An Idealized MetaML: Simpler, and More Expressive*

Eugenio Moggi[1], Walid Taha[2], Zine El-Abidine Benaissa[2], and Tim Sheard[2]

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

[2] Oregon Graduate Institute
Portland, OR, USA
{walidt,benaissa,sheard}@cse.ogi.edu

Abstract. MetaML is a multi-stage functional programming language featuring three constructs that can be viewed as statically-typed refinements of the back-quote, comma, and eval of Scheme. Thus it provides special support for writing code generators and serves as a semantically-sound basis for systems involving multiple interdependent computational stages. In previous work, we reported on an implementation of MetaML, and on a small-step semantics and type-system for MetaML. In this paper, we present An Idealized MetaML (AIM) that is the result of our study of a categorical model for MetaML. An important outstanding problem is finding a type system that provides the user with a means for manipulating both open and closed *code*. This problem has eluded efforts by us and other researchers for over three years. AIM solves the issue by providing two type constructors, one *classifies* closed code and the other open code, and describing how they *interact*.

1 Introduction

"If thought corrupts language, language can also corrupt thought"[1]. Staging computation into multiple steps is a well-known optimization technique used in many important algorithms, such as high-level program generation, compiled program execution, and partial evaluation. Yet few typed programming languages allow us to express staging in a natural and concise manner. MetaML was designed to fill this gap. Intuitively, MetaML has a special type for code that combines some features of both *open code*, that is, code that can contain free variables, and

* The research reported in this paper was supported by the USAF Air Materiel Command, contract #F19628-96-C-0161, NSF Grant IRI-9625462, DoD contract "Domain Specific Languages as a Carrier for Formal Methods", MURST progetto cofinanziato "Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software", ESPRIT WG APPSEM

[1] George Orwell, *Politics and the English Language*, 1946.

closed code, that is, code that contains no free variables. In a statically typed setting, open code and closed code have different properties, which we explain in the following section.

Open and Closed Code A number of typed languages for manipulating code fragments have been proposed in the literature. Some have types for open code [9, 6, 3, 12], and others have types for closed code [4, 13]. On one hand, languages with open code types play an important role in the study of partial evaluation. Typically, they provide two constructs, one for building a code fragment with free variables, and one for combining such fragments. Being able to construct open fragments allows the user to force computations "under a lambda". Generally, it has been hard for such languages to include constructs for executing such code fragments, because they can contain "not-yet-bound identifiers". On the other hand, languages with closed code types play an important role in the study of run-time (machine) code generation. Typically, they include constructs for building closed code, and for executing them. Generally, in such languages there is no mechanism for forcing computations "under a lambda".

The importance of having a way to execute code within a language is best illustrated by considering the eval of Scheme. In particular, Efficient implementations of Domain-Specific or "little" languages can be developed as follows: First, build a translator from the source language to Scheme, and then use eval to execute the generated Scheme code. For many languages, such an implementation would be almost as simple as an interpreter implementation (especially if back-quote and comma are utilized), but would incur almost non of the overhead associated with an interpretive implementation.

MetaML [12, 11] provides constructs for manipulating open code and executing it, but does not distinguish between open and closed code types. But open code cannot be executed because it may contain free variables that have not been bound yet. This means that in MetaML type information is not enough to decide whether or not we can safely execute a code fragment. In what follows, we introduce MetaML, explain what it allows us to express, and where it falls short.

MetaML MetaML has three staging annotations: Brackets $\langle \_ \rangle$, Escape $\tilde{}\ \_$ and Run run _. An expression $\langle e \rangle$ defers the computation of e; $\tilde{}\,e$ splices the deferred expression obtained by evaluating e into the body of a surrounding Bracketed expression; and run e evaluates e to obtain a deferred expression, and then evaluates it. Note that $\tilde{}e$ is only legal within lexically enclosing Brackets. Finally, Brackets in types such as `<int>` are read *"Code of int"*. To illustrate, consider the following interactive session:

```
-| val rec exp = fn n => fn x =>
      if n=0 then <1> else < ~x * ~(exp (n-1) x) >;
val exp = fn  : int -> <int> -> <int>
```

```
-| val exponent = fn n =>
      <fn a => ~(exp n <a>)>;
val exponent = fn  : int  -> <int -> int>

-| val cube = exponent 3;
val cube = <fn a => a * (a * (a * 1))> : <int -> int>

-| val program = <~cube 2>
val program = <(fn a => a * (a * (a * 1))) 2> : <int>

-| run program;
val it = 8 : int
```

The function `exp` returns a code fragment representing an exponent, given an integer power `n` and a code fragment representing a base `x`. The function `exponent` is very similar, but takes only a power and returns a code fragment representing a function that takes a base and returns the exponent. The code fragment `cube` is the specialization of `exponent` to the power `3`. Next, we construct the code fragment `program` which is an application of the code of `cube` to the base `2`. Finally, the last declaration executes this code fragment.

Unfortunately, there is a problem with the above example. In particular, the very last declaration is not typable with the basic type system of MetaML [11]. Intuitively, the type system for MetaML must keep track of free variables in a code fragment, so as to ensure that programs don't get stuck. But there is no way for the type system to know that `program` is closed, hence, a conservative approximation is made, and the term is rejected by the type system.

