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 $\lambda^O$, $\lambda^\Pi$, 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 models: 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 $\lambda^O$, $\lambda^\Pi$, 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:

- $\lambda^\Pi$ [5], proving constructs for the construction and the execution of closed code. Such a language is useful in machine-code generation.

- $\lambda^O$ [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.

$\lambda^\Pi$ and $\lambda^O$ already have clean, logical foundations (see [4, 5, 7, 6]): there is a Curry-Howard isomorphism between $\lambda^O$ and linear time temporal logic, and between $\lambda^\Pi$ 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\footnote{We replace the notion of symmetric monoidal adjunction with FP-adjunction}), we give a definition of what constitutes a categorical model for simply typed multi-level languages, namely $\lambda^\Pi$, $\lambda^O$, and AIM, and consider some examples.
We give the interpretation (denotational semantics) of \textit{AIM} without cross-stage persistence nor computational effects in an \textit{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 \textit{AIM} in the presence of computational effects, and discuss the subtleties involved in the interpretation of the \textit{run-with} construct.

\textbf{Notation 1.1} We introduce notation and terminology used throughout the paper.

- If \( C \) is a category, we write \([C]\) for the set of objects, \( C(A, B) \) 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 \( x \text{——>} \) for a full and faithful functor, and \( F \vdash G \) 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 \texttt{do}\{\( x_i \leftarrow e_i; \epsilon \)} and \texttt{ret} \( \epsilon \) for monads, instead of the notation \texttt{let} \( x_i \leftarrow e_i \) in \( \epsilon \) and \( [\epsilon] \) from [8]. If \( \texttt{op}: \prod_i A_i \rightarrow MB \), we write \( \texttt{M} \text{op}\prod_i A_i \rightarrow MB \) for its monadic extension, i.e. \( \text{M}\text{op}(u_i) \triangleq \text{do}\{x_i \leftarrow u_i; \text{op}(x_i)\} \).

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. \( \lambda^\square \), \( \lambda^\circ \), \textit{MetaML} and \textit{AIM}. We adopt the following unified notation for types:

\[
\tau \in \text{T} ::= b \mid t_1 \rightarrow t_2 \mid \langle t \rangle \mid [t]
\]

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

\[
\text{val } \sinN : \text{int} \to \text{real} \to \text{real}
\]

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:

\[
\text{val } \sinN : \text{int} \to <\text{real}> \to <\text{real}>
\]

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

\[
\text{fun } \sinN' n = \langle \text{fn } x => \langle(\sinN n x)\rangle\rangle;
\]

: \text{int} \to <\text{real} -> \text{real}>

which allows us to derive the expansion for any \( n \):

\[
\text{val } \sinN3 = \sinN' 3;
\]

\[
= \langle\text{fn } a =>
\begin{align*}
&\text{let } \text{val } b = a * a; \text{val } c = b * a; \\
&\text{val } d = b * c; \text{val } e = b * d \\
&\text{in } a/1.0 + c/-6.0 \\
&+ d/120.0 + e/-5040.0
\end{align*}
\]

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 pre-computed, 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 \( \lambda^2 \).

To execute \( \sinN3 \) we use the \text{Run} construct:

\[
\text{val } \text{sin } = \langle\text{run } \sinN3\rangle : \text{real} \to \text{real}
\]

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

\[
\text{val } \text{one } = \text{let } \text{val } a = <1> \text{ in } \text{run } a \text{ end}
\]

because, using the standard interpretation for \text{let}, it is the same as typing:

\[
\text{val } \text{one } = \langle\text{fn } a => \text{run } a\rangle <1>
\]

But (\langle\text{fn } a => \text{run } a\rangle) : \langle'a\rangle -> 'a is not derivable in \( \text{MetaML} \)'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 \text{let}-statement using direct substitution. This would make the first declaration for \text{one} typable, but impairs the efficiency of type-checking. In the existing implementation of \( \text{MetaML} \), ad hoc solutions were used to overcome this problem for top-level declarations (See [13]).

