructor (and more generally for closed types). The $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ typing rules for $\mathsf{MiniML}_{\mathsf{ref}}$ term constructs are closely related to those of $\mathsf{MiniML}_{\mathsf{ref}}$, namely:

- the typing rules operate uniformly at every level;
- the binders always bind variables declared in Γ;
- but the typing for the operations on references are restricted to closed types.

Finally, the typing rules for the new term constructs of $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ are mainly related to typing rules of $\lambda^{\bigcirc}$ or $\lambda^{\Box}$:

- (brck) and (esc) correspond directly to the following rules of $\lambda^{\bigcirc}$

$$\bigcirc\text{-I}\ \frac{\Gamma\vdash_{n+1} e:t}{\Gamma\vdash_n \mathsf{next}\ e:\bigcirc t} \qquad \bigcirc\text{-E}\ \frac{\Gamma\vdash_n e:\bigcirc t}{\Gamma\vdash_{n+1} \mathsf{prev}\ e:t}$$

- (run) and (csp) are consistent with the rules in Section 1.3; in MetaML cross-stage persistence is not embodied in a term construct like $\%e$, but it is implicit in the typing rule for variables $\dfrac{}{\Gamma \vdash_m x : t}\ \Gamma(x) = t^n$ and $m \geq n$
- (letc) and (closI2) are related to the following rules of $\lambda^{\Box}$

$$\Box\text{-I}\ \frac{\Delta; \emptyset \vdash e : t}{\Delta; \Gamma \vdash \mathsf{box}\ \ e : \Box t} \qquad \Box\text{-E}\ \frac{\Delta; \Gamma \vdash e_1 : \Box t_1 \quad \Delta, x : t; \Gamma \vdash e_2 : t_2}{\Delta; \Gamma \vdash \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 : t_2}$$

  Notice that $\mathsf{let_c}\ x = e_1\ \mathsf{in}\ e_2$ is not equivalent to $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \stackrel{\Delta}{=} (\lambda x.e_2)\ e_1$, since in the former x is declared in Δ, while in the latter it is declared in Γ.
- (closE) says that $[t]$ is a subset of t, and (closI1) says that $[c]$ and c are equal, i.e. they classify the same set of terms. These rules and (closI2) are not syntax directed, therefore the typing rules do not directly induce a type inference algorithm
- The typing of $(x)e$ has been discussed in Remark 3.2.

3.2 Operational Semantics

We give a small-step operational semantics in the style advocated in (Harper & Stone, 1997; Wright & Felleisen, 1994). Since $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ is a multi-level language, the semantics is given by a family $\stackrel{n}{\longmapsto}\ \subset (\mathsf{S}_0 \times \mathsf{E}) \times ((\mathsf{S}_0 \times \mathsf{E}) + \{\mathsf{err}\})$ of relations, one for each level $n \in \mathsf{N}$, defined in terms of two reductions $\stackrel{i}{\longrightarrow} \subset (\mathsf{S}_0 \times \mathsf{Red}^i) \times ((\mathsf{S}_0 \times \mathsf{E}) + \{\mathsf{err}\})$, with $i \in \{0, 1\}$, and evaluation contexts $E^n_i \in \mathsf{EC}^n_i$ (see Figure 4). The semantic categories involved in the definition of $\stackrel{n}{\longmapsto}$ are:

- $v^n \in \mathsf{V}^n \subset \mathsf{E}$ values at level $n \in \mathsf{N}$, i.e.

$$\begin{array}{rcl}
v^0 \in \mathsf{V}^0 & ::= & \lambda x.e \mid \mathsf{z} \mid \mathsf{s}\ v^0 \mid l \mid \langle v^1 \rangle \mid (x)v^0 \\
v^{n+1} \in \mathsf{V}^{n+1} & ::= & x \mid \lambda x.v^{n+1} \mid v^{n+1}_1 v^{n+1}_2 \mid \mathsf{fix}\ x.v^{n+1} \mid \\
 & & \mathsf{z} \mid \mathsf{s}\ v^{n+1} \mid \mathsf{case}\ v^{n+1}\ \mathsf{of}\ (\mathsf{z} \to v^{n+1}_1 \mid \mathsf{s}\ x \to v^{n+1}_2) \mid \\
 & & \mathsf{ref}\ v^{n+1} \mid !v^{n+1} \mid v^{n+1}_1 := v^{n+1}_2 \mid l \mid \\
 & & \langle v^{n+2} \rangle \mid \mathsf{run}\ v^{n+1} \mid \%v^n \mid \mathsf{let_c}\ x = v^{n+1}_1\ \mathsf{in}\ v^{n+1}_2 \mid (x)v^{n+1} \\
v^{n+2} \in \mathsf{V}^{n+2} & += & \tilde{}\,v^{n+1}
\end{array}$$

- closed stores $\mu \in \mathsf{S}_0 \stackrel{\Delta}{=} \mathsf{L} \stackrel{fin}{\to} \mathsf{V}^0_0$
- evaluation contexts $E^n_i \in \mathsf{EC}^n_i$ at level $n \in \mathsf{N}$ with hole at level $i \in \{0, 1\}$, i.e.

$$\begin{array}{rcl}
E^i_i \in \mathsf{EC}^i_i & ::= & \Box \\
E^n_i \in \mathsf{EC}^n_i & += & E^n_i e_2 \mid v^n_1 E^n_i \mid \mathsf{s}\ E^n_i \mid \mathsf{case}\ E^n_i\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \mid \\
 & & \mathsf{ref}\ E^n_i \mid !E^n_i \mid E^n_i := e_2 \mid v^n_1 := E^n_i \mid \\
 & & \langle E^{n+1}_i \rangle \mid \mathsf{run}\ E^n_i \mid \mathsf{let_c}\ x = E^n_i\ \mathsf{in}\ e_2 \mid (x)E^n_i \\
E^{n+1}_i \in \mathsf{EC}^{n+1}_i & += & \lambda x.E^{n+1}_i \mid \mathsf{fix}\ x.E^{n+1}_i \mid \\
 & & \mathsf{case}\ v^{n+1}\ \mathsf{of}\ (\mathsf{z} \to E^{n+1}_i \mid \mathsf{s}\ x \to e_2) \mid \\
 & & \mathsf{case}\ v^{n+1}\ \mathsf{of}\ (\mathsf{z} \to v^{n+1}_1 \mid \mathsf{s}\ x \to E^{n+1}_i) \mid \\
 & & \%E^n_i \mid \tilde{}\,E^n_i \mid \mathsf{let_c}\ x = v^{n+1}_1\ \mathsf{in}\ E^{n+1}_i
\end{array}$$

- redexes $r^i \in \mathsf{Red}^i \subset \mathsf{E}$ at level $i \in \{0,1\}$, i.e.

