Notions of computation and monads

Eugenio Moggi*

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages, since programs can be *identified* with λ-terms. However, if one goes further and uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification is introduced (programs are identified with total functions from *values* to *values*), that may jeopardise the applicability of theoretical results. In this paper we introduce calculi based on a categorical semantics for *computations*, that provide a correct basis for proving equivalence of programs, for a wide range of *notions of computation*.

Introduction

This paper is about logics for reasoning about programs, in particular for proving equivalence of programs. Following a consolidated tradition in theoretical computer science we identify programs with the closed λ-terms, possibly containing extra constants, corresponding to some features of the programming language under consideration. There are three semantic-based approaches to proving equivalence of programs:

- The **operational** approach starts from an **operational semantics**, e.g. a partial function mapping every program (i.e. closed term) to its resulting value (if any), which induces a congruence relation on open terms called **operational equivalence** (see e.g. [Plo75]). Then the problem is to prove that two terms are operationally equivalent.
- The **denotational** approach gives an interpretation of the (programming) language in a mathematical structure, the **intended model**. Then the problem is to prove that two terms denote the same object in the intended model.
- The **logical** approach gives a class of **possible models** for the (programming) language. Then the problem is to prove that two terms denotes the same object in all possible models.

The operational and denotational approaches give only a theory: the operational equivalence $\approx$ or the set Th of formulas valid in the intended model respectively. On the other hand, the logical approach gives a consequence relation $\vdash$, namely $Ax \vdash A$ iff the formula A is true in all models of the set of formulas Ax, which can deal with different programming languages (e.g. functional, imperative, non-deterministic) in a rather *uniform* way, by simply changing the set of axioms Ax, and possibly extending the language with new constants. Moreover, the relation $\vdash$ is often semidecidable, so it is possible to give a sound and complete formal system for it, while Th and $\approx$ are semidecidable only in oversimplified cases.

We do not take as a starting point for proving equivalence of programs the theory of $\beta\eta$-conversion, which identifies the denotation of a program (procedure) of type $A \to B$ with a total function from A to B, since this identification wipes out completely behaviours like non-termination, non-determinism or side-effects, that can be exhibited by real programs. Instead, we proceed as follows:

1. We take category theory as a general theory of functions and develop on top a **categorical semantics of computations** based on monads.

*Research partially supported by EEC Joint Collaboration Contract # ST2J-0374-C(EDB).

2. We consider simple formal systems matching the categorical semantics of computation.

3. We extend stepwise categorical semantics and formal system in order to interpret richer languages, in particular the λ-calculus.

4. We show that w.l.o.g. one may consider only (monads over) toposes, and we exploit this fact to establish conservative extension results.

The methodology outlined above is inspired by [Sco80][1], and it is followed in [Ros86, Mog86] to obtain the λ_{p}-calculus. The view that "category theory comes, logically, before the λ-calculus" led us to consider a categorical semantics of computations first, rather than to modify directly the rules of $\beta\eta$-conversion to get a *correct* calculus.

Related work

The operational approach to find *correct* λ-calculi w.r.t. an operational equivalence, was first considered in [Plo75] for call-by-value and call-by-name operational equivalence. This approach was later extended, following a similar methodology, to consider other features of computations like nondeterminism (see [Sha84]), side-effects and continuations (see [FFKD86, FF89]). The calculi based only on operational considerations, like the λ_{v}-calculus, are sound and complete w.r.t. the operational semantics, i.e. a program M has a value according to the operational semantics iff it is provably equivalent to a value (not necessarily the same) in the calculus, but they are too weak for proving equivalences of programs.

Previous work on axiom systems for proving equivalence of programs with side effects has shown the importance of the *let*-constructor (see [Mas88, MT89a, MT89b]). In the framework of the computational lambda-calculus the importance of *let* becomes even more apparent.

The denotational approach may suggest important principles, e.g. fix-point induction (see [Sco69, GMW79]), that can be found only after developing a semantics based on mathematical structures rather than term models, but it does not give clear criteria to single out the general principles among the properties satisfied by the model. Moreover, the theory at the heart of Denotational Semantics, i.e. Domain Theory (see [GS89, Mos89]), has focused on the mathematical structures for giving semantics to recursive definitions of types and functions (see [SP82]), while other structures, that might be relevant to a better understanding of programming languages, have been overlooked. This paper identify one of such structures, i.e. *monads*, but probably there are others just waiting to be discovered.

The categorical semantic of computations presented in this paper has been strongly influenced by the reformulation of Denotational Semantics based on the category of cpos, possibly without bottom, and partial continuous functions (see [Plo85]) and the work on categories of partial morphisms in [Ros86, Mog86]. Our work generalises the categorical account of partiality to other notions of computations, indeed *partial cartesian closed categories* turn out to be a special case of λ_{c}-*models* (see Definition 3.9).

A type theoretic approach to partial functions and computations is proposed in [CS87, CS88] by introducing a type-constructor $\bar{A}$, whose intuitive meaning is the set of *computations* of type A. Our categorical semantics is based on a similar idea. Constable and Smith, however, do not adequately capture the general axioms for computations (as we do), since their notion of model, based on an untyped partial applicative structure, accounts only for partial computations.