Solution: Closed Code. \( \text{AIM} \)'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 \text{Box} type to \( \text{MetaML} \). 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 \text{Boxed} to ensure that they do not have free variables. In the trivial example of \text{let}-binding, we simply rewrite our expression as:

\[
\text{val } \text{one } = \text{let } \text{val } a = \text{box} <1>
\]

\[
\text{in } \text{run } \text{unbox } b \text{ with } \{b=a\} \text{ end}
\]

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

\[
\text{val } \text{sinN} : \langle\text{int} -> <\text{real} -> <\text{real}>\rangle
\]

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

\[
\text{val } \text{sinN}' = \text{box } \text{fn } n => \langle\text{fn } x => \langle(s n x)\rangle\rangle
\]

\[
\text{with } \{s=\sinN\};
\]

: \langle\text{int} -> <\text{real} -> <\text{real}>\rangle
\]

\[
\text{val } \text{sin } = \text{run } \text{unbox } s \text{ 3 with } \{s=\sinN'\}
\]

: <real -> real

3 Categorical Models

In this section we define what is a categorical model for various multi-level languages, namely \( \lambda^2 \), \( \lambda^0 \) and \( \text{AIM} \) (see Definition 3.6, 3.8 and 3.10). At first we ignore computational effects, and focus on the \text{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** \( \mathcal{D} \xrightarrow{\mathcal{T}} \mathcal{C} \) is an FP-adjunction if 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 \( \mathcal{D} \xrightarrow{\mathcal{T}} \mathcal{C} \) is an FP-adjunction, then \( \mathcal{D} \) is an exponential ideal of \( \mathcal{C} \), i.e. \( Y^X \in \mathcal{D} \) (up to isomorphism) 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 \( \mathcal{S} : \mathcal{C}^\mathcal{D} \to \text{Cat} \):

- simple existential quantification \( \exists_Y A \equiv FY \times A \), i.e. \( \mathcal{S}_{X \times Y}(A, B) \equiv \mathcal{S}_X(\exists_Y A, B) \)
- exponentials, i.e. \( \mathcal{S}_X(C \times A, B) \equiv \mathcal{S}_X(C, BA) \), provided \( \mathcal{D} \) is CCC
- simple universal quantification \( \forall_Y A \equiv AFY \), i.e. \( \mathcal{S}_{X \times Y}(A, B) \equiv \mathcal{S}_X(A, \forall_Y B) \), provided \( \mathcal{D} \) is CCC
- simple comprehension, i.e. \( \mathcal{S}_X(1, A) \equiv \mathcal{C}(X, GA) \), provided \( F \dashv G \) is an FP-adjunction.

**Definition 3.6** A \( \lambda^\mathcal{D} \)-model is given by a CCC \( \mathcal{D} \) and an FP-adjunction \( \mathcal{D} \xrightarrow{\mathcal{T}} \mathcal{C} \).

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

\( [t] = FG[t] \) and \( [t_1 \to t_2] = [t_2]^{[t_1]} \)

and a term \( \{ x_j : t_j | j \in m \} \vdash c : t \) is by a map in \( \mathcal{S}(\prod_{j \in m} [t_j], [t]) \) where \( X \equiv (\prod_{j \in m} G[t_j]) \).

The FP-adjunction induces an FP-comonad \( B = FG \) on \( \mathcal{D} \). \( B \) is all that is needed for interpreting \( \lambda^\mathcal{D} \). In fact, the objects of \( \mathcal{C} \) relevant for the interpretation have the form \( GA \), and so we could take \( \mathcal{C} \) to be the co-Kleisli category \( \mathcal{D}_B \) for \( B \), which is always a CCC (however in a \( \lambda^\mathcal{D} \)-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 \( \Delta, x : t; \Gamma \vdash c : t' \) and those of the form \( \Delta, x : [t]; \Gamma \vdash c_{[t]} : t' \) given by

\[ c_1 \mapsto \text{let } x = x \in c_1 \quad c_2 \mapsto c_{[t]}[x = \text{box } x] \]

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

**Definition 3.8** A \( \lambda^\mathcal{C} \)-model is given by a CCC \( \mathcal{D} \) and an FP-adjunction \( \mathcal{D} \xrightarrow{\mathcal{T}} \mathcal{C} \).