$$\begin{array}{rl} r^0 \in \mathsf{Red}^0 ::= & x \mid v_1^0 v_2^0 \mid \mathsf{fix}\, x.e \mid \mathsf{case}\ v^0\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \mid \\ & \mathsf{ref}\ v^0 \mid !v^0 \mid v_1^0 := v_2^0 \mid \\ & \tilde{}e \mid \mathsf{run}\ v^0 \mid \%e \mid \mathsf{let_c}\ x = v_1^0\ \mathsf{in}\ e_2 \\ r^1 \in \mathsf{Red}^1 ::= & \tilde{}v^0 \end{array}$$

The reduction of $\mathsf{run}\ v^0$ makes use of an auxiliary operation on values:

Definition 3.5 (*Demotion*)
$\downarrow_n : \mathsf{V}^{n+1} \to \mathsf{E}$ is defined by induction on $v^{n+1} \in \mathsf{V}^{n+1}$

$$\begin{array}{rcl} x \downarrow_n & \triangleq & x \\ (\%v^0) \downarrow_0 & \triangleq & v^0[x := \%x | x \in \mathrm{FV}(v^0)] \\ (\%v^{n+1}) \downarrow_{n+1} & \triangleq & \%(v^{n+1} \downarrow_n) \\ \langle v^{n+2} \rangle \downarrow_n & \triangleq & \langle v^{n+2} \downarrow_{n+1} \rangle \\ (\tilde{}v^{n+1}) \downarrow_{n+1} & \triangleq & \tilde{}(v^{n+1} \downarrow_n) \end{array}$$

and $\downarrow_n$ commutes with the other term constructs.

In relation to the type system, $\downarrow_n$ lowers levels but preserves types, i.e.

Proposition 3.6 (*Demotion*)
If $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_{n+1} v : t$ and $v \in \mathsf{V}^{n+1}$, then $\Sigma; \Delta; \Gamma \vdash_n v \downarrow_n : t$.

Proof
By induction on derivation of $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_{n+1} v : t$. $\square$

The following property captures the *reflective* nature of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$, i.e. "a term at level n *is* a value at level $n+1$", on values where $\downarrow_n$ is the identity:

Proposition 3.7 (*Promotion*)
If $\Sigma; \Delta; \Gamma \vdash_n e : t$, then $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_{n+1} e : t$ and $e \in \mathsf{V}^{n+1}$ and $e \downarrow_n \equiv e$.

Proof
By induction on derivation of $\Sigma; \Delta; \Gamma \vdash_n e : t$. $\square$

The relations $\xrightarrow{i}$ and $\overset{n}{\longmapsto}$ enjoy the following progress properties:

- in a configuration (μ, r^i) with $r^i \in \mathsf{Red}^i$, the relation $\xrightarrow{i}$ can
  — either perform a computation step, yielding a configuration μ', e';
  — or report a run-time error err.
- in a configuration (μ, e) with $e \notin \mathsf{V}^n$, the relation $\overset{n}{\longmapsto}$ can
  — either perform a computation step, yielding a configuration μ', e';
  — or report a run-time error err.

These and other properties are stated formally in the following Lemmas.

Lemma 3.8 (*Progress for* $\xrightarrow{i}$)
If $(\mu, r^i) \in \mathsf{S}_0 \times \mathsf{Red}^i$, then there exists d such that $\mu, r^i \xrightarrow{i} d$.
If $\mu, r^i \xrightarrow{i} \mu', e'$, then $dom(\mu) \subseteq dom(\mu')$ and $\mathrm{FV}(e') \subseteq \mathrm{FV}(r^i)$.

$$\begin{array}{rcl}
\mu, ((X)\lambda x.e)v_2^0 & \xrightarrow{0} & \mu, ((X)e)[x := v_2^0] \\
\mu, \mathsf{fix}\ x.e & \xrightarrow{0} & \mu, e[x := \mathsf{fix}\ x.e] \\
\mu, \mathsf{case}\ (X)\mathsf{z}\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) & \xrightarrow{0} & \mu, e_1 \\
\mu, \mathsf{case}\ (X)\mathsf{s}\ v^0\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) & \xrightarrow{0} & \mu, e_2[x := \bullet v^0] \\
\mu, \mathsf{ref}\ v^0 & \xrightarrow{0} & (\mu, l : \bullet v^0), l \quad \text{with } l \notin \mathit{dom}(\mu) \\
\mu, !(X)l & \xrightarrow{0} & \mu, v^0 \quad \text{if } v^0 = \mu(l) \\
\mu, (X)l := v^0 & \xrightarrow{0} & \mu\{l = \bullet v^0\}, l \quad \text{if } l \in \mathit{dom}(\mu) \\
\mu, \mathsf{run}\ (X)\langle v^1 \rangle & \xrightarrow{0} & \mu, \bullet(v^1 \downarrow_0) \\
\mu, \mathsf{let_c}\ x = v^0\ \mathsf{in}\ e_2 & \xrightarrow{0} & \mu, e_2[x := \bullet v^0] \\
\mu, r^0 & \xrightarrow{0} & \mathsf{err} \quad \text{otherwise}
\end{array}$$

$$\begin{array}{rcl}
\mu, \tilde{\ }(X)\langle v^1 \rangle & \xrightarrow{1} & \mu, (X)v^1 \\
\mu, r^1 & \xrightarrow{1} & \mathsf{err} \quad \text{otherwise}
\end{array}$$

$$\frac{\mu, r^i \xrightarrow{i} \mu', e'}{\mu, E_i^n[r^i] \xmapsto{n} \mu', E_i^n[e']}\ E_i^n \in \mathsf{EC}_i^n \qquad \frac{\mu, r^i \xrightarrow{i} \mathsf{err}}{\mu, E_i^n[r^i] \xmapsto{n} \mathsf{err}}\ E_i^n \in \mathsf{EC}_i^n$$

Fig. 4. Reductions for $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$, where: $\bullet v^0$ Bind-closure of v^0, $\downarrow_0$: $\mathsf{V}^1 \to \mathsf{E}$ Demotion, and $E_i^n \in \mathsf{EC}_i^n$ Evaluation Context. In a reduction rule bind-closure is applied to (potentially open) terms that can be assigned a closed type, under the assumption that the redex is well-typed (see Remark 3.2).

Proof

By case analysis on $r^i \in \mathsf{Red}^i$, and the fact that what gets in or out a closed store μ is always a closed value. $\square$

Lemma 3.9 (*Unique Decomposition*)

If $n \in \mathsf{N}$ and $e \in \mathsf{E}$, then

- either $e \in \mathsf{V}^n$
- or exist (unique) $i \in \{0, 1\}$ and $E_i^n \in \mathsf{EC}_i^n$ and $r^i \in \mathsf{Red}^i$ such that $e \equiv E_i^n[r^i]$.

Proof

By induction on $e \in \mathsf{E}$. $\square$

Lemma 3.10 (*Progress for* $\xmapsto{n}$)

If $(\mu, e) \in \mathsf{S}_0 \times \mathsf{E}$, then either $e \in \mathsf{V}^n$ or there exists d such that $\mu, e \xmapsto{n} d$.
If $\mu, e \xmapsto{n} \mu', e'$, then $\mathit{dom}(\mu) \subseteq \mathit{dom}(\mu')$ and $\mathrm{FV}(e') \subseteq \mathrm{FV}(e)$.