1 A categorical semantics of computations

The basic idea behind the categorical semantics below is that, in order to interpret a programming language in a category $\mathcal{C}$, we distinguish the object A of values (of type A) from the object TA of

[1] "I am trying to find out where λ-calculus *should* come from, and the fact that the notion of a cartesian closed category is a late developing one (Eilenberg & Kelly (1966)), is not relevant to the argument: I shall try to explain in my own words in the next section why we should look to it *first*".

computations (of type A), and take as denotations of programs (of type A) the *elements* of TA. In particular, we identify the type A with the object of values (of type A) and obtain the object of computations (of type A) by applying an unary type-constructor T to A. We call T a *notion of computation*, since it abstracts away from the type of values computations may produce. There are many choices for TA corresponding to different notions of computations.

Example 1.1 We give few notions of computation in the category of sets.

- **partiality** $TA = A_\perp$ (i.e. $A + \{\perp\}$), where $\perp$ is the *diverging computation*
- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$
- **side-effects** $TA = (A \times S)^S$, where S is a set of states, e.g. a set U^L of stores or a set of input/output sequences U^*
- **exceptions** $TA = (A + E)$, where E is the set of exceptions
- **continuations** $TA = R^{(R^A)}$, where R is the set of results
- **interactive input** $TA = (\mu\gamma.A + \gamma^U)$, where U is the set of characters.
  More explicitly TA is the set of U-branching trees with finite branches and A-labelled leaves
- **interactive output** $TA = (\mu\gamma.A + (U \times \gamma))$.
  More explicitly TA is (isomorphic to) $U^* \times A$.

Further examples (in a category of cpos) could be given based on the denotational semantics for various programming languages (see [Sch86, GS89, Mos89]).

Rather than focusing on a specific T, we want to find the general properties common to all notions of computation, therefore we impose as only requirement that *programs* should form a category. The aim of this section is to convince the reader, with a sequence of informal argumentations, that such a requirement amounts to say that T is part of a Kleisli triple $(T, \eta, \_^*)$ and that the category of programs is the Kleisli category for such a triple.

Definition 1.2 ([Man76]) *A* **Kleisli triple** *over a category* $\mathcal{C}$ *is a triple* $(T, \eta, \_^*)$*, where* $T\colon \mathrm{Obj}(\mathcal{C}) \to \mathrm{Obj}(\mathcal{C})$*,* $\eta_A\colon A \to TA$ *for* $A \in \mathrm{Obj}(\mathcal{C})$*,* $f^*\colon TA \to TB$ *for* $f\colon A \to TB$ *and the following equations hold:*

- $\eta_A^* = \mathrm{id}_{TA}$
- $\eta_A; f^* = f$ *for* $f\colon A \to TB$
- $f^*; g^* = (f; g^*)^*$ *for* $f\colon A \to TB$ *and* $g\colon B \to TC$.

A Kleisli triple satisfies the **mono requirement** *provided* η_A *is mono for* $A \in \mathcal{C}$.

Intuitively η_A is the *inclusion* of values into computations (in several cases η_A is indeed a mono) and f^* is the *extension* of a function f from values to computations to a function from computations to computations, which first evaluates a computation and then applies f to the resulting value. In summary

$$a\colon A \overset{\eta_A}{\longmapsto} [a]\colon TA$$

$$\frac{a\colon A \overset{f}{\longmapsto} f(a)\colon TB}{c\colon TA \overset{f^*}{\longmapsto} (\mathrm{let}\; x{\Leftarrow}c \;\mathrm{in}\; f(x))\colon TB}$$

In order to justify the axioms for a Kleisli triple we have first to introduce a category $\mathcal{C}_T$ whose morphisms correspond to programs. We proceed by analogy with the categorical semantics for terms, where types are interpreted by objects and terms of type B with a parameter (free variable) of type A are interpreted by morphisms from A to B. Since the denotation of programs of type B are supposed to be elements of TB, programs of type B with a parameter of type A ought to be interpreted by morphisms with codomain TB, but for their domain there are two alternatives, either A or TA, depending on whether parameters of type A are identified with values or computations of type A. We choose the first alternative, because it entails the second. Indeed computations of type A are the same as values of type TA. So we take $\mathcal{C}_T(A,B)$ to be $\mathcal{C}(A,TB)$. It remains to define composition and identities in $\mathcal{C}_T$ (and show that they satisfy the unit and associativity axioms for categories).

Definition 1.3 *Given a Kleisli triple $(T,\eta,\_^*)$ over $\mathcal{C}$, the* **Kleisli category** *$\mathcal{C}_T$ is defined as follows:*

- *the objects of $\mathcal{C}_T$ are those of $\mathcal{C}$*
- *the set $\mathcal{C}_T(A,B)$ of morphisms from A to B in $\mathcal{C}_T$ is $\mathcal{C}(A,TB)$*
- *the identity on A in $\mathcal{C}_T$ is $\eta_A\colon A\to TA$*
- *$f\in\mathcal{C}_T(A,B)$ followed by $g\in\mathcal{C}_T(B,C)$ in $\mathcal{C}_T$ is $f;g^*\colon A\to TC$.*

It is natural to take η_A as the identity on A in the category $\mathcal{C}_T$, since it maps a parameter x to $[x]$, i.e. to x viewed as a computation. Similarly composition in $\mathcal{C}_T$ has a simple explanation in terms of the intuitive meaning of f^*, in fact

$$\frac{x\colon A \overset{f}{\longmapsto} f(x)\colon TB \qquad y\colon B \overset{g}{\longmapsto} g(y)\colon TC}{x\colon A \overset{f;g^*}{\longmapsto} (\mathrm{let}\; y{\Leftarrow}f(x)\;\mathrm{in}\; g(y))\colon TC}$$

i.e. f followed by g in $\mathcal{C}_T$ with parameter x is the program which first evaluates the program $f(x)$ and then feed the resulting value as parameter to g. At this point we can give also a simple justification for the three axioms of Kleisli triples, namely they are equivalent to the unit and associativity axioms for $\mathcal{C}_T$:

- $f;\eta_B^* = f$ for $f\colon A\to TB$
- $\eta_A; f^* = f$ for $f\colon A\to TB$
- $(f;g^*);h^* = f;(g;h^*)^*$ for $f\colon A\to TB$, $g\colon B\to TC$ and $h\colon C\to TD$.

