Logical Modalities and Multi-Stage Programming

Zine El-Abidine Benissa* (1), Eugenio Moggi† (2), Walid Taha* (1), Tim Sheard* (1)
(1) Oregon Graduate Inst., Portland, OR, USA (2) DISI, Univ. di Genova, Genova, Italy

Abstract. Multi-stage programming is a method for improving the performance of programs through the introduction of controlled program specialization. This paper makes a case for multi-stage programming with open code and closed values. We argue that a simple language exploiting interactions between two logical modalities is well suited for multi-stage programming, and report the results from our study of categorical models for multi-stage languages.

Keywords: Multi-stage programming, categorical models, semantics, type systems (multi-level typed calculi), combination of logics (modal and temporal).

1 Introduction

Multi-stage programming is a method for improving the performance of programs through the introduction of controlled program specialization [15, 13]. MetaML was the first language designed specifically to support this method. It provides a type constructor for “code” and staging annotations for building, combining, and executing code, thus allowing the programmer to have finer control of the evaluation order.

Unfortunately, the single type constructor (⊥) that MetaML provides for code does not allow for a natural way of executing code. This means that “generated code” cannot be easily integrated with the rest of the run-time system. Previously published attempts to provide a consistent type system for MetaML where code can be executed have had limited expressivity [14].

In this paper we propose a refined method based on a study of

1. The operational semantics and pragmatics of staged computation [15, 14, 11, 13],
2. Previous work on the applications of logical modalities to staged computation [5, 4],
3. Categorical models for multi-stage languages [9, 1].

Our study has been consolidated into a new language for multi-stage programming called λBN.

1.1 Multi-Stage Programming

The main steps of multi-stage programming are:

1. Write the conventional program

   \[ \text{program: } t_S \rightarrow t_D \rightarrow t \]

   where \( t_S \) is the type of the “static” or “known” parameters, \( t_D \) is the type of the “dynamic”, or “unknown” parameters, and \( t \) is the type of the result of the program.

2. Add staging annotations to the program to derive

   \[ \text{annotated \_ program: } t_S \rightarrow \langle t_D \rangle \rightarrow \langle t \rangle \]

   In partial evaluation, this step is performed automatically, and is known as Binding-Time Analysis. However, there are reasons for carrying it out interactively (see [15]).

3. Compose the annotated program with an unfolding combinator \text{back: } \langle (A \rightarrow \langle B \rangle) \rightarrow \langle A \rightarrow B \rangle \}

   \[ \text{code \_ generator: } t_S \rightarrow \langle t_D \rightarrow t \rangle \]

4. Construct or read the static inputs:

   \[ s: t_S \]

5. Apply the code generator to the static inputs to get

   \[ \text{specialized \_ code: } \langle t_D \rightarrow t \rangle \]

6. Run the specialized code to re-introduce the generated function as a first-class value in the current environment:

   \[ \text{specialized \_ program: } t_D \rightarrow t \]
Problem and Solution In MetaML the last step is generally carried out in *ad hoc* ways \[16\]. The problem is that there is no general way for going from MetaML’s code type \( \langle A \rangle \) to a MetaML value of type \( A \). We have not been able to find reasonable models where a function \texttt{unsafe}\_run:\( \langle A \rangle \rightarrow A \) exists. Operationally, a code fragment of type \( \langle A \rangle \) can contain “free dynamic variables”. Because the code type of MetaML does not provide us with any information as to whether or not there are “free dynamic variables” in the fragment, there is no way of ensuring that this code fragment can be safely executed.

At the level of language constructs, MetaML provides a construct \texttt{Run}. \texttt{Run} allows the execution of a code fragment. For example, \texttt{run} \((3 + 4)\) is well-typed and evaluates to 7, but this construct has limited expressivity, e.g. the lambda abstraction \( \lambda x.\texttt{run} \ x \) is not typable. Without such a function code fragments declared at top-level can never be executed using well-typed terms. At the same time, we don’t want a function like \texttt{unsafe}\_run:\( \langle A \rangle \rightarrow A \), since it would break type safety for MetaML \[14\].

The crucial advance presented in this paper is that there are useful models where a function \texttt{safe}\_run:\( \langle [A] \rangle \rightarrow [A] \) exists. \texttt{Safe-Run} has the same operational behavior that \texttt{unsafe}\_run was intended to achieve, namely namely \textit{running code}. The difference is \textit{only} in the typing of this function. In a nutshell, \texttt{safe}\_run allows the programmer to exploit the fact that \textit{closed code can be safely executed}.

1.2 The Refined Method

We propose a refinement of multi-stage programming with explicit annotations about \textit{closedness}, and where these assertions are checked by the type system:

1. Write the conventional program

\[
\textit{program: } t_S \rightarrow t_D \rightarrow t
\]

Exactly the same as before.

2. Add staging annotations to the program to achieve

\[
\textit{closed}\_\textit{annotated}\_\textit{program: } t_S \rightarrow \langle t_D \rangle \rightarrow \langle t \rangle
\]

Almost the same as before. The difference is that the programmer must use the \texttt{Closed} type constructor \( \downarrow \) to demonstrate to the type system that the annotated program does not introduce any “free dynamic variables”. This means that in constructing the annotated program, the programmer will only be allowed to use \texttt{Closed} values.

3. Compose the annotated program with an unfolding combinator to get

\[
\textit{closed}\_\textit{code}\_\textit{generator: } t_S \rightarrow \langle t_D \rightarrow t \rangle
\]

Now \texttt{back} must itself be closed if we are to use it inside a closed value, that is, we use a slightly different combinator \texttt{closed}\_\texttt{back: }\[\langle (A) \rightarrow (B) \rangle \rightarrow (A \rightarrow B)\]

4. Turn the closed code-generator into

\[
\textit{generator}\_\textit{of}\_\textit{closed}\_\textit{code: } t_S \rightarrow \langle t_D \rightarrow t \rangle
\]

This is done by applying a combinator \texttt{closed}\_\texttt{apply: }\[A \rightarrow B] \rightarrow [A] \rightarrow [B] \].

5. Construct or read the static inputs as \textit{closed values}:

\[
\textit{cs: } t_S
\]

This is similar to multi-stage programming with explicit annotations. However, requiring the value to be closed is much more specific than in the original method. This means that we have to make sure that all combinators used in constructing this value are themselves closed.

6. Apply the code generator to the static inputs to get

\[
\textit{closed}\_\textit{specialized}\_\textit{code: } \langle t_D \rightarrow t \rangle
\]

7. Run the above result to get:

\[
\textit{closed}\_\textit{specialized}\_\textit{program: } t_D \rightarrow t
\]

This step exploits an interaction between the closed and code types, and it is performed by applying a function \texttt{safe}\_\texttt{run: }\[A] \rightarrow [A].

8. Forget that the specialized program is closed:

\[
\textit{specialized}\_\textit{program: } t_D \rightarrow t
\]

The step is performed by applying a function \texttt{open: }\[A] \rightarrow A.

Various examples from \[15, 11\] can be fully developed in \( \lambda ^{\text{BN}} \) (the language proposed in this paper), but not in the core type system for MetaML \[14\].