Proof

By Unique Decomposition and Progress for $\xrightarrow{i}$. $\square$

Comparison with $\mathsf{MiniML_{ref}}$. Usually in $\mathsf{MiniML^{meta}_{ref}}$, one is interested only in execution (at level 0) of *complete programs*, i.e. $e \in \mathsf{E}_0$ (with no occurrences of l), starting from the empty store. Like in $\mathsf{MiniML_{ref}}$, the Progress Lemma for $\overset{0}{\longmapsto}$ tells us that from such configurations we will only reach *closed configurations*. However, there is an important difference: in an evaluation context for $\mathsf{MiniML^{meta}_{ref}}$ the hole can be within the scope of a binder, e.g. consider $\langle \lambda x.\tilde{}\square \rangle \in \mathsf{EC}^0_0$, and so $\mathrm{FV}(E^n_i[r^i]) = \emptyset$ does not imply $\mathrm{FV}(r^i) = \emptyset$. Therefore, for proving type safety in $\mathsf{MiniML^{meta}_{ref}}$ it does not suffice to consider only *closed redexes*.

3.3 Type Safety

This section establishes *weak soundness* for $\mathsf{MiniML^{meta}_{ref}}$, namely

Theorem 3.11 (*Weak Soundness*)
If $\emptyset; \emptyset; \emptyset \vdash_0 e : t$, then $(\emptyset, e) \not\overset{0}{\longmapsto}{}^* \mathsf{err}$.

We write $[t]^n$ to denote the n-fold application of $[\_]$, i.e. $[t]^0 \overset{\Delta}{\equiv} t$ and $[t]^{n+1} \overset{\Delta}{\equiv} [[t]^n]$. The following Lemmas are used in the proof of Safety for $\overset{i}{\longrightarrow}$.

Lemma 3.12 (*Structure*)
Given $v \in \mathsf{V}^0$ and $t \in \mathsf{T}$, if $\Sigma; \Delta; \Gamma \vdash_0 v : t$ is derivable, then for some $X \subseteq_{fin} \mathsf{X}$ and type-and-level assignments Δ_1 and Γ_1 s.t. $dom(\Delta_1, \Gamma_1) \subseteq X$ one of the following (mutually exclusive) possibilities holds:

1. $v \equiv (X)\lambda x.e$ and $t \equiv [t_1 \to t_2]^n$ and $\Sigma; \Delta, \Delta_1^{+1}; \Gamma, \Gamma_1^{+1}, x : t_1^0 \vdash_0 e : t_2$
2. $v \equiv (X)\mathsf{z}$ and $t \equiv [\mathsf{nat}]^n$
3. $v \equiv (X)\mathsf{s}\, v'$ and $t \equiv [\mathsf{nat}]^n$ and $\Sigma; \Delta, \Delta_1^{+1}; \Gamma, \Gamma_1^{+1} \vdash_0 v' : \mathsf{nat}$
4. $v \equiv (X)l$ and $t \equiv [\mathsf{ref}\ c]^n$ and $\Sigma(l) = \mathsf{ref}\ c$
5. $v \equiv (X)\langle v' \rangle$ and $t \equiv [\langle t' \rangle]^n$ and $\Sigma; \Delta, \Delta_1^{+1}; \Gamma, \Gamma_1^{+1} \vdash_1 v' : t'$

Proof
By induction on the derivation of $\Sigma; \Delta; \Gamma \vdash_0 v : t$. Because of the structure of a $v \in \mathsf{V}^0$ we have the following cases for the last typing rule in the derivation.

- Base cases: the following cases do not need the induction hypothesis.
  (cst) (lam) (zero) (succ) (brck)
- Inductive steps: the following cases use the induction hypothesis.
  (bind1) (bind2) (bind3)
  (closI1) (closI2) (closE)

We consider only one base case and two inductive steps. The other cases are similar.

Case (lam).
If the last typing rule is (lam), then it must be of the form

$$\frac{\Sigma; \Delta; \Gamma, x : t_1^0 \vdash_0 e : t_2}{\Sigma; \Delta; \Gamma \vdash_0 \lambda x.e : t_1 \to t_2}$$

where $t \equiv t_1 \to t_2$ and $v \equiv \lambda x.e$. Then possibility 1 applies with $X = \emptyset$, $n = 0$, $\Delta_1 = \emptyset$, $\Gamma_1 = \emptyset$.

Case (bind2).

If the last typing rule is (bind2), then it must be of the form

$$\frac{\Sigma; \Delta; \Gamma, x : t_1^{m+1} \vdash_0 v' : t}{\Sigma; \Delta; \Gamma \vdash_0 (x)v' : t}$$

where $t \in \mathsf{C}$ and $v \equiv (x)v'$. By induction hypothesis on the premise there are 5 possibilities; we consider only possibility 1. Then $v' \equiv (X)\lambda x'.e$ and $t \equiv [t_2 \to t_3]^n$ and $\Sigma; \Delta, \Delta_1^{+1}; \Gamma, x : t_1^{m+1}, \Gamma_1^{+1}, x' : t_2^0 \vdash_0 e : t_3$ with $dom(\Delta_1, \Gamma_1) \subseteq X$. We can write v as $(x, X)\lambda x'.e$, to prove possibility 1 for the conclusion we must show that there exist Δ_2, Γ_2 such that $dom(\Delta_2, \Gamma_2) \subseteq (x, X)$ and $\Sigma; \Delta, \Delta_2^{+1}; \Gamma, \Gamma_2^{+1}, x' : t_2^0 \vdash_0 e : t_3$. We take $\Delta_2 \equiv \Delta_1$ and $\Gamma_2 \equiv \Gamma_1, x : t_1^m$.

Case (closI2).

If the last typing rule is (closI2), then it must be of the form

$$\frac{\Sigma; \Delta^{\leq 0}; \emptyset \vdash_0 v : t}{\Sigma; \Delta; \Gamma \vdash_0 v : [t]}$$

By induction hypothesis on the premise there are 5 possibilities; we consider only possibility 5. Then $v \equiv (X)\langle v' \rangle$ and $t \equiv [\langle t' \rangle]^n$ and $\Sigma; \Delta^{\leq 0}, \Delta_1^{+1}; \Gamma_1^{+1} \vdash_1 v' : t'$ with $dom(\Delta_1, \Gamma_1) \subseteq X$. To prove possibility 5 for the conclusion we show that there exist n_2, Δ_2, Γ_2 such that $dom(\Delta_2, \Gamma_2) \subseteq X$ and $[t] \equiv [\langle t' \rangle]^{n_2}$ and $\Sigma; \Delta, \Delta_2^{+1}; \Gamma, \Gamma_2^{+1} \vdash_1 v' : t'$. By lemma 3.3 we have $\Sigma; \Delta, \Delta_1^{+1}; \Gamma, \Gamma_1^{+1} \vdash_1 v' : t'$, hence we take $n_2 \equiv n + 1$ and $\Delta_2 \equiv \Delta_1$ and $\Gamma_2 \equiv \Gamma_1$.