Example 1.4 We go through the notions of computation given in Example 1.1 and show that they are indeed part of suitable Kleisli triples.

- **partiality** $TA = A_\perp (= A+\{\perp\})$
  η_A is the inclusion of A into $A_\perp$
  if $f\colon A\to TB$, then $f^*(\perp)=\perp$ and $f^*(a)=f(a)$ (when $a\in A$)
- **nondeterminism** $TA=\mathcal{P}_{fin}(A)$
  η_A is the singleton map $a\mapsto\{a\}$
  if $f\colon A\to TB$ and $c\in TA$, then $f^*(c)=\cup_{x\in c}f(x)$
- **side-effects** $TA=(A\times S)^S$
  η_A is the map $a\mapsto(\lambda s\colon S.\langle a,s\rangle)$
  if $f\colon A\to TB$ and $c\in TA$, then $f^*(c)=\lambda s\colon S.(\mathrm{let}\;\langle a,s'\rangle=c(s)\;\mathrm{in}\;f(a)(s'))$
- **exceptions** $TA=(A+E)$
  η_A is the injection map $a\mapsto\mathrm{inl}(a)$
  if $f\colon A\to TB$, then $f^*(\mathrm{inr}(e))=e$ (when $e\in E$) and $f^*(\mathrm{inl}(a))=f(a)$ (when $a\in A$)
- **continuations** $TA=R^{(R^A)}$
  η_A is the map $a\mapsto(\lambda k\colon R^A.k(a))$
  if $f\colon A\to TB$ and $c\in TA$, then $f^*(c)=(\lambda k\colon R^B.c(\lambda a\colon A.f(a)(k)))$

- **interactive input** $TA = (\mu\gamma.A + \gamma^U)$
  η_A maps a to the tree consisting only of one leaf labelled with a
  if $f: A \to TB$ and $c \in TA$, then $f^*(c)$ is the tree obtained by replacing leaves of c labelled by a with the tree $f(a)$

- **interactive output** $TA = (\mu\gamma.A + (U \times \gamma))$
  η_A is the map $a \mapsto \langle \epsilon, a \rangle$
  if $f: A \to TB$, then $f^*(\langle s, a \rangle) = \langle s * s', b \rangle$, where $f(a) = \langle s', b \rangle$ and $s * s'$ is the concatenation of s followed by s'.

Kleisli triples are just an alternative description for monads. Although the formers are easy to justify from a computational perspective, the latters are more widely used in the literature on Category Theory and have the advantage of being defined only in terms of funtors and natural transformations, which make them more suitable for abstract manipulation.

Definition 1.5 ([Mac71]) *A* **monad** *over a category $\mathcal{C}$ is a triple (T, η, μ), where $T: \mathcal{C} \to \mathcal{C}$ is a functor, $\eta: \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu: T^2 \dot{\to} T$ are natural transformations and the following diagrams commute:*

$$\begin{tikzcd}
T^3A \arrow[r, "\mu_{TA}"] \arrow[d, "T\mu_A"'] & T^2A \arrow[d, "\mu_A"] \\
T^2A \arrow[r, "\mu_A"'] & TA
\end{tikzcd}
\qquad
\begin{tikzcd}
TA \arrow[r, "\eta_{TA}"] \arrow[dr, "\mathrm{id}_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA \arrow[l, "T\eta_A"'] \arrow[dl, "\mathrm{id}_{TA}"] \\
& TA &
\end{tikzcd}$$

Proposition 1.6 ([Man76]) *There is a one-one correspondence between Kleisli triples and monads.*

Proof Given a Kleisli triple $(T, \eta, \_^*)$, the corresponding monad is (T, η, μ), where T is the extension of the function T to an endofunctor by taking $T(f) = (f; \eta_B)^*$ for $f: A \to B$ and $\mu_A = \mathrm{id}^*_{TA}$. Conversely, given a monad (T, η, μ), the corresponding Kleisli triple is $(T, \eta, \_^*)$, where T is the restriction of the functor T to objects and $f^* = (Tf); \mu_B$ for $f: A \to TB$. ∎

Remark 1.7 In general the categorical semantics of partial maps, based on a category $\mathcal{C}$ equipped with a *dominion* $\mathcal{M}$ (see [Ros86]), cannot be reformulated in terms of a Kleisli triple over $\mathcal{C}$ satisfying some additional properties, unless $\mathcal{C}$ has *lifting*, i.e. the inclusion functor from $\mathcal{C}$ into the category of partial maps $\mathrm{P}(\mathcal{C}, \mathcal{M})$ has a right adjoint $\__\perp$ characterised by the natural isomorphism

$$\mathcal{C}(A, B_\perp) \stackrel{\cdot}{\cong} \mathrm{P}(\mathcal{C}, \mathcal{M})(A, B)$$

This mismatch disappears when considering partial cartesian closed categories.