Contribution and Organization of this Paper In previous work [12], we reported on the implementation and applications of MetaML, and later [11] presented an axiomatic semantics and a type system for MetaML and proved type-safety. However, there were still a number of drawbacks:

1. As discussed above, there is a typing problem with executing a separately-declared code fragment. While this problem is addressed in the implementation using a sound rule for top-level declarations, this solution is *ad hoc*.
2. Only a call-by-value semantics could be defined for MetaML, because substitution was a partial function, only defined when variables are substituted with values.
3. The type judgment used two indices. Moreover, it has been criticized for not being based on a standard logical system [13].

This paper describes the type system and operational semantics of An Idealized MetaML (AIM), whose design is *inspired* by a categorical model for MetaML (such a model will be the subject of another paper). AIM is strictly more expressive than any known typed multi-level language, and features:

1. An open code type $\langle t \rangle$, which corresponds to $\bigcirc t$ of $\lambda^{\bigcirc}$ [3] and $\langle t \rangle$ of MetaML;
2. A closed code type $[t]$, which corresponds to $\Box t$ of $\lambda^{\Box}$ [4];
3. Cross-stage persistence of MetaML;
4. A Run-with construct, generalizing Run of MetaML.

This work is the first to achieve a *semantically sound* integration of Davies and Pfenning's $\lambda^{\Box}$ [4] and Davies' $\lambda^{\bigcirc}$ [3], and to identify useful interactions between them. Moreover, we present important simplifications over MetaML [11], which overcome the problems mentioned above:

1. The type system uses only one level annotation, like the $\lambda^{\bigcirc}$type system [3];
2. The level Promotion and level Demotion lemmas, and the Substitution lemma, are proven in full generality and not just for the cases restricted to values. This development is crucial for a call-by-name semantics. Such a semantics seems to play an important role in the formal theory of Normalization by Evaluation [1] and Type Directed Partial Evaluation [2];
3. The big-step semantics is defined in the style in which $\lambda^{\bigcirc}$was defined [3], and does not make explicit use of a stateful renaming function;
4. Terms have no explicit level annotations.

Finally, it is straight forward to extend AIM with new base types and constants, therefore it provides a general setting for investigating *staging combinators*.

In the rest of the paper, we present the type system and establish several syntactic properties. We give a big-step semantics of AIM, including a call-by-name variant, and prove type-safety. We present embeddings of $\lambda^{\bigcirc}$, MetaML and $\lambda^{\Box}$ into AIM. Finally, we discuss related works.

2 AIM: An Idealized MetaML

The definition of AIM's types $t \in T$ and terms $e \in E$ is parameterized with respect to a signature consisting of a set of **base types** b and **constants** c:

$$
\begin{aligned}
t \in T &::= b \mid t_1 \to t_2 \mid \langle t \rangle \mid [t] \\
e \in E &::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}e \mid \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \\
&\qquad \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{unbox}\ e
\end{aligned}
$$

Where m is a natural number, and it is identified with the set of its predecessors.

The first four constructs are the standard ones in a call-by-value λ-calculus with constants. Bracket and Escape are the same as in MetaML [12, 11]. Run-With generalizes Run of MetaML, in that allows the use of additional variables x_i in the body of e if they satisfy certain typing requirements that are made explicit in the next section. Box-With and unbox are not in MetaML, but are motivated by $\lambda^{\Box}$of Davies and Pfenning [4]. We use some abbreviated forms:

$$
\begin{array}{rcl}
\mathsf{run}\ e & \textit{for} & \mathsf{run}\ e\ \mathsf{with}\ \emptyset \\
\mathsf{box}\ e & \textit{for} & \mathsf{box}\ e\ \mathsf{with}\ \emptyset \\
\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i & \textit{for} & \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \\
\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i & \textit{for} & \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\}
\end{array}
$$

$$\Gamma \vdash c\colon t_c^n \qquad \Gamma \vdash x\colon t^n \text{ if } \Gamma\ x = t^m \text{ and } m \le n \qquad \frac{\Gamma, x\colon t_1^n \vdash e\colon t_2^n}{\Gamma \vdash \lambda x.e\colon (t_1 \to t_2)^n}$$

$$\frac{\Gamma \vdash e_1\colon (t_1 \to t_2)^n \quad \Gamma \vdash e_2\colon t_1^n}{\Gamma \vdash e_1\ e_2\colon t_2^n} \qquad \frac{\Gamma \vdash e\colon t^{n+1}}{\Gamma \vdash \langle e\rangle\colon \langle t\rangle^n} \qquad \frac{\Gamma \vdash e\colon \langle t\rangle^n}{\Gamma \vdash \tilde{}e\colon t^{n+1}}$$

$$\frac{\Gamma \vdash e_i\colon [t_i]^n \quad \Gamma^{+1}, \{x_i\colon [t_i]^n | i \in m\} \vdash e\colon \langle t\rangle^n}{\Gamma \vdash \mathsf{run}\ e\ \mathsf{with}\ x_i = e_i\colon t^n}$$

$$\frac{\Gamma \vdash e_i\colon [t_i]^n \quad \{x_i\colon [t_i]^0 | i \in m\} \vdash e\colon t^0}{\Gamma \vdash \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i\colon [t]^n} \qquad \frac{\Gamma \vdash e\colon [t]^n}{\Gamma \vdash \mathsf{unbox}\ e\colon t^n}$$