□

Lemma 3.13 (*Closedness*)
If t is not closed and v is of the form $\lambda x.e$ or $\langle v^1 \rangle$, then

1. $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 (X)v : [t]^{n+1}$ implies $\mathrm{FV}(v) = \emptyset$
2. $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 (X)v : t$ implies $X \cap \mathrm{FV}(v) = \emptyset$

Proof
By induction on the derivation of the typing judgments. For part 1 we have the following cases for the last typing rule in the derivation:

- If the last rule is (bind1) or (bind2) or (bind3) or (closE), then the result follows by part 1 for the premise of the rule.
- If $n > 0$ and the last rule is (closI1) or (closI2), then the result follows again by part 1 for the premise of the rule.
- The only other case is $n = 0$ and the last rule is (closI2), in fact (closI1) is not applicable because t is not closed. The premise of the rule is $\Sigma; \emptyset; \emptyset \vdash_0 (X)v : t$, therefore $\mathrm{FV}((X)v) = \emptyset$, and by part 2 we have $X \cap \mathrm{FV}(v) = \emptyset$. So we conclude that $\mathrm{FV}(v) = \emptyset$.

For part 2, if $X = \emptyset$ the conclusion is immediate. Otherwise, let X be the sequence x, X'. The last typing rule applied must be (bind1) or (closE), because t is not closed. In the first case, part 2 on the premise implies $X' \cap \mathrm{FV}(v) = \emptyset$ and $x \notin FV((X')v)$, hence $X \cap \mathrm{FV}(v) = \emptyset$. In the second case, part 1 on the premise gives the result. □

The rest of the proof of weak soundness follows the same pattern sketched for $\mathsf{MiniML}_{\mathsf{ref}}$. In the case of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$, it does not suffice to consider only closed redexes. However, by our use of Bind in the reduction rules for the operations on references, we need to consider only closed stores. We define well-formedness of closed stores in the obvious way:

- $\Sigma \models \mu \quad \overset{\Delta}{\iff} \quad \begin{cases} \mathit{dom}(\Sigma) = \mathit{dom}(\mu) \text{ and,} \\ \mu(l) = v \wedge \Sigma(l) = \mathsf{ref}\ c \implies \Sigma; \emptyset; \emptyset \vdash_0 v : c \end{cases}$

Lemma 3.14 (*Safety for* $\xrightarrow{i}$)
If $\Sigma \models \mu$ and $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_i r^i : t$ with $r^i \in \mathsf{Red}^i$, then

- $\mu, r^i \not\xrightarrow{i} \mathsf{err}$
- $\mu, r^i \xrightarrow{i} \mu', e' \implies$ there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \models \mu'$ and $\Sigma'; \Delta^{+1}; \Gamma^{+1} \vdash_i e' : t$

Proof
By case analysis on $r^i \in \mathsf{Red}^i$ and induction on the derivation of $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_i r^i : t$. If the last rule in the derivation of $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_i r^i : t$ is not syntax directed (i.e. closI1, closI2 or closE), then we can simply exploit the induction hypothesis for the derivation of the premise (which must be of the form $\Sigma; \Delta_1^{+1}; \Gamma_1^{+1} \vdash_i r^i : t_1$, with $\Delta_1 \subseteq \Delta$ and $\Gamma_1 \subseteq \Gamma$).

In all other cases we cannot exploit the induction hypothesis, instead we use basic properties of the type systems and the following Lemmas: Structure (in the cases $v_1^0 v_2^0$, $\mathsf{case}\ v^0\ \mathsf{of}\ (\mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2)$ $!v^0$, $v_1^0 := v_2^0$, $\mathsf{run}\ v^0$ and $\tilde{}\,v^0$), Closedness (in the cases $v_1^0 v_2^0$, $\mathsf{run}\ v^0$ and $\tilde{}\,v^0$), and Demotion (in the case $\mathsf{run}\ v^0$). We consider only few interesting cases of redexes r^i, and assume that the last rule in the derivation of $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_i r^i : t$ is syntax directed.

Case $r^0 \equiv v_1^0 v_2^0$ with $v_1^0, v_2^0 \in \mathsf{V}^0$.
If $v_1^0 v_2^0$ is well-typed, then the typing rule (app) must have been applied with conclusion $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 v_1^0 v_2^0 : t_2$ and premises $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 v_1^0 : t_1 \to t_2$ and $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 v_2^0 : t_1$. Then lemma 3.12 implies that v_1^0 must be of the form $(X)\lambda x.e$ (assume $x \notin X$ by alpha conversion) and there exist Δ_1, Γ_1 such that $\mathit{dom}(\Delta_1, \Gamma_1) \subseteq X$ and $\Sigma; \Delta^{+1}, \Delta_1^{+1}; \Gamma^{+1}, \Gamma_1^{+1}, x : t_1^0 \vdash_0 e : t_2$. There are two cases. If $t_2 \in \mathsf{C}$ then repeated application of rules (bind2) and (bind3) gives $\Sigma; \Delta^{+1}; \Gamma^{+1}, x : t_1^0 \vdash_0 (X)e : t_2$. If $t_2 \notin \mathsf{C}$, then $(t_1 \to t_2) \notin \mathsf{C}$, hence lemma 3.13 gives $X \cap FV(\lambda x.e) = \emptyset$, thus $\Sigma; \Delta^{+1}; \Gamma^{+1}, x : t_1^0 \vdash_0 e : t_2$ holds and repeated application of rule (bind1) gives $\Sigma; \Delta^{+1}; \Gamma^{+1}, x : t_1^0 \vdash_0 (X)e : t_2$ like for case $t_2 \in \mathsf{C}$. The reduction rule is $\mu, ((X)\lambda x.e)v_2^0 \xrightarrow{0} \mu, ((X)e)[x := v_2^0]$. To conclude the case, lemma 3.4 implies $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 ((X)e)[x := v_2^0] : t_2$.

Case $r^0 \equiv \mathsf{ref}\ v^0$ with $v^0 \in \mathsf{V}^0$.
If $\mathsf{ref}\ v^0$ is well-typed, then the typing rule (ref) must have been applied with conclusion $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 \mathsf{ref}\ v^0 : \mathsf{ref}\ c$ and premise $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 v^0 : c$. The reduction rule is $\mu, \mathsf{ref}\ v^0 \xrightarrow{0} (\mu, l : \bullet v^0), l$ with $l \notin \mathit{dom}(\mu)$. By repeated use of rules (bind2) and (bind3) we get $\Sigma; \emptyset; \emptyset \vdash_0 \bullet v^0 : c$. Let Σ' be $(\Sigma, l : \mathsf{ref}\ c)$; then $\Sigma' \models (\mu, l : \bullet v^0)$ and $\Sigma'; \Delta^{+1}; \Gamma^{+1} \vdash_0 l : \mathsf{ref}\ c$.

Case $r^0 \equiv \mathsf{run}\ v^0$ with $v^0 \in \mathsf{V}^0$.