2 Simple languages for monads

In this section we consider two formal systems motivated by different objectives: reasoning about programming languages and reasoning about programs in a fixed programming language. When reasoning about programming languages one has different monads (for simplicity we assume that they are over the same category), one for each programming language, and the main aim is to study how they relate to each other. So it is natural to base a formal system on a *metalanguage* for a category and treat monads as unary type-constructors. When reasoning about programs one has only one monad, because the programming language is fixed, and the main aim is to prove properties of programs. In this case the obvious choice for the term language is the *programming language* itself, which is more naturally interpreted in the Kleisli category.

Remark 2.1 We regard the metalanguage as more fundamental. In fact, its models are more general, as they don't have to satisfy the mono requirement, and the interpretation of programs (of some given programming language) can be defined simply by translation into (a suitable extension of) the metalanguage. It should be pointed out that the mono requirement cannot be axiomatised in the metalanguage, as we would need conditional equations $[x]_T = [y]_T \to x = y$, and that existence assertions cannot be translated into formulas of the metalanguage, as we would need existentially quantified formulas $(e\downarrow_\sigma)^\circ \equiv (\exists! x{:}\,\sigma. e^\circ = [x]_T)$[2].

In Section 2.3 we will explain once for all the correspondence between theories of a simple programming language and categories with a monad satisfying the mono requirement. For other programming languages we will give only their translation in a suitable extension of the metalanguage. In this way, issues like call-by-value versus call-by-name affect the translation, but not the metalanguage.

In Categorical Logic it is common practice to identify a *theory* $\mathcal{T}$ with a category $\mathcal{F}(\mathcal{T})$ with additional structure such that there is a one-one correspondence between *models* of $\mathcal{T}$ in a category $\mathcal{C}$ with additional structure and structure preserving functors from $\mathcal{F}(\mathcal{T})$ to $\mathcal{C}$ (see [KR77])[3]. This identification was originally proposed by Lawvere, who also showed that algebraic theories can be viewed as categories with finite products.

In Section 2.2 we give a class of theories that can be viewed as categories with a monad, so that any category with a monad is, up to *equivalence* (of categories with a monad), one of such theories. Such a reformulation in terms of theories is more suitable for formal manipulation and more appealing to those unfamiliar with Category Theory. However, there are other advantages in having an alternative presentation of monads. For instance, natural extensions of the syntax may suggest extensions of the categorical structure that may not be immediate to motivate and justify otherwise (we will exploit this in Section 3). In Section 2.3 we take a programming language perspective and establish a correspondence between theories (with equivalence and existence assertions) for a simple programming language and categories with a monad satisfying the mono requirement, i.e. η_A mono for every A.

As starting point we take *many sorted monadic equational logic*, because it is more primitive than many sorted equational logic, indeed monadic theories are equivalent to categories without any additional structure.

2.1 Many sorted monadic equational logic

The language and formal system of many sorted monadic equational logic are parametric in a signature, i.e. a set of base types A and unary function symbols $\mathrm{f}{:}\, \mathrm{A}_1 \to \mathrm{A}_2$. The language is made of types $\vdash \mathrm{A}$ type, terms $x{:}\,\mathrm{A}_1 \vdash e{:}\,\mathrm{A}_2$ and equations $x{:}\,\mathrm{A}_1 \vdash e_1 =_{\mathrm{A}_2} e_2$ defined by the following formation rules:

$$\mathrm{A}\quad \frac{}{\vdash \mathrm{A}\ \mathrm{type}}\ \mathrm{A}\ \text{base type}$$

$$\mathrm{var}\quad \frac{\vdash \mathrm{A}\ \mathrm{type}}{x{:}\,\mathrm{A} \vdash x{:}\,\mathrm{A}}$$

$$\mathrm{f}\quad \frac{x{:}\,\mathrm{A} \vdash e_1{:}\,\mathrm{A}_1}{x{:}\,\mathrm{A} \vdash \mathrm{f}(e_1){:}\,\mathrm{A}_2}\ \mathrm{f}{:}\,\mathrm{A}_1 \to \mathrm{A}_2$$

$$\mathrm{eq}\quad \frac{x{:}\,\mathrm{A}_1 \vdash e_1{:}\,\mathrm{A}_2 \qquad x{:}\,\mathrm{A}_1 \vdash e_2{:}\,\mathrm{A}_2}{x{:}\,\mathrm{A}_1 \vdash e_1 =_{\mathrm{A}_2} e_2}$$

[2] The uniqueness of x s.t. $e^\circ = [x]_T$ follows from the mono requirement.

[3] In [LS86] a stronger relation is sought between theories and categories with additional structure, namely an equivalence between the category of theories and translations and the category of small categories with additional structure and structure preserving functors. In the case of typed λ-calculus, for instance, such an equivalence between λ-theories and cartesian closed categories requires a modification in the definition of λ-theory, which allows not only equations between λ-terms but also equations between type expressions.

