Logical Modalities and Multi-Stage Programming

Zine El-Abidine Benaissa*(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 $\langle \_ \rangle$ 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 λ^{BN}.

1.1 Multi-Stage Programming

The main steps of multi-stage programming are:

1. Write the conventional program
$$program\colon t_S \to t_D \to t$$
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
$$annotated\_program\colon t_S \to \langle t_D \rangle \to \langle t \rangle$$
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 $\mathsf{back}\colon (\langle A \rangle \to \langle B \rangle) \to \langle A \to B \rangle$
$$code\_generator\colon t_S \to \langle t_D \to t \rangle$$
4. Construct or read the static *inputs*:
$$s\colon t_S$$
5. Apply the code generator to the static inputs to get
$$specialized\_code\colon \langle t_D \to t \rangle$$
6. Run the specialized code to re-introduce the generated function as a first-class value in the current environment:
$$specialized\_program\colon t_D \to t$$

*Research supported by the NSF Grant IRI-9625462, The Department of Defense, USAF Air Materiel Command contract # F19628-93-C-0069, and The NSF Grant CDA-9703218.

†Research partially supported by MURST progetto cofinanziato "Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software", ESPRIT WG APPSEM.

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 $\mathsf{unsafe\_run}\colon \langle A \rangle \to A$ 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 Run. Run allows the execution of a code fragment. For example, $\mathsf{run}\ \langle \mathsf{3+4} \rangle$ is well-typed and evaluates to $\mathsf{7}$, but this construct has limited expressivity, e.g. the lambda abstraction $\lambda x.\mathsf{run}\ x$ 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 $\mathsf{unsafe\_run}\colon \langle A \rangle \to A$, 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 $\mathsf{safe\_run}\colon [\langle A \rangle] \to [A]$ exists. Safe-Run has the same operational behavior that $\mathsf{unsafe\_run}$ was intended to achieve, namely *running* code. The difference is *only* in the typing of this function. In a nutshell, $\mathsf{safe\_run}$ allows the programmer to exploit the fact that *closed code can be safely executed.*

1.2 The Refined Method

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

1. Write the conventional program

   $$program\colon t_S \to t_D \to t$$

   Exactly the same as before.

2. Add staging annotations to the program to achieve

   $$closed\_annotated\_program\colon [t_S \to \langle t_D \rangle \to \langle t \rangle]$$

   Almost the same as before. The difference is that the programmer must use the Closed type constructor $[\_]$ 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 Closed values.

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

   $$closed\_code\_generator\colon [t_S \to \langle t_D \to t \rangle]$$

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

4. Turn the closed code-generator into

   $$generator\_of\_closed\_code\colon [t_S] \to [\langle t_D \to t \rangle]$$

   This is done by applying a combinator $\mathsf{closed\_apply}\colon [A \to B] \to [A] \to [B]$.

5. Construct or read the static inputs *as closed values*:

   $$cs\colon [t_S]$$

   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

   $$closed\_specialized\_code\colon [\langle t_D \to t \rangle]$$

7. Run the above result to get:

   $$closed\_specialized\_program\colon [t_D \to t]$$

   This step exploits an interaction between the closed and code types, and it is performed by applying a function $\mathsf{safe\_run}\colon [\langle A \rangle] \to [A]$.

8. Forget that the specialized program is closed:

   $$specialized\_program\colon t_D \to t$$

   The step is performed by applying a function $\mathsf{open}\colon [A] \to A$.

Various examples from [15, 11] can be fully developed in λ^{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 $\lambda^{\Box}$, $\lambda^{\bigcirc}$, and λ^{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 λ^{BN} in the presence of computational effects.

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

$\mathcal{C}, \mathcal{D}$	Categories
$\lvert\mathcal{C}\rvert$	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)
GF	$G \circ F$
GFA	$G(FA)$
F^n	n times composition of functor F
$\hookrightarrow$	Full and faithful functor
$F \dashv G$	An adjunction. F and G are the left- and right-adjoints
$(x_n \mid n \in N)$	Infinite sequence
$(x_i \mid i \in m)$	Finite sequence of length m
x_i	Short for $(x_i \mid i \in m)$ when m is clear from context
$x{::}\,s$	The sequence obtained by adding x in front of s
$n{+}, n{+}{+}$	$n+1, n+2$
$\mathsf{do}\{x_i \leftarrow e_i; e\}$, $\mathsf{ret}\ e$	$\mathsf{let}\ x_i \Leftarrow e_i\ \mathsf{in}\ e$ and $[e]$ of [8]
$\overline{op}\colon \prod_i MA_i \to MB$	$\overline{op}(u_i) \overset{\Delta}{=} \mathsf{do}\{x_i \leftarrow u_i; op(x_i)\}$, i.e., the *monadic extension* of $op\colon \prod_i A_i \to MB$

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:

- $\lambda^{\Box}$ [5], provides constructs for building and then executing closed code. Such a language is useful in machine-code generation.
- $\lambda^{\bigcirc}$ [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 $\lambda^{\bigcirc}$ 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.

$\lambda^{\Box}$ and $\lambda^{\bigcirc}$ have clean, logical foundations [5, 4, 7, 6]: there is a Curry-Howard isomorphism between $\lambda^{\bigcirc}$ and linear time temporal logic, and between $\lambda^{\Box}$ 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 $\lambda^{\Box}$ or $\lambda^{\bigcirc}$ 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 λ^{BN} Types, Syntax, and Semantics

λ^{BN} has the following 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 values. The syntax of λ^{BN} is as follows:

$$\begin{aligned} e \in E{::=}\ & c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e\rangle \mid \tilde{}e \mid \mathsf{up}\ e \mid \\ & \mathsf{close}\ e\ \mathsf{with}\ \{x_i = e_i \mid i \in m\} \mid \mathsf{open}\ e \mid \\ & \mathsf{safe\_run}\ e \end{aligned}$$

The first four constructs are the standard ones in a λ-calculus with constants. Bracket and Escape allow building and combining open code. Brackets construct code, and Escape splices a code fragment into the context of a bigger code fragment. A term such as `(fn x => <(~x,~x)>) <5>` yields `<(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 λ^{BN} is given in Figure 1. Typing judgments have the form $\Gamma \vdash e\colon t^n$, where $\Gamma \equiv \{x_i\colon t_i^{n_i} \mid i \in m\}$ and n is a natural called the *level* of

$$\Gamma \vdash c\colon t_c^n \qquad \Gamma \vdash x\colon t^n \;\text{ if } t^n = \Gamma(x)$$