**Remark 3.9** The pattern for interpreting \( \lambda^\mathcal{C} \) is to interpret a type \( t \) by an object \( [t] \) of \( \mathcal{D} \), namely

\( [t] = N[t] \) and \( [t_1 \to t_2] = [t_2]^{[t_1]} \)

and a term \( \{ x_j : t_j | j \in m \} \vdash c : t' \) by a map in \( \mathcal{D}(\prod_{j \in m} N[t_j], N[t']) \).
The assumption “N is full and faithful” ensures that N preserves the whole CCC structure (see Proposition 3.3), therefore one may safely confuse N^n[t_1 → t_2] with (N^n[t_2])^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 λ^2 and λ^Q interact. The basic idea is that a model for AIM is a λ^2-model where the category D has the structure of a λ^Q-model parameterized w.r.t. C. The precise formulation uses the simple indexed category of Definition 3.4.

**Definition 3.10** An AIM-model is given by a CCC D, an FP-adjunction D \[ \xrightarrow{G} C \], and a C-indexed FP-adjunction S \[ \xrightarrow{\mathbf{T}} \mathbf{S} \].

**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 \rightarrow 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 λ^Q, i.e. a type t is interpreted by an object [t] of D, namely

\[ [[t]] = FG[t], \quad [[(t)]] \Rightarrow N[[t]] \quad \text{and} \quad [[t_1 \rightarrow t_2]] = [[t_2]]^{[[t_1]]} \]

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

**Proposition 3.12** In any AIM-model there are two canonical isomorphisms compile: GNA \rightarrow GA and down: PFX \rightarrow FX.

**Remark 3.13** These isomorphisms suggest an extension of AIM with up: [t] \rightarrow (|[t]|), i.e. cross-stage persistence for close code types, and compile: |[t]| \rightarrow [t].

### 3.1 Examples

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

**Example 3.14** Let \( \mathbb{N} \) be the set of naturals. Given a CCC C with \( \mathbb{N} \)-indexed products, take

- **D \triangleq C^{\mathbb{N}}**, hence an object \( A \in [D] \) is a sequence \( (A_n)_{n \in \mathbb{N}} \) and a map \( f \in D(A,B) \) is a sequence \( (f_n)_{n \in \mathbb{N}} \).
- **NA \triangleq 1: A**, where 1 is the terminal object of C, while \( PA \triangleq (A_n \mid n \in \mathbb{N}) \).
- **FX \triangleq (X \mid n \in \mathbb{N})**, i.e. the sequence which is constantly X, while \( GA \triangleq \prod_{n \in \mathbb{N}} A_n \).

Example 3.14 does not support cross-stage persistence. Therefore, it is suitable for interpreting λ^Q, but not MetaML or AIM (as defined in [12, 11]).

**Example 3.15** Let \( \omega^{op} \) be the category of natural numbers with the reverse order, i.e.

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

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

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

\[
\begin{array}{ccccccc}
A_0 & \xrightarrow{a_0} & A_1 & \ldots & A_n & \xrightarrow{a_n} & A_{n+} & \ldots \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & \\
B_0 & \xrightarrow{b_0} & B_1 & \ldots & B_n & \xrightarrow{b_n} & B_{n+} & \ldots
\end{array}
\]

while an object of D is a sequence of maps in C.

- **NA \triangleq !_{A_0} : A**, where \( !_{A_0} \) is the map \( 1 \leftarrow A_0 \) in C, while \( PA \triangleq (A_n \mid n \in \mathbb{N}) \).
- **FX \triangleq (id:X \leftarrow X \mid n \in \mathbb{N})**, i.e. the sequence which is constantly idX, while \( GA \triangleq \lim_{n \in \omega^{op}} A_n \).

In this model we can define the natural transformation up: A \rightarrow NA modeling cross-stage persistence, namely \( up_0 \triangleq !: A_0 \leftarrow 1 \) and \( up_{n+} \triangleq a_n : A_{n+} \leftarrow A_n \).

Note that exponentials in D are not defined pointwise. However, existence of exponentials and finite limits in C ensures that 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

\[
[[t]] = B[t], \quad [[(t)]] = N[[t]] \quad \text{and} \quad [[t_1 \rightarrow t_2]] = [[t_2]]^{[[t_1]]}
\]
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 \( \lambda \)-calculus in a CCC.

- \( c_n : 1 \to N^n A \) where \( c : 1 \to A \) is a global element of \( A \) (e.g. the interpretation of a constant). Since \( N \) preserves finite products, we define \( c_n \triangleq N^n c \).

- \( \lambda_n : (N^n B)^{N^n A} \to N^n B^A \). Since \( N \) preserves the CCC structure, \( \lambda_n \) is the iso \( (N^n B)^{N^n A} \to N^n B^A \).

