January 1998

A categorical analysis of multi-level languages (extended abstract)

Zine El-Abidine Benaissa

Eugenio Moggi

Walid Taha

Tim Sheard

A Categorical Analysis of Multi-Level Languages (Extended Abstract)

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
E.Moggi, DISI, Univ. di Genova, v. Dodecaneso 35, 16146 Genova, Italy
tel: +39-010-353 6629, fax: +39-010-353 6699, e-mail: moggi@disi.unige.it

Abstract. We propose categorical models for $\lambda^{\bigcirc}$, $\lambda^{\Box}$, *MetaML*, and *AIM*. First, we focus on the underlying logical modalities and the interactions between them, then we investigate the interactions between logical modalities and computational monads. We give two examples of categorical model: one simpler but with some limitations, the other more complex but able to model all features of *AIM*.

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

1 Introduction

This paper proposes a categorical semantics for multi-level languages like $\lambda^{\bigcirc}$, $\lambda^{\Box}$, *MetaML* and *AIM* (see [4, 5, 12, 11]). Developing such a semantics has a number of benefits, including:

- Suggesting simplifications and extensions. We have already simplified the type system of *MetaML* and proposed an extension with *closed code types* called *AIM* (see [11]).
- Validating equational reasoning principles. In this paper we have not established any computational adequacy results, and therefore we cannot formally claim that equality in a model entails observational equivalence (where code inspection is not among the allowed observations). However, we expect such results to hold, and their proof should exploit Kripke logical relations (see [10]).
- Explaining multi-level languages in terms of more primitive concepts, namely *logical* modalities (in the sense that the modalities are characterized by universal properties) and *computational monads*.

Multi-level languages provide generic constructs for the manipulation of code fragments. They can be viewed as *instances* of two-level languages, in which the object language is the multi-level language itself. We study four multi-level languages:

- $\lambda^{\Box}$ [5], proving constructs for the construction and the execution of closed code. Such a language is useful in machine-code generation.
- $\lambda^{\bigcirc}$ [4], providing constructs for manipulating open code fragments. Such a language is useful in high-level program generation and inlining.
- *MetaML* [13, 12], providing an additional construct for the execution of such fragments, and cross-stage persistence. Cross-stage persistence is the ability to use at one level a variable declared at a lower level. Both features are important for pragmatic reasons.
- *AIM* [11], revising and extending *MetaML* with a closed code type for expressivity and modularity.

$\lambda^{\Box}$ and $\lambda^{\bigcirc}$ already have clean, logical foundations (see [4, 5, 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* had no such foundations, nor the formal hygiene they often promote. Indeed, *MetaML* had a complex type system and a number of ad hoc restrictions (see [12]), which demanded deeper investigation and possibly simplification. Starting from the categorical account of two-level languages [9], we arrive at a number of results for multi-level languages:

- 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)[1], we give a definition of what constitutes a categorical model for *simply typed* multi-level languages, namely $\lambda^{\Box}$, $\lambda^{\bigcirc}$, and *AIM*, and consider some examples.

[1] We replace the notion of symmetric monoidal adjunction with FP-adjunction.

- We give the interpretation (denotational semantics) of *AIM* without cross-stage persistence nor computational effects in an *AIM*-model.

- 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 *AIM* in the presence of computational effects, and discuss the subtleties involved in the interpretation of the *run-with* construct.

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

- If $\mathcal{C}$ is a category, we write $|\mathcal{C}|$ for the set of objects, $\mathcal{C}(A, B)$ for the hom-set of maps from A to B.

- We write GF for $G \circ F$ and GFA for $G(FA)$, when F and G are functors/functions and A an object.

- We write arrow $\hookrightarrow$ for a full and faithful functor, and $F \dashv G$ for an *adjunction*, where F is the left-adjoint and G the right-adjoint.

- We write $(x_n | n \in N)$ for an infinite sequence, and $(x_i | i \in m)$ for a finite sequence of length m (we identify the natural number m with the set of its predecessors). Sometimes we write x_i for $(x_i | i \in m)$ when m is clear from the context. If s is a sequence and x an element, we write $x::s$ for the sequence obtained by adding x in front of s.

- We write $n+$ for $n+1$ and $n-$ for $n-1$.

- We use Haskell's notation $\mathsf{do}\{x_i \leftarrow e_i; e\}$ and $\mathsf{ret}\ e$ for monads, instead of the notation $\mathsf{let}\ x_i \Leftarrow e_i\ \mathsf{in}\ e$ and $[e]$ from [8]. If $op: \prod_i A_i \to MB$, we write $\overline{op}: \prod_i MA_i \to MB$ for its **monadic extension**, i.e. $\overline{op}(u_i) \stackrel{\Delta}{=} \mathsf{do}\{x_i \leftarrow u_i; op(x_i)\}$.