1.3 Overview

The paper is organized as follows: To begin, we introduce multi-stage languages, and review the connections that have been established between code types and logical modalities.
Next, we analyze, from a categorical point of view, the logical modalities and how they interact. Borrowing ideas from the work by Benton and others on categorical models for linear logic (and more specifically the adjoint calculus [2, 3]), we give a definition of what constitutes a categorical model for three simply typed multi-stage languages, namely $\lambda^\square$, $\lambda^\square$, and $\lambda^{BN}$, and consider some examples.

Then, we investigate the interaction between modalities and computational monads, since computational effects are a pervasive feature of programming languages. In particular, we refine the interpretation of $\lambda^{BN}$ in the presence of computational effects.

**Notation 1.1** We introduce notation and terminology used throughout the paper.

- $C, \mathcal{D}$: Categories
- $|C|$: Objects of the category
- $A, B$: Objects (in some category)
- $\mathcal{C}(A, B)$: The set of maps from $A$ to $B$
- $F, G$: Functors (between categories)
- $G F$: $G \circ F$
- $G F A$: $G (F A)$
- $F^n$: $n$ times composition of functor $F$
- $F \dashv G$: Full and faithful functor
- $F \dashv G$: An adjunction. $F$ and $G$ are the left- and right-adjoints
- $(x_n | n \in N)$: Infinite sequence
- $(x_i | i \in m)$: Finite sequence of length $m$
- $x_i$: Short for $(x_i | i \in m)$
- $x :: s$: The sequence obtained by adding $x$ in front of $s$
- $n++, n++$: $n + 1, n + 2$
- $\text{do}\{x_i \leftarrow e_i ; e\}$, ret $e$: let $x_i \leftarrow e_i$ in $e$ and $[e]$ of [8]
- $\sigma p: \prod_i MA_i \rightarrow MB$: $\sigma p(\{u_i \}) \Delta \text{ do}\{x_i \leftarrow u_i; op(x_i)\}$, i.e., the monadic extension of $\text{op}: \prod_i A_i \rightarrow MB$

### 2 Multi-Stage Languages

Multi-stage languages provide generic constructs building, combining and executing code fragments. The three languages we have studied have the following main features:

- $\lambda^\square$ [5], provides constructs for building and then executing closed code. Such a language is useful in machine-code generation.
- $\lambda^\square$ [4], provides constructs for building and combining open code fragments. Such a language is useful in high-level program generation and in-lining.
- MetaML [15, 14], extends $\lambda^\square$ by providing an additional construct for the execution of code fragments, and cross-stage persistence. Cross-stage persistence is the ability to bind a variable at "stage" $n$ and use it at "stage" $n + 1$. Both features are important for pragmatic reasons.

$\lambda^\square$ and $\lambda^\square$ have clean, logical foundations [3, 4, 7, 6]: there is a Curry-Howard isomorphism between $\lambda^\square$ and linear time temporal logic, and between $\lambda^\square$ and modal logic S4. MetaML emphasizes the pragmatic importance of being able to combine cross-stage persistence, "evaluation under lambda" (or "symbolic computation"), and being able to execute code. The combination of these three goals is not achieved by either $\lambda^\square$ or $\lambda^\square$ separately (see [15]). At the same time, MetaML had no logical foundations, nor the formal hygiene that such foundations often promote. For example, MetaML could only avoid the expressivity problem of Run through ad hoc extensions (see [16]), which demanded deeper investigation and possibly simplification.

#### 2.1 $\lambda^{BN}$ Types, Syntax, and Semantics

$\lambda^{BN}$ has the following types:

$$t \in T::= b \mid t_1 \rightarrow t_2 \mid \langle t \rangle \mid [t]$$

i.e. base types, functions, (open) code fragments, and closed values. The syntax of $\lambda^{BN}$ is as follows:

$$e \in E::= e \mid x \mid \lambda x.e \mid e_1 ; e_2 \mid \langle e \rangle \mid " e \mid \text{ up e} \mid \text{ close e with } \{x_i = e_i | i \in m\} \mid \text{ open e} \mid \text{ safe_run e}$$

The first four constructs are the standard ones in a $\lambda$-calculus with constants. Bracket and Escape allow building and combining open code. Brackets construct code, and Escape splits a code fragment into the context of a bigger code fragment. A term such as $(\text{fn x} \Rightarrow \langle ^{\sim} x, ^{\sim} x \rangle) <S>$ yields $<S, 5, 5>$ when evaluated. Up allows us to use any boxed expression at a higher level, thus providing cross-stage persistence for boxed-values. The Close-With construct asserts that $e$ is closed except for a set of variables $x_i$, each of which is bound to a closed term $e_i$. Open forgets the closedness assertion on $e$. Finally Safe-Run executes a closed code fragment and returns a closed value.

The type system of $\lambda^{BN}$ is given in Figure 1. Typing judgments have the form $\Gamma \vdash c : t^n$, where $\Gamma \equiv \{x_i : t^n_i | i \in m\}$ and $n$ is a natural called the level of
\[ \Gamma \vdash c : t^n \quad \Gamma \vdash x : t^m \text{ if } t^n = \Gamma(x) \]
\[
\Gamma, x : t^1 \vdash e : t^2
\]
\[
\Gamma \vdash \lambda x. e \,(t_1 \rightarrow t_2)^n
\]
\[
\Gamma \vdash e_1 : (t_1 \rightarrow t_2)^n \quad \Gamma \vdash e_2 : t_1^n
\]
\[
\Gamma \vdash e : (t)^n
\]
\[
\Gamma \vdash (e) : (t)^n
\]
\[
\Gamma \vdash \downarrow e : t^n
\]
\[
\Gamma \vdash \uparrow e : t^n
\]
\[
\Gamma \vdash \text{open } e : t^n
\]
\[
\Gamma \vdash \text{safe-run } e : [t]^n
\]

Figure 1: \( \lambda^{BN} \) Type System

the term. The level of a program fragment (i.e. a subterm of a closed term) is determined by the context, namely it is the number of surrounding Brackets less the number of surrounding Escapes and Ups. The big-step operational semantics of \( \lambda^{BN} \) is given in Figure 2. The set of values for \( \lambda^{BN} \) is defined as follows:

\[
v^0 \in V^0 \quad ::= \quad \lambda x.e \mid \langle v^1 \rangle \mid \text{close } v^0
\]
\[
v^1 \in V^1 \quad ::= \quad c \mid x \mid v^1 \mid \lambda x.v^1 \mid \langle v^2 \rangle \mid \text{close } e \text{ with } x_i = v^1 \mid \text{open } v^1 \mid \text{safe-run } v^1
\]
\[
v^{n+2} \in V^{n+2} \quad ::= \quad c \mid x \mid v^{n+2} \mid v^{n+2} \mid \lambda x.v^{n+2} \mid \langle v^{n+3} \rangle \mid \langle v^{n+1} \rangle \mid \text{up } v^{n+1} \mid \text{close } e \text{ with } x_i = v^{n+2} \mid \text{open } v^{n+2} \mid \text{safe-run } v^{n+2}
\]

3 Categorical Models

In this section we define what is a model of \( \lambda^{BN} \) (see Definition 3.9). We ignore computational effects, and focus instead on the logical modalities underpinning this multi-stage language.

