Closed Types as a Simple Approach to Safe Imperative Multi-Stage Programming

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, Göteborg, Sweden
taha@cs.chalmers.se

Abstract. Safely adding computational effects to a multi-stage language has been an open problem. In previous work, a *closed type constructor* was used to provide a safe mechanism for executing dynamically generated code. This paper proposes a general notion of *closed type* as a simple approach to safely introducing computational effects into multi-stage languages. We demonstrate this approach formally in a core language called $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$. This core language combines safely multi-stage constructs and ML-style references. In addition to incorporating state, $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ also embodies a number of technical improvements over previously proposed core languages for multi-stage programming.

1 Introduction

Many important software applications require the manipulation of open code at run-time. Examples of such applications include high-level program generation, compilation, and partial evaluation [JGS93]. But having a notion of values that includes open code (that is, possibly containing free variables) complicates both the (untyped) operational semantics and type systems for programming languages designed to support such applications. This paper advocates a simple and direct approach for safely adding computational effects into languages that manipulate open code. The approach capitalises on a single type constructor that guarantees that a given term will evaluate to a *closed* value at run-time. We demonstrate our approach in the case of ML-style references [MTHM97].

We extend recent studies into the semantics and type systems for multi-level and multi-stage languages. **Multi-level** languages [GJ91,GJ96,Mog98,Dav96] provide a mechanism for constructing and combining open code. **Multi-stage** languages [TS97,TBS98,MTBS99,BMTS99,Tah99,Tah00] extend multi-level languages with a construct for executing the code generated at run-time. Multi-stage programming can be illustrated using **MetaML** [TS97,Met00], an extension of

⋆ Research partially supported by MURST and ESPRIT WG APPSEM.

⋆⋆ Postdoctoral Fellow funded by the Swedish Research Council for Engineering Sciences (TFR), grant number 221-96-403.

```
-| 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>               (* annotated program *)
     | p_a (s n) x y = <~(p_a n x y); ~y:=~x * !~y>;
val p_a = fn : nat -> <real> -> <real ref> -> <unit>

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

-| val p_sc = p_cg 3;                              (* specialised code *)
val p_sc = <fn x y => (y:=1.0; y:=x*!y; y:=x*!y; y:=x*!y)>
         : <real -> real ref -> unit>

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

Fig. 1. Example of multi-stage programming with references in MetaML

SML [MTHM97] with a type constructor $\langle\_\rangle$ for open code. MetaML provides three basic staging constructs that operate on this type: Brackets $\langle\_\rangle$, Escape ˜ _ and Run run _. Brackets defers the computation of its argument; Escape splices its argument into the body of surrounding Brackets; and Run executes its argument.

Figure 1 lists a sequence of declarations illustrating the multi-stage programming method [TS97,BMTS99] in an imperative setting:

- `p` is a conventional "single-stage" program, which takes a natural `n`, a real `x`, a reference `y`, and stores $\mathtt{x}^{\mathtt{n}}$ in `y`.
- `p_a` is a "two-stage" *annotated* version of `p`, which requires the natural `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. When the first argument is zero, no assignment is performed, instead a piece of code for performing an assignment at a later time is generated. When the first argument is greater than zero, code is generated for performing an assignment at a later time, and moreover the recursive call to `p_a` is performed so that the whole code-generation is performed in full.
- `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 powerful capability of "evaluation under lambda" is an essential feature of multi-stage programming languages.

– `p_sc` is the *specialised code* generated by applying `p_cg` to a particular natural number (in this case 3). The generated (high-level) code corresponds closely to machine code, and should compile into a light-weight subroutine.
– `p_sp` is the *specialised program*, the ultimate goal of run-time code generation. The function `p_sp` is a specialised version of `p` applied to 3, which does not have unnecessary run-time overheads.

Problem Safely adding computational effects to multi-stage languages has been an open problem[1]. For example, when adding ML-style references to a multi-stage language like MetaML, one can have that "dynamically bound" variables go out of the scope of their binder [TS00]. Consider the following MetaML[2] 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>
```

In evaluating the second declaration, the variable `x` goes outside the scope of the binding lambda, and the result of the third line is wrong, since x is not bound in the environment, even though the session is well-typed according to naive extensions of previously proposed type systems for MetaML. This form of **scope extrusion** is specific to multi-level and multi-stage languages, and it does *not* arise in traditional programming languages, where evaluation is generally restricted to closed terms (e.g. see [Plo75] and many subsequent studies.) The the problem lies in the *run-time interaction between free variables and references.*