2 Multi-Level Languages

We begin by describing the syntax and type systems of the four multi-level languages investigated in this paper, i.e. $\lambda^\square$, $\lambda^\bigcirc$, *MetaML* and *AIM*. We adopt the following unified notation for types:

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

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

$\lambda^\square$ of [5] features function and closed code types. Typing judgments have the form $\Delta; \Gamma \vdash e:t$, where $\Delta, \Gamma \equiv \{x_i : t_i | i \in m\}$. The syntax for $\lambda^\square$ 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 $\lambda^\square$ is given in Figure 1.

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

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

The first four constructs are the standard ones in a λ-calculus with constants. Bracket and Escape (called Next and Prev in [4]) allow the construction and combination of open code. Brackets construct code, and Escapes splice a code fragment into the context a bigger code fragment. A term such as `(fn x => <(~x,~x)>) <5>` yields `<(5,5)>` when executed. The rules for constants, variables, and applications are essentially standard.

MetaML [13, 12] 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 $\lambda^\bigcirc$ with

$$e \in E ::= \quad \ldots \mid \mathsf{run}\ e$$

Run allows the execution of a code fragment. For example, `run <3+4>` is well-typed and evaluates to `7`.

AIM [11] extends *MetaML* with an analog of the Box type of $\lambda^\square$ yielding a more expressive language, and yet has a simpler type judgment than *MetaML*[2]. The syntax of *AIM* extends that of *MetaML* as follows:

$$\begin{aligned} e \in E ::= \quad & \ldots \mid \mathsf{run}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \\ & \mathsf{box}\ e\ \mathsf{with}\ \{x_i = e_i | i \in m\} \mid \mathsf{unbox}\ e \end{aligned}$$

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 2, 3 and 4, while the big-step operational semantics of *AIM* and its sub-languages is in Figure 5.

Now that the basic multi-level constructs have been introduced, we illustrate the need for both open and closed code types in staged programming.

[2] The presentation of *MetaML* in this paper uses the simpler type judgment of *AIM*, for reasons of space.

Uses of open code: Taylor Series. Consider generating code for an embedded system (e.g. the controller of a robot) that requires computing the *sin* function using Taylor series polynomial around 0:

$$\sum_{k=0}^{n} \frac{-1^k\ x^{2k+1}}{(2k+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} \ldots$$

First we write a function to add the first n coefficients:

```
val sinN : int -> real -> real
```

If we determine n at the time of generating our program, Brackets and Escapes can be used to derive a similar function that manipulates "representations" of x instead of the value of x itself, and where the result is a representation of the desired polynomial:

```
val sinN : int -> <real> -> <real>
```

To construct the definition of the desired code fragment, we need the following construction:

```
fun sinN' n = <fn x => ~(sinN n <x>)>;
   : int -> <real -> real>
```

which allows us to derive the expansion for any n:

```
val sinN3 = sinN' 3;
= <(fn a =>
     let val b = a * a; val c = b * a;
         val d = b * c; val e = b * d
      in a/1.0 + c/-6.0
       + d/120.0 + e/-5040.0
     end)> : <real -> real>
```

where b is bound to x^2, c to x^3, d to x^5, and so on. In this code, the factorial expressions have been precomputed, and fairly efficient code was generated to perform this computation. Thus, the construction of the desired expression is performed symbolically, once and for all, *before* we know the value of x.

To achieve this kind of "unfolding" ("symbolic computation", or "reduction under lambda"), it is necessary to apply `sinN` to the *open code fragment* `<x>`, where `x` has not yet been bound, and is therefore still *a free variable*. Such unfolding *cannot* be achieved in $\lambda^{\Box}$.

To execute `sinN3` we use the Run construct:

```
val sin = (run sinN3) : real -> real
```

Caveat: Typing Run Unfortunately, typing the above use of Run is problematic. In fact, typing the use of Run on a code fragment constructed in a previous declaration is problematic, even in the trivial example

```
val one = let val a = <1> in run a end
```

because, using the standard interpretation for let, it is the same as typing:

```
val one = (fn a => run a) <1>
```

But `(fn a => run a) : <'a> -> 'a` is not derivable in *MetaML*'s type system, and for good reason: *An open code fragment, in general, cannot be executed.* One solution is to use (for type checking purposes only) an interpretation of the let-statement using direct substitution. This would make the first declaration for `one` typable, but impairs the efficiency of type-checking. In the existing implementation of *MetaML*, *ad hoc* solutions were used to overcome this problem for top-level declarations (See [13]).