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

$$\frac{\Gamma \vdash e_1\colon (t_1 \to t_2)^n \quad \Gamma \vdash e_2\colon t_1^n}{\Gamma \vdash e_1\ e_2\colon t_2^n}$$

$$\frac{\Gamma \vdash e\colon t^{n+}}{\Gamma \vdash \langle e\rangle\colon \langle t\rangle^n} \qquad \frac{\Gamma \vdash e\colon \langle t\rangle^n}{\Gamma \vdash \,\tilde{}\, e\colon t^{n+}}$$

$$\frac{\Gamma \vdash e\colon [t]^n}{\Gamma \vdash \mathsf{up}\ e\colon [t]^{n+}}$$

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

$$\frac{\Gamma \vdash e\colon [t]^n}{\Gamma \vdash \mathsf{open}\ e\colon t^n} \qquad \frac{\Gamma \vdash e\colon [\langle t\rangle]^n}{\Gamma \vdash \mathsf{safe\_run}\ e\colon [t]^n}$$

Figure 1: λ^{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 λ^{BN} is given in Figure 2. The set of values for λ^{BN} is defined as follows:

$$\begin{array}{rcl}
v^0 \in V^0 & ::= & \lambda x.e \mid \langle v^1\rangle \mid \mathsf{close}\ v^0 \\
v^1 \in V^1 & ::= & c \mid x \mid v^1\ v^1 \mid \lambda x.v^1 \mid \langle v^2\rangle \mid \\
 & & \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i^1 \mid \mathsf{open}\ v^1 \mid \\
 & & \mathsf{safe\_run}\ v^1 \\
v^{n+2} \in V^{n+2} & ::= & c \mid x \mid v^{n+2}\ v^{n+2} \mid \lambda x.v^{n+2} \mid \\
 & & \langle v^{n+3}\rangle \mid \,\tilde{}\, v^{n+1} \mid \mathsf{up}\ v^{n+1} \mid \\
 & & \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i^{n+2} \mid \\
 & & \mathsf{open}\ v^{n+2} \mid \mathsf{safe\_run}\ v^{n+2}
\end{array}$$

3 Categorical Models

In this section we define what is a model of λ^{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 $F \dashv G$ is simply an adjunction 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 Pfenning between $\lambda^{\bigcirc}$ (Figure 6) and linear time temporal logic, and between $\lambda^{\Box}$ (Figure 5) and S4 modal logic.

Definition 3.3 *A $\lambda^{\bigcirc}$-model is given by a CCC $\mathcal{D}$ and a full and faithful CCC-functor*

$$\begin{tikzcd} \mathcal{D} \arrow[r, hook, "\mathsf{N}"] & \mathcal{D} \end{tikzcd}$$

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

$$[\![\langle t\rangle]\!] = \mathsf{N}[\![t]\!] \text{ and } [\![t_1 \to t_2]\!] = [\![t_2]\!]^{[\![t_1]\!]}$$

and a term $\{x_i\colon t_i^{n_i} | i \in m\} \vdash_{\bigcirc} e\colon t^n$ by a map in $\mathcal{D}(\prod_{i \in m} \mathsf{N}^{n_i}[\![t_i]\!], \mathsf{N}^n[\![t]\!])$.

The properties of N ensure that one may safely confuse $\mathsf{N}^n[\![t_1 \to t_2]\!]$ with $(\mathsf{N}^n[\![t_2]\!])^{\mathsf{N}^n[\![t_1]\!]}$. This iso formalizes the property of $\bigcirc$ in linear time temporal logic, and was also observed in [15].

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

$$\begin{tikzcd} \mathcal{D} \arrow[r, shift left, "\mathsf{G}"] \arrow[r, phantom, "\top"] & \mathcal{C} \arrow[l, shift left, "\mathsf{F}"] \end{tikzcd}$$

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

$$[\![[t]]\!] = \mathsf{B}[\![t]\!] \text{ and } [\![t_1 \to t_2]\!] = [\![t_2]\!]^{[\![t_1]\!]}$$

and a term $\{x_i\colon t_i | i \in m\}; \{x_j\colon t_j | j \in n\} \vdash_{\Box} e\colon t$ is interpreted by a map in $\mathcal{D}(\mathsf{B}(\prod_{i\in m}[\![t_i]\!]) \times (\prod_{j \in n}[\![t_j]\!]), [\![t]\!])$. 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 $\Delta, x\colon t; \Gamma \vdash_{\Box} e_1\colon t'$ and those of the form $\Delta; x\colon [t], \Gamma \vdash_{\Box} e_2\colon t'$ given by

$$e_1 \mapsto \mathsf{let\ box}\ x = x\ \mathsf{in}\ e_1 \qquad e_2 \mapsto e_2[x\colon = \mathsf{box}\ x]$$

By analogy with the adjoint calculus [2, 3]), one may consider a variant of $\lambda^{\Box}$ in which the category $\mathcal{C}$ and context separation have a more prominent role.

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x:=v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e \qquad \frac{e \overset{0}{\hookrightarrow} \langle v \rangle}{\textasciitilde e \overset{1}{\hookrightarrow} v} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ v'}{\mathsf{up}\ e \overset{1}{\hookrightarrow} \mathsf{close}\ v'}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i:=v_i] \overset{0}{\hookrightarrow} v}{\mathsf{close}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \mathsf{close}\ v} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ v}{\mathsf{open}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{close}\ \langle v' \rangle \quad v' \overset{0}{\hookrightarrow} v}{\mathsf{safe\_run}\ e \overset{0}{\hookrightarrow} \mathsf{close}\ v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\langle e \rangle \overset{n}{\hookrightarrow} \langle v \rangle}$$

$$x \overset{n+}{\hookrightarrow} x \qquad c \overset{n+}{\hookrightarrow} c \qquad \frac{e_1 \overset{n+}{\hookrightarrow} v_1 \quad e_2 \overset{n+}{\hookrightarrow} v_2}{e_1\ e_2 \overset{n+}{\hookrightarrow} v_1\ v_2} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\textasciitilde e \overset{n++}{\hookrightarrow} \textasciitilde v} \qquad \frac{e \overset{n+}{\hookrightarrow} v'}{\mathsf{up}\ e \overset{n++}{\hookrightarrow} \mathsf{up}\ v'}$$

$$\frac{e_i \overset{n+}{\hookrightarrow} v_i}{\mathsf{close}\ e\ \mathsf{with}\ x_i = e_i \overset{n+}{\hookrightarrow} \mathsf{close}\ e\ \mathsf{with}\ x_i = v_i} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{open}\ e \overset{n+}{\hookrightarrow} \mathsf{open}\ v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{safe\_run}\ e \overset{n+}{\hookrightarrow} \mathsf{safe\_run}\ v}$$