- \( \oplus_n : N^n B^A \times N^n A \to N^n B \). \( \oplus_n \) is essentially an instance of evaluation \( \text{eval} : (N^n B)^{N^n A} \times N^n A \to N^n B \).

- \( \text{unbox}_n : N^n B A \to N^n A \). Since \( B \) is a comonad with \( co\text{-unit} \) \( c : B A \to A \) and \( co\text{-multiplication} \( \delta : B A \to B^2 A \), then \( \text{unbox}_n \triangleq N^n c \).

- \( \text{box}_n(f) : \prod_i N^n B A_i \to N^n B B \) when \( f : \prod_i B A_i \to B \). Since all functors preserve finite products, it suffices to say that \( \text{box}_n(f) \triangleq N^n (\delta f) ; N^n B A \to N^n B B \) when \( f : B A \to B \) and \( A \triangleq \prod_i A_i \).

- \( \text{run}_n(f) : C \times \prod_i N^n B A_i \to N^n B \) when \( f : C \times \prod_i N^n B A_i \to N^n B \). As in case of \( \text{box}_n(f) \) it suffices to give \( \text{run}_n(f) \equiv C \times \prod_i N^n B A_i \to N^n B \) when \( f : C \times \prod_i N^n B A_i \to N^n B \) and \( A \triangleq \prod_i A_i \).

By the canonical iso down (see Proposition 3.12) we have \( C \times N^n B A \cong C \times N^n P B A \). We have an FP-monad \( I_n \triangleq N^n P^n \) on \( \mathcal{D} \) with unit \( \eta^0_n : A \to I_n A \) induced by the FP-adjunction \( P^n \dashv N^n \). Moreover, we have an iso \( P^n A \to A \) given by the co-unit of the adjunction \( P \dashv N \), since \( N \) is full and faithful. Therefore, modulo some canonical iso \( \text{run}_n(f) \) is

\[
C \times N^n P B A \overset{\eta^0_n}{\longrightarrow} I_n C \times N^n P B A \overset{I_n P f}{\longrightarrow} N^n B
\]

Figure 6 defines the interpretation of a well-formed term \( \Gamma \vdash c : t^n \) by induction on the typing derivation in the type system of Figure 4.

5 Modalities and monads

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

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

The idea is that \( M \) models computation at level 0. We extend the \( \text{AIM}\)-models of Examples 3.14 and 3.15 to monadic \( \text{AIM}\)-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 \( \mathcal{C}^{\times \omega} \), namely \( MA \) is given by

\[
MA_0 \leftarrow MA_1 \ldots MA_n \leftarrow MA_{n+} \ldots
\]

The additional requirement holds, provided the monad \( M \) over \( \mathcal{C} \) preserves pullbacks and the commuting square \( M^! \) is a pullback, where

\[
M1 \overset{k}{\longrightarrow} (M!)^A
\]

\( \epsilon(u) \triangleq \lambda x : A. d \circ \{ f \leftarrow u; \text{ret}(fx) \} \) and \( k(u) \triangleq \lambda x : A. u \).

**Remark 5.4** The interaction of \( M \) with pullbacks is important, because exponentials in \( \mathcal{C}^{\times \omega} \) 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:

\[
[[t]] = B.M[[t]], [[t]] = N.M[[t]], [[t_1 \to t_2]] = (M[[t_2]])^{[t_1]}
\]

We introduce the shorthand \( N_t \) for \( MN \) and \( M_n \) for \( (MN)^n \). We call \( M_n A \) the **type of \( n \)-stage computations** returning (at stage \( n \)) a value of type \( A \).

In a monadic \( \text{AIM}\)-model a term \( \{x_i : t_i^n \mid i \in m \} : c : t^n \) is interpreted by a map in \( \mathcal{D}(\prod_{i \in m} N^n t_i, M_n [t]) \).
Remark 5.5 This interpretation is a refinement of the interpretation given in Section 4, which is recovered by replacing $\mathcal{M}$ with the identity monad, and it extends the CBV interpretation of the simply typed λ-calculus (in a CCC with a strong monad). $\mathcal{M}_n$ 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 $\text{run}_n(f)$, which we have been unable to define in general, but will be given for specific models. (We use notation introduced in Notation 1.1.)