Solution: Closed Code *AIM*'s type system addresses the cause of the typing problem described above: *to ensure that a code fragment can be executed, we ensure that it is closed.* This is achieved by adding the Box type to *MetaML*. From the programmer's viewpoint the main new concept is that all code fragments and functions used in the construction of a new closed fragment, must be Boxed to ensure that they do not have free variables. In the trivial example of let-binding, we simply rewrite our expression as:

```
val one = let val a = box <1>
           in run (unbox b) with {b=a} end
```

In our example, the basic function must have the type:

```
val sinM : [int -> <real> -> <real>]
```

This is easily accomplished by surrounding the definitions of the symbolic `sinN` by `box (...)`. Now, we can describe the desired computation using the following *well-typed AIM* terms:

```
val sinM' = box fn n => <fn x => ~(s n <x>)>
             with {s=sinM});
   : [int -> <real -> real>]
val sin = run (unbox s) 3 with {s=sinM'}
   : real -> real
```

3 Categorical Models

In this section we define what is a categorical model for various multi-level languages, namely $\lambda^{\Box}$, $\lambda^{\bigcirc}$ and *AIM* (see Definition 3.6, 3.8 and 3.10). At first we ignore computational effects, and focus on the *logical modalities* underpinning these languages. Previous work by

Davies and Pfenning has already established a correspondence between closed code types and the necessity modality of S4, and between open code types and the next modality of linear time temporal logic. We show that these modalities can be described in terms of *FP-adjunctions*, and explain how they should interact to provide a model for *AIM*.

Definition 3.1 *$\mathcal{D} \substack{\xrightarrow{\;G\;} \\ \top \\ \xleftarrow[\;F\;]{}} \mathcal{C}$ is an FP-adjunction iff it is an adjunction in the 2-category of categories with finite products and functors preserving them (or equivalently it is an adjunction where the left adjoint F preserves finite products).*

Remark 3.2 We use the FP- prefix to indicate any 2-categorical notion (e.g. category, functor, monad, adjunction) specialized to the 2-category introduced above.

An FP-adjunction is a special case of a *symmetric monoidal adjunction*, which has been used to give an elegant definition of what is a categorical model for intuitionistic linear logic (see [1, 2, 3]).

We recall some properties of FP-adjunctions (and FP-functors), which will be exploited in the sequel.

Proposition 3.3 *If $\mathcal{C}$ is a CCC and $\mathcal{D} \substack{\hookrightarrow \\ \top \\ \xleftarrow[\;F\;]{}} \mathcal{C}$ is an FP-adjunction, then $\mathcal{D}$ is an exponential ideal of $\mathcal{C}$, i.e. $Y^X \in \mathcal{D}$ (up to iso) for any $Y \in \mathcal{D}$ and $X \in \mathcal{C}$.*

Definition 3.4 *An FP-functor $F: \mathcal{C} \to \mathcal{D}$ induces the following* simple *$\mathcal{C}$-indexed FP-category $\mathcal{S}: \mathcal{C}^{op} \to$* **Cat**

- *$|\mathcal{S}_X| \triangleq |\mathcal{D}|$ and $\mathcal{S}_X(A, B) \triangleq \mathcal{D}(FX \times A, B)$.*
- *$h \circ_X g \triangleq h \circ \langle \pi_1, g \rangle \in \mathcal{S}_X(A, C)$, where $g \in \mathcal{S}_X(A, B)$, $h \in \mathcal{S}_X(B, C)$ and $\pi_1: (FX) \times A \to FX$ is the first projection. While the identity for A in $\mathcal{S}_X$ is the second projection $\pi_2: FX \times A \to A$.*
- *substitution $f^*: \mathcal{S}_X \to \mathcal{S}_Y$ along $f \in \mathcal{C}(Y, X)$ is given by $f^*(A) \triangleq A$ and $f^*(g) \triangleq g \circ (Ff \times \mathrm{id})$.*

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

Proposition 3.5 *The simple indexed category $\mathcal{S}$ of Definition 3.4 has the following categorical structure:*

- *finite products, i.e. $\prod_{i \in m} \mathcal{S}_X(A, B_i) \cong \mathcal{S}_X(A, \prod_{i \in m} B_i)$*
- *simple existential quantification $\exists_Y A \triangleq FY \times A$, i.e. $\mathcal{S}_{X \times Y}(A, B) \cong \mathcal{S}_X(\exists_Y A, B)$*
- *exponentials, i.e. $\mathcal{S}_X(C \times A, B) \cong \mathcal{S}_X(C, B^A)$, provided $\mathcal{D}$ is CCC*
- *simple universal quantification $\forall_Y A \triangleq A^{FY}$, i.e. $\mathcal{S}_{X \times Y}(A, B) \cong \mathcal{S}_X(A, \forall_Y B)$, provided $\mathcal{D}$ is CCC*
- *simple comprehension, i.e. $\mathcal{S}_X(1, A) \cong \mathcal{C}(X, GA)$, provided $F \dashv G$ is an FP-adjunction.*

Definition 3.6 *A $\lambda^{\Box}$-model is given by a CCC $\mathcal{D}$ and an FP-adjunction $\mathcal{D} \substack{\xrightarrow{\;\mathsf{G}\;} \\ \top \\ \xleftarrow[\;\mathsf{F}\;]{}} \mathcal{C}$.*

Remark 3.7 The pattern for interpreting $\lambda^{\Box}$ is to interpret a type t by an object $[\![t]\!]$ of $\mathcal{D}$, namely

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

and a term $\{x_i : t_i | i \in m\}; \{x_j : t_j | j \in n\} \vdash_{\Box} e : t$ is by a map in $\mathcal{S}_X(\prod_{j \in n} [\![t_j]\!], [\![t]\!])$ where $X \triangleq (\prod_{i \in m} \mathsf{G}[\![t_i]\!])$.