If $\mathsf{run}\ v^0$ is well-typed, then the typing rule (run) must have been applied with conclusion $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 \mathsf{run}\ v^0 : [t]$, and the premise is $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 v^0 : [\langle t \rangle]$; so lemma 3.12 implies that v^0 must be of the form $(X)\langle v^1 \rangle$ for $v^1 \in \mathsf{V}^1$. Then lemma 3.13 gives $FV(\langle v^1 \rangle) = \emptyset$, hence $\Sigma; \emptyset; \emptyset \vdash_1 v^1 : t$. Now, proposition 3.6 implies $\Sigma; \emptyset; \emptyset \vdash_0 (v^1 \downarrow_0) : t$. The reduction rule is $\mu, \mathsf{run}\ (X)\langle v^1 \rangle \xrightarrow{0} \mu, \bullet(v^1 \downarrow_0)$. Since $FV(v^1 \downarrow_0) = FV(v^1) = \emptyset$, $\bullet(v^1 \downarrow_0) \equiv v^1 \downarrow_0$, so using rule (closI2) and lemma 3.3 we conclude with $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_0 \bullet(v^1 \downarrow_0) : [t]$.

□

Lemma 3.15 (*Replacement for Evaluation Contexts*)
If $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_n E_i^n[e] : t$ with $E_i^n \in \mathsf{EC}_i^n$, then exist Δ_1 and Γ_1 and t_1 such that

- $\Sigma; \Delta_1^{+1}; \Gamma_1^{+1} \vdash_i e : t_1$
- $\Sigma'; \Delta_1^{+1}; \Gamma_1^{+1} \vdash_i e' : t_1 \implies \Sigma'; \Delta^{+1}; \Gamma^{+1} \vdash_n E_i^n[e'] : t$ for any $\Sigma' \supseteq \Sigma$ and e'

Proof
By induction on the structure of $E_i^n \in \mathsf{EC}_i^n$ (and the derivation of $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_n E_i^n[e] : t$), using Weakening and the fact that in an evaluation context the hole □ can only be within the scope of binders which bind variables at level > 0. □

Comparison with Replacement for $\mathsf{MiniML_{ref}}$. In $\mathsf{MiniML_{ref}^{meta}}$ (and multi-level languages like $\lambda^{\bigcirc}$) one can reduce within the scope of a binder, thus $\Delta, \Gamma = \emptyset$ does not imply $\Delta_1, \Gamma_1 = \emptyset$. Nevertheless, Replacement for evaluation contexts is more informative than Replacement for contexts with one hole, since it says that the hole can only be within the scope of binders which bind variables at level > 0.

Lemma 3.16 (*Safety for* $\overset{n}{\longmapsto}$)
If $\Sigma \models \mu$ and $\Sigma; \Delta^{+1}; \Gamma^{+1} \vdash_n e : t$, then

- $\mu, e \overset{n}{\not\longmapsto} \mathsf{err}$
- $\mu, e \overset{n}{\longmapsto} \mu', e' \implies$ there exists $\Sigma' \supseteq \Sigma$ such that $\Sigma' \models \mu'$ and $\Sigma'; \Delta^{+1}; \Gamma^{+1} \vdash_n e' : t$

Proof
By Unique Decomposition, Replacement, and Safety for $\xrightarrow{i}$. □

Proof of Weak Soundness
$(\emptyset, e) \overset{n}{\longmapsto}{}^m d$ implies $d \not\equiv \mathsf{err}$, by induction on m, using Safety for $\overset{n}{\longmapsto}$. □

3.4 Conservative Extension Result

This section shows that typing and operational semantics for $\mathsf{MiniML_{ref}^{meta}}$ are a *conservative extension* of those for $\mathsf{MiniML_{ref}}$. To distinguish the syntactic categories of $\mathsf{MiniML_{ref}^{meta}}$ from those of $\mathsf{MiniML_{ref}}$ we use the prefix $\_^{\mathsf{meta}}$ for the former. For example, $\mathsf{E^{meta}}$ denotes the set of $\mathsf{MiniML_{ref}^{meta}}$ terms, while E denotes the set of $\mathsf{MiniML_{ref}}$ terms. The conservative extension result is stated for $e \in \mathsf{E}_0$, i.e. for complete programs in $\mathsf{MiniML_{ref}}$:

Theorem 3.17 (*Conservative Extension*)
If $e \in \mathsf{E}_0$, $t \in \mathsf{T}$ and $d \in (\mathsf{S}_0^{\mathsf{meta}} \times \mathsf{E}_0^{\mathsf{meta}}) + \{\mathsf{err}\}$, then

1. $\emptyset; \emptyset \vdash e : t \Longleftrightarrow \emptyset; \emptyset; \emptyset \vdash_0 e : t$;
2. $\emptyset, e \longmapsto^* d \Longleftrightarrow \emptyset, e \overset{0}{\longmapsto}{}^* d$.

The rest of the Section establishes several facts, which combined together imply the desired result. We have the following inclusions between syntactic categories:

Lemma 3.18
$\mathsf{T} \subseteq \mathsf{C}^{\mathsf{meta}}$, $\mathsf{E} \subseteq \mathsf{E}^{\mathsf{meta}}$, $\mathsf{V} \subseteq \mathsf{V}^{0^{\mathsf{meta}}}$, $\mathsf{S}_0 \subseteq \mathsf{S}_0{}^{\mathsf{meta}}$, $\mathsf{Red} \subseteq \mathsf{Red}^{0^{\mathsf{meta}}}$ and $\mathsf{EC} \subseteq \mathsf{EC}_0^{0^{\mathsf{meta}}}$.

Proof
All by easy inductions. □

A typing judgment $\Sigma; \Gamma \vdash e : t$ for $\mathsf{MiniML_{ref}}$ it is not appropriate for $\mathsf{MiniML_{ref}^{meta}}$, because $\Gamma : \mathsf{X} \overset{fin}{\to} \mathsf{T}$ and e lack the level information. We introduce the operation

$$\{x_i : t_i | i \in m\}^n \overset{\Delta}{=} \{x_i : t_i^n | i \in m\}$$

which turns a type assignment into a type-and-level assignment, i.e. Γ^n assigns level n to all variables declared in Γ.

Proposition 3.19
$\Sigma; \Gamma \vdash e : t$ implies $\Sigma; \emptyset; \Gamma^0 \vdash_0 e : t$.

Proof
Easy induction on the derivation of $\Sigma; \Gamma \vdash e : t$. □

An immediate consequence of Proposition 3.19 is one direction of Part 1 of Theorem 3.17. For the other direction, we need a translation from $\mathsf{T}^{\mathsf{meta}}$ to T.

Definition 3.20
The function $\|\_\|$ from $\mathsf{T}^{\mathsf{meta}}$ to T is defined as

$$\begin{array}{rcl} \|[t]\| & \overset{\Delta}{=} & \|t\| \\ \|\langle t \rangle\| & \overset{\Delta}{=} & \|t\| \end{array}$$

and it commutes with all other type-constructors of $\mathsf{MiniML_{ref}^{meta}}$. The extension to signatures Σ is point-wise; $\|\Gamma\|(x) = \|t\|$ when $\Gamma(x) = t^n$ and similarly for Δ.