RULE	SYNTAX		SEMANTICS
A	$\dfrac{}{\vdash \mathrm{A}\ \mathrm{type}}$	$=$	$[\![\mathrm{A}]\!]$
var	$\vdash \mathrm{A}\ \mathrm{type}$	$=$	c
	$\overline{x{:}\,\mathrm{A} \vdash x{:}\,\mathrm{A}}$	$=$	id_c
$\mathrm{f}{:}\,\mathrm{A}_1 \to \mathrm{A}_2$	$x{:}\,\mathrm{A} \vdash e_1{:}\,\mathrm{A}_1$	$=$	g
	$\overline{x{:}\,\mathrm{A} \vdash \mathrm{f}(e_1){:}\,\mathrm{A}_2}$	$=$	$g; [\![\mathrm{f}]\!]$
eq	$x{:}\,\mathrm{A}_1 \vdash e_1{:}\,\mathrm{A}_2$	$=$	g_1
	$x{:}\,\mathrm{A}_1 \vdash e_2{:}\,\mathrm{A}_2$	$=$	g_2
	$\overline{x{:}\,\mathrm{A}_1 \vdash e_1 =_{\mathrm{A}_2} e_2}$	$\Longleftrightarrow$	$g_1 = g_2$

Table 1: Interpretation of Many Sorted Monadic Equational Language

Remark 2.2 Terms of (many sorted) monadic equational logic have exactly one free variable (the one declared in the context) which occurs exactly once, and equations are between terms with the same free variable.

An interpretation $[\![\_]\!]$ of the language in a category $\mathcal{C}$ is parametric in an interpretation of the symbols in the signature and is defined by induction on the derivation of well-formedness for (types,) terms and equations (see Table 1) according to the following general pattern:

- the interpretation $[\![\mathrm{A}]\!]$ of a base type A is an object of $\mathcal{C}$
- the interpretation $[\![\mathrm{f}]\!]$ of an unary function $\mathrm{f}{:}\,\mathrm{A}_1 \to \mathrm{A}_2$ is a morphism from $[\![\mathrm{A}_1]\!]$ to $[\![\mathrm{A}_2]\!]$ in $\mathcal{C}$; similarly for the interpretation of a term $x{:}\,\mathrm{A}_1 \vdash e{:}\,\mathrm{A}_2$
- the interpretation of an assertion $x{:}\,\mathrm{A} \vdash \phi$ (in this case just an equation) is either true or false.

Remark 2.3 The interpretation of equations is standard. However, if one want to consider more complex assertions, e.g. formulas of first order logic, then they should be interpreted by subobjects; in particular equality $\_ = \_{:}\,\mathrm{A}$ should be interpreted by the diagonal $\Delta_{[\![\mathrm{A}]\!]}$.

The formal consequence relation on the set of equations is generated by the inference rules for equivalences ((refl), (simm) and (trans)), congruence and substitutivity (see Table 2). This formal consequence relation is *sound and complete* w.r.t. interpretation of the language in categories, i.e. an equation is formally derivable from a set of equational axioms if and only if all the interpretations satisfying the axioms satisfy the equation. Soundness follows from the admissibility of the inference rules in any interpretation, while completeness follows from the fact that any theory $\mathcal{T}$ (i.e. a set of equations closed w.r.t. the inference rules) is the set of equations satisfied by *the canonical interpretation* in the category $\mathcal{F}(\mathcal{T})$, i.e. $\mathcal{T}$ viewed as a category.

Definition 2.4 *Given a monadic equational theory $\mathcal{T}$, the category $\mathcal{F}(\mathcal{T})$ is defined as follows:*

- *objects are (base) types* A,
- *morphisms from* A_1 *to* A_2 *are equivalence classes* $[x{:}\,\mathrm{A}_1 \vdash e{:}\,\mathrm{A}_2]_{\mathcal{T}}$ *of terms w.r.t. the equivalence relation induced by the theory $\mathcal{T}$, i.e.*

$$(x{:}\,\mathrm{A}_1 \vdash e_1{:}\,\mathrm{A}_2) \equiv (x{:}\,\mathrm{A}_1 \vdash e_2{:}\,\mathrm{A}_2) \Longleftrightarrow (x{:}\,\mathrm{A}_1 \vdash e_1 =_{\mathrm{A}_2} e_2) \in \mathcal{T}$$

$$\text{refl}\ \frac{x{:}\,\mathrm{A} \vdash e{:}\,\mathrm{A}_1}{x{:}\,\mathrm{A} \vdash e =_{\mathrm{A}_1} e}$$

$$\text{symm}\ \frac{x{:}\,\mathrm{A} \vdash e_1 =_{\mathrm{A}_1} e_2}{x{:}\,\mathrm{A} \vdash e_2 =_{\mathrm{A}_1} e_1}$$

$$\text{trans}\ \frac{x{:}\,\mathrm{A} \vdash e_1 =_{\mathrm{A}_1} e_2 \qquad x{:}\,\mathrm{A} \vdash e_2 =_{\mathrm{A}_1} e_3}{x{:}\,\mathrm{A} \vdash e_2 =_{\mathrm{A}_1} e_3}$$

$$\text{congr}\ \frac{x{:}\,\mathrm{A} \vdash e_1 =_{\mathrm{A}_1} e_2}{x{:}\,\mathrm{A} \vdash \mathrm{f}(e_1) =_{\mathrm{A}_2} \mathrm{f}(e_2)}\ \mathrm{f}{:}\,\mathrm{A}_1 \to \mathrm{A}_2$$

$$\text{subst}\ \frac{x{:}\,\mathrm{A} \vdash e{:}\,\mathrm{A}_1 \qquad x{:}\,\mathrm{A}_1 \vdash \phi}{x{:}\,\mathrm{A} \vdash [e/x]\phi}$$

Table 2: Inference Rules of Many Sorted Monadic Equational Logic

- *composition is substitution, i.e.*

$$[x{:}\,\mathrm{A}_1 \vdash e_1{:}\,\mathrm{A}_2]_{\mathcal{T}};[x{:}\,\mathrm{A}_2 \vdash e_2{:}\,\mathrm{A}_3]_{\mathcal{T}} = [x{:}\,\mathrm{A}_1 \vdash [e_1/x]e_2{:}\,\mathrm{A}_3]_{\mathcal{T}}$$

- *identity over* A *is* $[x{:}\,\mathrm{A} \vdash x{:}\,\mathrm{A}]_{\mathcal{T}}$.