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, the objects of $\mathcal{C}$ relevant for the interpretation have the form $\mathsf{G}A$, and so we could take $\mathcal{C}$ to be the co-Kleisli category $\mathcal{D}_{\mathsf{B}}$ for B, which is always a CCC (however in a $\lambda^{\Box}$-model $\mathcal{C}$ is not required to be a CCC).

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 $\Delta, x : t; \Gamma \vdash_{\Box} e_1 : t'$ and those of the form $\Delta; x : [t], \Gamma \vdash_{\Box} e_2 : t'$ given by

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

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

Definition 3.8 *A $\lambda^{\bigcirc}$-model is given by a CCC $\mathcal{D}$ and an FP-adjunction $\mathcal{D} \substack{\xhookrightarrow{\;\mathsf{N}\;} \\ \top \\ \xleftarrow[\;\mathsf{P}\;]{}} \mathcal{D}$.*

Remark 3.9 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 $\{x_i : t_i^{n_i} | i \in m\} \vdash_{\bigcirc} e : t^n$ by a map in $\mathcal{D}(\prod_{i \in m} \mathsf{N}^{n_i}[\![t_i]\!], \mathsf{N}^n[\![t]\!])$.

The assumption "N is full and faithful" ensures that N preserves the whole CCC structure (see Proposition 3.3), therefore one may safely confuse $\mathsf{N}^n[\![t_1 \to t_2]\!]$ with $(\mathsf{N}^n[\![t_2]\!])^{\mathsf{N}^n[\![t_1]\!]}$ (formalizing Section 8 of [13]).

In *AIM* closed and open code types coexists, and so the key point is to clarify how the modalities of $\lambda^{\Box}$ and $\lambda^{\bigcirc}$ interact. The basic idea is that a model for *AIM* is a $\lambda^{\Box}$-model where the category $\mathcal{D}$ has the structure of a $\lambda^{\bigcirc}$-model *parameterized* w.r.t. $\mathcal{C}$. The precise formulation uses the simple indexed category of Definition 3.4.

Definition 3.10 *An* AIM-*model is given by a CCC* $\mathcal{D}$*, an FP-adjunction*

$$\begin{tikzcd} \mathcal{D} \arrow[r, shift left=1.5ex, "\mathsf{G}"] \arrow[r, phantom, "\top"] & \mathcal{C} \arrow[l, shift left=1.5ex, "\mathsf{F}"] \end{tikzcd}$$

, and a $\mathcal{C}$*-indexed FP-adjunction*

$$\begin{tikzcd} \mathcal{S} \arrow[r, hook, shift left=1.5ex, "\mathsf{N}"] \arrow[r, phantom, "\top"] & \mathcal{S} \arrow[l, shift left=1.5ex, "\mathsf{P}"] \end{tikzcd}.$$

Remark 3.11 The above definition of an *AIM*-model fails to capture cross-stage persistence. This can be easily fixed by requiring a natural transformation $up: A \to \mathsf{N}A$ (satisfying some additional properties), but we prefer not to include *up* in the definition of an *AIM*-model (we will see also models without *up*).

The pattern for interpreting *AIM* mimics that for $\lambda^{\bigcirc}$, i.e. a type t is interpreted by an object $[\![t]\!]$ of $\mathcal{D}$, namely

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

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

Proposition 3.12 *In any* AIM-*model there are two canonical isomorphisms* $compile: \mathsf{GN}A \to \mathsf{G}A$ *and* $down: \mathsf{PF}X \to \mathsf{F}X$.

Remark 3.13 These isomorphisms suggest an extension of *AIM* with $\mathsf{up} : [t] \to \langle [t] \rangle$, i.e. cross-stage persistence for close code types, and $\mathsf{compile}: [\langle t \rangle] \to [t]$.

3.1 Examples

We give examples of *AIM*-models parameterized w.r.t. the category $\mathcal{C}$, making explicit what additional structure or properties are needed. For each example we define the category $\mathcal{D}$, the action on objects of the functors N, P, F and G.

Example 3.14 Let N be the set of naturals. Given a CCC $\mathcal{C}$ with N-indexed products, take

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

Example 3.14 does not support cross-stage persistence. Therefore, it is suitable for interpreting $\lambda^{\bigcirc}$, but not *MetaML* or *AIM* (as defined in [12, 11]).

Example 3.15 Let ω^{op} be the category of natural numbers with the reverse order, i.e.

$$0 \longleftarrow 1 \quad \dots \quad n \longleftarrow n+ \quad \dots$$

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

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

In this model we can define the natural transformation $up: A \to \mathsf{N}A$ modeling cross-stage persistence, namely $up_0 \triangleq !: A_0 \to 1$ and $up_{n+} \triangleq a_n : A_{n+} \to A_n$.

Note that exponentials in $\mathcal{D}$ are not defined pointwise. However, existence of exponentials and finite limits in $\mathcal{C}$ ensures that $\mathcal{D}$ has exponentials (and finite limits).