Fig. 1. Typing Rules

2.1 Type System

An AIM typing judgment has the form $\Gamma \vdash e\colon t^n$, where $t \in T$, $n \in N$ and Γ is a type assignment, that is, a finite set $\{x_i\colon t_i^{n_i} | i \in m\}$ with the x_i distinct. The reading of $\Gamma \vdash e\colon t^n$ is *"term e has type t at level n in the type assignment Γ"*. We say that $\Gamma\ x = t^n$ if $x\colon t^n$ is in Γ. Furthermore, we write Γ^{+r} for the type assignment obtained by incrementing the level annotations in Γ by r, that is, $\Gamma^{+r}\ x = t^{n+r}$ if and only if $\Gamma\ x = t^n$. Figure 1 gives the typing rules for AIM. The Constant rule says that a constant c of type t_c, which has to be given in the signature, can be used at any level n. The Variable rule incorporates cross-stage persistence, therefore if x is introduced at level m it can be used later, that is, at level $n \ge m$, but not before. The Abstraction and Application rules are standard. The Bracket and Escape rules establish an *isomorphism* between t^{n+1} and $\langle t\rangle^n$. Typing Run in MetaML [11] introduces an extra index-annotation on types for counting the number of Runs surrounding an expression (see Figure 3). We avoid this extra annotation by incrementing the level of all variables in Γ. In particular, the Run rule of MetaML becomes

$$\frac{\Gamma^{+1} \vdash e\colon \langle t\rangle^n}{\Gamma \vdash \mathsf{run}\ e\colon t^n}$$

The Box rule ensures that there are no "late" free variables in the term being Boxed. This ensures that when a Boxed term is evaluated in a type-safe context, the resulting value is a closed term. The Box rule ensures that only the variables explicitly bound in the Box statement can occur free in the term e. At the same time, it ensures that no "late" free variable can infiltrate the body of a Box using one of these variables. This is accomplished by forcing the With-bound variables themselves to have a Boxed type. Note that in $\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i$ the term e may contain other free variables besides the x_i.

2.2 Properties of the Type System

The following level Promotion, level Demotion and Substitution lemmas are needed for proving Type Preservation.

Lemma 1 (Promotion). *If* $\Gamma_1, \Gamma_2 \vdash e\colon t^n$ *then* $\Gamma_1, \Gamma_2^{+1} \vdash e\colon t^{n+1}$.

Meaning that if we increment the level of a well-formed term e it remains well-formed. Furthermore, we can simultaneously increment the level of an arbitrary subset of the variables in the environment.
Demotion on e at n, written $e{\downarrow_n}$, lowers the level of e from level $n+1$ down to level n, and is well-defined on all terms, unlike demotion for MetaML [11].

Definition 1 (Demotion). $e{\downarrow_n}$ *is defined by induction on* e*:*

$$\begin{aligned}
c{\downarrow_n} &= c \\
x{\downarrow_n} &= x \\
(e_1\ e_2){\downarrow_n} &= e_1{\downarrow_n}\ \ e_2{\downarrow_n} \\
(\lambda x.e){\downarrow_n} &= \lambda x.e{\downarrow_n} \\
\langle e\rangle{\downarrow_n} &= \langle e{\downarrow_{n+1}}\rangle \\
\tilde{\ } e{\downarrow_0} &= \mathsf{run}\ e \\
(\tilde{\ } e){\downarrow_{n+1}} &= \tilde{\ }(e{\downarrow_n}) \\
(\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i){\downarrow_n} &= \mathsf{run}\ e{\downarrow_n}\ \ \mathsf{with}\ x_i = e_i{\downarrow_n} \\
(\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i){\downarrow_n} &= \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i{\downarrow_n} \\
(\mathsf{unbox}\ e){\downarrow_n} &= \mathsf{unbox}\ e{\downarrow_n}
\end{aligned}$$

The key for making demotion total on all terms is handling the case for Escape $\tilde{\ } e{\downarrow_0}$: Escape is simply replaced by Run. It should also be noted that demotion does not go into the body of Box.

Lemma 2 (Demotion). *If* $\Gamma^{+1} \vdash e\colon t^{n+1}$ *then* $\Gamma \vdash e{\downarrow_n}\colon t^n$.

Meaning that demotion of a well-formed term e is well-formed, provided the level of all free variables is decremented.
In this paper, proofs are omitted for brevity (Please see technical report for proof details [8]).

Lemma 3 (Weakening). *If* $\Gamma_1, \Gamma_2 \vdash e_2\colon t_2^n$ *and* x *is* fresh, *then* $\Gamma_1, x\colon t_1^{n'}, \Gamma_2 \vdash e_2\colon t_2^n$.

Lemma 4 (Substitution). *If* $\Gamma_1 \vdash e_1\colon t_1^{n'}$ *and* $\Gamma_1, x\colon t_1^{n'}, \Gamma_2 \vdash e_2\colon t_2^n$ *then* $\Gamma_1, \Gamma_2 \vdash e_2[x\colon= e_1]\colon t_2^n$.