Remark 1. In the type system we propose (see Figure 2) the above session is not well-typed. First, `ref <1>` cannot be typed, because `<1>` is not of a closed type. Second, if we add some closedness annotation to make the first line well-typed, i.e. `val a = ref [<1>]`, then the type of `a` becomes `ref [<int>]`, and we can no longer type `a:=<x>` in the third line. Now, there is no way to add closedness annotations, e.g. `a:=[<x>]`, to make the third line well-typed, in fact the (close)-rule is not applicable to derive $a\colon \mathsf{ref\ nat}^0; x\colon \mathsf{nat}^1 \vdash [\langle x\rangle]\colon [\langle \mathsf{nat}\rangle]^0$.

Contributions and organisation of this paper This paper shows that *multi-stage and imperative features can be combined safely in the same programming*

[1] The current release of MetaML [Met00] is a substantial language, supporting most features of SML and a host of novel meta-programming constructs. In this release, safety is not guaranteed for meta-programs that use Run or effects. We hope to incorporate the ideas presented in this paper into the next MetaML release.
[2] The observation made here also applies to $\lambda^{\bigcirc}$ [Dav96].

language. We demonstrate this formally using a core language, that we call $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$, which extends Mini-ML [CDDK86] with ML-style references and[3]

- *A code type constructor* $\langle \_ \rangle$ [TS97,TBS98,Dav96].
- *A closed type constructor* $[\_]$ [BMTS99], but with improved syntax borrowed from $\lambda^{\Box}$ [DP96].
- *A term construct* $\mathsf{run}\ \_$ [TS97] typed with $[\_]$.

The key technical result is **type safety** for $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$, i.e. evaluation of well-typed programs does not raise an error (see Theorem 1). The type system of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ is simpler than some related systems for binding-time analysis (BTA), and it is also more expressive than most proposals for such systems (Section 3).

In principle the additional features of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ should not prevent us from writing programs like those in normal imperative languages. This can be demonstrated by giving an embedding of $\mathsf{Mini\text{-}ML}_{\mathsf{ref}}$ into our language, omitted for brevity. We expect the simple approach of using closed types to work in relation to other computational effects, for example: only closed values can be packaged with exceptions, only closed values can be communicated between processes.

Note on Previous Work The results presented here are a significant generalisation of a recently proposed solution to the problem of assigning a sound type to Run. The naive typing $\mathsf{run} : \langle t \rangle \to t$ of Run is unsound (see [TBS98]), since it allows to execute an arbitrary piece of code, including "dummy variables" such as `<x>`. The **closed type constructor** $[\_]$ proposed in [BMTS99] allows to give a sound typing $\mathsf{run} : [\langle t \rangle] \to t$ for Run, since one can *guarantee* that values of type $[\tau]$ will be *closed*. In this paper, we generalise this property of the closed type constructor to a bigger set of types, that we call **closed types**, and we also exploit these types to avoid the scope extrusion problem in the setting of imperative multi-stage programming.

2 $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$

This section describes the syntax, type system and operational semantics of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$, and establishes safety of well-typed programs. The types τ and closed types σ are defined as

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

Intuitively, a term can only be assigned a closed type σ when it will evaluate to a closed value (see Lemma 4). Values of type $[\tau]$ are always closed, but relying only on the close type constructor makes programming verbose [MTBS99,BMTS99,Tah00]. The generalised notion of closed type greatly improves the usability of the language (see Section 2.3). The set of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$

[3] $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ can incorporate also MetaML's *cross-stage persistence* [TS97]. This can be done by adding an up, similar to that of λ^{BN} [MTBS99], and by introducing a demotion operation. This development is omitted for space reasons.

terms is parametric in an infinite set of variables $x \in \mathsf{X}$ and an infinite set of locations $l \in \mathsf{L}$

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

The first line lists the $\mathsf{Mini\text{-}ML}$ terms: variables, abstraction, application, fixpoint for recursive definitions, zero, successor, and case-analysis on natural numbers. The second line lists the three multi-stage constructs of MetaML [TS97]: *Brackets* $\langle e \rangle$ and *Escape* $\tilde{}\,e$ are for building and splicing code, and *Run* is for executing code. The second line also lists the two "closedness annotations": *Close* $[e]$ is for marking a term as being closed, and *Let-Close* is for forgetting these markings. The third line lists the three SML operations on references, constants l for locations, and a constant fault for a program that crashes. The constants l and fault are not allowed in user-defined programs, but they are instrumental to the operational semantics of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$.