4 Interpretation of terms

We have already given the interpretation of types for *AIM* without computational effects or cross-stage persistence in an *AIM*-model, 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 morphisms, which simplify the definition of the interpretation, and clarify the similarities with the interpretation of the λ-calculus in a CCC.

- $c_n : 1 \to \mathsf{N}^n A$ where $c : 1 \to A$ is a global element of A (e.g. the interpretation of a constant). Since N preserve finite products, we define $c_n \triangleq \mathsf{N}^n c$.
- $\lambda_n : (\mathsf{N}^n B)^{\mathsf{N}^n A} \to \mathsf{N}^n B^A$. Since N preserves the CCC structure, λ_n is the iso $(\mathsf{N}^n B)^{\mathsf{N}^n A} \to \mathsf{N}^n B^A$.
- $@_n : \mathsf{N}^n B^A \times \mathsf{N}^n A \to \mathsf{N}^n B$. $@_n$ is essentially an instance of evaluation $eval : (\mathsf{N}^n B)^{\mathsf{N}^n A} \times \mathsf{N}^n A \to \mathsf{N}^n B$.
- $unbox_n : \mathsf{N}^n \mathsf{B} A \to \mathsf{N}^n A$. Since B is a comonad with co-unit $\epsilon : \mathsf{B}A \to A$ and co-multiplication $\delta : \mathsf{B}A \to \mathsf{B}^2 A$, then $unbox_n \triangleq \mathsf{N}^n \epsilon$.
- $box_n(f) : \prod_i \mathsf{N}^n \mathsf{B} A_i \to \mathsf{N}^n \mathsf{B} B$ when $f : \prod_i \mathsf{B} A_i \to B$. Since all functors preserve finite products, it suffices to say that $box_n(f) \triangleq \mathsf{N}^n((\mathsf{B} f) \circ \delta) : \mathsf{N}^n \mathsf{B} A \to \mathsf{N}^n \mathsf{B} B$ when $f : \mathsf{B} A \to B$ and $A \triangleq \prod_i A_i$.
- $run_n(f) : C \times \prod_i \mathsf{N}^n \mathsf{B} A_i \to \mathsf{N}^n B$ when $f : \mathsf{N} C \times \prod_i \mathsf{N}^n \mathsf{B} A_i \to \mathsf{N}^{n+} B$. As in case of $box_n(f)$ it suffices to give $run_n(f) : C \times \mathsf{N}^n \mathsf{B} A \to \mathsf{N}^n B$ when $f : \mathsf{N} C \times \mathsf{N}^n \mathsf{B} A \to \mathsf{N}^{n+} B$ and $A \triangleq \prod_i A_i$.

  By the canonical iso *down* (see Proposition 3.12) we have $C \times \mathsf{N}^n \mathsf{B} A \cong C \times \mathsf{N}^n \mathsf{P} \mathsf{B} A$. We have an FP-monad $I_n \triangleq \mathsf{N}^n \mathsf{P}^n$ on $\mathcal{D}$ with unit $\eta^I_n : A \to I_n A$ induced by the FP-adjunction $\mathsf{P}^n \dashv \mathsf{N}^n$. Moreover, we have an iso $\mathsf{P}\mathsf{N} A \to A$ given by the co-unit of the adjunction $\mathsf{P} \dashv \mathsf{N}$, since N is full and faithful. Therefore, modulo some canonical isos $run_n(f)$ is

  $$C \times \mathsf{N}^n \mathsf{P} \mathsf{B} A \xrightarrow{\eta^I_n} I_n C \times \mathsf{N}^n \mathsf{P} \mathsf{B} A \xrightarrow{I_n \mathsf{P} f} \mathsf{N}^n B$$

Figure 6 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 4.

5 Modalities and monads

We have given a simplified interpretation of *AIM* (and other multi-level languages) in the absence of *computational effects*. This interpretation is the analogue of the interpretation of the simply typed λ-calculus in a CCC. However, we are interested in multi-level programming languages, like *Mini-ML*$^{\Box}$ *Mini-ML*$^{\bigcirc}$, and *MetaML* (see [5, 4, 13]), where logical modalities co-exist with computational effects. In this section we define a *CBV monadic* interpretation of *AIM* in an *AIM*-model equipped with a strong monad (see [8]).

Definition 5.1 *A monadic* AIM*-model is a* AIM*-model with a strong monad M over $\mathcal{D}$ s.t. the canonical morphism $M\mathsf{N}B^A \to (M\mathsf{N}B)^{\mathsf{N}A}$ is an iso, and we call $\lambda_* : (M\mathsf{N}B)^{\mathsf{N}A} \to M\mathsf{N}B^A$ its inverse.*

The idea is that M models computation at level 0. We extend the *AIM*-models of Examples 3.14 and 3.15 to monadic *AIM*-models.