Notation 3.1 We use the FP- prefix to indicate any 2-categorical notion (e.g. category, functor, monad, adjunction) specialized to the 2-category of categories with finite products and functors preserving them. Similarly we use the CCC- prefix to indicate any 2-categorical notion specialized to the 2-category of cartesian closed categories and functors preserving the CCC structure.

Remark 3.2 An FP-adjunction \( F \rightarrow G \) is simply an adjuction such that the left adjoint \( F \) is an FP-functor.

Definitions 3.3 and 3.5 recast in categorical terms the correspondence established by Davies and Pfening between \( \lambda^Q \) (Figure 6) and linear time temporal logic, and between \( \lambda^Q \) (Figure 5) and S4 modal logic.

Definition 3.3 A \( \lambda^Q \)-model is given by a CCC \( \mathcal{D} \) and a full and faithful CCC-functor

\[
\mathcal{D} \longrightarrow \mathcal{N} \quad \longrightarrow \quad \mathcal{D}
\]

Remark 3.4 The pattern for interpreting \( \lambda^Q \) is to interpret a type \( t \) by an object \([t]\) of \( \mathcal{D} \), namely

\[
[[t]] = N[[t]] \text{ and } [t_1 \rightarrow t_2] = [t_2]^{[[t_1]]}
\]

and a term \( \{x_i : t_i^m | i \in m\} \vdash e : t^n \) by a map in \( \mathcal{D}(\prod_{i \in m} N^m[[t_i]], N^m[[t]]) \).

The properties of \( \mathcal{N} \) ensure that one may safely confuse \( N^m[[t_1 \rightarrow t_2]] \) with \( (N^m[[t_2]])^{N^m[[t_1]]} \). This is formalizes the property of \( \Box \) in linear time temporal logic, and was also observed in [15].

Definition 3.5 A \( \lambda^Q \)-model is given by an FP-category \( \mathcal{C} \), a CCC \( \mathcal{D} \) and an FP-adjunction

\[
\mathcal{D} \longrightarrow \mathcal{T} \quad \longrightarrow \quad \mathcal{C}
\]

Remark 3.6 The definition of \( \lambda^Q \)-model is consistent with the topos-theoretic approach to modalities of [12]. The FP-adjunction induces an FP-comonad \( \mathcal{B} = \mathcal{F}G \) on \( \mathcal{D} \). \( \mathcal{B} \) is all that is needed for interpreting \( \lambda^Q \). In fact, a type \( t \) is interpreted by an object \([[t]] \) of \( \mathcal{D} \), namely

\[
[[t]] = \mathcal{B}[[t]] \text{ and } [[t_1 \rightarrow t_2]] = [[t_2]]^{[[t_1]]}
\]

and a term \( \{x_i : t_i^m | i \in m\} \vdash e : t \) is interpreted by a map in \( \mathcal{D}(\mathcal{B}(\prod_{i \in m} [[t_i]]), [[t]]) \).

The separation of typing contexts in two parts is not essential. In fact, there is a bijection (modulo semantic equality) between terms of the form \( \Delta, x : t; \Gamma \vdash e_1 : t' \) and those of the form \( \Delta; x : [t]; \Gamma \vdash e_2 : t' \), given by

\[
e_1 \mapsto \text{let } x = x \in e_1 \quad e_2 \mapsto e_2[x := \text{box } x]
\]

By analogy with the adjoint calculus \([2, 3])\), one may consider a variant of \( \lambda^Q \) in which the category \( \mathcal{C} \) and context separation have a more prominent role.
In $\lambda^{\text{BN}}$ the closed types and open code types coexist, so the key point is to clarify how the modalities of $\lambda^\square$ and $\lambda^\Diamond$ interact. The basic idea is that a $\lambda^{\text{BN}}$-model is a $\lambda^\square$ model where the category $D$ has the structure of a $\lambda^\Diamond$-model parameterized w.r.t. $C$. To formalize this notion of parameterization, we introduce the following auxiliary definition:

**Definition 3.7** Given an FP-functor $F: C \to D$ the simple $C$-indexed FP-category $D^F : \mathcal{C}^{op} \to \mathbf{Cat}$ is defined as follows

- $|D^F_X| \triangleq |D|$ and $D^F_X(A, B) \triangleq D(FX \times A, B)$.
- Composition of $g \in D^F_X(A, B)$ and $h \in D^F_X(B, C)$ is $FX \times A \xrightarrow{\langle \pi_1, g \rangle} FX \times B \xrightarrow{h} C \in D^F_X(A, C)$, while the identity for $A$ in $D^F_X$ is the second projection $\pi_2 : FX \times A \to A$.
- Substitution functor $f^* : D^F_X \to D^F_Y$ along $f \in C(Y, X)$ is given by $f^*(A) \triangleq A$ and $f^*(g) \triangleq g \circ (Ff \times \text{id})$.

$D^F$ is called simple because the action on objects of the substitution functor $f^*$ is the identity.

**Proposition 3.8** The simple indexed category $D^F$ of Definition 3.7 has:

- finite products, i.e. $\prod_{i \in m} D^F_X(A_i, B_i) \cong D^F_X(\prod_{i \in m} A_i, B_i)$
- exponentials, i.e. $D^F_X(C \times A, B) \cong D^F_X(C, BA)$, provided $D$ is CCC
- simple comprehension, i.e. $D^F_X(1, A) \cong C(X, GA)$, provided $F \dashv G$ is an FP-adjunction (c.f. [9]).

**Definition 3.9** A $\lambda^{\text{BN}}$-model is given by an FP-category $C$, a CCC $D$, an FP-adjunction

\[
\begin{array}{ccc}
G & \downarrow \ & T \\
D & \longrightarrow & C \\
\end{array}
\]

and a $C$-indexed full and faithful CCC-functor

\[
D^F \leftarrow N \rightarrow D^F
\]

Intuitively, the category $C$ is the closed universe, where actual evaluation takes place, while $D$ is the open universe, where one can define symbolic evaluation. The functor $F : C \to D$ says how the closed universe embeds into the open one, and is the key data for defining the parameterization of $D$ by $C$.

**Remark 3.10** We work mainly in $D$, therefore we write $N$ for functor on $D$ corresponding to $N_!$ via the isomorphism of categories $\mathcal{D} \cong D^F$. Furthermore, we write $NA$ for $N_X A$ when $A \in |D^F_X| = |D|$, since the action on objects (but not on maps) of the functors $N_X$ is independent of $X$.

The pattern for interpreting $\lambda^{\text{BN}}$ is like that for $\lambda^\square$ ($\lambda^{\text{BN}}$ has no splitting of contexts like in $\lambda^\Diamond$), i.e. a type $t$ is interpreted by an object $[t]$ of $D$, namely

\[
[[t]] = B[t], [\langle t \rangle] = N[t] \quad \text{and} \quad [t_1 \to t_2] = [t_2][t_1]
\]

where $B$ is the FP-comonad induced by the FP-adjunction $F \dashv G$, and a term $x_i : t^n_i \mid i \in m \vdash e : t^n$ is interpreted by a map in $D(\prod_{i \in m} N^n_i [t_i], N^n [e]_i)$.

Any $\lambda^{\text{BN}}$ model supports both cross-stage persistence for a restricted class of types (including the closed types), and the possibility of executing closed code.
Proposition 3.11 (Cross-stage persistence) In a \( \lambda^{BN} \)-model there is a canonical map \( up:FX \to NF X \) in \( \mathcal{D} \).