Figure 2: Big-Step Operational Semantics of λ^{BN}

In λ^{BN} the closed types and open code types coexist, so the key point is to clarify how the modalities of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ interact. The basic idea is that a λ^{BN}-model is a $\lambda^{\Box}$-model where the category $\mathcal{D}$ has the structure of a $\lambda^{\bigcirc}$-model *parameterized* w.r.t. $\mathcal{C}$. To formalize this notion of parameterization, we introduce the following auxiliary definition:

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

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

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

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

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

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

$$\begin{tikzcd} \mathcal{D} \arrow[r, shift left, "\mathsf{G}"] \arrow[r, phantom, "\top"] & \mathcal{C} \arrow[l, shift left, "\mathsf{F}"] \end{tikzcd}$$

and a $\mathcal{C}$-indexed full and faithful CCC-functor

$$\begin{tikzcd} \mathcal{D}^{\mathsf{F}} \arrow[r, hook, "\mathsf{N}"] & \mathcal{D}^{\mathsf{F}} \end{tikzcd}$$

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

Remark 3.10 We work mainly in $\mathcal{D}$, therefore we write N for functor on $\mathcal{D}$ corresponding to N_1 via the isomorphism of categories $\mathcal{D} \cong \mathcal{D}^{\mathsf{F}}_1$. Furthermore, we write $\mathsf{N}A$ for $\mathsf{N}_X A$ when $A \in |\mathcal{D}^{\mathsf{F}}_X| = |\mathcal{D}|$, since the action on objects (but not on maps) of the functors N_X is independent of X.

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

$$[\![[t]]\!] = \mathsf{B}[\![t]\!]\ ,\ [\![\langle t \rangle]\!] = \mathsf{N}[\![t]\!] \text{ and } [\![t_1 \to t_2]\!] = [\![t_2]\!]^{[\![t_1]\!]}$$

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

Any λ^{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* λ^{BN}*-model there is a* canonical *map up*: $\mathsf{F}X \to \mathsf{NF}X$ *in* $\mathcal{D}$.

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

$$\begin{array}{ll}
\mathcal{D}(\mathsf{F}X, \mathsf{F}Y) & \text{by definition of } \mathcal{D}^{\mathsf{F}}_X \\
\cong \mathcal{D}^{\mathsf{F}}_X(1, \mathsf{F}Y) & \text{because } \mathsf{N}_X \text{ is full and faithful} \\
\cong \mathcal{D}^{\mathsf{F}}_X(\mathsf{N}1, \mathsf{NF}Y) & \text{because } \mathsf{N}_X \text{ is a CCC-functor} \\
\cong \mathcal{D}^{\mathsf{F}}_X(1, \mathsf{NF}Y) & \text{by definition of } \mathcal{D}^{\mathsf{F}}_X \\
\cong \mathcal{D}(\mathsf{F}X, \mathsf{NF}Y) &
\end{array}$$

We define *up*: $\mathsf{F}X \to \mathsf{NF}X$ as the map corresponding to the identity over $\mathsf{F}X$ when $Y = X$ (in general *up* is not an iso). ∎

Proposition 3.12 (Compile) *In a* λ^{BN}*-model there is a* canonical *iso compile*: $\mathsf{GN}A \to \mathsf{G}A$ *in* $\mathcal{C}$.

Proof: We have the natural isomorphisms

$$\begin{array}{ll}
\mathcal{C}(X, \mathsf{G}A) & \text{by } \mathsf{F} \dashv \mathsf{G} \\
\cong \mathcal{D}(\mathsf{F}X, A) & \text{by definition of } \mathcal{D}^{\mathsf{F}}_X \\
\cong \mathcal{D}^{\mathsf{F}}_X(1, A) & \text{because } \mathsf{N}_X \text{ is full and faithful} \\
\cong \mathcal{D}^{\mathsf{F}}_X(\mathsf{N}1, \mathsf{N}A) & \text{because } \mathsf{N}_X \text{ is a CCC-functor} \\
\cong \mathcal{D}^{\mathsf{F}}_X(1, \mathsf{N}A) & \text{by definition of } \mathcal{D}^{\mathsf{F}}_X \\
\cong \mathcal{D}(\mathsf{F}X, \mathsf{N}A) & \text{by } \mathsf{F} \dashv \mathsf{G} \\
\cong \mathcal{C}(X, \mathsf{GN}A) &
\end{array}$$

We define *compile*: $\mathsf{GN}A \to \mathsf{G}A$ as the map corresponding to the identity over $\mathsf{GN}A$ when $X = \mathsf{GN}A$, while the inverse of *compile* is the map corresponding to the identity over $\mathsf{G}A$ when $X = \mathsf{G}A$. ∎

3.1 Examples

We give examples of λ^{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

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

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

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

$$\begin{tikzcd}
0 & 1 \arrow[l] & \dots & n & n+ \arrow[l] & \dots
\end{tikzcd}$$

Given a CCC $\mathcal{A}$ with finite and ω^{op}-limits, take

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

$$\begin{tikzcd}
A_0 \arrow[d, "f_0"] & A_1 \arrow[l, "a_0"'] \arrow[d, "f_1"] & \dots & A_n \arrow[d, "f_n"] & A_{n+} \arrow[l, "a_n"'] \arrow[d, "f_{n+}"] & \dots \\
B_0 & B_1 \arrow[l, "b_0"] & \dots & B_n & B_{n+} \arrow[l, "b_n"] & \dots
\end{tikzcd}$$

  while an object of $\mathcal{D}$ is a sequence of maps in $\mathcal{A}$.
- $\mathsf{N}A \triangleq !_{A_0} :: A$, where $!_{A_0}$ is the map $1 \leftarrow A_0$ in $\mathcal{A}$
- $\mathsf{F}X \triangleq (\mathrm{id}: X \leftarrow X | n \in N)$, i.e. the sequence which is constantly id_X, while $\mathsf{G}A \triangleq \lim_{n \in \omega^{op}} A_n$.