- $\eta_n: N^n A \to N^n A$ is given by induction:
  
  \[
  \begin{align*}
  0 & \quad \xymatrix{A \ar[r]^-{\text{id}} & A} \\
  n+ & \quad \xymatrix{N^n A \ar[r]^-{\eta} & M N^n A \ar[r]^-{MN \eta_n} & N^n A}
  \end{align*}
  \]
  where $\eta: A \to MA$ is the unit of the monad $\mathcal{M}$.

- $\psi_n: \prod_i N^n A_i \to N^n \prod_i A_i$ is given by induction:
  
  \[
  \begin{align*}
  0 & \quad \xymatrix{\prod_i A_i \ar[r]^-{\text{id}} & \prod_i A_i} \\
  n+ & \quad \xymatrix{\prod_i N^n A_i \ar[r]^-{\psi} & N^n \prod_i A_i \ar[r]^-{N^n \psi_n} & N^n \prod_i A}
  \end{align*}
  \]
  where $\psi: \prod_i A_i \to M(\prod_i A_i)$ is given by $\psi(u_i|i) \triangleq \text{do}\{x_i \leftarrow u_i; \text{ret}\ (x_i|i)\}$, and we exploit preservation of finite products by $\eta$.

- $c_n \triangleq \begin{cases} N^n \text{c} & N^n MA \eta_n \triangleq N^n MA \equiv M_n A, \quad \text{where } c: 1 \to MA \text{ is a global element of } MA. \\
\end{cases}$

- $\text{var}_n \triangleq N^n A \xrightarrow{\eta} N^n MA \xrightarrow{\eta} N^n MA \equiv M_n A.$

- $\lambda_n: (\mathcal{M}_n B)^{N^n A} \to \mathcal{M}_n (MB)^A$ is given by induction:
  
  \[
  \begin{align*}
  0 & \quad \xymatrix{(MB)^A \ar[r]^-{\eta} & M (MB)^A} \\
  n+ & \quad \xymatrix{(\mathcal{M}_n B)^{N^n A} \ar[r]^-{\lambda_n} & N^A (\mathcal{M}_n B)^{N^n} A \ar[r]^-{N^A \lambda_n} & M_{n+} (MB)^A}
  \end{align*}
  \]

- $\otimes_n: \mathcal{M}_n (MB)^A \times M_n A \to M_n B$ is given by $(N^n (\text{eval}) \circ \psi_n) \circ (\text{eval}): (MB)^A \times A \to MB$ is an instance of evaluation.

- $\text{unbox}_n: \mathcal{M}_n BMA \to M_n A$ is given by $N^n (\text{eval})$, where $c:BMA \to MA$ is an instance of the co-unit for $\mathcal{B}$.

- $\text{box}_n(f): \prod_i \mathcal{M}_n B A_i \to \mathcal{M}_n B MB$ is given by $N^n ((\mathcal{B}) \circ \delta) \circ \psi_n$, where $f: \prod_i BMA_i \to MB$, $\delta$ is an instance of the co-multiplication for $\mathcal{B}$, and we exploit preservation of finite products by $\mathcal{B}$.

The interpretation of terms. Figure 7 defines the interpretation of a well-formed term $\Gamma \vdash e: \mathcal{T}^n$ 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 $\text{AIM}$-models of Example 5.2 and 5.3. To interpret run-with we need an auxiliary morphism

- $\text{run}_n(f): C \times \prod_i \mathcal{M}_n B A_i \to \mathcal{M}_n B$ for any $f: NC \times \prod_i N^n B A_i \to M_{n+B}$.

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

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

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

Let $g \triangleq \text{run}_n(f): N^n C \times M_n BMA \to M_n B$, we define its $m$th component $g_m$ (a map in $\mathcal{C}$) by case-analysis:

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

2. $n$: $g_m(x:C_b,v:MX) \triangleq \text{do}\{y \leftarrow v; f_n(*,y); f_{n+}(x,y)\}$

   where $X \triangleq (\prod_i M A_i) \setminus f: 1 \times X \to M1$ and $f_{n+}: C_b \times X \to MB_b$