Remark 2. Realistic implementations should erase closedness annotations, by mapping $[e]$ to e and $(\mathsf{let}\ [x] = e_1\ \mathsf{in}\ e_2)$ to $(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2)$.

The constant fault is used in the rules for symbolic evaluation of binders, e.g. we write $\dfrac{\mu, e \overset{n+1}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+1}{\hookrightarrow} \mu'[x := \mathsf{fault}], \lambda x.v}$ instead of $\dfrac{\mu, e \overset{n+1}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+1}{\hookrightarrow} \mu', \lambda x.v}$.
This more *hygienic* handling of scope extrusion is compatible with the identification of terms modulo α-conversion, and prevents new free variable to appear as effect of the evaluation (see Lemma 3). On the other hand, in implementations there is no need to use the more hygienic rules, because during evaluation of a well-typed program (starting from the empty store) only closed values get stored.

Note 1. We will use the following notation and terminology

- Term equivalence, written $\equiv$, is α-conversion. Substitution of e for x in e' (modulo $\equiv$) is written $e'[x := e]$.
- m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} \mid i < m\}$ 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 $dom(f)$.
- $\Sigma : \mathsf{L} \overset{fin}{\to} \mathsf{T}$ is a *signature* (for locations only), written $\{l_i : \mathsf{ref}\ \sigma_i \mid i \in m\}$.
- $\Delta, \Gamma : \mathsf{X} \overset{fin}{\to} (\mathsf{T} \times \mathsf{N})$ are type-and-level assignments, written $\{x_i : \tau_i^{n_i} \mid i \in m\}$. We use the following operations on type-and-level assignments:
  $\{x_i : \tau_i^{n_i} \mid i \in m\}^{+n} \triangleq \{x_i : \tau_i^{n_i+n} \mid i \in m\}$ adds n to the level of the x_i;
  $\{x_i : \tau_i^{n_i} \mid i \in m\}^{\leq n} \triangleq \{x_i : \tau_i^{n_i} \mid n_i \leq n \wedge i \in m\}$ removes the x_i with level $> n$.
- $\mu : \mathsf{L} \overset{fin}{\to} \mathsf{E}$ is a *store*.
- $\Sigma, l : \mathsf{ref}\ \sigma$, $\Gamma, x : \tau^n$ and $\mu\{l = e\}$ denote extension of a signature, assignment and store respectively.

$$\frac{}{\Sigma, \Delta; \Gamma \vdash x : \tau^n}\ \Delta(x) = \tau^n \qquad \frac{}{\Sigma, \Delta; \Gamma \vdash x : \tau^n}\ \Gamma(x) = \tau^n$$

$$\frac{\Sigma, \Delta; \Gamma, x : \tau_1^n \vdash e : {\tau_2}^n}{\Sigma, \Delta; \Gamma \vdash \lambda x.e : \tau_1 \to {\tau_2}^n} \qquad \frac{\Sigma, \Delta; \Gamma \vdash e_1 : \tau_1 \to {\tau_2}^n \quad \Sigma, \Delta; \Gamma \vdash e_2 : {\tau_1}^n}{\Sigma, \Delta; \Gamma \vdash e_1\ e_2 : {\tau_2}^n}$$

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

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

$$\frac{\Sigma, \Delta; \Gamma \vdash e : \tau^{n+1}}{\Sigma, \Delta; \Gamma \vdash \langle e \rangle : \langle \tau \rangle^n} \qquad \frac{\Sigma, \Delta; \Gamma \vdash e : \langle \tau \rangle^n}{\Sigma, \Delta; \Gamma \vdash\ \tilde{}e : \tau^{n+1}} \qquad \frac{\Sigma, \Delta; \Gamma \vdash e : [\langle \tau \rangle]^n}{\Sigma, \Delta; \Gamma \vdash \mathsf{run}\ e : \tau^n}$$