Example 5.2 A strong monad M over $\mathcal{C}$ induces a strong monad M over $\mathcal{C}^N$ given by $(MA)_0 \triangleq MA_0$ and $(MA)_{n+} \triangleq A_{n+}$. It is immediate to check that the additional requirement is always satisfied, since exponentiation in $\mathcal{C}^N$ is pointwise.

Example 5.3 A strong monad M over $\mathcal{C}$ induces a strong monad M over $\mathcal{C}^{\omega^{op}}$, namely MA is given by

$$\begin{tikzcd} MA_0 & MA_1 \arrow[l, "Ma_0"'] & \ldots & MA_n & MA_{n+} \arrow[l, "Ma_n"'] & \ldots \end{tikzcd}$$

The additional requirement holds, provided the monad M over $\mathcal{C}$ 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 $e(u) \triangleq \lambda x : A.\mathsf{do}\{f \leftarrow u; \mathsf{ret}\ (fx)\}$ and $k(u) \triangleq \lambda x : A.u$.

Remark 5.4 The interaction of M with pullbacks is important, because exponentials in $\mathcal{C}^{\omega^{op}}$ are computed using exponentials and pullbacks in $\mathcal{C}$. Many monads over the category of cpos (e.g. lifting, state and exception monad) satisfy the properties required in Example 5.3, but notable exceptions are power-domains and continuations.

Interpretation of types. A type t is interpreted (as usual) by an object $[\![t]\!]$ of $\mathcal{D}$, namely:

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

We introduce the shorthand N_* for $M\mathsf{N}$ and M_n for $(M\mathsf{N})^n M$. We call $M_n A$ the **type of n-stage computations** returning (at stage n) a value of type A. In a monadic *AIM*-model a term $\{x_i : t_i^{n_i} | i \in m\} \vdash e : t^n$ is interpreted by a map in $\mathcal{D}(\prod_{i \in m} \mathsf{N}^{n_i}[\![t_i]\!], M_n[\![t]\!])$.

Remark 5.5 This interpretation is a *refinement* of the interpretation given in Section 4, which is recovered by replacing M with the identity monad, and it *extends* the CBV interpretation of the simply typed λ-calculus (in a CCC with a strong monad). M_n is always a functor, but in general it is not a monad.

Auxiliary morphisms. We introduce some auxiliary morphisms, similar to those given in Section 4. The only exception is the morphism $run_n(f)$, which we have been unable to define in general, but will be given for specific models. (We use notation introduced in Notation 1.1.)

- $\eta_n: \mathsf{N}^n A \to \mathsf{N}^n_* A$ is given by induction:

  0) $A \xrightarrow{\mathrm{id}} A$

  $n+$) $\mathsf{N}^{n+}A \xrightarrow{\eta} M\mathsf{N}^{n+}A \xrightarrow{M\mathsf{N}\eta_n} \mathsf{N}^{n+}_* A$

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

- $\psi_n: \prod_i \mathsf{N}^n_* A_i \to \mathsf{N}^n_* \prod_i A_i$ is given by induction:

  0) $\prod_i A_i \xrightarrow{\mathrm{id}} \prod_i A_i$

  $n+$) $\prod_i \mathsf{N}^{n+}_* A_i \xrightarrow{\psi} \mathsf{N}_* \prod_i \mathsf{N}^n_* A_i \xrightarrow{\mathsf{N}_*\psi_n} \mathsf{N}^{n+}_* \prod_i A_i$

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

- $c_n \triangleq 1 \xrightarrow{\mathsf{N}^n c} \mathsf{N}^n MA \xrightarrow{\eta_n} \mathsf{N}^n_* MA \equiv M_n A$, where $c: 1 \to MA$ is a global element of MA.

- $var_n \triangleq \mathsf{N}^n A \xrightarrow{\mathsf{N}^n \eta} \mathsf{N}^n MA \xrightarrow{\eta_n} \mathsf{N}^n_* MA \equiv M_n A$.

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

  0) $(MB)^A \xrightarrow{\eta} M(MB)^A$

  $n+$)
$$\begin{tikzcd} (M_{n+}B)^{\mathsf{N}^{n+}A} \arrow[r, "\lambda_*"] & \mathsf{N}_*(M_n B)^{\mathsf{N}^n A} \arrow[d, "\mathsf{N}_*\lambda_n"] \\ & M_{n+}(MB)^A \end{tikzcd}$$

- $@_n: M_n(MB)^A \times M_n A \to M_n B$ is given by $(\mathsf{N}^n_*(\overline{eval})) \circ \psi_n$, where $eval: (MB)^A \times A \to MB$ is an instance of evaluation.

- $unbox_n: M_n \mathsf{B} MA \to M_n A$ is given by $\mathsf{N}^n_*(\overline{\epsilon})$, where $\epsilon: \mathsf{B}MA \to MA$ is an instance of the co-unit for B.

- $box_n(f): \prod_i M_n \mathsf{B} MA_i \to M_n \mathsf{B} MB$ is given by $\mathsf{N}^n_*(\overline{(\mathsf{B}f) \circ \delta}) \circ \psi_n$, where $f: \prod_i \mathsf{B}MA_i \to MB$, δ is an instance of the co-multiplication for B, and we exploit preservation of finite products by B.