3. $> n$: $g_m(x:C_k,v:MX) \triangleq \text{do}\{y \leftarrow v; f_{n+}(x,y)\}$, 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 $\text{run}_n(f)$ by applying the functor $N^n P_{\mathcal{T}^+}$ to $f$. In $\mathcal{C}^n$ this functor replaces the $m$th component $f_m$ with $1$, 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 $g = \text{run}_n(f)$ does not lose 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. $\bot: 1 \to A_\bot$) 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 $\text{AIM}$-model based on $\mathcal{C}^\omega$ we define $\text{run}_n(f)$ without imposing any restriction on $C$. In this model we have

\[
(M_n A)_m = \begin{cases}
M^{m+1} & \text{when } m < n \\
M^n + A_{m-n} & \text{when } m = n
\end{cases}
\]
Let $X \triangleq C.M.A$, then $f: NC \times N^nFX \to M_nB$ and we have to define $\text{run}_n(f): C \times M_nFX \to M_nB$:

- first we define $F:C \to P M_n(N, M B)^FX$ as
  $$P(NC \xrightarrow{\lambda f} (M_n + B)^{nFX} \xrightarrow{\lambda f} M_n(N, M B)^{FX})$$

- then we define $R: PM_n(N, M B)^FX \to M_n(M B)^FX$, namely its $n$th component $R_m$, by case-analysis:
  
  \begin{align*}
  & < n \Rightarrow R_m \triangleq M^{m+n} : M^{m+n+1} \to M^{m+1} \\
  & = n \Rightarrow R_m \triangleq M^n! : M^n(M1)^X \to M^n1 \\
  & \geq n \Rightarrow R_m \triangleq M^{n+m}X : M^{n+m}(M^2B_k)^X \to M^{n+m}(MB_k)^X, \\
  \end{align*}

  where $k = m - n$

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

- finally we define $\text{run}_n(f): C \times M_nFX \to M_nB$ as
  $$C \times M_nFX \xrightarrow{R \circ F \circ \text{id}} M_n(M B)^FX \times M_nFX \xrightarrow{\text{run}_m} M_nB$$

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

**Monadic interpretation of compile.** In any $\text{AIM}$-model there is an iso $\text{compile}: \text{BNA} \to \text{BA}$ (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 $\text{AIM}$-models described above there is a morphism $\text{compile}: \text{BMA} \to M \text{BMA}$ suitable for interpreting $\text{compile}:[\langle [t] \rangle] \to [t]$ with the following operational semantics:

\[
\text{compile} \ e \bowtie \text{box} \ e' \quad \text{if } e \xRightarrow{\omega} (e')
\]

We define $\text{compile} e'$ in $\mathcal{CN}$ (in the other model one must assume that $M$ over $\mathcal{C}$ preserves $\omega$-$\omega$-limits). First, note that $\text{(BMA)}_m = X \triangleq M A_0 \times \prod_{i=0}^{m+1} A_n$ and $\text{(BMA)}_m = M1 \times X$. It is now easy to define the $m$th component $\text{compile} e'_m$ by case-analysis:

\[
\begin{align*}
0 \& \text{ compile} e'_0 (u: M 1, v: X) \triangleq \text{do} \{u; \text{ret} v\} \\
> 0 \& \text{ compile} e'_m (u: M 1, v: X) \triangleq v.
\end{align*}
\]

\[
\Delta; \Gamma \vdash \text{c : } t_e \quad \Delta; \Gamma \vdash x: t \text{ if } t = \Delta(x) \text{ or } \Gamma(x)
\]

\[
\begin{align*}
\Delta; \Gamma \vdash \alpha \cdot x: t_1 \to t_2 & \quad \Delta; \Gamma \vdash \text{c : } t \\
\Delta; \Gamma \vdash \alpha \cdot x: t_1 \to t_2 & \quad \Delta; \Gamma \vdash \text{c : } t_1 \\
\end{align*}
\]

\[
\Delta; \Gamma \vdash \text{box } c: [t]
\]

\[
\Delta; \Gamma \vdash c_1: t_1 \to t_2 \quad \Delta; \Gamma \vdash c_1: t_1 \to t_2
\]

\[
\Delta; \Gamma \vdash \text{let } x = c_1 \text{ in } c_2: t_2
\]

**Figure 1:** $\lambda^n$ Type System

\[
\begin{align*}
\Delta; \Gamma \vdash e_1: t_1^n & \quad \Delta; \Gamma \vdash x: t_1^n \text{ if } t_1^n = \Gamma(x) \\
\Gamma, x: t_1^n & \vdash e_1: t_1^n \\
\Gamma & \vdash \lambda x. x: (t_1 \to t_2)^n \\
\Gamma & \vdash e_1: (t_1 \to t_2)^n \\
\Gamma & \vdash e_2: t_1^n \\
\Gamma, r: (t_1^n) & \vdash e: (t_1^n) \\
\Gamma + \vdash e: t_1^n \\
\Gamma & \vdash \text{run } c: t_1
\end{align*}
\]