$$(\text{close})\ \frac{\Sigma, \Delta^{\leq n}; \emptyset \vdash e : \tau^n}{\Sigma, \Delta; \Gamma \vdash [e] : [\tau]^n} \qquad \frac{\Sigma, \Delta; \Gamma \vdash e_1 : [\tau_1]^n \quad \Sigma, \Delta, x : \tau_1^n; \Gamma \vdash e_2 : {\tau_2}^n}{\Sigma, \Delta; \Gamma \vdash (\mathsf{let}\ [x] = e_1\ \mathsf{in}\ e_2) : {\tau_2}^n}$$

$$\frac{\Sigma, \Delta; \Gamma \vdash e : \sigma^n}{\Sigma, \Delta; \Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \sigma^n} \qquad \frac{\Sigma, \Delta; \Gamma \vdash e : \mathsf{ref}\ \sigma^n}{\Sigma, \Delta; \Gamma \vdash !\ e : \sigma^n}$$

$$(\text{set})\ \frac{\Sigma, \Delta; \Gamma \vdash e_1 : \mathsf{ref}\ \sigma^n \quad \Sigma, \Delta; \Gamma \vdash e_2 : \sigma^n}{\Sigma, \Delta; \Gamma \vdash e_1 := e_2 : \mathsf{ref}\ \sigma^n} \qquad \frac{}{\Sigma, \Delta; \Gamma \vdash l : \mathsf{ref}\ \sigma^n}\ \Sigma(l) = \mathsf{ref}\ \sigma$$

$$(\text{fix*})\ \frac{\Sigma, \Delta^{\leq n}, x : \tau^n; \emptyset \vdash e : \tau^n}{\Sigma, \Delta; \Gamma \vdash \mathsf{fix}\ x.e : \tau^n} \qquad (\text{close*})\ \frac{\Sigma, \Delta; \Gamma \vdash e : \sigma^n}{\Sigma, \Delta; \Gamma \vdash [e] : [\sigma]^n}$$

Fig. 2. Type System for $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$

2.1 Type System

Figure 2 gives the rules for the type system of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$. A typing judgement has the form $\Sigma, \Delta; \Gamma \vdash e : \tau^n$, read *"e has type τ and level n in Σ, Δ; Γ"*. Σ gives the type of locations which can be used in e, Δ and Γ (must have disjoint domains and) give the type and level of variables which may occur free in e.

Remark 3. Splitting the context into two parts (Δ and Γ) is borrowed from $\lambda^{\Box}$ [DP96], and allows us to replace the cumbersome closedness annotation (close e with $\{x_i = e_i | i \in m\}$) of λ^{BN} [BMTS99] with the more convenient $[e]$ and ($\mathsf{let}\ [x] = e_1\ \mathsf{in}\ e_2$). Informally, a variable $x : \tau^n$ declared in Γ ranges over values of type τ at level n (see Definition 1), while a variable $x : \tau^n$ declared in Δ ranges over *closed* values (i.e. without free variables) of type τ at level n.

Most typing rules are similar to those for related languages [Dav96,BMTS99], but there are some notable exceptions:

- (close) is the *standard* rule for $[e]$, the restricted context $\Sigma, \Delta^{\leq n}; \emptyset$ in the premise prevents $[e]$ to depend on variables declared in Γ (like in $\lambda^{\Box}$ [DP96]) or variables of level $> n$. The stronger rule (close*) applies only to closed types, and it is *justified* in Remark 5.
- (fix) is the *standard* rule for fix $x.e$, while (fix*) makes a stronger assumption on x, and thus can type recursive definitions (e.g. of closed functions) that

are not typable with (fix). For instance, from $\emptyset; f'{:}\,[\tau_1 \to \tau_2]^n, x{:}\,\tau_1^n \vdash e{:}\,{\tau_2}^n$ we cannot derive $\mathsf{fix}\ f'.[\lambda x.e]{:}\,[\tau_1 \to \tau_2]^n$, while the following modified term $\mathsf{fix}\ f'.(\mathsf{let}\ [f] = f'\ \mathsf{in}\ [\lambda x.e[f'{:}= [f]]])$ has the right type, but the wrong behaviour (it diverges!). On the other hand, the stronger rule (fix*) allows to type $[\mathsf{fix}\ f.\lambda x.e[f'{:}= [f]]]$, which has the desired operational behaviour.

- There is a *weaker* variant of (case*), which we ignore, where the assumption $x{:}\,\mathsf{nat}^n$ is in Γ instead of Δ.
- (set) does not assign to $e_1{:}= e_2$ type unit, simply to avoid adding a unit type to $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$.