Proof: We have the natural isomorphisms (note that the adjunction \( F \dashv G \) is not used)

\[
\begin{align*}
\mathcal{D}(FX, FY) & \cong \mathcal{D}_X^F(1, FY) \quad \text{by definition of } \mathcal{D}_X^F, \\
\cong \mathcal{D}_X^F(1, NFY) & \quad \text{because } NF_X \text{ is full and faithful} \\
\cong \mathcal{D}_X^F(N1, NFY) & \quad \text{because } NF_X \text{ is a CCC-functor} \\
\cong \mathcal{D}_X^F(1, NFY) & \quad \text{by definition of } \mathcal{D}_X^F, \\
\cong \mathcal{D}(FX, NFY) & \quad \text{by } F \dashv G.
\end{align*}
\]

We define \( up:FX \to NF X \) as the map corresponding to the identity over \( FX \) when \( Y = X \) (in general \( up \) is not an iso).

Proposition 3.12 (Compile) In a \( \lambda^{BN} \)-model there is a canonical iso compile: \( GNA \to GA \) in \( \mathcal{C} \).

Proof: We have the natural isomorphisms

\[
\begin{align*}
\mathcal{C}(X, GA) & \cong \mathcal{D}(FX, A) \quad \text{by } F \dashv G, \\
\cong \mathcal{D}_X^F(1, A) & \quad \text{by definition of } \mathcal{D}_X^F, \\
\cong \mathcal{D}_X^F(N1, NA) & \quad \text{because } NF_X \text{ is full and faithful} \\
\cong \mathcal{D}_X^F(1, NA) & \quad \text{by definition of } \mathcal{D}_X^F, \\
\cong \mathcal{D}(FX, NA) & \quad \text{by } F \dashv G, \\
\cong \mathcal{C}(X, GNA).
\end{align*}
\]

We define compile: \( GNA \to GA \) as the map corresponding to the identity over \( GNA \) when \( X = GNA \), while the inverse of compile is the map corresponding to the identity over \( GA \) when \( X = GA \).

3.1 Examples

We give examples of \( \lambda^{BN} \)-models parameterized w.r.t. a category \( \mathcal{A} \), making explicit what additional structure or properties are needed on \( \mathcal{A} \) in each instance. For each example we define the categories \( \mathcal{C} \) and \( \mathcal{D} \), and the action on objects of the functors \( N, F \) and \( G \).

Example 3.13 Let \( N \) be the set of naturals. Given a CCC \( \mathcal{A} \) with \( N \)-indexed products, take

- \( \mathcal{C} \cong \mathcal{A} \) and \( \mathcal{D} \cong \mathcal{A}^N \), hence an object \( A \in |\mathcal{D}| \) is a sequence \( (A_n \in |\mathcal{A}|) \) and a map \( f \in \mathcal{D}(A, B) \) is a sequence \( (f_n \in \mathcal{A}(A_n, B_n)) \).
- \( NA \cong 1:1 \), where \( 1 \) is the terminal object of \( \mathcal{A} \).
- \( FX \cong (X|n \in N) \), i.e. the sequence which is constantly \( X \), while \( GA \cong \prod_{n \in N} A_n \).

Exponentials in \( \mathcal{D} \) are defined pointwise in terms of exponentials in \( \mathcal{A} \), i.e. \( (BA)_n = B^A_n \).

Example 3.14 Let \( \omega^{op} \) be the category of natural numbers with the reverse order, i.e.

\[
0 \quad \vdash \quad 1 \quad \ldots \quad n \quad \vdash \quad n+1 \quad \ldots
\]

Given a CCC \( \mathcal{A} \) with finite and \( \omega^{op} \)-limits, take

- \( \mathcal{C} \cong \mathcal{A} \) and \( \mathcal{D} \cong \mathcal{A}^\omega \), hence a map \( f \in \mathcal{D}(A, B) \) amounts to a commuting diagram

\[
\begin{align*}
A_0 & \quad \leftarrow \quad a_0 \quad A_1 \quad \ldots \quad A_n \quad \leftarrow \quad a_n \quad A_{n+1} \quad \ldots \\
& \quad \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \\
B_0 & \quad \leftarrow \quad b_0 \quad B_1 \quad \ldots \quad B_n \quad \leftarrow \quad b_n \quad B_{n+1} \quad \ldots
\end{align*}
\]

while an object of \( \mathcal{D} \) is a sequence of maps in \( \mathcal{A} \).

- \( NA \cong !_{A_0}: A \), where \( !_{A_0} \) is the map 1 \( \dashv A_0 \) in \( \mathcal{A} \).
- \( FX \cong (id: X \dashv X|n \in N) \), i.e. the sequence which is constantly \( id_X \), while \( GA \cong \lim_{n \in \omega^{op}} A_n \).

Exponentials in \( \mathcal{D} \) are not defined pointwise, but existence of exponentials and finite limits in \( \mathcal{A} \) ensures that \( \mathcal{D} \) has exponentials (and finite limits). In this model we have a natural transformation \( up: \mathcal{A} \to NA \), namely \( up_0 \cong !_{A_0}: A_0 \to 1 \) and \( up_{n+1} \cong a_n: A_{n+1} \to A_n \), which provides cross-stage persistence for arbitrary types.

Finally we give an example which is both a \( \lambda^{BN} \)- and \( \lambda^{C} \)-model, but fails to be a \( \lambda^{BN} \)-model. More precisely, it has the structure of a \( \lambda^{BN} \)-model, but the \( \mathcal{C} \)-indexed functor \( N \) fails to be full and faithful, so we do not have the iso compile: \( GNA \to GA \) of Proposition 3.12.

Example 3.15 Given a CCC \( \mathcal{A} \) with finite limits, take

- \( \mathcal{C} \cong \mathcal{A}^\omega \) and \( \mathcal{D} \cong \mathcal{A}^N \)
- \( NA \cong 1:1 \), where \( 1 \) is the terminal object of \( \mathcal{A} \).
- \( (FX)_n \cong X_n \), i.e. forget the maps \( x_n: X_{n+1} \to X_n \), while \( (GA)_n \cong \prod_{i \in n} A_i \) with the obvious projection \( \pi: (GA)_{n+1} \to (GA)_n \).

\[ \text{6} \]
\[ [\Gamma \vdash e : t^n] \triangleq \pi_n : C \rightarrow N^n[t_n] \]
\[ [\Gamma \vdash \text{run } e : t^n] \triangleq \text{run}_n \circ f : C \rightarrow N^n(BA) \]
\[ [\Gamma \vdash \text{safe-run } e : [t^n]] \triangleq \text{run}_n \circ f : C \rightarrow N^n(BA) \]

The action of \( N_X \) on maps of \( D_X^\land \), sends \( f \in D_X^\land(A, B) \) to \( g \in D_X^\land(NA, NB) \), where \( g_0 = \lambda X_0 \times 1 \rightarrow 1 \) and
\[
g_{n+} = X_{n+} \times A \xrightarrow{x_n \times id} X_n \times A \xrightarrow{f_n} B_n.
\]

3.2 Interpretation of terms

We have already given the interpretation of types in a \( \lambda^{BN} \)-model without computational effects, namely
\[
[[t]] = B[t] , [[t^n]] = N^n[t] \text{ and } [t_1 \rightarrow t_2] = [t_2][t_1^n]
\]
This section gives the corresponding interpretation of terms. Before doing that, we introduce some auxiliary maps in \( D \), which simplify the definition of the interpretation, and clarify the similarities with the interpretation of the simply typed \( \lambda \)-calculus in a CCC.