This is the expected substitution property, that is, a variable x can be replaced by a term e_1, provided e_1 meets the type requirements on x.

Evaluation.

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x{:}= v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i{:}= v_i] \overset{0}{\hookrightarrow} \langle v' \rangle \quad v'\!\downarrow_0 \overset{0}{\hookrightarrow} v}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\mathord{\sim} e \overset{1}{\hookrightarrow} v}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i}{\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \mathsf{box}\ e[x_i{:}= v_i]} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{0}{\hookrightarrow} v}$$

Building.

$$\frac{e \overset{n+1}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{n+1}{\hookrightarrow} \mathsf{unbox}\ v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\lambda x.e \overset{n+1}{\hookrightarrow} \lambda x.v} \qquad x \overset{n+1}{\hookrightarrow} x$$

$$\frac{e \overset{n+1}{\hookrightarrow} v \qquad e_i \overset{n+1}{\hookrightarrow} v_i}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \mathsf{run}\ v\ \mathsf{with}\ x_i = v_i} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\mathord{\sim} e \overset{n+2}{\hookrightarrow} \mathord{\sim} v} \qquad \frac{e \overset{n+1}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle}$$

$$\frac{e_i \overset{n+1}{\hookrightarrow} v_i}{\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \overset{n+1}{\hookrightarrow} \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i} \qquad \frac{e_1 \overset{n+1}{\hookrightarrow} v_1 \quad e_2 \overset{n+1}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+1}{\hookrightarrow} v_1\ v_2} \qquad c \overset{n+1}{\hookrightarrow} c$$

Stuck.

$$\frac{e \overset{0}{\hookrightarrow} v \not\equiv \mathsf{box}\ e'}{\mathsf{unbox}\ e \overset{0}{\hookrightarrow} err} \qquad \frac{e_1 \overset{0}{\hookrightarrow} v \not\equiv \lambda x.e}{e_1 e_2 \overset{0}{\hookrightarrow} err} \qquad x \overset{0}{\hookrightarrow} err$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i{:}= v_i] \overset{0}{\hookrightarrow} v \not\equiv \langle e' \rangle}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} err} \qquad \frac{e \overset{0}{\hookrightarrow} v \not\equiv \langle e' \rangle}{\mathord{\sim} e \overset{1}{\hookrightarrow} err} \qquad \mathord{\sim} e \overset{0}{\hookrightarrow} err$$

Fig. 2. Big-Step Semantics

3 Big-Step Semantics

The big-step semantics for MetaML [12] reflects the existing implementation: it is complex, and hence not very suitable for formal reasoning. Figure 2 presents a concise big-step semantics for AIM, which is presented at the same level of abstraction as that for $\lambda^{\bigcirc}$ [3]. We avoid the explicit use of a *gensym* or *newname* for renaming bound variables: this is implicitly done by substitution.

Definition 2 (Values).

$$
\begin{array}{lcl}
v^0 \in V^0 & ::= & \lambda x.e \mid \langle v^1 \rangle \mid \mathsf{box}\ e \\
v^1 \in V^1 & ::= & c \mid x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \mathsf{run}\ v^1\ \mathsf{with}\ x_i = v_i^1 \mid \\
 & & \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i^1 \mid \mathsf{unbox}\ v^1 \\
v^{n+2} \in V^{n+2} & ::= & c \mid x \mid v^{n+2}\ v^{n+2} \mid \lambda x.v^{n+2} \mid \langle v^{n+3} \rangle \mid \tilde{}v^{n+1} \mid \\
 & & \mathsf{run}\ v^{n+2}\ \mathsf{with}\ x_i = v_i^{n+2} \mid \mathsf{box}\ e\ \mathsf{with}\ x_i = v_i^{n+2} \mid \mathsf{unbox}\ v^{n+2}
\end{array}
$$

Values have three important properties: First, a value at level 1 can be a Bracketed or a Boxed expression, reflecting the fact that terms representing open and closed code are both considered acceptable results from a computation. Second, values at level $n+1$ can contain Applications such as $\langle(\lambda y.y)\ (\lambda x.x)\rangle$, reflecting the fact that Brackets defer computations. Finally, there are no level 1 Escapes in values, reflecting the fact that having such an Escape in a term would mean that evaluating the term has not yet been completed. This is true, for example, in terms like $\langle\tilde{}(f\ x)\rangle$.

Lemma 5 (Orthogonality). *If $v \in V^0$ and $\Gamma \vdash v\colon [t]^0$ then $\emptyset \vdash v\colon [t]^0$.*

Theorem 1 (Type Preservation). *If $\Gamma^{+1} \vdash e\colon t^n$ and $e \overset{n}{\hookrightarrow} v$ then $v \in V^n$ and $\Gamma^{+1} \vdash v\colon t^n$.*

Note that in AIM (unlike ordinary programming languages) we cannot restrict the evaluation rules to closed terms, because at levels above 0 evaluation is *symbolic* and can go inside the body of binders. On the other hand, evaluation of a variable at level 0 is an error! The above theorem strikes the right balance, namely it allows open terms provided their free variables are at level above 0 (this is reflected by the use of Γ^{+1} in the typing judgment).