Proposition 3.21
If $e \in \mathsf{E}$, $t \in \mathsf{T}$, $\Delta : \mathsf{X} \overset{fin}{\to} (\mathsf{C}^{\mathsf{meta}} \times \mathsf{N})$, $\Gamma : \mathsf{X} \overset{fin}{\to} (\mathsf{T}^{\mathsf{meta}} \times \mathsf{N})$ and $n \in \mathsf{N}$, then

- $\Sigma; \Delta; \Gamma \vdash_n e : t$ implies $\|\Sigma\|; \|\Delta\|, \|\Gamma\| \vdash e : \|t\|$

Proof
By induction on the derivation of $\Sigma; \Delta; \Gamma \vdash_n e : t$. □

Finally we show that, starting from a *closed* $\mathsf{MiniML_{ref}}$ configuration, the transitions allowed in $\mathsf{MiniML_{ref}}$ and in $\mathsf{MiniML_{ref}^{meta}}$ are the same.

Lemma 3.22

1. $\mathsf{V} = \mathsf{V}^{0^{\mathsf{meta}}} \cap \mathsf{E}$;
2. $\mathsf{Red} = \mathsf{Red}^{0^{\mathsf{meta}}} \cap \mathsf{E}$;
3. $E^0_i[e] \in \mathsf{E}$ implies $i = 0$, $e \in \mathsf{E}$ and $E^0_i \in \mathsf{EC}$.

Proof
By induction on the definition of $\mathsf{V}^{0^{\mathsf{meta}}}$, $\mathsf{Red}^{0^{\mathsf{meta}}}$ and E^0_i, respectively. $\square$

Lemma 3.23
If $r \in \mathsf{Red}_0$, $\mu \in \mathsf{S}_0$ and $d \in (\mathsf{S}^{\mathsf{meta}}_0 \times \mathsf{E}^{\mathsf{meta}}_0) + \{\mathsf{err}\}$, then

- $\mu, r \longrightarrow d \iff \mu, r \xrightarrow{0} d$

Proof
Case analysis on the definition of $\longrightarrow$ and $\xrightarrow{0}$. $\square$

Proposition 3.24
If $e \in \mathsf{E}_0$, $\mu \in \mathsf{S}_0$ and $d \in (\mathsf{S}^{\mathsf{meta}}_0 \times \mathsf{E}^{\mathsf{meta}}_0) + \{\mathsf{err}\}$, then

- $\mu, e \longmapsto d \iff \mu, e \overset{0}{\longmapsto} d$

Proof
By definition of $\longmapsto$ and $\overset{0}{\longmapsto}$, using Lemmas 3.18, 3.22 and 3.23. $\square$

This implies Part 2 of Theorem 3.17.

4 Examples of Imperative Multi-stage Programming

We present several examples of imperative multi-stage programming. The examples are described in $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ extended with whatever features of SML and its standard library are most appropriate, e.g. polymorphism, data-types, pattern matching, arrays. All the examples make essential use of closed types, and could not be reproduced in full in other meta-programming formalisms. In particular, a sequence of top-level declarations corresponds to nested letc-bindings (evaluated at level 0), thus identifiers declared at the top-level are in the Δ part of a typing context $\Sigma;\Delta;\Gamma$, and have a closed type. More formally, `val x = e; p` stands for $\mathsf{let_c}\ x = e\ \mathsf{in}\ p$ and has the following derived rules for typing and reduction

$$\frac{\Sigma;\Delta^{\leq 0};\emptyset \vdash_0 e : c \qquad \Sigma;\Delta^{\leq 0}, x : c^0;\emptyset \vdash_0 p : t}{\Sigma;\Delta^{\leq 0};\emptyset \vdash_0 (\mathtt{val}\ x\ \mathtt{=}\ e\mathtt{;}\ p) : t} \qquad \mu, (\mathtt{val}\ x\ \mathtt{=}\ v^0\mathtt{;}\ p) \xrightarrow{0} \mu, p[x := \bullet v^0]$$

Warnings. The extensions of $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ with polymorphism and data-types have not been formally investigated, yet. Although we do not foresee major technical difficulties in the study of such extensions, the examples relying on them should be taken with a grain of salt. The current release of MetaML displays values of code types, so that a programmer can check the quality of the generated code. However, in $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ code is not observable, where observability means that for any pair of syntactically different values of a code type one can find a context that distinguish them. For instance, the values `<1>` and `<(fn x => x) 1>` of type `<int>` are indistinguishable.

```
- datatype nat = z | s of nat;                          (* natural numbers*)
datatype nat

- fun p z     x y = (y := 1.0)                     (* conventional program *)
    | p (s n) x y = (p n x y; y := x * !y);
val p = fn : nat -> real -> real ref -> unit

- fun p_a z     x y = <~y := %1.0>    (* staged program with annotations *)
    | p_a (s n) x y = <~(p_a n x y); ~y:=~x %* !~y>;
val p_a = fn : [nat -> <real> -> <real ref> -> <unit>]

- fun p_cg n =  <fn x y => ~(p_a n <x> <y>)>;           (* code generator *)
val p_cg = fn : [nat  -> <real -> real ref -> unit>]

- val p_sc = p_cg 2;                                  (* specialized code *)
val p_sc = <fn x y => (y := %1.0; y := x %* !y; y := x %* !y)>
         : [<real -> real ref -> unit>]

- val p_sp = run p_sc;                             (* specialized program *)
val p_sp = fn : real -> real ref -> unit

- fun p_o n = letc n=n in run(p_cg n)               (* optimized program*)
val p_o = fn : nat -> real -> real ref -> unit
```

Fig. 5. The multi-stage programming method: the imperative power function

4.1 An Imperative Power Function

Figure 5 illustrates the multi-stage programming method (Taha & Sheard, 1997; Benaissa *et al.*, 1999) in an imperative setting, by adapting the classic example of the power function:

- nat is the datatype for natural numbers.
- p is a conventional "single-stage" program, which takes a natural number n, a real x, a reference y, and stores x^n in y. It uses the predefined identifiers (constants) 1.0:real and *:real->real->real.
- p_a is a "two-stage" *annotated* version of p, which requires the natural number n (as before), but uses only symbolic representations for the real x and the reference y. p_a builds a representation of the desired computation. In this representation the predefined identifiers declared at level 0, i.e. 1.0 and *, are lifted to level 1 using cross-stage persistence.
- p_cg is the *code generator*. Given a natural number, the code generator proceeds by building a piece of code that contains a lambda abstraction, and then, using Escape, performs an unfolding of the annotated program p_a over the "dummy variables" <x> and <y>. This unfolding is possible because of "symbolic evaluation under lambda".
- p_sc is the *specialized code* generated by applying p_cg to a particular natural number (in this case 2). The generated (high-level) code corresponds closely to machine code, and should compile into a light-weight subroutine.

- `p_sp` is the *specialized program*, the ultimate goal of run-time code generation. The function `p_sp` is a specialized version of `p` applied to 2, which does not have unnecessary run-time overheads.