Given \( \prod_i A_i \rightarrow A \) we define
\[
op_n \triangleq N^n op \prod_i N^n A_i \rightarrow N^n A
\]
where we exploit that \( N \) preserves finite products.

- \( \lambda_n : (N^n B)^{N^n A} \rightarrow N^n B^A \). Since \( N \) preserves \( \text{CCC} \) structure, \( \lambda_n \) is the iso \( (N^n B)^{N^n A} \rightarrow N^n B^A \).
- \( @_n \triangleq \text{eval}_n : N^n B^A \times N^n A \rightarrow N^n B \). Since \( N \) preserves \( \text{CCC} \) structure, \( @_n \) is (up to iso) an instance of evaluation \( \text{eval} : (N^n B)^{N^n A} \times N^n A \rightarrow N^n B \).
- \( \text{open}_n \triangleq s_n : N^n B^A \rightarrow N^n A \), where \( e : BA \rightarrow A \) is the co-unit of the co-monad \( B \).
- \( \text{close}_n(f) \triangleq (Bf \circ \delta_n) : \prod_i N^n BA_i \rightarrow N^n BC \), where \( f : \prod_i BA_i \rightarrow C \) and \( \delta : BA \rightarrow B^2A \) is the co-multiplication of \( B \), and we exploit that \( N \) and \( B \) preserve finite products.
- \( \text{up}_n : N^n BA \rightarrow N^{n+} BA \), where \( \text{up} : BA \rightarrow NBA \) is the map of Proposition 3.11.
- \( \text{run}_n \triangleq (\text{F} \, \text{compile})_n : N^n BA \rightarrow N^n BA \), where \( \text{compile} : GNA \rightarrow GA \) is the iso of Proposition 3.12.

Figure 3 defines the interpretation of a well-formed term \( \Gamma \vdash e : t^n \) by induction on the typing derivation in the type system of Figure 1.
4 Modalities and monads

We have given a simplified interpretation of $\lambda^{BN}$ (and its multi-stage sub-languages) in the absence of computational effects. This interpretation is the analog of the interpretation of the simply typed $\lambda$-calculus in a CCC. However, we are interested in multi-stage programming languages, like Mini-ML$^\diamond$ [5], Mini-ML$^\circ$ [4], and MetaML [15], where logical modalities coexist with computational effects. In this section we define monadic $\lambda^{BN}$-models and give a monadic CBV interpretation of $\lambda^{BN}$ in such models, extending the interpretation sketched in [8].

Definition 4.1 A monadic $\lambda^{BN}$-model consists of a $\lambda^{BN}$-model equipped with

- a strong monad $\mathcal{M}$ over $\mathcal{D}$ such that the canonical map $\mathcal{M}N\mathcal{B}A \rightarrow (\mathcal{M}\mathcal{N}\mathcal{B})^{\mathcal{N}A}$ is an iso, and we call $\lambda_\ast: (\mathcal{M}\mathcal{N}\mathcal{B})^{\mathcal{N}A} \rightarrow \mathcal{M}\mathcal{N}\mathcal{B}A$ its inverse
- a strong natural transformation $\sigma: \mathcal{B}\mathcal{M}\mathcal{A} \rightarrow \mathcal{M}\mathcal{B}\mathcal{A}$ respecting the structure of the strong monad $\mathcal{M}$ and the FP-coMonad $\mathcal{B}$, i.e.

\[
\begin{array}{c c c c c c}
\mathcal{M}\mathcal{A} & \mathcal{B}\mathcal{M}^2\mathcal{A} & \sigma & \mathcal{M}\mathcal{B}\mathcal{A} & \mathcal{M}\sigma & \mathcal{M}^2\mathcal{B}\mathcal{A} \\
\mathcal{M}\epsilon & \mathcal{B}\mu & & \mathcal{M}\mu & \\
\mathcal{B}\mathcal{M}\mathcal{A} & \sigma & \mathcal{M}\mathcal{B}\mathcal{A} & \mathcal{B}\mu & \mathcal{M}\delta & \eta \\
\mathcal{B}^2\mathcal{M}\mathcal{A} & \mathcal{B}\sigma & \mathcal{M}\mathcal{B}\mathcal{A} & \sigma & \mathcal{B}\mathcal{M}^2\mathcal{A} & \mathcal{B}\mathcal{A}
\end{array}
\]

Remark 4.2 The intuition here is that the strong monad $\mathcal{M}$ models conventional computations, i.e. those at level 0. Staged computations are achieved by an alternation of $\mathcal{M}$ and $\mathcal{N}$, namely we define the type $\mathcal{M}_n\mathcal{A}$ of n-staged computations of type $\mathcal{A}$ as $(\mathcal{M}\mathcal{N})^n\mathcal{M}\mathcal{A}$. In general, $\mathcal{M}_n$ is not a monad, however the natural transformations $\eta$ and $\mu$ for $\mathcal{M}$ induce natural transformations $\eta_n: \mathcal{N}^n\mathcal{A} \rightarrow \mathcal{M}_n\mathcal{A}$ and $\mu_n: (\mathcal{M}^2\mathcal{N})^n\mathcal{M}^2\mathcal{A} \rightarrow \mathcal{M}_n\mathcal{A}$. The iso $\lambda_\ast: (\mathcal{M}\mathcal{N})^{\mathcal{N}A} \rightarrow \mathcal{M}\mathcal{N}^A$ refines the iso $(\mathcal{N}^A)^{\mathcal{N}A} \rightarrow \mathcal{N}^A$ corresponding to preservation of exponentials by $\mathcal{N}$. $\lambda_\ast$ is essential to define the interpretation of $\lambda$-abstraction at level $n > 0$, and also to define the analog of let: $(\mathcal{M}^A)^{\mathcal{X}} \times \mathcal{M}\mathcal{A} \rightarrow \mathcal{M}\mathcal{B}$ for $\mathcal{M}_n$, namely let$_n: ((\mathcal{M}\mathcal{N})^{\mathcal{N}A})^{\mathcal{X}A} \times \mathcal{M}_n\mathcal{A} \rightarrow \mathcal{M}_n\mathcal{B}$.

The natural transformation $\sigma: \mathcal{B}\mathcal{M}\mathcal{A} \rightarrow \mathcal{M}\mathcal{B}\mathcal{A}$ and its properties can be intuitively justified as follows. Think of $\mathcal{B}$ as the subset of values of type $\mathcal{A}$ without “free dynamic variables”, so $\sigma$ is saying that a computation without “free dynamic variables” is guaranteed to return a value without “free dynamic variables”. $\sigma$ is essential for interpreting Safe-Run and Close-With.