Having no level 1 escapes ensures that demotion is the identity on V^{n+1} as shown in following lemma. Thus, we don't need to perform demotion in the evaluation rule for Run when evaluating a well-formed term.

Lemma 6 (Value Demotion). *If $v \in V^{n+1}$ then $v{\downarrow_n} \equiv v$.*

A good property for multi-level languages is the existence of a bijection between programs $\emptyset \vdash e\colon t^0$ and program representations $\emptyset \vdash \langle v\rangle\colon \langle t\rangle^0$. This property holds for AIM, in fact it is a consequence of the following result:

Proposition 1 (Reflection). *If $\Gamma \vdash e\colon t^n$, then $\Gamma^{+1} \vdash e\colon t^{n+1}$ and $e \in V^{n+1}$. Conversely, if $v \in V^{n+1}$ and $\Gamma^{+1} \vdash v\colon t^{n+1}$, then $\Gamma \vdash v\colon t^n$.*

3.1 Call-by-Name

The difference between the call-by-name semantics and the call-by-value semantics for AIM is only in the evaluation rule for Application at level 0. For call-by-name, this rule becomes

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e[x\colon= e_2] \overset{0}{\hookrightarrow} v}{e_1\ e_2 \overset{0}{\hookrightarrow} v}$$

The Type Preservation proof must be changed for this case. However, this not problematic, since the Substitution Lemma for the AIM's type system has no *value restriction*.

Theorem 2 (CBN Type Preservation). *If* $\Gamma^{+1} \vdash e\colon t^n$ *and* $e \overset{n}{\hookrightarrow} v$ *then* $v \in V^n$ *and* $\Gamma^{+1} \vdash v\colon t^n$.

3.2 Expressiveness

MetaML's type system has one Code type constructor, which tries to combine the features of the Box and Circle type constructors of Davies and Pfenning. However, this *combination* leads to the typing problem discussed in the introduction. In contrast, AIM's type system incorporates both Box and Circle type constructors, thereby providing *correct semantics* for the following functions:

1. $\mathsf{unbox} : [t] \to t$. This function executes closed code. AIM has no function of type $t \to [t]$, thus we avoid the "collapse" of types in the recent work of Wickline, Lee, and Pfenning [13]. Such a function does not exist in MetaML.
2. $\mathsf{up} : t \to \langle t \rangle$. This function corresponds to cross-stage persistence [12], in fact it embeds any value into an open fragment, including values of functional type. Such a function does not exist in $\lambda^{\bigcirc}$. At the same time, AIM has no function of type $\langle t \rangle \to t$, reflecting the fact that open code cannot be executed. up is expressible as $\lambda x.\langle x \rangle$.
3. $\mathsf{weaken}\colon [t] \to \langle t \rangle$. The composite of the two functions above. weaken reflects the fact that closed code can always be viewed as open code. AIM has no function of type $\langle t \rangle \to [t]$.
4. $\mathsf{execute}\colon [\langle t \rangle] \to t$. This function executes closed code, and it can be defined in AIM as $\lambda x.\mathsf{run}\ x\ \mathsf{with}\ x = x$.
5. $\mathsf{build}\colon [\langle t \rangle] \to [\langle t \rangle]$ This function forces the building of an open fragment known to be closed. build is not expressible in the language, but it can be added as a new combinator with the following semantics:

$$\frac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} \langle v \rangle}{\mathsf{build}\ e \overset{0}{\hookrightarrow} \mathsf{box}\ \langle v \rangle}$$

Type Preservation is still valid with such an extension.

Now, the MetaML example presented in the Introduction can be expressed in AIM as follows:

```
-| val rec exp = box (fn n => fn x =>
      if n=0 then <1> else < ~x * ~((unbox exp) (n-1) x) >)
      with {exp=exp};
val exp = [fn]  : [int -> <int> -> <int>]

-| val exponent = box (fn n =>
```

```
      <fn a => ~((unbox exp) n <a>)>)
      with {exp=exp};
val exponent = [fn]  : [int  -> <int -> int>]

-| val cube = build (box ((unbox exponent) 3)
                     with {exponent=exponent});
val cube = [<fn a => a * (a * (a * 1))>] : [<int -> int>]

-| val program = build (box <~(unbox cube) 2>
                        with {cube=cube})
val program = [<(fn a => a * (a * (a * 1))) 2>] : [<int>]

-| execute program;
val it = 8 : int
```

In AIM, asserting that a code fragment is closed (using Box) has become part of the responsibilities of the programmer. Furthermore, Build is needed to explicitly overcome the default lazy behavior of Box. If Build was not used in the above examples, the (Boxed code) values returned for `cube` and `program` would contain level 0 Escapes. In general, it appears that the lazy behavior of Box is not needed when our primary concern is high-level program generation.

Unfortunately, the syntax is verbose compared to that of MetaML. In future work, we hope to improve the syntax based on experience using AIM.

4 Embedding Results

This section shows that other languages for staging computations can be translated into AIM, and that the embedding *respects* the typing and evaluation. The languages we consider are $\lambda^{\bigcirc}$ [3], MetaML [11], and $\lambda^{\Box}$ [4].