The type system enjoys the following basic properties:

Lemma 1 (Weakening).

1. *If* $\Sigma, \Delta; \Gamma \vdash e{:}\,{\tau_2}^n$ *and* x fresh, *then* $\Sigma, \Delta; \Gamma, x{:}\,\tau_1^m \vdash e{:}\,{\tau_2}^n$
2. *If* $\Sigma, \Delta; \Gamma \vdash e{:}\,{\tau_2}^n$ *and* x fresh, *then* $\Sigma, \Delta, x{:}\,\tau_1^m; \Gamma \vdash e{:}\,{\tau_2}^n$
3. *If* $\Sigma, \Delta; \Gamma \vdash e{:}\,{\tau_2}^n$ *and* l fresh, *then* $\Sigma, l{:}\,\mathsf{ref}\ \sigma_1, \Delta; \Gamma \vdash e{:}\,{\tau_2}^n$

Proof. Part 1 is proved by induction on the derivation of $\Sigma, \Delta; \Gamma \vdash e{:}\,{\tau_2}^n$. The other two parts are proved similarly.

Lemma 2 (Substitution).

1. *If* $\Sigma, \Delta; \Gamma \vdash e{:}\,{\tau_1}^m$ *and* $\Sigma, \Delta; \Gamma, x{:}\,\tau_1^m \vdash e'{:}\,{\tau_2}^n$, *then* $\Sigma, \Delta; \Gamma \vdash e'[x{:}= e]{:}\,{\tau_2}^n$
2. *If* $\Sigma, \Delta^{\leq m}; \emptyset \vdash e{:}\,{\tau_1}^m$ *and* $\Sigma, \Delta, x{:}\,\tau_1^m; \Gamma \vdash e'{:}\,{\tau_2}^n$, *then* $\Sigma, \Delta; \Gamma \vdash e'[x{:}= e]{:}\,{\tau_2}^n$

Proof. Part 1 is proved by induction on the derivation of $\Delta; \Gamma, x{:}\,\tau_1^m \vdash e'{:}\,{\tau_2}^n$. Part 2 is proved similarly.

2.2 CBV Operational Semantics

Figure 3 gives the evaluation rules for the call-by-value (CBV) operational semantics of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$. Evaluation of a term e at level n can lead to

- a *result* v and a new store μ', when we can derive $\mu, e \overset{n}{\hookrightarrow} \mu', v$,
- a *run-time error*, when we can derive $\mu, e \overset{n}{\hookrightarrow} \mathsf{err}$, or
- *divergence*, when the search for a derivation goes into an *infinite regress.*

We will show that the second case (error) does not occur for well-typed programs (see Theorem 1). In general v ranges over terms, but under appropriate assumptions on μ, v could be restricted to *value at level n*.

Definition 1. *We define the set* $\mathsf{V}^n \subset \mathsf{E}$ *of* **values at level** n *by the BNF*

$$\begin{aligned}
v^0 \in \mathsf{V}^0 &::= \lambda x.e \mid \mathsf{z} \mid \mathsf{s}\ v^0 \mid \langle v^1\rangle \mid [v^0] \mid l \\
v^{n+1} \in \mathsf{V}^{n+1} &::= x \mid \lambda x.v^{n+1} \mid v_1^{n+1} v_2^{n+1} \mid \mathsf{fix}\ x.v^{n+1} \mid \\
&\quad\ \mathsf{z} \mid \mathsf{s}\ v^{n+1} \mid (\mathsf{case}\ v^{n+1}\ \mathsf{of}\ \mathsf{z} \to v_1^{n+1} \mid \mathsf{s}\ x \to v_2^{n+1}) \mid \\
&\quad\ \langle v^{n+2}\rangle \mid \mathsf{run}\ v^{n+1} \mid [v^{n+1}] \mid (\mathsf{let}\ [x] = v^{n+1}\ \mathsf{in}\ v^{n+1}) \mid \\
&\quad\ \mathsf{ref}\ v^{n+1} \mid !\ v^{n+1} \mid v_1^{n+1}{:}= v_2^{n+1} \mid l \mid \mathsf{fault} \\
v^{n+2} \in \mathsf{V}^{n+2} &+= \tilde{}\, v^{n+1}
\end{aligned}$$

Normal Evaluation

We give an *exhaustive* set of rules for evaluation of terms $e \in \mathsf{E}$ at level 0