Example 3.13 fails to extend to a monadic $\lambda^{BN}$-model for monads as simple as lifting, because $\sigma$ does not exists. However, we can extend the $\lambda^{BN}$-model of Example 3.14:

Example 4.3 Given a strong monad $\mathcal{M}$ over $\mathcal{A}$ such that

- $\mathcal{M}$ preserves pullbacks, and the commuting square

\[
\begin{array}{c c c c}
\mathcal{M}(\mathcal{B}^A) & \mathcal{M}(\mathcal{B})^A \\
\mathcal{M}\mathcal{I} & (\mathcal{M}\mathcal{I})^A \\
\mathcal{M}\mathcal{I} & (\mathcal{M}\mathcal{I})^A \\
\end{array}
\]

is a pullback, where $k(u) \triangleq \lambda x: \mathcal{A}\mathcal{.}u$ and $e(u) \triangleq \lambda x: \mathcal{A}\.f \leftarrow u; \text{ret}(fx)$

- $\mathcal{M}$ preserves $\omega^{op}$-limits

the induced strong monad $\mathcal{M}$ over $\mathcal{A}^{\omega^{op}}$, i.e.

\[
\begin{array}{c c c c c c c}
\mathcal{M}\mathcal{A} & \mathcal{M}\mathcal{A}_0 & \mathcal{M}\mathcal{A}_1 & \cdots & \mathcal{M}\mathcal{A}_n & \mathcal{M}\mathcal{A}_{n+1} & \cdots \\
\end{array}
\]

satisfies the additional requirements for having a monadic $\lambda^{BN}$-model, namely

- the first property of $\mathcal{M}$ over $\mathcal{A}$ ensures the existence of the iso $\lambda_\ast: (\mathcal{M}\mathcal{N})^{\mathcal{N}A} \rightarrow \mathcal{M}\mathcal{N}^A$ (exponentials in $\mathcal{A}^{\omega^{op}}$ are computed using exponentials and pullbacks in $\mathcal{A}$);

- the second property of $\mathcal{M}$ over $\mathcal{A}$ allows us to define the natural transformation $\sigma: \mathcal{B}\mathcal{M}\mathcal{A} \rightarrow \mathcal{M}\mathcal{B}\mathcal{A}$ as the iso corresponding to preservation of $\omega^{op}$-limits by $\mathcal{M}$.

Remark 4.4 Many monads over the category of cpos (e.g. lifting) satisfy the additional properties required in Example 4.3, moreover several monad transformers (e.g. for adding a global state or exceptions) preserve such properties. However, there are monads which fail to satisfy the additional properties, notably powerdomains and continuations.
Interpretation of types. A type \( t \) is interpreted
(as usual) by an object \([t]\) of \( \mathcal{D} \), namely:
\[
[t] = B[t'], [\bar{t}] = \text{NM}([t], [t_1 \rightarrow t_2]) = (M[t_2])^{[t_1]}I
\]
In a monadic \( \lambda^{\text{BN}} \)-model a term \( \{x_i; t^n_i | i \in m\} \vdash e : t^n \)
is interpreted by a map in \( \mathcal{D}(\prod_{i \in m} \text{NM}\{t_i, M_n[t_i]\}) \).

Remark 4.5 This interpretation is a refinement of the
interpretation given in Section 3.2, which is recovered
by replacing \( M \) with the identity monad, and it extends
the monadic CBV interpretation of the simply typed \( \lambda \)-calculus (in a CCC with a strong monad).

Auxiliary maps. We introduce some auxiliary
maps in \( \mathcal{D} \) (see also Notation 1.1), similar to those
given in Section 3.2. Given \( op : \prod_i A_i \rightarrow MB \)
we define \( \psi \eta_n : \prod_i M_n A_i \rightarrow M_n B \) by induction on \( n \):
\[
\begin{align*}
0 & : \prod_i M A_i \xrightarrow{\psi \eta} MB \\
n+ & : \prod_i M_{n+1} A_i \xrightarrow{\psi \eta_n} M N \prod_i M_n A_i \xrightarrow{M N \psi \eta_n} M_{n+1} B
\end{align*}
\]
where \( \psi : \prod_i MA_i \rightarrow M(\prod_i A_i) \) is given by \( \psi(u_i | i) \triangleq \text{do}\{x_i \leftarrow u_i; \text{ret}(x_i | i)\} \), and we exploit that \( N \) preserves
finite products.

\[ \eta : N^n A \rightarrow M^n A \]
is given by induction:
\[
\begin{align*}
0 & : A \xrightarrow{\eta} MA \\
n+ & : N^{n+1} A \xrightarrow{\eta} M N^{n+1} A \xrightarrow{M N \eta_n} M_{n+1} A
\end{align*}
\]
where \( \eta : A \rightarrow MA \) is the unit of the monad \( M \).

\[ \lambda : (M_B)^n A \rightarrow M^n (MB)^A \]
is given by induction:
\[
\begin{align*}
0 & : (MB)^A \xrightarrow{\eta} M (MB)^A \\
n+ & : (M_{n+1} B)^{n+1} A \xrightarrow{\lambda \eta_n} M N (M_B)^n A \xrightarrow{M N \lambda} M_{n+1} (MB)^A
\end{align*}
\]

\[ \sigma_n \triangleq \text{eval}_n : M_n (MB)^A \times M_n A \rightarrow M_n B, \]
where \( \text{eval} : (MB)^A \times A \rightarrow MB \) is evaluation.

\[ \text{open}_n \triangleq \text{open}_n : M_n e : M_n B A \rightarrow M_n A, \]
where \( e : B A \rightarrow A \) is the co-unit for \( B \).

5 Discussion
Searching for a categorical semantics of \( \text{MetaML} \) has
benefited us in a number of ways:

- It has suggested simplifications and extensions. We
  have simplified the type system of \( \text{MetaML} \) and
  proposed an extension with closed code types called
  \( AIM \) (see [11]). \( \lambda^{\text{BN}} \) is the result of further simplification
  of \( AIM \) and the associated model.

- Explaining multi-stage languages in terms of more
  primitive concepts, namely logical modalities (in the
  sense that the modalities are characterized by
  universal properties) and computational monads. We
  have shown that a simple interaction between the
  two modalities in our models accounts for execution
  of closed code. Finally, we have pointed out what
  kinds of computational effects should be expected to
  integrate easily with multi-stage languages.

5.1 \( \lambda^{\text{BN}} \) Compared to \( AIM \)
Recently, we presented \( AIM \) (An Idealized MetaML)
[11], which extends \( \text{MetaML} \) with an analog of
the Box type of \( \lambda^{\Box} \) yielding a more expressive language,
yet has a simpler typing judgment than \( \text{MetaML} \). We
have shown that we can embed all three languages into
\( AIM \). \( \lambda^{\text{BN}} \) can be viewed as a cut-down version of \( AIM \)
which we believe is sufficiently expressive for the
purposes of multi-stage programming.

The Closed type constructor of \( \lambda^{\text{BN}} \) is essentially a
strict version of the Box type constructor of \( AIM \). In
\( AIM \), the Box construct delayed its argument. To the
programmer, this meant that there were two “code”
types. This causes some confusion from the point of view of multi-stage programming, because manipulating values of type $\llbracket A \rrbracket$ would be read as “closed code of open code of $A$”. Not only is this a cumbersome reading, it makes it harder for the programmer to reason about when computations are performed. In $\lambda^B$, we have made the pragmatic decision that Closed should not delay its argument, and so, types such as $\llbracket A \rrbracket$ can be read as simply “closed code of $A$”. In other words, we propose to use the Necessity modality only for asserting closedness, and not for delaying evaluation.