4.1 Embedding of $\lambda^{\bigcirc}$

The embedding of $\lambda^{\bigcirc}$ into AIM is straight forward. In essence, $\lambda^{\bigcirc}$ corresponds to the **Open fragment** of AIM:

$$t \in T_{Open} ::= b \mid t_1 \to t_2 \mid \langle t \rangle$$
$$e \in E_{Open} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e \rangle \mid \tilde{}\,e$$

The translation $(\_^{\bigcirc})$ between $\lambda^{\bigcirc}$and AIM is as follows: $(\bigcirc t)^{\bigcirc} = \langle (t^{\bigcirc}) \rangle$, $(\mathsf{next}\ e)^{\bigcirc} = \langle e^{\bigcirc} \rangle$, and $(\mathsf{prev}\ e)^{\bigcirc} = \tilde{}\,(e^{\bigcirc})$. With these identifications the typing and evaluation rules for $\lambda^{\bigcirc}$ are those of AIM restricted to the relevant fragment. The only exception is the typing rule for variables, which in $\lambda^{\bigcirc}$ is simply $\Gamma \vdash x\colon t^n$ if $\Gamma\ x = t^n$ (this reflects the fact that $\lambda^{\bigcirc}$ has no cross-stage persistence).

We write $\Gamma \vdash_{\bigcirc} e\colon t$ and $e \overset{n}{\hookrightarrow}_{\bigcirc} v$ for the typing and evaluation judgments of $\lambda^{\bigcirc}$, so that they are not confused with the corresponding judgments of AIM.

$$\Gamma \vdash c\colon (t_c, r)^n \qquad \Gamma \vdash x\colon (t,r)^n \text{ if } \Gamma\ x = (t,p)^m \text{ and } m + r \le n + p$$

$$\frac{\Gamma, x\colon (t_1, r)^n \vdash e\colon (t_2, r)^n}{\Gamma \vdash \lambda x.e\colon (t_1 \to t_2, r)^n} \qquad \frac{\Gamma \vdash e_1\colon (t_1 \to t_2, r)^n \quad \Gamma \vdash e_2\colon (t_1, r)^n}{\Gamma \vdash e_1\ e_2\colon (t_2, r)^n}$$

$$\frac{\Gamma \vdash e\colon (t,r)^{n+1}}{\Gamma \vdash \langle e\rangle\colon (\langle t\rangle, r)^n} \qquad \frac{\Gamma \vdash e\colon (\langle t\rangle, r)^n}{\Gamma \vdash \tilde{\ } e\colon (t,r)^{n+1}} \qquad \frac{\Gamma \vdash e\colon (\langle t\rangle, r+1)^n}{\Gamma \vdash \mathsf{run}\ e\colon (t,r)^n}$$

Fig. 3. MetaML Typing rules

Proposition 2 (Temporal Type Embedding). *If $\Gamma \vdash_{\bigcirc} e\colon t^n$ is derivable in $\lambda^{\bigcirc}$, then $\Gamma^{\bigcirc} \vdash e^{\bigcirc}\colon (t^{\bigcirc})^n$ is derivable in AIM.*

Proposition 3 (Temporal Semantics Embedding). *If $e \overset{n}{\hookrightarrow}_{\bigcirc} v$ is derivable in $\lambda^{\bigcirc}$, then $e^{\bigcirc} \overset{n}{\hookrightarrow} v^{\bigcirc}$ is derivable in AIM.*

4.2 Embedding of MetaML

The difference between MetaML and AIM is in the type system. We show that while AIM's typing judgments are simpler, what is typable in MetaML remains typable in AIM.

$$t \in T_{MetaML} ::= b \mid t_1 \to t_2 \mid \langle t\rangle$$
$$e \in E_{MetaML} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \langle e\rangle \mid \tilde{\ } e \mid \mathsf{run}\ e$$

A MetaML's typing judgment has the form $\Delta \vdash e\colon (t,r)^n$, where $t \in T$, $n, r \in N$ and Δ is a type assignment, that is, a finite set $\{x_i\colon (t_i, r_i)^{n_i} | i \in m\}$ with the x_i distinct. Figure 3 recalls the MetaML [11].

Definition 3 (Acceptable Judgment). *We say that a MetaML typing judgment $\{x_i\colon (t_i, r_i)^{n_i} | i \in m\} \vdash e\colon (t,r)^n$ is* **acceptable** *if and only if $\forall i \in m.\ r_i \le r$.*

Remark 1. A careful analysis of MetaML's typing rules shows that typing judgments occurring in the derivation of a judgment $\emptyset \vdash e\colon (t,r)^n$ are acceptable: In MetaML typing rules are acceptable whenever its conclusion is acceptable, simply because the index r never decreases when we go from the conclusion of a type rule to its premise, thus, we never get an environment binding with an r higher than that of the judgment.