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

$$\mu, \lambda x.e \overset{0}{\hookrightarrow} \mu, \lambda x.e$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv \lambda x.e}{\mu, e_1\ e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e[x{:}= \mathsf{fix}\ x.e] \overset{0}{\hookrightarrow} \mu', v}{\mu, \mathsf{fix}\ x.e \overset{0}{\hookrightarrow} \mu', v} \qquad \mu, \mathsf{z} \overset{0}{\hookrightarrow} \mu, \mathsf{z} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v}{\mu, \mathsf{s}\ e \overset{0}{\hookrightarrow} \mu', \mathsf{s}\ v}$$

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

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \mathsf{s}\ v \quad \mu', e_2[x{:}= v] \overset{0}{\hookrightarrow} \mu'', v'}{\mu, (\mathsf{case}\ e\ \mathsf{of}\ \mathsf{z} \to e_1 \mid \mathsf{s}\ x \to e_2) \overset{0}{\hookrightarrow} \mu'', v'} \qquad \frac{\mu, e \overset{1}{\hookrightarrow} \mu', v}{\mu, \langle e \rangle \overset{0}{\hookrightarrow} \mu', \langle v \rangle} \qquad \mu, \tilde{\ } e \overset{0}{\hookrightarrow} \mathsf{err}$$

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', [\langle v \rangle] \quad \mu', v \overset{0}{\hookrightarrow} \mu'', v'}{\mu, \mathsf{run}\ e \overset{0}{\hookrightarrow} \mu'', v'} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv [\langle e' \rangle]}{\mu, \mathsf{run}\ e \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v}{\mu, [e] \overset{0}{\hookrightarrow} \mu', [v]}$$

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', [v] \quad \mu', e_2[x{:}= v] \overset{0}{\hookrightarrow} \mu'', v'}{\mu, (\mathsf{let}\ [x] = e_1\ \mathsf{in}\ e_2) \overset{0}{\hookrightarrow} \mu'', v'} \qquad \frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv [e]}{\mu, (\mathsf{let}\ [x] = e_1\ \mathsf{in}\ e_2) \overset{0}{\hookrightarrow} \mathsf{err}}$$

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

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

$$\frac{\mu, e_1 \overset{0}{\hookrightarrow} \mu', v \not\equiv l \in dom(\mu')}{\mu, e_1{:}= e_2 \overset{0}{\hookrightarrow} \mathsf{err}} \qquad \mu, l \overset{0}{\hookrightarrow} \mu, l \qquad \mu, \mathsf{fault} \overset{0}{\hookrightarrow} \mathsf{err}$$

Symbolic Evaluation

$$\frac{\mu, e \overset{0}{\hookrightarrow} \mu', \langle v \rangle}{\mu, \tilde{\ } e \overset{1}{\hookrightarrow} \mu', v} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \not\equiv \langle e' \rangle}{\mu, \tilde{\ } e \overset{1}{\hookrightarrow} \mathsf{err}} \qquad \frac{\mu, e \overset{n+2}{\hookrightarrow} \mu', v}{\mu, \langle e \rangle \overset{n+1}{\hookrightarrow} \mu', \langle v \rangle} \qquad \frac{\mu, e \overset{n+1}{\hookrightarrow} \mu', v}{\mu, \tilde{\ } e \overset{n+2}{\hookrightarrow} \mu', \tilde{\ } v}$$

In all other cases symbolic evaluation is applied to the immediate sub-terms from left to right without changing level $\dfrac{\mu, e_1 \overset{n+1}{\hookrightarrow} \mu', v_1 \quad \mu', e_2 \overset{n+1}{\hookrightarrow} \mu'', v_2}{\mu, e_1\ e_2 \overset{n+1}{\hookrightarrow} \mu'', v_1\ v_2}$ and bound variables that have *leaked* in the store are replaced by fault $\dfrac{\mu, e \overset{n+1}{\hookrightarrow} \mu', v}{\mu, \lambda x.e \overset{n+1}{\hookrightarrow} \mu'[x{:}= \mathsf{fault}], \lambda x.v}$

Error Propagation

For space reasons, we omit the rules for error propagation. These rules follow the ML-convention for exceptions propagation.