Another difference is that AIM was a “superset” of the three languages that we had studied (i.e. $\lambda^O$, $\lambda^B$, and MetaML), while $\lambda^B$ is not. We have shown that $\lambda^O$ can be embedded into the open fragment of AIM, and $\lambda^B$ into the closed fragment. This establishes a strong relation between the closed code and open code types of AIM and the Necessity and Next modalities of modal and temporal logic. The embedding of $\lambda^O$ and $\lambda^B$ in AIM can be turned into an embedding in $\lambda^B$ (the embedding of $\lambda^B$ needs to be modified to take into account the fact that Closed in $\lambda^B$ is strict, i.e. it does not delay the evaluation of its argument). On the other hand, the embedding of MetaML in AIM cannot be adapted for the following two reasons:

1. $\lambda^B$ does not have full cross-stage persistence. Not having cross-stage persistence simplifies the model. At the same time, from the pragmatic point of view, cross-stage persistence for closed types is sufficient.

2. $\lambda^B$ does not have Run. We were not able to find a general categorical interpretation for this construct, though [1] shows how to interpret Run in $\mathcal{A}^\omega$. At the same time, the pragmatic need for Run disappears in the presence of safe run.

Acknowledgment. We would to thank PacSoft members for their comments on the different drafts of this paper. We would also like to thank the referees for their suggestions and corrections.

References


\[ \Delta; \Gamma \vdash c : t_c \quad \Delta; \Gamma \vdash x : t \text{ if } t = \Delta(x) \text{ or } \Gamma(x) \]
\[ \Delta; \Gamma, x : t_1 \vdash c : t_2 \quad \Delta; \Gamma \vdash e : t \]
\[ \Delta; \Gamma \vdash \lambda x. e : t_1 \rightarrow t_2 \quad \Delta; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \]

Figure 5: \( \lambda^\square \) Type System

\[ \Gamma \vdash e : t^n \quad \Gamma \vdash x : t^n \text{ if } t^n = \Gamma(x) \]
\[ \Gamma, x : t^n_1 \vdash e : t^n_2 \]
\[ \Gamma \vdash \lambda x. e : (t_1 \rightarrow t_2)^n \]
\[ \Gamma \vdash e_1 : (t_1 \rightarrow t_2)^n \quad \Gamma \vdash e_2 : t^n_1 \]
\[ \Gamma \vdash e_1 \ e_2 : t^n_2 \]
\[ \Gamma \vdash \langle e \rangle : \langle t \rangle^n \quad \Gamma \vdash \langle e \rangle : \langle t \rangle^n \]

Figure 6: \( \lambda^\square \) Type System

\[ \Gamma \vdash x : t^n \text{ if } t^n = \Gamma(x) \text{ and } m \leq n \]
\[ \Gamma^+ \vdash e : \langle t \rangle^n \]
\[ \Gamma \vdash \text{run } e : t^n \]

Figure 7: MetaML Type System (\(+ Figure 6\))

\[ \Gamma \vdash x : t^n \text{ if } t^n = \Gamma(x) \text{ and } m \leq n \]
\[ \Gamma \vdash e_i : [t_i]^m \quad \Gamma^+, \{ x_i : [t_i]^m | i \in m \} \vdash e : \langle t \rangle^n \]
\[ \Gamma \vdash \text{run } e \text{ with } x_i = e_i : t^n \]
\[ \Gamma \vdash e_i : [t_i]^m \quad \{ x_i : [t_i]^0 | i \in m \} \vdash e : t^0 \]
\[ \Gamma \vdash \text{box } e \text{ with } x_i = e_i : t^n \]
\[ \Gamma \vdash \text{unbox } e : t^n \]

Figure 8: AIM Type System (\(+ Figure 6\))
A Multi-Stage Languages

For completeness, this appendix reproduces the syntax and type system of the multi-stage languages $\lambda^\Box$, $\lambda^\Box$, MetaML and AIM (see [11, 10] for details).

We adopt the following unified notation for types:

$$t \in T::= b \mid t_1 \to t_2 \mid \langle t \rangle \mid [t]$$

i.e. base types, functions, open code fragments, and closed code fragments.

$\lambda^\Box$ of [5] features function and closed code types. Typing judgments have the form $\Delta; \Gamma \vdash e : t$, where $\Delta; \Gamma \equiv \{x_i : t_i \mid i \in m\}$. The syntax for $\lambda^\Box$ is as follows:

$$e \in E::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \text{box } e \mid \text{let } box \ x = e_1 \text{ in } e_2$$

The type system of $\lambda^\Box$ is given in Figure 5.

$\lambda^\Box$, MetaML and AIM feature function and open code types. Typing judgments have the form $\Gamma \vdash e : t^n$, where $\Gamma \equiv \{x_i : t^n_i \mid i \in m\}$ and $n$ is a natural called the level of the term. The syntax for $\lambda^\Box$ is as follows:

$$e \in E::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e \rangle \mid ^* e$$

MetaML [15, 14] uses a more relaxed type rule for variables than $\lambda^\Box$, in that variables can be bound at a level lower than the level where they are used. This is called cross-stage persistence. Furthermore, MetaML extends the syntax of $\lambda^\Box$ to

$$e \in E::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e \rangle \mid ^* e \mid \text{run } e$$

AIM [11] extends MetaML with an analog of the Box type of $\lambda^\Box$ yielding a more expressive language, and yet has a simpler typing judgment than MetaML. The syntax of AIM extends that of MetaML as follows:

$$e \in E::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e \rangle \mid ^* e \mid \text{run } e$$

Run-With generalizes Run of MetaML, in that it allows the use of additional variables $x_i$ in the body of $e$ if they satisfy certain typing requirements.

The type systems of $\lambda^\Box$, MetaML and AIM are given in Figure 6, 7 and 8, while the big-step operational semantics of AIM and its sub-languages is in Figure 9.

B Technical Lemmas

We state some technical lemmas for $\lambda^\Box$, which are adaptations of those established for AIM in [10]. They are needed to establish Subject Reduction.

First, the properties of the type system:

**Lemma B.1** (Weakening) If $\Gamma_1, \Gamma_2 \vdash e_2 : t_2^n$ and $x$ is fresh, then $\Gamma_1, x : t_1^n, \Gamma_2 \vdash e_1 e_2 : t_2^n$.

**Proposition B.2** (Substitution) If $\Gamma_1 \vdash e : t_1^n$ and $\Gamma_1, x : t_1^n, \Gamma_2 \vdash e : t_2^n$ then $\Gamma_1, \Gamma_2 \vdash e[x : e'] : t_2^n$.

**Proof:** By induction over the derivation of the judgment $\Gamma_1, x : t_1^n, \Gamma_2 \vdash e : t_2^n$.

Second, the properties of the operational semantics:

**Lemma B.3** (Values)

1. If $v \in V^n$ then $v \vdash v$
2. If $e \vdash e'$ then $e' \in V^n$
Proof: The first is by induction over the derivation of $v \in V^m$. The second is by induction over the derivation of $e \xrightarrow{n} e'$.