Finally, one can define an *optimized* program `p_o` (i.e. `p_o 2` is `p_sp`) with the same type of the conventional program `p`. The definition of `p_o` relies on a general trick.

Remark 4.1 (*The Letc-trick*)
According to the typing rule for $\lambda x.e$, the λ-bound x must be declared in Γ. However, when x has a closed type, one can replace $\lambda x.e$ with $\lambda x.\mathsf{let_c}\ x = x\ \mathsf{in}\ e$. This transformation does not change the operational behaviour, but it allows to infer more types. In fact, to assign type $c \to t$ (at level n) to $\lambda x.\mathsf{let_c}\ x = x\ \mathsf{in}\ e$ it suffices to assign type t to e under the assignment $\Sigma; \Delta, x : c^n; \Gamma$, rather than the less accurate assignment $\Sigma; \Delta; \Gamma, x : c^n$. For instance, in relation to the typing of `p_o`, if $t \overset{\Delta}{\equiv} \mathsf{real} \to \mathsf{ref\ real} \to \mathtt{unit}$ and $\Delta \overset{\Delta}{\equiv} p_{cg} : [\mathsf{nat} \to \langle t \rangle]^0$, then the judgement $\emptyset; \Delta, n : \mathsf{nat}^0; \emptyset \vdash_0 p_{cg} n : [\langle t \rangle]$ is derivable, but $\emptyset; \Delta; n : \mathsf{nat}^0 \vdash_0 p_{cg} n : [\langle t \rangle]$ is not.

4.2 Lightweight and Generative Components

(Kamin *et al.*, 2000) proposes to describe components as higher-order macros written in a *functional* meta-language JR (with Bracket and Escape constructs similar to those of MetaML) to generate code in an *imperative* object-language (for instance Java). It is easy to recast in (an extension of) $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ several examples of components given in JR. For brevity, we consider only the example of a generative sort component. The main function is written in SML extended with arrays. It is a generic sort function with type

```
- fun sortfun size lessfun arg = ... ;
val sortfun: int -> ('a -> 'a -> bool) -> 'a array -> unit.
```

In our extended language we can express a generative sort *component* (Kamin *et al.*, 2000), and assign to it the following $\mathsf{MiniML}_{\mathsf{ref}}^{\mathsf{meta}}$ closed type (which is more informative than that assigned in JR, where all object-code has type `Code`):

```
- fun sortcomp size lesscomp arg =
      if size=2 (* in-line *)
      then <if ~(lesscomp <~arg[1]> <~arg[0]>)
            then "swap ~arg[1] ~arg[0]">
      else <%sortfun %size (fn x y => ~(lesscomp <x> <y>)) ~arg>;
val sortcomp:[int -> (<'a> -> <'a> -> <bool>) -> <'a array> -> <unit>]
```

Note that the type variable `'a` should range over *closed types*, since `array`, like the `ref` type constructor, can be applied only to closed types. The main advantages of `sortcomp` over `sortfun` are:

- the component can generate optimized code according to the value of `size`, e.g. we in-line the sorting code when `size` is small, and we call the generic sort function `sortfun` otherwise;

- the client of `sortcom` can give optimized comparison code, instead of calling a comparison function `lessfun`, i.e. `fun lesscomp x y = <%lessfun ~x ~y>;`
- the client can in-line the generated code instead of wrapping it in a procedure, i.e. `<fn A => ~(sortcomp size lesscomp <A>)>`.

Given a component `sortcomp` one can exploit the trick described in Remark 4.1, and define a generic sort function `sortfun_o` optimized with respect to `size` (but the other advantages offered by the generative component are lost):

```
- fun sortfun_o size lessfun arg =
      letc size=size in
      letc lessfun=lessfun in
      letc arg=arg in
          run(sortcomp size (fn x y => <%lessfun ~x ~y>) <%arg>);
val sortfun_o: int -> ('a -> 'a -> bool) -> 'a array -> unit
```

4.3 References to Generative Components

The examples above do not need references to code (or functions returning code). The use of generative components, advocated by (Kamin *et al.*, 2000), suggests obvious reasons why such type of references are useful. First of all, a generative component has always a type of the form $\ldots \to \langle t\rangle$. If a component is declared at the top-level (or is part of a library), then it can be assigned the more accurate closed type $[\ldots \to \langle t\rangle]$. Now suppose that there are several generative components $GC_i : [GCT_i]$ located at some remote sites, and one wants to download them only if needed. What one can do is to provide stubs $GCS_i : \mathsf{unit} \to [GCT_i]$, that download the components and cache them locally for repeated use

```
- datatype 'a maybe = fail | ok of 'a;
- local val cache : [GCT_i] maybe ref = ref fail
  in fun GCS_i () = case !cache of
                      fail => letc gc_i = "downloaded GC_i"
                              in cache := ok gc_i; gc_i
                    | ok gc_i => gc_i;
val GCS_i: unit -> [GCT_i]
```

This is well-typed in $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$, because we have assumed that the component has a closed type `[GCT_i]`, and the type `t maybe` is closed when `t` is.

5 Related Work

Multi-level languages (Gomard & Jones, 1991; Glück & Jørgensen, 1996; Davies, 1996; Moggi, 1998) provide mechanisms for constructing and combining open code. Multi-stage languages extend multi-level languages with a construct for executing code at run-time (Taha, 1999). The scope extrusion problem identified in Section 1.1 also applies to a naive imperative extension of $\lambda^{\bigcirc}$ (Davies, 1996), which allows open code and symbolic evaluation under lambda (but has no construct for executing code). Binding-Time Analyses (BTAs) for imperative languages must also address such problems. Intuitively, a BTA takes a single-stage program and produces a two-stage one (Jones *et al.*, 1993; Taha, 2000b).

(Thiemann & Dussart, 1999) describes an off-line partial evaluator for a higher-order language with first-class references, where a two-level language with regions is used to specify a BTA. This two-level language 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). The two-level language of (Thiemann & Dussart, 1999) and $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$ provide incomparable approaches to Type Safety of imperative multi-level languages (for partial evaluation): the first uses regions and effects, the second uses closed types (and introduces a new type constructor $[\_]$).

(Calcagno & Moggi, 2000) gives a big-step operational semantics for $\mathsf{MiniML}^{\mathsf{meta}}_{\mathsf{ref}}$, which uses as primitive term construct Bind-Closure $\bullet e$ instead of Bind $(x)e$, but Type Safety for such a semantics fails! The problem (recast in a small-step operational semantics) is in the rule $\mu, (\bullet\lambda x.e)v_2^0 \xrightarrow{0} \mu, \bullet(e[x := v_2^0])$, whose contractum is always a closed term (in particular the free variables in v_2^0 get bound by $\bullet$). This is in contrast to rule $\mu, ((X)\lambda x.e)v_2^0 \xrightarrow{0} \mu, ((X)e)[x := v_2^0]$ of Figure 4. For instance, consider the well-typed redex $\emptyset; \emptyset; x : t^1 \vdash_0 (\bullet\lambda x.x)\langle x\rangle : \langle t\rangle$. The reduction rule in (Calcagno & Moggi, 2000) yields the contractum $\bullet\langle x\rangle$, which is not typable. While the reduction rule of Figure 4 yields $\langle x\rangle$, because $\bullet\lambda x.x \equiv \lambda x.x$.