The interpretation of terms. Figure 7 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 4 (without run-with). We give the interpretation of run-with in the monadic *AIM*-models of Example 5.2 and 5.3. To interpret run-with we need an auxiliary morphism

- $run_n(f): C \times \prod_i M_n \mathsf{B} MA_i \to M_n B$ for any $f: \mathsf{N}C \times \prod_i \mathsf{N}^n \mathsf{B} MA_i \to M_{n+} B$.

For simplicity, in the sequel we assume that there is only one A_i, and call it A.

Example 5.6 In the monadic *AIM*-model based on $\mathcal{C}^N$ we can define $run_n(f)$ only when C is replaced by $\mathsf{N}^n C$. In this model we have

$$(M_n A)_m = \begin{cases} M1 & \text{when } m < n \\ MA_0 & \text{when } m = n \\ A_{m-n} & \text{when } m > n \end{cases}$$

Let $g \triangleq run_n(f): \mathsf{N}^n C \times M_n \mathsf{B} MA \to M_n B$, we define its mth component g_m (a map in $\mathcal{C}$) by case-analysis:

$< n$) $g_m(x: 1, v: M1) = \mathsf{do}\{y \leftarrow v; f_m(x, y)\}$, where $f_m: 1 \times 1 \to M1$

$= n$) $g_n(x: C_0, v: MX) = \mathsf{do}\{y \leftarrow v; f_n(*, y); f_{n+}(x, y)\}$ where $X \triangleq (\prod_n MA_n)$, $f_n: 1 \times X \to M1$ and $f_{n+}: C_0 \times X \to MB_0$

$> n$) $g_m(x: C_k, v: MX) = \mathsf{do}\{y \leftarrow v; f_{m+}(x, y)\}$, where $k = m - n$ and $f_{m+}: C_k \times X \to MB_k$.

Remark 5.7 In the absence of computational effects we defined $run_n(f)$ by applying the functor $\mathsf{N}^n \mathsf{P}^{n+}$ to f. In $\mathcal{C}^N$ this functor replaces the mth component f_m with !, when $m \leq n$. If the codomain of f_m is the terminal object 1, we don't lose any information. However, in the monadic interpretation the codomain of f_m is not 1 but $M1$. Informally speaking, the above definition of $g = run_n(f)$ does not loose information, because it maps f_m to g_m when $m < n$, *collapses* f_n and f_{n+} into g_n, and maps f_{m+} to g_m when $m > n$. The interpretation in $\mathcal{C}^N$ has a serious caveat, namely if we have a natural transformation $c: 1 \to MA$ in $\mathcal{C}$ (e.g. $\perp: 1 \to A_\perp$) there is no *generic* way of lifting it to a natural transformation $c_n: 1 \to M_n A$ in $\mathcal{C}^N$.

Example 5.8 In the monadic *AIM*-model based on $\mathcal{C}^{\omega^{op}}$ we define $run_n(f)$ without imposing any restriction on C. In this model we have

$$(M_n A)_m = \begin{cases} M^{m+}1 & \text{when } m < n \\ M^{n+}A_{m-n} & \text{when } m \geq n \end{cases}$$

Let $X \triangleq \mathsf{G}MA$, then $f\colon \mathsf{N}C \times \mathsf{N}^n\mathsf{F}X \to M_{n+}B$ and we have to define $run_n(f)\colon C \times M_n\mathsf{F}X \to M_nB$:

- first we define $F\colon C \to \mathsf{P}M_n(\mathsf{N}_*MB)^{\mathsf{F}X}$ as

$$\mathsf{P}(\ \mathsf{N}C \xrightarrow{\Lambda f} (M_{n+}B)^{\mathsf{N}^n\mathsf{F}X} \xrightarrow{\lambda_n} M_n(\mathsf{N}_*MB)^{\mathsf{F}X}\ )$$

- then we define $R\colon \mathsf{P}M_n(\mathsf{N}_*MB)^{\mathsf{F}X} \to M_n(MB)^{\mathsf{F}X}$, namely its mth component R_m, by case-analysis:

$< n-)$ $R_m \triangleq M^{m+}!\colon M^{m++}1 \to M^{m+}1$

$= n-)$ $R_{n-} \triangleq M^n!\colon M^{n+}(M1)^X \to M^n 1$

$\geq n)$ $R_m \triangleq M^{n+}\mu^X\colon M^{n+}(M^2B_k)^X \to M^{n+}(MB_k)^X$, where $k = m - n$

although exponentiation in $\mathcal{C}^{\omega^{op}}$ is not pointwise, in the special case of exponentiation by $\mathsf{F}X$ it is.