In $\lambda^{BN}$ one cannot define demotion (as in AIM). This is not a problem, since the Demotion Lemma was used only in the case run $e$ with $x_i = e_i$ of Subject Reduction for AIM.

**Proposition B.4 (Reflection)**

1. If $v \in V^{n+}$ and $\Gamma^+ \vdash v: t^{n+}$, then $\Gamma \vdash v: t^n$.
2. If $\Gamma \vdash e: t^n$ then $\Gamma^+ \vdash e: t^{n+}$ and $e \in V^{n+}$.

**Proof:** The first is by induction on the derivation of $\Gamma^+ \vdash v: t^{n+}$, and case analysis on $v \in V^{n+}$. The second is by induction on the derivation of $\Gamma^+ \vdash e: t^n$.

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

**Proof:** Trivial from the definition of values.

**Theorem B.6 (Type Preservation)** If $\Gamma^+ \vdash e: t^n$ and $e \xrightarrow{n} v$ then $\Gamma^+ \vdash v: t^n$.

**Proof:** Induction on the derivation of $e \xrightarrow{n} v$. The case for application uses Substitution. The case for Up involves Orthogonality, Reflection, Weakening, in addition to applying the induction hypothesis. The case for Safe-Run involves Reflection.

## C An example

In this section, we illustrate the multi-stage programming approach as presented in the introduction, subsection 1.1. We apply the approach to the power function defined recursively as follows:

(* int -> real -> real *)

fun exp n x =
  if n = 0 then 1.0
  else if even(n) then sqx (exp (n div 2) x)
      else x * (exp (n-1) x);

(* real -> real *)

fun sqx x = x * x

First we develop the example in MetaML, and show the limitations of its type system when building and executing code are combined in one function. Then we develop the same example in $\lambda^{BN}$, showing how the problems are overcome. The examples are written in SML syntax extended with multi-stage annotations.

### C.1 In MetaML

1. Annotating the function exp with staging annotations requires analysis of the program with respect to static and dynamic computations. The annotations distinguish computations that can be performed using only the static argument (static computations) and computations that need to be delayed to the next stage (dynamic computations). Some or all of the program might be reformulated to improve the quality of the generated program. For instance, the function sqx has been rewritten in order to avoid code duplication.

   (* int -> <real> -> <real> *)
   fun exp1 n x =
      if n = 0 then 1.0
      else if even(n) then sqx1 (exp1 (n div 2) x)
         else x * (exp1 (n-1) x);

   (* <real> -> <real> *)
   fun sqx1 x = <let val y = *x in y * y end>

2. The third step consists only of the composition of the functions back and sqx1. The function back is polymorphic in the MetaML system, which has been extended to support Hindley-Milner polymorphism. This extension is beyond the scope of this paper.

   (* (<'a> -> <'b>) -> <'a -> 'b> *)
   fun back f = <fn x => *f x>

   (* int -> <real -> real> *)
   fun exp2 = back o exp1

3. Now, we specialize the function exp2 to a specific exponent (in our case 5). The generated code is pretty printed by MetaML system.

   (* <real -> real> *)
   val code_of_power5 = exp2 5

   val it = <fn x =>
         <let val a =
           <let val b = x * x
           in b * b end
         in x * a end>

4. Unfortunately, power5 as defined below cannot be typed in the core MetaML type system because the Run construct disallows the use of free variables such as exp2.

   The MetaML system can type power5 using an ad hoc extension of the type system specially designed for top-level let-bindings.
(* real -> real *)
val power5 = run(exp2 5)

A cut-and-paste of the generated code allows us to run it. We obtain the function power5 of type real -> real.

(* real -> real *)
val power5 = run <fn x ->
  let val a =
  let val b = x * x
  in b * b end
in x * a end>

(* int -> real -> real *)
fun power n = run (exp2 n)

The function power above cannot be expressed in the MetaML system because of the free variable n and exp2. power composes the specialization of the program on the first argument and the execution of the specialized code on the second argument. However, MetaML fails to provide such desirable functions in a multi-stage programming language.

C.2 In \(\lambda^B_N\)

1. Annotating the function exp in \(\lambda^B_N\) is similar to MetaML but a bit more tedious than MetaML. We need to tell the type system that the functions sqri and exp1 are closed. Note that exp1 uses the function even which also needs to be of closed type in order to be used within a closed expression.

(* [int -> <real -> <real>] *)
val exp1' =
close let val even = open(even')
  fun sqri x =
    <let val y = ~x in y * y end
  fun exp1 n x =
    if n=0 then <1.0>
    else if even(n)
      then sqri(exp1 (n div 2) x)
      else <"x * ~"(exp1 (n-1) x)>
    in exp1 end
  with even' = even'

2. This step is also similar to MetaML. In \(\lambda^B_N\), we need to compose two closed functions. We use for the combinators closed-compose of type ['a -> 'b] -> ['b -> 'c] -> ['a -> 'c] for composition.

(* [int -> <real -> real>] *)
val exp2' = closed_compose closed_back exp1'

fun closed_compose f' g' =
close let val f = open f'
  val g = open g'
  in fn x => f(g(x)) end
  with f' = f', g' = g'

3. The next step consists of the distribution of close type in order to extract the static argument.

(* [int] -> [<real -> real>] *)
val exp3' = closed_apply(exp2')

(* ['a -> 'b] -> ['a] -> ['b] *)
fun closed_apply f = fn x =>
close let val f = open f'
  val x = open x'
  in (f x) with x' = x', f' = f'

4. Now we apply the static argument, which is [5]. The specialized code is the result of this application.

(* [<real -> real>] *)
val code_of_power5 = exp3' [5]
val it = [fn x =>
  let val a =
    let val b = x * x
    in b * b end
  in x * a end>]

5. We execute the code using the safe_run construct. The result of this execution is a function of type [real -> real].

(* [real -> real] *)
val closed_power5 = safe_run(exp3' [5])

6. Finally, we forget that closed_power5 is closed using the open construct.

(* real -> real *)
val power5 = open closed_power5

(* [int] -> real -> real *)
val power n = open(safe_run (exp3' n))

In contrast to MetaML, power can be expressed in \(\lambda^B_N\). It first specializes the program on the exponent then executes the specialized program on the second argument.

14
To conclude, the combination of the construction and execution of code within one function is the major contribution of $\lambda^{BN}$ over MetaML. The price of achieving this combination is a bit more cumbersome syntactic sugar.

Contents

1 Introduction 1
   1.1 Multi-Stage Programming . . . . . . . 1
   1.2 The Refined Method . . . . . . . 2
   1.3 Overview . . . . . . . 2

2 Multi-Stage Languages 3
   2.1 $\lambda^{BN}$ Types, Syntax, and Semantics . . . 3

3 Categorical Models 4
   3.1 Examples . . . . . . . 6
   3.2 Interpretation of terms . . . . . . . 7

4 Modalities and monads 8

5 Discussion 9
   5.1 $\lambda^{BN}$ Compared to AIM . . . . . . . 9

A Multi-Stage Languages 12

B Technical Lemmas 12

C An example 13
   C.1 In MetaML . . . . . . . . . . . . . 13
   C.2 In $\lambda^{BN}$ . . . . . . . . . . . . . 14