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*: $A \to \mathsf{N}A$, namely $up_0 \triangleq !: A_0 \to 1$ and $up_{n+} \triangleq a_n: A_{n+} \to A_n$, which provides cross-stage persistence for arbitrary types.

Finally we give an example which is both a $\lambda^{\Box}$- and $\lambda^{\bigcirc}$-model, but fails to be a λ^{BN}-model. More precisely, it has the structure of a λ^{BN}-model, but the $\mathcal{C}$-indexed functor N fails to be full and faithful, so we do not have the iso *compile*: $\mathsf{GN}A \to \mathsf{G}A$ of Proposition 3.12.

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

- $\mathcal{C} \triangleq \mathcal{A}^{\omega^{op}}$ and $\mathcal{D} \triangleq \mathcal{A}^N$
- $\mathsf{N}A \triangleq 1 :: A$, where 1 is the terminal object of $\mathcal{A}$
- $(\mathsf{F}X)_n \triangleq X_n$, i.e. forget the maps $x_n: X_{n+} \to X_n$, while $(\mathsf{G}A)_n \triangleq \prod_{i \leq n} A_i$ with the obvious projection $\pi: (\mathsf{G}A)_{n+} \to (\mathsf{G}A)_n$.

$$[\![\Gamma \vdash c{:}\, t_c^n]\!] \triangleq [\![c]\!]_n \circ !{:}\, C \to \mathsf{N}^n[\![t_c]\!] \qquad [\![\Gamma \vdash x{:}\, t^n]\!] \triangleq \pi_x{:}\, C \to \mathsf{N}^n A \text{ if } t^n = \Gamma(x)$$

$$\frac{[\![\Gamma \vdash e{:}\, [t]^n]\!] = f{:}\, C \to \mathsf{N}^n(\mathsf{B}A)}{[\![\Gamma \vdash \mathsf{up}\ e{:}\, [t]^{n+}]\!] \triangleq up_n \circ f{:}\, C \to \mathsf{N}^n(\mathsf{NB}A)} \qquad \frac{[\![\Gamma \vdash e{:}\, [\langle t\rangle]^n]\!] = f{:}\, C \to \mathsf{N}^n(\mathsf{BN}A)}{[\![\Gamma \vdash \mathsf{safe\_run}\ e{:}\, [t]^n]\!] \triangleq run_n \circ f{:}\, C \to \mathsf{N}^n(\mathsf{B}A)}$$

$$\frac{[\![\Gamma, x{:}\, t^n \vdash e{:}\, t'^n]\!] = f{:}\, C \times \mathsf{N}^n A \to \mathsf{N}^n B}{[\![\Gamma \vdash \lambda x.e{:}\, t \to t'^n]\!] \triangleq \lambda_n \circ (\Lambda f){:}\, C \to \mathsf{N}^n(B^A)} \qquad \frac{[\![\Gamma \vdash e{:}\, [t]^n]\!] = f{:}\, C \to \mathsf{N}^n(\mathsf{B}A)}{[\![\Gamma \vdash \mathsf{open}\ e{:}\, t^n]\!] \triangleq open_n \circ f{:}\, C \to \mathsf{N}^n A}$$

$$\frac{[\![\Gamma \vdash e{:}\, t^{n+}]\!] = f{:}\, C \to \mathsf{N}^{n+} A}{[\![\Gamma \vdash \langle e\rangle{:}\, \langle t\rangle^n]\!] \triangleq f{:}\, C \to \mathsf{N}^n(\mathsf{N}A)} \qquad \frac{[\![\Gamma \vdash e{:}\, \langle t\rangle^n]\!] = f{:}\, C \to \mathsf{N}^n(\mathsf{N}A)}{[\![\Gamma \vdash \tilde{}e{:}\, t^{n+}]\!] \triangleq f{:}\, C \to \mathsf{N}^{n+} A}$$

$$\frac{[\![\Gamma \vdash e_1{:}\, t \to t'^n]\!] = f_1{:}\, C \to \mathsf{N}^n(B^A) \qquad [\![\Gamma \vdash e_2{:}\, t^n]\!] = f_2{:}\, C \to \mathsf{N}^n A}{[\![\Gamma \vdash e_1\ e_2{:}\, t'^n]\!] \triangleq @_n \circ \langle f_1, f_2\rangle{:}\, C \to \mathsf{N}^n B}$$

$$\frac{[\![\Gamma \vdash e_i{:}\, [t_i]^n]\!] = f_i{:}\, C \to \mathsf{N}^n(\mathsf{B}A_i) \qquad [\![\{x_i{:}\, [t_i]^0 | i\} \vdash e{:}\, t^0]\!] = f{:}\, \prod_i \mathsf{B}A_i \to A}{[\![\Gamma \vdash \mathsf{close}\ e\ \mathsf{with}\ x_i = e_i{:}\, [t]^n]\!] \triangleq close_n(f) \circ \langle f_i | i\rangle{:}\, C \to \mathsf{N}^n(\mathsf{B}A)}$$

where $C \triangleq [\![\Gamma]\!]$, $A \triangleq [\![t]\!]$, $B \triangleq [\![t']\!]$ and $A_i \triangleq [\![t_i]\!]$.

Figure 3: Pure Interpretation in λ^{BN}-Models

The action of N_X on maps of $\mathcal{D}_X^{\mathsf{F}}$ sends $f \in \mathcal{D}_X^{\mathsf{F}}(A, B)$ to $g \in \mathcal{D}_X^{\mathsf{F}}(\mathsf{N}A, \mathsf{N}B)$, where $g_0 = !{:}\, X_0 \times 1 \to 1$ and

$$g_{n+} = X_{n+} \times A_n \xrightarrow{x_n \times \mathrm{id}} X_n \times A_n \xrightarrow{f_n} B_n$$

It is easy to see that N_X is a CCC-functor on $\mathcal{D}_X^{\mathsf{F}}$, but it fails to be full and faithful unless all x_n are isos (or equivalently both epis and split monos). In fact, if x_n is not epi, then exist $C \in |\mathcal{A}|$ and $h, h'{:}\, X_n \to C$ such that $h \neq h'$ but $h \circ x_n = h' \circ x_n$. One can show that N_X is not faithful, i.e. $f \neq f'$ but $\mathsf{N}_X f = \mathsf{N}_X f'$, by taking $f_m = f'_m = !{:}\, X_m \times 1 \to 1$ for $m \neq n$ and $f_n = h, f'_m = h'{:}\, X_m \times 1 \to C$. On the other hand, if x_n is not a split mono, then one can show that N_X is not full by taking $g_{m+}{:}\, X_{m+} \times 1 \to 1$ for $m \neq n$ and $g_{n+} = \pi_1{:}\, X_{n+} \times 1 \to X_{n+}$.