(Hatcliff & Danvy, 1997) proposes a partial evaluator for a computational meta-language, and they formalize existing techniques in a uniform framework by abstracting from dynamic computational effects. However, this partial evaluator does not seem to allow interesting computational effects at specialization time.

There is a simpler approach to imperative multi-stage programming based on $\lambda^{\Box}$ (Davies & Pfenning, 1996; Wickline *et al.*, 1998). In fact, this language allows closed code and run-time code generation, but does not allow evaluation under lambda. Therefore, adding references to $\lambda^{\Box}$ is as easy as to MiniML. The price for this simplicity is the lack of symbolic evaluation (typical of partial evaluation) necessary for optimization at *specialization time.*

(Xi, 1999) uses (dependent) types for eliminating dead code. Our Bind construct is a mere dead code annotation, used for handling scope extrusion, and we made no attempt to exploit it for dead code elimination. The typing rules for Bind (apart from the trivial one), make sense only in a multi-level language. However, the Bind annotation and its operational meaning (see Remark 3.2) is not specific to multi-level languages.

Acknowledgements

We would like to thank the anonymous referees for their valuable comments (any failure to fully exploit them is our fault). We would like to thank in particular Walid Taha for the many discussions and useful comments on this paper, and Jeff Polakow for pointing out errors/typos in the final version.

References

Benaissa, Zine El-Abidine, Moggi, Eugenio, Taha, Walid, & Sheard, Tim. (1999). Logical modalities and multi-stage programming. *Federated logic conference (FLoC) satellite workshop on intuitionistic modal logics and applications (IMLA).*

Calcagno, Cristiano, & Moggi, Eugenio. (2000). Multi-stage imperative languages: A conservative extension result. *Pages 92–107 of: (Taha, 2000a).*

Calcagno, Cristiano, Moggi, Eugenio, & Taha, Walid. (2000). Closed types as a simple approach to safe imperative multi-stage programming. *Pages 25–36 of: the international colloquium on automata, languages, and programming (ICALP '00).* Lecture Notes in Computer Science, vol. 1853. Geneva: Springer-Verlag.

Cardelli, Luca. (1997). Type systems. Tucker, Allen B. Jr (ed), *The computer science and engineering handbook.* CRC Press.

Clement, Dominique, Despeyroux, Joelle, Despeyroux, Thierry, & Kahn, Gilles. (1986). A simple applicative language: Mini-ML. *Pages 13–27 of: Proceedings of the 1986 ACM conference on lisp and functional programming.* ACM press.

Davies, Rowan. (1996). A temporal-logic approach to binding-time analysis. *Pages 184–195 of: the symposium on logic in computer science (LICS '96).* IEEE Computer Society Press, New Brunswick.

Davies, Rowan, & Pfenning, Frank. (1996). A modal analysis of staged computation. *Pages 258–270 of: the symposium on principles of programming languages (POPL '96).*

Glück, Robert, & Jørgensen, Jesper. (1996). Fast binding-time analysis for multi-level specialization. *Pages 261–272 of:* Bjørner, Dines, Broy, Manfred, & Pottosin, Igor V. (eds), *Perspectives of system informatics.* Lecture Notes in Computer Science, vol. 1181. Springer-Verlag.

Gomard, Carsten K., & Jones, Neil D. (1991). A partial evaluator for untyped lambda calculus. *Journal of functional programming*, **1**(1), 21–69.

Harper, Robert, & Stone, Chris. (1997). *A type-theoretic account of Standard ML 1996 (version 2).* Tech. rept. CMU–CS–97–147. Carnegie Mellon University, Pittsburgh, PA.

Hatcliff, John, & Danvy, Olivier. (1997). A computational formalization for partial evaluation. *Mathematical structures in computer science*, **7**(5), 507–541.

Jones, Neil D., Gomard, Carsten K, & Sestoft, Peter. (1993). *Partial evaluation and automatic program generation.* Prentice-Hall.

Kamin, Sam, Callahan, Miranda, & Clausen, Lars. (2000). Lightweight and generative components II: Binary-level components. *Pages 28–50 of: (Taha, 2000a).*

MHP. (2000). *The MetaML Home Page.* Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**, 348–375.

Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, David. (1997). *The Definition of Standard ML (revised).* MIT Press.

Moggi, Eugenio. (1998). Functor categories and two-level languages. *Foundations of software science and computation structures (FoSSaCS).* Lecture Notes in Computer Science, vol. 1378. Springer Verlag.

Moggi, Eugenio, Taha, Walid, Benaissa, Zine El-Abidine, & Sheard, Tim. (1999). An idealized MetaML: Simpler, and more expressive. *Pages 193–207 of: European symposium on programming (ESOP).* Lecture Notes in Computer Science, vol. 1576. Springer-Verlag.

Smith, Brian Cantwell. (1982). *Reflection and semantics in a procedural language.* Ph.D. thesis, Massachusetts Institute of Technology.

Taha, Walid. (1999). *Multi-stage programming: Its theory and applications.* Ph.D.

thesis, Oregon Graduate Institute of Science and Technology. Available from ftp://cse.ogi.edu/pub/tech-reports/README.html.

Taha, Walid (ed). (2000a). *Semantics, applications, and implementation of program generation.* Lecture Notes in Computer Science, vol. 1924. Montréal: Springer-Verlag.

Taha, Walid. (2000b). A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. *Proceedings of the workshop on partial evaluation and semantics-based program maniplation (PEPM).* Boston: ACM Press.

Taha, Walid, & Sheard, Tim. (1997). Multi-stage programming with explicit annotations. *Pages 203–217 of: Proceedings of the symposium on partial evaluation and semantic-based program manipulation (PEPM).* Amsterdam: ACM Press.

Taha, Walid, & Sheard, Tim. (2000). MetaML: Multi-stage programming with explicit annotations. *Theoretical computer science*, **248**(1-2).

Taha, Walid, Benaissa, Zine-El-Abidine, & Sheard, Tim. (1998). Multi-stage programming: Axiomatization and type-safety. *Pages 918–929 of: 25th international colloquium on automata, languages, and programming (ICALP).* Lecture Notes in Computer Science, vol. 1443.

Thiemann, Peter, & Dussart, Dirk. (1999). *Partial evaluation for higher-order languages with state.* Available from http://www.informatik.uni-freiburg.de/~thiemann/papers/index.html.

Wickline, Philip, Lee, Peter, & Pfenning, Frank. (1998). Run-time code generation and Modal-ML. *Pages 224–235 of: Proceedings of the conference on programming language design and implementation (PLDI).*

Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Information and computation*, **115**(1), 38–94.

Xi, Hongwei. (1999). Dead code elimination through dependent types. *First international workshop on practical aspects of declarative languages.* Lecture Notes in Computer Science, vol. 1551.