There is also a correspondence in the opposite direction, namely every category $\mathcal{C}$ (with additional structure) can be viewed as a theory $\mathcal{T}_{\mathcal{C}}$ (i.e. the *theory* of $\mathcal{C}$ over the language for $\mathcal{C}$), so that $\mathcal{C}$ and $\mathcal{F}(\mathcal{T}_{\mathcal{C}})$ are equivalent as categories (with additional structure). Actually, in the case of monadic equational theories and categories, $\mathcal{C}$ and $\mathcal{F}(\mathcal{T}_{\mathcal{C}})$ are isomorphic.

In the sequel we consider other equational theories. They can be viewed as categories in the same way described above for monadic theories; moreover, these categories are equipped with additional structure, depending on the specific nature of the theories under consideration.

2.2 The Simple metalanguage

We extend many sorted monadic equational logic to match categories equipped with a monad (or equivalently a Kleisli triple). Although we consider only one monad, it is conceptually straightforward to have several monads at once.

The first step is to extend the language. This could be done in several ways without affecting the correspondence between theories and monads, we choose a presentation inspired by Kleisli triples, more specifically we introduce an unary type-constructor T and the two term-constructors, $[\_]$ and let, used informally in Section 1. The definition of signature is slightly modified, since the domain and codomain of an unary function symbol $\mathrm{f}{:}\,\tau_1 \to \tau_2$ can be any type, not just base types (the fact is that in many sorted monadic logic the only types are base types). An interpretation $[\![\_]\!]$ of the language in a category $\mathcal{C}$ with a Kleisli triple $(T, \eta, \_^*)$ is parametric in an interpretation of the symbols in the signature and is defined by induction on the derivation of well-formedness for types, terms and equations (see Table 3). Finally we add to many sorted monadic equational logic appropriate inference rules capturing axiomatically the properties of the new type- and term-constructors after interpretation (see Table 4).

Proposition 2.5 *Every theory $\mathcal{T}$ of the simple metalanguage, viewed as a category $\mathcal{F}(\mathcal{T})$, is equipped with a Kleisli triple $(T, \eta, \_^*)$:*

- $T(\tau) = T\tau$,
- $\eta_\tau = [x{:}\,\tau \vdash_{ml} [x]_T{:}\,T\tau]_{\mathcal{T}}$,
- $([x{:}\,\tau_1 \vdash_{ml} e{:}\,T\tau_2]_{\mathcal{T}})^* = [x'{:}\,T\tau_1 \vdash_{ml} (\mathrm{let}_T\, x{\Leftarrow}x' \,\mathrm{in}\, e){:}\,T\tau_2]_{\mathcal{T}}$.

RULE	SYNTAX	SEMANTICS
A	$\dfrac{}{\vdash_{ml} \mathrm{A}\ \mathrm{type}}$	$= [\![\mathrm{A}]\!]$
T	$\dfrac{\vdash_{ml} \tau\ \mathrm{type}}{\vdash_{ml} T\tau\ \mathrm{type}}$	$= c$ $= Tc$
var	$\dfrac{\vdash_{ml} \tau\ \mathrm{type}}{x\colon\tau \vdash_{ml} x\colon\tau}$	$= c$ $= \mathrm{id}_c$
$\mathrm{f}\colon \tau_1 \to \tau_2$	$\dfrac{x\colon\tau \vdash_{ml} e_1\colon\tau_1}{x\colon\tau \vdash_{ml} \mathrm{f}(e_1)\colon\tau_2}$	$= g$ $= g; [\![\mathrm{f}]\!]$
$[\_]_T$	$\dfrac{x\colon\tau \vdash_{ml} e\colon\tau'}{x\colon\tau \vdash_{ml} [e]_T\colon T\tau'}$	$= g$ $= g; \eta_{[\![\tau']\!]}$
let	$\dfrac{x\colon\tau \vdash_{ml} e_1\colon T\tau_1 \quad x_1\colon\tau_1 \vdash_{ml} e_2\colon T\tau_2}{x\colon\tau \vdash_{ml} (\mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, e_2)\colon T\tau_2}$	$= g_1$ $= g_2$ $= g_1; g_2^*$
eq	$\dfrac{x\colon\tau_1 \vdash_{ml} e_1\colon\tau_2 \quad x\colon\tau_1 \vdash_{ml} e_2\colon\tau_2}{x\colon\tau_1 \vdash_{ml} e_1 =_{\tau_2} e_2}$	$= g_1$ $= g_2$ $\iff g_1 = g_2$