3.2 Interpretation of terms

We have already given the interpretation of types in a λ^{BN}-model without computational effects, namely

$$[\![[t]]\!] = \mathsf{B}[\![t]\!]\ ,\ [\![\langle t\rangle]\!] = \mathsf{N}[\![t]\!] \text{ and } [\![t_1 \to t_2]\!] = [\![t_2]\!]^{[\![t_1]\!]}$$

This section gives the corresponding interpretation of terms. Before doing that, we introduce some auxiliary maps in $\mathcal{D}$, which simplify the definition of the interpretation, and clarify the similarities with the interpretation of the simply typed λ-calculus in a CCC.

Given $op{:}\, \prod_i A_i \to A$ we define

$$op_n \triangleq \mathsf{N}^n op{:}\, \prod_i \mathsf{N}^n A_i \to \mathsf{N}^n A$$

where we exploit that N preserves finite products.

- $\lambda_n{:}\, (\mathsf{N}^n B)^{\mathsf{N}^n A} \to \mathsf{N}^n B^A$. Since N preserves the CCC structure, λ_n is the iso $(\mathsf{N}^n B)^{\mathsf{N}^n A} \to \mathsf{N}^n B^A$.
- $@_n \triangleq eval_n{:}\, \mathsf{N}^n B^A \times \mathsf{N}^n A \to \mathsf{N}^n B$. Since N preserves the CCC structure, $@_n$ is (up to iso) an instance of evaluation $eval{:}\, (\mathsf{N}^n B)^{\mathsf{N}^n A} \times \mathsf{N}^n A \to \mathsf{N}^n B$.
- $open_n \triangleq \epsilon_n{:}\, \mathsf{N}^n\mathsf{B}A \to \mathsf{N}^n A$, where $\epsilon{:}\, \mathsf{B}A \to A$ is the co-unit of the co-monad B.
- $close_n(f) \triangleq (\mathsf{B}f \circ \delta)_n{:}\, \prod_i \mathsf{N}^n\mathsf{B}A_i \to \mathsf{N}^n\mathsf{B}C$, where $f{:}\, \prod_i \mathsf{B}A_i \to C$ and $\delta{:}\, \mathsf{B}A \to \mathsf{B}^2 A$ is the co-multiplication of B, and we exploit that N and B preserve finite products.
- $up_n{:}\, \mathsf{N}^n\mathsf{B}A \to \mathsf{N}^{n+}\mathsf{B}A$, where $up{:}\, \mathsf{B}A \to \mathsf{NB}A$ is the map of Proposition 3.11.
- $run_n \triangleq (\mathsf{F}\ compile)_n{:}\, \mathsf{N}^n\mathsf{BN}A \to \mathsf{N}^n\mathsf{B}A$, where $compile{:}\, \mathsf{GN}A \to \mathsf{G}A$ 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 λ^{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 λ-calculus in a CCC. However, we are interested in multi-stage programming languages, like *Mini-ML*$^\Box$ [5], *Mini-ML*$^\bigcirc$ [4], and *MetaML* [15], where logical modalities coexist with computational effects. In this section we define *monadic* λ^{BN}-models and give a monadic CBV interpretation of λ^{BN} in such models, extending the interpretation sketched in [8].

Definition 4.1 *A monadic λ^{BN}-model consists of a λ^{BN}-model equipped with*

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

$$\begin{tikzcd}
MA & \mathsf{B}M^2A \arrow[r, "\sigma"] \arrow[dr, "\mathsf{B}\mu"] & M\mathsf{B}MA \arrow[r, "M\sigma"] & M^2\mathsf{B}A \arrow[d, "\mu"] \\
\mathsf{B}MA \arrow[u, "\epsilon"] \arrow[r, "\sigma"] \arrow[d, "\delta"] & M\mathsf{B}A \arrow[ul, "M\epsilon"] \arrow[dr, "M\delta"] & \mathsf{B}MA \arrow[r, "\sigma"] & M\mathsf{B}A \\
\mathsf{B}^2MA \arrow[r, "\mathsf{B}\sigma"'] & \mathsf{B}M\mathsf{B}A \arrow[r, "\sigma"'] & M\mathsf{B}^2A & \mathsf{B}A \arrow[u, "\eta"'] \arrow[ul, "\mathsf{B}\eta"]
\end{tikzcd}$$

Remark 4.2 The intuition here is that the strong monad M models conventional computations, i.e. those at level 0. Staged computations are achieved by an alternation of M and N, namely we define the type M_nA of **n-staged computations** of type A as $(M\mathsf{N})^nMA$. In general, M_n is not a monad, however the natural transformations η and μ for M induce natural transformations $\eta_n\colon \mathsf{N}^nA \to M_nA$ and $\mu_n\colon (M^2\mathsf{N})^nM^2A \to M_nA$.
The iso $\lambda_*\colon (M\mathsf{N}B)^{\mathsf{N}A} \to M\mathsf{N}B^A$ refines the iso $(\mathsf{N}B)^{\mathsf{N}A} \to \mathsf{N}B^A$ corresponding to preservation of exponentials by N. λ_* is essential to define the interpretation of λ-abstraction at level $n > 0$, and also to define the analog of $let\colon (MB)^A \times MA \to MB$ for M_n, namely $let_n\colon (M_nB)^{\mathsf{N}^nA} \times M_nA \to M_nB$.