Fig. 3. Operational Semantics for $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$

Remark 4. Values at level 0 can be classified according to the five kinds of types:

types	$\tau_1 \to \tau_2$	nat	$\langle\tau\rangle$	$[\tau]$	ref σ
values	$\lambda x.e$	z, s v^0	$\langle v^1\rangle$	$[v^0]$	l

Because of $\langle v^1\rangle$ the definition of value at level 0 involves values at higher levels. Values at level > 0, called *symbolic values*, are almost like terms. The differences between the BNF for V^{n+1} and E is in the productions for $\langle e\rangle$ and $\tilde{}e$:

- $\langle v^{n+1}\rangle$ is a value at level n, rather than level $n+1$
- $\tilde{}v^{n+1}$ is a value at level $n+2$, rather than level $n+1$.

Note 2. We will use the following auxiliary notation to describe stores:

- μ is **value store** $\stackrel{\Delta}{\Longleftrightarrow} \mu{:}\,\mathsf{L} \stackrel{fin}{\to} \mathsf{V}^0$;
- $\Sigma \models \mu \stackrel{\Delta}{\Longleftrightarrow} \mu$ is a value store and $dom(\Sigma) = dom(\mu)$ and $\Sigma;\emptyset \vdash \mu(l){:}\,\sigma^0$ whenever $l \in dom(\mu)$.

The following result establishes basic facts about the operational semantics, which are independent of the type system.

Lemma 3 (Values). *$\mu, e \stackrel{n}{\hookrightarrow} \mu', v$ implies $dom(\mu) \subseteq dom(\mu')$ and $\mathrm{FV}(\mu', v) \subseteq \mathrm{FV}(\mu, e)$; moreover, if μ is a value store, then $v \in \mathsf{V}^n$ and μ' is a value store.*

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

The following property justifies why a $\sigma \in \mathsf{C}$ is called a closed type.

Lemma 4 (Closedness). *$\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash v^0{:}\,\sigma^0$ implies $\mathrm{FV}(v^0) = \emptyset$.*

Proof. By induction on the derivation of $\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash v^0{:}\,\sigma^0$.

Remark 5. Let $\mathsf{V}_\tau \stackrel{\Delta}{=} \{v \in \mathsf{V}^0 | \Sigma, \Delta^{+1}; \Gamma^{+1} \vdash v{:}\,\tau^0\}$ be the set of values of type τ (in a given context $\Sigma, \Delta^{+1}; \Gamma^{+1}$). It is easy to show that the mapping $[v] \mapsto v$ is an injection of $\mathsf{V}_{[\tau]}$ into V_τ, and moreover it is *represented* by the term $open \stackrel{\Delta}{=} \lambda x.(\mathsf{let}\ [x] = x\ \mathsf{in}\ x)$, i.e. $open{:}\,[\tau] \to \tau$ and $open\ [v] \stackrel{0}{\hookrightarrow} v$.
Note also that the Closedness Lemma implies the mapping $[v] \mapsto v$ is a bijection when τ is a closed type. A posteriori, this property justifies the typing rule (close*), which in turn ensures that term $close \stackrel{\Delta}{=} \lambda x.[x]$, representing the inverse mapping $v \mapsto [v]$, has type $\sigma \to [\sigma]$.

Evaluation of Run at level 0 requires to view a value at level 1 as a term to be evaluated at level 0. The following result says that this confusion in the levels is compatible with the type system.

Lemma 5 (Demotion). *$\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash v^{n+1}{:}\,\tau^{n+1}$ implies $\Sigma, \Delta; \Gamma \vdash v^{n+1}{:}\,\tau^n$.*

Proof. By induction on the derivation of $\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash v^{n+1}{:}\,\tau^{n+1}$.

```
datatype  nat
val p = fn : nat -> real -> real ref -> unit

val p_a = fn : nat -> <real> -> <real ref> -> <unit>
val p_cg = fn : nat  -> <real -> real ref -> unit>
val p_sc = <fn x y => (y:=1.0; y:=x*!y; y:=x*!y; y:=x*!y)>
         : <real -> real ref -> unit>

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

> val p_pg = fn n => let [n]=[n] in run[p_cg n]; (* program generator *)
val p_pg = fn : nat -> real -> real ref -> unit
```

Fig. 4. The Example Written in $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$

To fully claim the *reflective* nature of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ we need also a Promotion Lemma (which, however, is not relevant to the proof of Type Safety).