**Figure 2:** $\lambda^n$ Type System (+ Figure 2)

**Figure 3:** MetaML Type System (+ Figure 2)

\[
\begin{align*}
\Gamma & \vdash \lambda x: t_1^n \vdash e: (t_1^n) \\
\Gamma & \vdash \text{run } e \text{ with } x_1 := c_2: t_1^n \\
\Gamma & \vdash e_1: (t_1^n) \\
\Gamma & \vdash e: [t_1^n] \\
\Gamma & \vdash \text{unbox } c: t_1
\end{align*}
\]

**Figure 4:** $\text{AIM}$ Type System (+ Figure 2)
Figure 5: Big-Step Operational Semantics

\[
\begin{align*}
\Gamma &\vdash c : t^0 \quad \overset{\Delta}{=} \quad \sigma_n : C \rightarrow N^n [t] \\
\Gamma, x : t^0 + c : t^0 &\vdash f : C \times N^0 A \rightarrow N^0 B \\
\Gamma &\vdash \lambda x. e : t \rightarrow t^0 \quad \overset{\Delta}{=} \quad \lambda x \circ (\Lambda f) : C \rightarrow N^n (B^A) \\
\Gamma &\vdash e : t^0 + \quad \overset{\Delta}{=} \quad f : C \rightarrow N^0 + A \\
\Gamma &\vdash \langle e \rangle : t^0 \quad \overset{\Delta}{=} \quad f : C \rightarrow N^n (NA) \\
\Gamma &\vdash x_i : [t_i]^n \quad \overset{\Delta}{=} \quad f : C \rightarrow N^n (BA_i) \\
\Gamma &\vdash \langle x_i : [t_i]^n \rangle \alpha_i \quad \overset{\Delta}{=} \quad \Gamma \vdash \langle t_i \rangle \quad \overset{\Delta}{=} \quad \alpha_n \circ \langle f_1, f_2 \rangle : C \rightarrow N^n B \\
\Gamma &\vdash \text{run } e \quad \overset{\Delta}{=} \quad \text{run}_n (f) \circ \langle x \rangle : C \rightarrow N^n A
\end{align*}
\]

where \( C \overset{\Delta}{=} \Gamma \), \( A \overset{\Delta}{=} [t] \), \( B \overset{\Delta}{=} [t'] \) and \( A_i \overset{\Delta}{=} [t_i] \).

Figure 6: Pure Interpretation in AIM-Models

\[
\begin{align*}
\Gamma &\vdash c : t^0 \quad \overset{\Delta}{=} \quad [c]_n : C \rightarrow M^n I_c \\
\Gamma, x : t^0 + c : t^0 &\vdash \text{var}_n \circ \sigma_n : C \rightarrow M^n A \text{ if } t^0 = \Gamma(x) \\
\Gamma &\vdash \lambda x. e : t \rightarrow t^0 \quad \overset{\Delta}{=} \quad \lambda x \circ (\Lambda f) : C \rightarrow M^n (BMA) \\
\Gamma &\vdash e : t^0 + \quad \overset{\Delta}{=} \quad f : C \rightarrow M^n + A \\
\Gamma &\vdash \langle e \rangle : t^0 \quad \overset{\Delta}{=} \quad f : C \rightarrow M^n (NA) \\
\Gamma &\vdash x_i : [t_i]^n \quad \overset{\Delta}{=} \quad f_i : C \rightarrow M^n (BA_i) \\
\Gamma &\vdash \langle x_i : [t_i]^n \rangle \quad \overset{\Delta}{=} \quad \prod_i M^n (BA_i) \\
\Gamma &\vdash \text{run } e \quad \overset{\Delta}{=} \quad \text{run}_n (f) \circ \langle x \rangle : C \rightarrow M^n A
\end{align*}
\]

where \( C \overset{\Delta}{=} \Gamma \), \( A \overset{\Delta}{=} [t] \), \( B \overset{\Delta}{=} [t'] \) and \( A_i \overset{\Delta}{=} [t_i] \).

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