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

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

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

- M preserves pullbacks, and the commuting square

$$\begin{tikzcd}
M(B^A) \arrow[r, "e"] \arrow[d, "M!"'] \arrow[dr, phantom, "(M*)"] & (MB)^A \arrow[d, "(M!)^A"] \\
M1 \arrow[r, "k"'] & (M1)^A
\end{tikzcd}$$

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

- M preserves ω^{op}-limits

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

$$MA \triangleq MA_0 \xleftarrow{Ma_0} MA_1 \quad \ldots \quad MA_n \xleftarrow{Ma_n} MA_{n+} \quad \ldots$$

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

- the first property of M over $\mathcal{A}$ ensures the existence of the iso $\lambda_*\colon (M\mathsf{N}B)^{\mathsf{N}A} \to M\mathsf{N}B^A$ (exponentials in $\mathcal{A}^{\omega^{op}}$ are computed using exponentials and pullbacks in $\mathcal{A}$);
- the second property of M over $\mathcal{A}$ allows us to define the natural transformation $\sigma\colon \mathsf{B}MA \to M\mathsf{B}A$ as the iso corresponding to preservation of ω^{op}-limits by 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:

$$[\![[t]]\!] = \mathsf{B}[\![t]\!], [\![\langle t\rangle]\!] = \mathsf{N}M[\![t]\!], [\![t_1 \to t_2]\!] = (M[\![t_2]\!])^{[\![t_1]\!]}$$

In a monadic λ^{BN}-model a term $\{x_i\colon t_i^{n_i}|i \in m\} \vdash e\colon t^n$ is interpreted by a map in $\mathcal{D}(\prod_{i\in m} \mathsf{N}^{n_i}[\![t_i]\!], M_n[\![t]\!])$.

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 λ-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 $op\colon \prod_i A_i \to MB$ we define $\overline{op}_n\colon \prod_i M_nA_i \to M_nB$ by induction on n:

$$\begin{tikzcd} 0) & \prod_i MA_i \arrow[r, "\overline{op}"] & MB & \\ n+) & \prod_i M_{n+}A_i \arrow[r, "\psi"] & M\mathsf{N}\prod_i M_nA_i \arrow[dr, "M\mathsf{N}\overline{op}_n"] & \\ & & & M_{n+}B \end{tikzcd}$$

where $\psi\colon \prod_i MA_i \to M(\prod_i A_i)$ is given by $\psi(u_i|i) \triangleq \mathsf{do}\{x_i \leftarrow u_i; \mathsf{ret}\ (x_i|i)\}$, and we exploit that N preserves finite products.

- $\eta_n\colon \mathsf{N}^nA \to M_nA$ is given by induction:

  $$\begin{tikzcd} 0) & A \arrow[r, "\eta"] & MA & \\ n+) & \mathsf{N}^{n+}A \arrow[r, "\eta"] & M\mathsf{N}^{n+}A \arrow[r, "M\mathsf{N}\eta_n"] & M_{n+}A \end{tikzcd}$$

  where $\eta\colon A \to MA$ is the unit of the monad M.

- $\lambda_n\colon (M_nB)^{\mathsf{N}^nA} \to M_n(MB)^A$ is given by induction:

  $$\begin{tikzcd} 0) & (MB)^A \arrow[r, "\eta"] & M(MB)^A & \\ n+) & (M_{n+}B)^{\mathsf{N}^{n+}A} \arrow[r, "\lambda_*"] & M\mathsf{N}(M_nB)^{\mathsf{N}^nA} \arrow[dr, "M\mathsf{N}\lambda_n"] & \\ & & & M_{n+}(MB)^A \end{tikzcd}$$

- $@_n \triangleq \overline{eval}_n\colon M_n(MB)^A \times M_nA \to M_nB$, where $eval\colon (MB)^A \times A \to MB$ is evaluation.

- $open_n \triangleq M_n\epsilon\colon M_n\mathsf{B}A \to M_nA$, where $\epsilon\colon \mathsf{B}A \to A$ is the co-unit for B.

- $close_n(f) \triangleq \overline{(\sigma \circ (\mathsf{B}f) \circ \delta)}_n\colon \prod_i M_n\mathsf{B}A_i \to M_n\mathsf{B}C$, where $f\colon \prod_i \mathsf{B}A_i \to MC$ and $\delta\colon \mathsf{B}A \to \mathsf{B}^2A$ is the co-multiplication for B, and we exploit that B preserves finite products.

- $up_n \triangleq \overline{(\eta \circ up)}_n\colon M_n\mathsf{B}A \to M_{n+}\mathsf{B}A$, where $up\colon \mathsf{B}A \to \mathsf{NB}A$ is the map of Proposition 3.11.

- $run_n \triangleq \overline{(\sigma \circ \mathsf{F}\ compile)}_n\colon M_n\mathsf{BN}MA \to M_n\mathsf{B}A$, where $compile\colon \mathsf{GN}A \to \mathsf{G}A$ is the iso of Proposition 3.12.

The interpretation of terms. Figure 4 defines the interpretation of a well-formed term $\Gamma \vdash e\colon t^n$ by induction on the typing derivation in the type system of Figure 1.

5 Discussion

Searching for a categorical semantics of *MetaML* has benefited us in a number of ways:

- It has suggested simplifications and extensions. We have simplified the type system of *MetaML* and proposed an extension with *closed code types* called *AIM* (see [11]). λ^{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 λ^{BN} Compared to *AIM*

Recently, we presented *AIM* (An Idealized MetaML) [11], which extends *MetaML* with an analog of the Box type of $\lambda^{\Box}$ yielding a more expressive language, yet has a simpler typing judgment than *MetaML*. We have shown that we can embed all three languages into *AIM*. λ^{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 λ^{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"

$$\llbracket \Gamma \vdash c\colon t_c^n \rrbracket \triangleq \overline{\llbracket c \rrbracket}_n \circ !\colon C \to M_n \llbracket t_c \rrbracket \qquad \llbracket \Gamma \vdash x\colon t^n \rrbracket \triangleq \eta_n \circ \pi_x \colon C \to M_n A \text{ if } t^n = \Gamma(x)$$

$$\frac{\llbracket \Gamma \vdash e\colon [t]^n \rrbracket = f\colon C \to M_n(\mathsf{B}A)}{\llbracket \Gamma \vdash \mathsf{up}\ e\colon [t]^{n+} \rrbracket \triangleq up_n \circ f\colon C \to M_{n+}(\mathsf{B}A)} \qquad \frac{\llbracket \Gamma \vdash e\colon [\langle t \rangle]^n \rrbracket = f\colon C \to M_n(\mathsf{BN}MA)}{\llbracket \Gamma \vdash \mathsf{safe\_run}\ e\colon [t]^n \rrbracket \triangleq run_n \circ f\colon C \to M_n(\mathsf{B}A)}$$