Lemma 6. $\Sigma, \Delta; \Gamma \vdash e\colon \tau^n$ *implies* $e \in \mathsf{V}^{n+1}$ *and* $\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash e\colon \tau^{n+1}$.

Finally, we establish the key result relating the type system to the operational semantics. This result entails that evaluation of a well-typed program $\emptyset; \emptyset \vdash e\colon \tau^0$ cannot raise an error, i.e. $\emptyset, e \overset{0}{\hookrightarrow} \mathsf{err}$ is not derivable.

Theorem 1 (Safety). $\mu, e \overset{n}{\hookrightarrow} d$ *and* $\Sigma \models \mu$ *and* $\Sigma, \Delta^{+1}; \Gamma^{+1} \vdash e\colon \tau^n$ *imply that there exist* μ' *and* v^n *and* Σ' *such that* $d \equiv (\mu', v^n)$ *and* $\Sigma, \Sigma' \models \mu'$ *and* $\Sigma, \Sigma', \Delta^{+1}; \Gamma^{+1} \vdash v^n\colon \tau^n$.

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

2.3 The power function

While ensuring the safety of $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ requires a relatively non-trivial type system, the power examples presented at the beginning of this paper can still be expressed just as concisely as in MetaML. First, we introduce the following top-level derived forms:

- `val x = e; p` stands for ($\mathsf{let}\ [x] = [e]\ \mathsf{in}\ p$), with the following derived rules for typing and evaluation at level 0

$$\frac{\Sigma, \Delta^{\leq n}; \emptyset \vdash e\colon {\tau_1}^n \qquad \Sigma, \Delta, x\colon \tau_1^n; \Gamma \vdash p\colon {\tau_2}^n}{\Sigma, \Delta; \Gamma \vdash (\mathtt{val}\ x\ \mathtt{=}\ e\mathtt{;}\ p)\colon {\tau_2}^n} \qquad \frac{\mu, e \overset{0}{\hookrightarrow} \mu', v \qquad \mu', p[x\colon= v] \overset{0}{\hookrightarrow} \mu'', v'}{\mu, (\mathtt{val}\ x\ \mathtt{=}\ e\mathtt{;}\ p) \overset{0}{\hookrightarrow} \mu'', v'}$$

- 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).

Note that this means identifiers declared at top-level go in the closed part Δ of a context $\Sigma, \Delta; \Gamma$. We assume to have a predefined closed type real with a function times $*\colon \mathsf{real} \to \mathsf{real} \to \mathsf{real}$ and a constant $1.0\colon \mathsf{real}$. Figure 4 reconsider the example of Figure 1 used in the introduction in $\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$:

- the declarations of `p`, `p_a`, `p_cg` and `p_sc` do not require any change;
- in the declaration of `p_sp` one closedness annotation has been added;
- `p_pg` is a program generator with the same type of the conventional program `p`, but applied to a natural, say 3, returns a specialised program (i.e. `p_sp`).

3 Related Work

The problem we identify at the beginning of this paper also applies to Davies's $\lambda^{\bigcirc}$ [Dav96], which allows open code and symbolic evaluation under lambda (but has no construct for running code). Therefore, the naive addition of references leads to the same problem of scope extrusion pointed out in the Introduction.

$\mathsf{Mini\text{-}ML}^{\mathsf{BN}}_{\mathsf{ref}}$ is related to Binding-Time Analyses (BTAs) for imperative languages. Intuitively, a BTA takes a single-stage program and produces a two-stage one (often in the form of a two-level program) [JGS93,Tah00]. Thiemann and Dussart [TD] describe 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. Their 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). While both this BTA and our type system ensure that no run-time error (such as 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 seem to allow interesting computational effects at specialisation time.

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.

[CDDK86] Dominique Clement, Joelle Despeyroux, Thierry Despeyroux, and Gilles 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, 11$^{\text{th}}$ Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, July 1996. IEEE Computer Society Press.

[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.

[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.

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

[Met00] The MetaML Home Page, 2000. Provides source code and documentation online at http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html.

[Mog98] Eugenio Moggi. Functor categories and two-level languages. In *FoSSaCS '98*, volume 1378 of *Lecture Notes in Computer Science.* Springer Verlag, 1998.

[MTBS99] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim 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.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Tah99] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, July 1999.

[Tah00] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. Or, the theory of MetaML is non-trivial. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics Based Program Manipulation*, Boston, January 2000.

[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.

[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.

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