- finally we define $run_n(f)\colon C \times M_n\mathsf{F}X \to M_nB$ as

$$C \times M_n\mathsf{F}X \xrightarrow{R \circ F \times \mathrm{id}} M_n(MB)^{\mathsf{F}X} \times M_n\mathsf{F}X \xrightarrow{@_n} M_nB$$

Remark 5.9 The monadic *AIM*-model in $\mathcal{C}^{\omega^{op}}$ does not have the serious caveat we mentioned for $\mathcal{C}^N$. Moreover, it has a property that we call *cross-stage persistence of computational effects*, i.e. there exists an iso $down_M\colon M\mathsf{P}A \to \mathsf{P}MA$ (commuting with the monad structure).

Monadic interpretation of compile. In any *AIM*-model there is an iso $compile\colon \mathsf{BN}A \to \mathsf{B}A$ (see Proposition 3.12), and therefore the pure interpretation of $[\langle t\rangle]$ and $[t]$ are isomorphic. Although the monadic interpretations of these types are not isomorphic, in the monadic *AIM*-models described above there is a morphism $compile'\colon \mathsf{B}M\mathsf{N}MA \to M\mathsf{B}MA$ suitable for interpreting $\mathsf{compile}\colon [\langle t\rangle] \to [t]$ with the following operational semantics $\dfrac{e \overset{0}{\hookrightarrow} \mathsf{box}\ e' \quad e' \overset{0}{\hookrightarrow} \langle v'\rangle}{\mathsf{compile}\ e \overset{0}{\hookrightarrow} \mathsf{box}\ v'_0}$.

We define $compile'$ in $\mathcal{C}^N$ (in the other model one must assume that M over $\mathcal{C}$ preserves ω^{op}-limits). First, note that $(\mathsf{B}MA)_m = X \triangleq MA_0 \times \prod_n A_{n+1}$ and $(\mathsf{B}M\mathsf{N}MA)_m = M1 \times X$. It is now easy to define the mth component $compile'_m$ by case-analysis:

0) $compile'_0(u\colon M1, v\colon X) \triangleq \mathsf{do}\{u; \mathsf{ret}\ v\}$

> 0) $compile'_m(u\colon M1, v\colon X) \triangleq v$.

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

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

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

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

Figure 1: $\lambda^{\Box}$ 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 2: $\lambda^{\bigcirc}$ 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 3: *MetaML* Type System (+ Figure 2)

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

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

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

Figure 4: *AIM* Type System (+ Figure 2)

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

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

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

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

Figure 5: Big-Step Operational Semantics

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

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

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

$$
\frac{\begin{array}{c}[\![\Gamma \vdash e_i : [t_i]^n]\!] = f_i : C \to \mathsf{N}^n (\mathsf{B} A_i) \\ [\![\{x_i : [t_i]^0 | i\} \vdash e : t^0]\!] = f : \prod_i \mathsf{B} A_i \to A\end{array}}{[\![\Gamma \vdash \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i : [t]^n]\!] \triangleq box_n(f) \circ \langle f_i | i \rangle : C \to \mathsf{N}^n (\mathsf{B} A)}
\qquad
\frac{\begin{array}{c}[\![\Gamma \vdash e_1 : t \to t'^n]\!] = f_1 : C \to \mathsf{N}^n (B^A) \\ [\![\Gamma \vdash e_2 : t^n]\!] = f_2 : C \to \mathsf{N}^n A\end{array}}{[\![\Gamma \vdash e_1\ e_2 : t'^n]\!] \triangleq @_n \circ \langle f_1, f_2 \rangle : C \to \mathsf{N}^n B}
$$

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

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

Figure 6: Pure Interpretation in *AIM*-Models

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

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

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

$$
\frac{\begin{array}{c}[\![\Gamma \vdash e_i : [t_i]^n]\!] = f_i : C \to M_n (\mathsf{B} M A_i) \\ [\![\{x_i : [t_i]^0 | i\} \vdash e : t^0]\!] = f : \prod_i \mathsf{B} M A_i \to M A\end{array}}{[\![\Gamma \vdash \mathsf{box}\ e\ \mathsf{with}\ x_i = e_i : [t]^n]\!] \triangleq box_n(f) \circ \langle f_i | i \rangle : C \to M_n (\mathsf{B} M A)}
\qquad
\frac{\begin{array}{c}[\![\Gamma \vdash e_1 : t \to t'^n]\!] = f_1 : C \to M_n (M B)^A \\ [\![\Gamma \vdash e_2 : t^n]\!] = f_2 : C \to M_n A\end{array}}{[\![\Gamma \vdash e_1\ e_2 : t'^n]\!] \triangleq @_n \circ \langle f_1, f_2 \rangle : C \to M_n B}
$$

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

Figure 7: Monadic Interpretation in *AIM*-Models without run

References

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

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

[3] G. M. Bierman. What is a categorical model of intuitionistic linear logic? *LNCS*, 902, 1995.

[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. Functor categories and two-level languages. In *FoSSaCS '98*, volume 1378 of *LNCS*. Springer Verlag, 1998.

[11] E. Moggi, W. Taha, Z. Benaissa, and T. Sheard. An idealized MetaML: Simpler, and more expressive (includes proofs). Technical Report CSE-98-017, OGI, October 1998.

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

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