$$\frac{\llbracket \Gamma, x\colon t^n \vdash e\colon t'^n \rrbracket = f\colon C \times \mathsf{N}^n A \to M_n B}{\llbracket \Gamma \vdash \lambda x.e\colon t \to t'^n \rrbracket \triangleq \lambda_n \circ (\Lambda f)\colon C \to M_n(MB)^A} \qquad \frac{\llbracket \Gamma \vdash e\colon [t]^n \rrbracket = f\colon C \to M_n(\mathsf{B}A)}{\llbracket \Gamma \vdash \mathsf{open}\ e\colon t^n \rrbracket \triangleq open_n \circ f\colon C \to M_n A}$$

$$\frac{\llbracket \Gamma \vdash e\colon t^{n+} \rrbracket = f\colon C \to M_{n+} A}{\llbracket \Gamma \vdash \langle e \rangle\colon \langle t \rangle^n \rrbracket \triangleq f\colon C \to M_n(\mathsf{N}MA)} \qquad \frac{\llbracket \Gamma \vdash e\colon \langle t \rangle^n \rrbracket = f\colon C \to M_n(\mathsf{N}MA)}{\llbracket \Gamma \vdash \tilde{}e\colon t^{n+} \rrbracket \triangleq f\colon C \to M_{n+} A}$$

$$\frac{\llbracket \Gamma \vdash e_1\colon t \to t'^n \rrbracket = f_1\colon C \to M_n(MB)^A \qquad \llbracket \Gamma \vdash e_2\colon t^n \rrbracket = f_2\colon C \to M_n A}{\llbracket \Gamma \vdash e_1\ e_2\colon t'^n \rrbracket \triangleq @_n \circ \langle f_1, f_2 \rangle\colon C \to M_n B}$$

$$\frac{\llbracket \Gamma \vdash e_i\colon [t_i]^n \rrbracket = f_i\colon C \to M_n(\mathsf{B}A_i) \qquad \llbracket \{x_i\colon [t_i]^0 | i\} \vdash e\colon t^0 \rrbracket = f\colon \prod_i \mathsf{B}A_i \to MA}{\llbracket \Gamma \vdash \mathsf{close}\ e\ \mathsf{with}\ x_i = e_i\colon [t]^n \rrbracket \triangleq close_n(f) \circ \langle f_i | i \rangle\colon C \to M_n(\mathsf{B}A)}$$

$$\text{where } C \triangleq \llbracket \Gamma \rrbracket,\ A \triangleq \llbracket t \rrbracket,\ B \triangleq \llbracket t' \rrbracket \text{ and } A_i \triangleq \llbracket t_i \rrbracket.$$

Figure 4: Monadic Interpretation in λ^{BN}-Models

types. This causes some confusion from the point of view of multi-stage programming, because manipulating values of type $[\langle A \rangle]$ 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 λ^{BN}, we have made the pragmatic decision that Closed should not delay its argument, and so, types such as $[\langle A \rangle]$ 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. $\lambda^{\bigcirc}$, $\lambda^{\Box}$, and *MetaML*), while λ^{BN} is *not*. We have shown that $\lambda^{\bigcirc}$ can be embedded into the *open fragment* of *AIM*, and $\lambda^{\Box}$ 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 $\lambda^{\bigcirc}$ and $\lambda^{\Box}$ in *AIM* can be turned into an embedding in λ^{BN} (the embedding of $\lambda^{\Box}$ needs to be modified to take into account the fact that Closed in λ^{BN} 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. λ^{BN} 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. λ^{BN} 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 $\mathcal{A}^{\omega^{op}}$. 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

[1] Z. Benaissa, E. Moggi, W. Taha, and T. Sheard. A categorical analysis of multi-level languages (extended abstract). Technical Report CSE-98-018, Oregon Graduate Institute, December 1998. `ftp://cse.ogi.edu/pub/tech-reports/`.

[2] N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. *LNCS*, 933, 1995.

[3] N. Benton and P. Wadler. Linear logic, monads and the lambda calculus. In *11*th *LICS*, New Brunswick, New Jersey, 27–30 July 1996. IEEE Computer Society Press.

[4] R. Davies. A temporal-logic approach to binding-time analysis. In *11*$^{\text{th}}$ *LICS*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[5] R. Davies and F. Pfenning. A modal analysis of staged computation. In *23rd POPL*, St.Petersburg Beach, Florida, January 1996.

[6] S. Martini and A. Masini. A computational interpretation of modal proofs. In H. Wansing, editor, *Proof Theory of Modal Logic*. Kluwer, 1996.

[7] A. Masini. 2-Sequent calculus: Intuitionism and natural deduction. *Journal of Logic and Computation*, 3(5), 1993.

[8] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[9] E. Moggi. A categorical account of two-level languages. In *MFPS 1997*, 1997.

[10] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, Oregon Graduate Institute, October 1998. `ftp://cse.ogi.edu/pub/tech-reports/`.

[11] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). In *European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*. Springer-Verlag, 1999.

[12] G.E. Reyes and H. Zolfaghari. Topos-theoretic approaches to modalities. In A. Carboni, C. Pedicchio, and G. Rosolini, editors, *Conference on Category Theory '90*, volume 1488 of *LNM*. Springer-Verlag, 1991.

[13] W. Taha. *Multi-Stage Programming: Its Theory and Applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, June 1999. To appear.

[14] W. Taha, Z. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type-safety. In *25th ICALP*, Aalborg, Denmark, 1998.

[15] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *PEPM*. ACM, 1997.

[16] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. Technical Report CSE-99-007, Oregon Graduate Institute, January 1999. `ftp://cse.ogi.edu/pub/tech-reports/`.

$$\Delta;\Gamma \vdash c\colon t_c \qquad \Delta;\Gamma \vdash x\colon t \ \text{ if } t = \Delta(x) \text{ or } \Gamma(x)$$

$$\frac{\Delta;\Gamma, x\colon t_1 \vdash e\colon t_2}{\Delta;\Gamma \vdash \lambda x.e\colon t_1 \to t_2} \qquad \frac{\Delta;\emptyset \vdash e\colon t}{\Delta;\Gamma \vdash \mathsf{box}\ e\colon [t]}$$

$$\frac{\Delta;\Gamma \vdash e_1\colon t_1 \to t_2 \quad \Delta;\Gamma \vdash e_2\colon t_1}{\Delta;\Gamma \vdash e_1 e_2\colon t_2}$$

$$\frac{\Delta;\Gamma \vdash e_1\colon [t_1] \quad \Delta, x\colon t_1;\Gamma \vdash e_2\colon t_2}{\Delta;\Gamma \vdash \mathsf{let\ box}\ x = e_1\ \mathsf{in}\ e_2\colon t_2}$$