Table 3: Interpretation of the Simple Metalanguage

$$[\_].\xi \quad \frac{x\colon\tau \vdash_{ml} e_1 =_{\tau_1} e_2}{x\colon\tau \vdash_{ml} [e_1]_T =_{T\tau_1} [e_2]_T}$$

$$\mathrm{let}.\xi \quad \frac{x\colon\tau \vdash_{ml} e_1 =_{T\tau_1} e_2 \qquad x'\colon\tau_1 \vdash_{ml} e_1' =_{T\tau_2} e_2'}{x\colon\tau \vdash_{ml} (\mathrm{let}_T\, x' \Leftarrow e_1\, \mathrm{in}\, e_1') =_{T\tau_2} (\mathrm{let}_T\, x' \Leftarrow e_2\, \mathrm{in}\, e_2')}$$

$$\mathrm{ass} \quad \frac{x\colon\tau \vdash_{ml} e_1\colon T\tau_1 \qquad x_1\colon\tau_1 \vdash_{ml} e_2\colon T\tau_2 \qquad x_2\colon\tau_2 \vdash_{ml} e_3\colon T\tau_3}{x\colon\tau \vdash_{ml} (\mathrm{let}_T\, x_2 \Leftarrow (\mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, e_2)\, \mathrm{in}\, e_3) =_{T\tau_3} (\mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, (\mathrm{let}_T\, x_2 \Leftarrow e_2\, \mathrm{in}\, e_3))}$$

$$\mathrm{T}.\beta \quad \frac{x\colon\tau \vdash_{ml} e_1\colon\tau_1 \qquad x_1\colon\tau_1 \vdash_{ml} e_2\colon T\tau_2}{x\colon\tau \vdash_{ml} (\mathrm{let}_T\, x_1 \Leftarrow [e_1]_T\, \mathrm{in}\, e_2) =_{T\tau_2} [e_1/x_1]e_2}$$

$$\mathrm{T}.\eta \quad \frac{x\colon\tau \vdash_{ml} e_1\colon T\tau_1}{x\colon\tau \vdash_{ml} (\mathrm{let}_T\, x_1 \Leftarrow e_1\, \mathrm{in}\, [x_1]_T) =_{T\tau_1} e_1}$$

Table 4: Inference Rules of the Simple Metalanguage

Proof We have to show that the three axioms for Kleisli triples are valid. The validity of each axiom amounts to the derivability of an equation. For instance, $\eta^*_\tau = \mathrm{id}_{T\tau}$ is valid provided $x' \colon T\tau \vdash_{ml} (\mathrm{let}_T\, x \Leftarrow x' \,\mathrm{in}\, [x]_T) =_{T\tau} x'$ is derivable, indeed it follows from $(T.\eta)$. The reader can check that the equations corresponding to the axioms $\eta_\tau; f^* = f$ and $f^*; g^* = (f; g^*)^*$ follow from $(T.\beta)$ and (ass) respectively. ∎

2.3 A Simple Programming Language

In this section we take a programming language perspective by introducing a simple programming language, whose terms are interpreted by morphisms of the Kleisli category for a monad. Unlike the metalanguage of Section 2.2, the programming language does not allow to consider more than one monad at once.

The interpretation in the Kleisli category can also be given indirectly via a translation in the simple metalanguage of Section 2.2 mapping *programs* of type τ into *terms* of type $T\tau$. If we try to establish a correspondence between *equational theories* of the simple programming language and categories with one monad (as done for the metalanguage), then we run into problems, since there is no way (in general) to recover $\mathcal{C}$ from $\mathcal{C}_T$. What we do instead is to establish a correspondence between theories with *equivalence* and *existence* assertions and categories with one monad satisfying the *mono requirement*, i.e. η_A is mono for every object A (note that η_{TA} is always a mono, because $\eta_{TA}; \mu_A = \mathrm{id}_{TA}$). The intended extension of the existence predicate on computations of type A is the set of computations of the form $[v]$ for some value v of type A, so it is natural to require η_A to be mono and interpret the existence predicate as the *subobject* corresponding to η_A.

The simple programming language is parametric in a signature, i.e. a set of base types and unary command symbols. To stress that the interpretation is in $\mathcal{C}_T$ rather than $\mathcal{C}$, we use unary command symbols $\mathrm{p}\colon \tau_1 \rightharpoonup \tau_2$ (instead of unary function symbols $\mathrm{f}\colon \tau_1 \to \tau_2$), we call $x\colon \tau_1 \vdash_{pl} e\colon \tau_2$ a program (instead of a term) and write $\_ \equiv_\tau \_$ (instead of $\_ =_{T\tau} \_$) as equality of computations of type τ. Given a category $\mathcal{C}$ with a Kleisli triple $(T, \eta, \_^*)$ satisfying the mono requirement, an interpretation $[\![\_]\!]$ of the programming language is parametric in an interpretation of the symbols in the signature and is defined by induction on the derivation of well-formedness for types, terms and equations (see Table 5) following the same pattern given for many sorted monadic equational logic, but with $\mathcal{C}$ replaced by $\mathcal{C}_T$, namely:

- the interpretation $[\![\tau]\!]$ of a (base) type τ is an object of $\mathcal{C}_T$, or equivalently an object of $\mathcal{C}$
- the interpretation $[\![\mathrm{p}]\!]$ of an unary command $\mathrm{p}\colon \tau_1 \rightharpoonup \tau_2$ is a morphism from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$ in $\mathcal{C}_T$, or equivalently a morphism from $[\![\tau_1]\!]$ to $T[\![\tau_2]\!]$ in $\mathcal{C}$; similarly for the interpretation of a program $x\colon \tau_1 \vdash_{pl} e\colon \tau_2$
- the interpretation of an equivalence or existence assertion is a truth value.