Proposition 4 (MetaML Type Embedding). *If $\{x_i\colon (t_i, r_i)^{n_i} | i \in m\} \vdash e\colon (t,r)^n$ is acceptable, then it is derivable in MetaML if and only if $\{x_i\colon t_i^{n_i + r - r_i} | i \in m\} \vdash e\colon t^n$ is derivable in AIM.*

4.3 Embedding of $\lambda^{\Box}$

Figure 4 summarizes the language $\lambda^{\Box}$ [4]. We translate $\lambda^{\Box}$ into the **Closed**

Syntax

$$\begin{aligned}\text{Types } t \in T_\Box ::= &\ b \mid t_1 \to t_2 \mid \Box t\\ \text{Expressions } e \in E_\Box ::= &\ x \mid \lambda x.e \mid e_1\ e_2 \mid \mathsf{box}\ e \mid \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2\\ \text{Type assignments } \Gamma, \Delta ::= &\ \{x_i : t_i | i \in m\}\end{aligned}$$

Type System

$$\Delta; \Gamma \vdash_\Box x : t \ \text{ if } \Delta x = t \qquad\qquad \Delta; \Gamma \vdash_\Box x : t \ \text{ if } \Gamma x = t$$

$$\frac{\Delta; (\Gamma, x : t') \vdash_\Box e : t}{\Delta; \Gamma \vdash_\Box \lambda x.e : t' \to t} \qquad \frac{(\Delta; x : t'), \Gamma \vdash_\Box e_2 : t \quad \Delta; \Gamma \vdash_\Box e_1 : \Box t'}{\Delta; \Gamma \vdash_\Box \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 : t}$$

$$\frac{\Delta; \Gamma \vdash_\Box e_1 : t' \to t \quad \Delta; \Gamma \vdash_\Box e_2 : t'}{\Delta; \Gamma \vdash_\Box e_1\, e_2 : t} \qquad \frac{\Delta; \emptyset \vdash_\Box e : t}{\Delta; \Gamma \vdash_\Box \mathsf{box}\ e : \Box t}$$

Big-Step Semantics

$$\frac{e_1 \hookrightarrow_\Box \lambda x.e \quad e_2 \hookrightarrow_\Box v' \quad e[x := v'] \hookrightarrow_\Box v}{e_1,\ e_2 \hookrightarrow_\Box v} \qquad \lambda x.e \hookrightarrow_\Box \lambda x.e$$

$$\frac{e_1 \hookrightarrow_\Box \mathsf{box}\ e \quad e_2[x := e] \hookrightarrow_\Box v}{\mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2 \hookrightarrow_\Box v} \qquad \mathsf{box}\ e \hookrightarrow_\Box \mathsf{box}\ e$$

Fig. 4. Description of $\lambda^\Box$

fragment of AIM:

$$\begin{aligned}&t \in T_{Closed} ::= b \mid t_1 \to t_2 \mid [t]\\ &e \in E_{Closed} ::= c \mid x \mid e_1\ e_2 \mid \lambda x.e \mid \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \mid \mathsf{unbox}\ e\end{aligned}$$

Furthermore, we consider only typing judgments of the form $\{x_i : t_i^0 | i \in m\} \vdash e : t^0$ and evaluation judgments of the form $e \overset{0}{\hookrightarrow} v$. These restrictions are possible for two reasons. If the conclusion of a typing rule is of the form $\{x_i : t_i^0 | i \in m\} \vdash e : t^0$ with types and terms in the Closed fragment, then also the premises of the typing rule enjoy such properties. When e is a closed term in the Closed fragment, the only judgments $e' \overset{n}{\hookrightarrow} v'$ that can occur in the derivation of $e \overset{0}{\hookrightarrow} v$ are such that $n = 0$ and e' and v' are closed terms in the Closed fragment.

Definition 4 (Modal Type Translation). *The translation of $\lambda^\Box$ types is given by*

$$b^\Box = b \qquad (t_1 \to t_2)^\Box = t_1^\Box \to t_2^\Box \qquad (\Box t)^\Box = [t^\Box]$$

The translation of $\lambda^{\Box}$ terms depends on a set X of variables, namely those declared in the modal context Δ.

$$\begin{aligned}
x^{\Box X} &= \mathsf{unbox}\ x \qquad \textit{if } x \in X \\
y^{\Box X} &= y \qquad\qquad\ \ \textit{if } y \notin X \\
(\mathsf{box}\ e)^{\Box X} &= \mathsf{box}\ e^{\Box X}\ \mathsf{with}\ \{x = x | x \in \mathrm{FV}(e) \cap X\} \\
(\mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2)^{\Box X} &= (\lambda x. e_2^{\Box X \cup \{x\}})\, e_1^{\Box X} \\
(\lambda y.e)^{\Box X} &= \lambda y. e^{\Box X} \quad \textit{where } y \notin X \\
(e_1\, e_2)^{\Box X} &= e_1^{\Box X}\, e_2^{\Box X}
\end{aligned}$$

Proposition 5 (Modal Type Embedding). *If $\Delta; \Gamma \vdash_{\Box} e{:}\,t$ is derivable in $\lambda^{\Box}$, then $[\Delta^{\Box}], \Gamma^{\Box} \vdash e^{\Box X}{:}\,t^{\Box}$ is derivable in AIM's Closed fragment, where X is the set of variables declared in Δ, $\{x_i{:}\,t_i | i \in m\}^{\Box}$ is $\{x_i{:}\,t_i^{\Box} | i \in m\}$, and $[\{x_i{:}\,t_i | i \in m\}]$ is $\{x_i{:}\,[t_i] | i \in m\}$.*

The translation of $\lambda^{\Box}$ into the AIM's Closed fragment does not preserve evaluation *on the nose* (that is, up to syntactic equality). Therefore, we need to consider an *administrative* reduction.