Figure 5: $\lambda^{\Box}$ Type System

$$\Gamma \vdash c\colon t_c^n \qquad \Gamma \vdash x\colon t^n \ \text{ if } t^n = \Gamma(x)$$

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

$$\frac{\Gamma \vdash e_1\colon (t_1 \to t_2)^n \quad \Gamma \vdash e_2\colon t_1^n}{\Gamma \vdash e_1\ e_2\colon t_2^n}$$

$$\frac{\Gamma \vdash e\colon t^{n+}}{\Gamma \vdash \langle e\rangle\colon \langle t\rangle^n} \qquad \frac{\Gamma \vdash e\colon \langle t\rangle^n}{\Gamma \vdash \tilde{}e\colon t^{n+}}$$

Figure 6: $\lambda^{\bigcirc}$ Type System

$$\Gamma \vdash x\colon t^n \ \text{ if } t^m = \Gamma(x) \text{ and } m \leq n$$

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

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

$$\Gamma \vdash x\colon t^n \ \text{ if } t^m = \Gamma(x) \text{ and } m \leq n$$

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

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

$$\frac{\Gamma \vdash e\colon [t]^n}{\Gamma \vdash \mathsf{unbox}\ e\colon t^n}$$

Figure 8: *AIM* Type System (+ Figure 6)

$$\frac{e_1 \overset{0}{\hookrightarrow} \lambda x.e \quad e_2 \overset{0}{\hookrightarrow} v_1 \quad e[x\colon= v_1] \overset{0}{\hookrightarrow} v_2}{e_1\ e_2 \overset{0}{\hookrightarrow} v_2} \qquad \lambda x.e \overset{0}{\hookrightarrow} \lambda x.e \qquad \frac{e \overset{0}{\hookrightarrow} \langle v\rangle}{{\sim}e \overset{1}{\hookrightarrow} v}$$

$$\frac{e_i \overset{0}{\hookrightarrow} v_i}{\mathsf{box}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} \mathsf{box}\ e[x_i\colon= v_i]} \qquad \frac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{0}{\hookrightarrow} v} \qquad \frac{e_i \overset{0}{\hookrightarrow} v_i \quad e[x_i\colon= v_i] \overset{0}{\hookrightarrow} \langle v'\rangle \quad v'_0 \overset{0}{\hookrightarrow} v}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{0}{\hookrightarrow} v}$$

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

$$\frac{e \overset{n+}{\hookrightarrow} v}{\lambda x.e \overset{n+}{\hookrightarrow} \lambda x.v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{{\sim}e \overset{n++}{\hookrightarrow} {\sim}v} \qquad \frac{e \overset{n+}{\hookrightarrow} v}{\mathsf{unbox}\ e \overset{n+}{\hookrightarrow} \mathsf{unbox}\ v} \qquad \frac{e_i \overset{n+}{\hookrightarrow} v_i \quad e \overset{n+}{\hookrightarrow} v}{\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i \overset{n+}{\hookrightarrow} \mathsf{run}\ v\ \mathsf{with}\ x_i = v_i}$$

Figure 9: Big-Step Operational Semantics for *AIM* (including $\lambda^{\bigcirc}$ and *MetaML*)

A Multi-Stage Languages

For completeness, this appendix reproduces the syntax and type system of the multi-stage languages $\lambda^{\Box}$, $\lambda^{\bigcirc}$, *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 $\Delta;\Gamma \vdash e\colon t$, where $\Delta, \Gamma \equiv \{x_i\colon t_i | i \in m\}$. The syntax for $\lambda^{\Box}$ is as follows:

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

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

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

$$e \in E ::= \quad c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e\rangle \mid {\sim}e$$

MetaML [15, 14] uses a more relaxed type rule for variables than $\lambda^{\bigcirc}$, 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 $\lambda^{\bigcirc}$ to

$$e \in E ::= \quad c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e\rangle \mid {\sim}e \mid \mathsf{run}\ e$$

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:

$$\begin{aligned} e \in E ::= \quad & c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \langle e\rangle \mid {\sim}e \mid \\ & \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \\ & \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{unbox}\ e \end{aligned}$$

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 $\lambda^{\bigcirc}$, *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 λ^{BN}, 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* $\Gamma_1, \Gamma_2 \vdash e_2\colon t_2^n$ *and* x *is* fresh, *then* $\Gamma_1, x\colon t_1^{n'}, \Gamma_2 \vdash e_2\colon t_2^n$.

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

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

Second, the properties of the operational semantics:

Lemma B.3 (Values)

1. If $v \in V^n$ *then* $v \overset{n}{\hookrightarrow} v$

2. If $e \overset{n}{\hookrightarrow} e'$ *then* $e' \in V^n$

Proof: The first is by induction over the derivation of $v \in V^n$. The second is by induction over the derivation of $e \overset{n}{\hookrightarrow} e'$. ∎

In λ^{BN} one cannot define demotion (as in *AIM*). This is not a problem, since the Demotion Lemma was used only in the case $\mathsf{run}\ e\ \mathsf{with}\ x_i = e_i$ 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 $e \overset{n}{\hookrightarrow} v$ then $\Gamma^+ \vdash v{:}\, t^n$.*

Proof: Induction on the derivation of $e \overset{n}{\hookrightarrow} v$. 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 sqr (exp (n div 2) x)
                 else x * (exp (n-1) x);

(* real -> real *)
fun sqr 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 λ^{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 `sqr` 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 sqr1(exp1 (n div 2) x)
      else <~x * ~(exp1 (n-1) x)>

(* <real> -> <real> *)
fun sqr1 x = <let val y = ~x in  y * y end>
```

2. The third step consists only of the composition of the functions `back` and `sqr1`. The function `back` is polymorphic in the *MetaML* system, which has been extended to support Hindley-Milner polymorphism. This extension is beyond of 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 [1] because the Run construct disallows the use of free variables such as `exp2`.

[1]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 λ^{BN}

1. Annotating the function `exp` in λ^{BN} is similar to *MetaML* but a bit more tedious than *MetaML*. We need to tell the type system that the functions `sqr1` 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 sqr1 x =
             <let val y = ~x in y * y end>
           fun exp1 n x =
             if n=0 then <1.0>
             else if even(n)
                  then sqr1(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 λ^{BN}, 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 λ^{BN}. It first specializes the program on the exponent then executes the specialized program on the second argument.

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

Contents