Definition 5 (Box-Reduction). *$\rightarrow_{box}$ reduction is given by the rewrite rules*

$$\mathsf{unbox}\ (\mathsf{box}\ e) \rightarrow e$$
$$\mathsf{box}\ e'\ \mathsf{with}\ x_i = e_i, x = \mathsf{box}\ e, x_j = e_j \rightarrow \mathsf{box}\ e'[x{:}= \mathsf{box}\ e]\ \mathsf{with}\ x_i = e_i, x_j = e_j$$

where e is a closed term of the Closed fragment.

Lemma 7 (Properties of $\rightarrow_{box}$). *The $\rightarrow_{box}$ reduction on the Closed fragment satisfies the following properties:*

- *Subject Reduction, that is, $\Gamma \vdash e{:}\,t$ and $e \rightarrow e'$ imply $\Gamma \vdash e'{:}\,t$*
- *Confluence and Strong Normalization*
- *Compatibility with Evaluation on closed terms, that is, $e_1 \hookrightarrow v_1$ and $e_1 \overset{*}{\longrightarrow}_{box} e_2$ imply that exists v_2 s.t. $v_1 \overset{*}{\longrightarrow}_{box} v_2$ and $e_2 \hookrightarrow v_2$.*

Lemma 8 (Substitutivity). *Given a closed term $e_0 \in E_{\Box}$ the following holds:*
- *$e^{\Box X}[y{:}= e_0^{\Box\emptyset}] \equiv (e[y{:}= e_0])^{\Box X}$, where $y \notin X$*
- *$e^{\Box X \cup \{x\}}[x{:}= \mathsf{box}\ e_0^{\Box\emptyset}] \overset{*}{\longrightarrow}_{box} (e[x{:}= e_0])^{\Box X}$*

Proposition 6 (Modal Semantics Embedding). *If $e \in E_{\Box}$ is closed and $e \hookrightarrow_{\Box} v$ is derivable in $\lambda^{\Box}$, then there exists v' such that $e^{\Box\emptyset} \overset{0}{\hookrightarrow} v'$ and $v' \overset{*}{\longrightarrow}_{box} v^{\Box\emptyset}$.*

5 Related Work

Multi-stage programming techniques have been used in a wide variety of settings [12], including run-time specialization of C programs [10].

Nielson and Nielson present a seminal detailed study into a two-level functional programming language [9]. This language was developed for studying code generation. Davies and Pfenning show that a generalization of this language to a multi-level language called $\lambda^{\Box}$ gives rise to a type system related to a modal logic, and that this type system is equivalent to the binding-time analysis of Nielson and Nielson [4]. Intuitively, $\lambda^{\Box}$ provides a natural framework where Scheme's back-quote and eval can be present in a language. The semantics of our Box and Unbox correspond closely to those of back-quote and eval, respectively.

Gomard and Jones [6] use a statically-typed two-level language for partial evaluation of the untyped λ-calculus. This language is the basis for many binding-time analyses.

Glück and Jørgensen study partial evaluation in the generalized context where inputs can arrive at an arbitrary number of times rather than just two (namely, specialization-time and run-time) [5], and demonstrate that binding-time analysis in a multi-level setting can be done with efficiency comparable to that of two-level binding time analysis.

Davies extends the Curry-Howard isomorphism to a relation between temporal logic and the type system for a multi-level language [3]. Intuitively, $\lambda^{\bigcirc}$ provides a good framework for formalizing the presence of back-quote and comma in a statically typed language. The semantics of our Bracket and Escape correspond closely to those of back-quote and comma, respectively.

Moggi [7] advocates a categorical approach to two-level languages based on indexed categories, and stresses formal analogies with a categorical account of phase distinction and module languages.

References

1. U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In Rao Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, Loss Alamitos, 1991.
2. Olivier Danvy. Type-directed partial evaluation. In *ACM Symposium on Principles of Programming Languages*, pages 242–257, Florida, January 1996. New York: ACM.
3. 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, New Jersey, July 1996. IEEE Computer Society Press.
4. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *23rd Annual ACM Symposium on Principles of Programming Languages (POPL'96)*, St.Petersburg Beach, Florida, January 1996.
5. Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, 1997.

6. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.
7. Eugenio Moggi. A categorical account of two-level languages. In *MFPS 1997*, 1997.
8. Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, Department of Computer Science, Oregon Graduate Institute, October 1998. Available from `http://www.cse.ogi.edu/cgi-bin/adm/masterform`.
9. Flemming Nielson and Hanne Rijs Nielson. *Two-Level Functional Languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
10. Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. Microlanguages for operating system specialization. In *Proceedings of the SIGPLAN Workshop on Domain-Specific Languages*, Paris, January 1997.
11. 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*, Aalborg, Denmark, 13–17July 1998.
12. 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.
13. Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, Montreal, Canada, 17–19 June 1998.