Remark 2.6 The let-constructor play a fundamental role: operationally it corresponds to sequential evaluation of programs and categorically it corresponds to composition in the Kleisli category $\mathcal{C}_T$ (while substitution corresponds to composition in $\mathcal{C}$). In the λ_v-calculus $(\mathrm{let}\; x \Leftarrow e \;\mathrm{in}\; e')$ is treated as syntactic sugar for $(\lambda x.e')e$. We think that this is not the right way to proceed, because it explains the let-constructor (i.e. sequential evaluation of programs) in terms of constructors available only in functional languages. On the other hand, $(\mathrm{let}\; x \Leftarrow e \;\mathrm{in}\; e')$ cannot be treated as syntactic sugar for $[e/x]e'$ (involving only the more primitive substitution) without collapsing computations to values.

The existence predicate $e\downarrow$ is inspired by the logic of partial terms/elements (see [Fou77, Sco79, Mog88]); however, there are important differences, e.g.

$$\text{strict}\ \frac{x\colon \tau \vdash_{pl} \mathrm{p}(e)\downarrow_{\tau_2}}{x\colon \tau \vdash_{pl} e\downarrow_{\tau_1}}\ \mathrm{p}\colon \tau_1 \rightharpoonup \tau_2$$

is admissible for partial computations, but not in general. For certain notions of computation there may be other predicates on computations worth considering, or the existence predicate itself may have a more specialised meaning, for instance:

RULE	SYNTAX		SEMANTICS
A	$\dfrac{}{\vdash_{pl} \mathrm{A}\ \mathrm{type}}$	$=$	$[\![\mathrm{A}]\!]$
T	$\dfrac{\vdash_{pl} \tau\ \mathrm{type}}{\vdash_{pl} T\tau\ \mathrm{type}}$	$\begin{array}{c} = \\ = \end{array}$	$\begin{array}{l} c \\ Tc \end{array}$
var	$\dfrac{\vdash_{pl} \tau\ \mathrm{type}}{x\colon\tau \vdash_{pl} x\colon\tau}$	$\begin{array}{c} = \\ = \end{array}$	$\begin{array}{l} c \\ \eta_c \end{array}$
$\mathrm{p}\colon\tau_1 \rightharpoonup \tau_2$	$\dfrac{x\colon\tau \vdash_{pl} e_1\colon\tau_1}{x\colon\tau \vdash_{pl} \mathrm{p}(e_1)\colon\tau_2}$	$\begin{array}{c} = \\ = \end{array}$	$\begin{array}{l} g \\ g;[\![\mathrm{p}]\!]^* \end{array}$
$[\_]$	$\dfrac{x\colon\tau \vdash_{pl} e\colon\tau'}{x\colon\tau \vdash_{pl} [e]\colon T\tau'}$	$\begin{array}{c} = \\ = \end{array}$	$\begin{array}{l} g \\ g;\eta_{T[\![\tau']\!]} \end{array}$
μ	$\dfrac{x\colon\tau \vdash_{pl} e\colon T\tau'}{x\colon\tau \vdash_{pl} \mu(e)\colon\tau'}$	$\begin{array}{c} = \\ = \end{array}$	$\begin{array}{l} g \\ g;\mu_{[\![\tau']\!]} \end{array}$
let	$\dfrac{\begin{array}{l} x\colon\tau \vdash_{pl} e_1\colon\tau_1 \\ x_1\colon\tau_1 \vdash_{pl} e_2\colon\tau_2 \end{array}}{x\colon\tau \vdash_{pl} (\mathrm{let}\, x_1 \Leftarrow e_1 \,\mathrm{in}\, e_2)\colon\tau_2}$	$\begin{array}{c} = \\ = \\ = \end{array}$	$\begin{array}{l} g_1 \\ g_2 \\ g_1;{g_2}^* \end{array}$
eq	$\dfrac{\begin{array}{l} x\colon\tau_1 \vdash_{pl} e_1\colon\tau_2 \\ x\colon\tau_1 \vdash_{pl} e_2\colon\tau_2 \end{array}}{x\colon\tau_1 \vdash_{pl} e_1 \equiv_{\tau_2} e_2}$	$\begin{array}{c} = \\ = \\ \iff \end{array}$	$\begin{array}{l} g_1 \\ g_2 \\ g_1 = g_2 \end{array}$
ex	$\dfrac{x\colon\tau_1 \vdash_{pl} e\colon\tau_2}{x\colon\tau_1 \vdash_{pl} e\downarrow_{\tau_2}}$	$\begin{array}{c} = \\ \iff \end{array}$	$\begin{array}{l} g \\ \exists! h\colon [\![\tau_1]\!] \to [\![\tau_2]\!] \text{ s.t. } g = h;\eta_{[\![\tau_2]\!]} \end{array}$

Table 5: Interpretation of the Simple Programming Language

$$\text{refl}\ \frac{x{:}\tau \vdash_{pl} e{:}\tau_1}{x{:}\tau \vdash_{pl} e \equiv_{\tau_1} e}$$

$$\text{symm}\ \frac{x{:}\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2}{x{:}\tau \vdash_{pl} e_2 \equiv_{\tau_1} e_1}$$

$$\text{trans}\ \frac{x{:}\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2 \qquad x{:}\tau \vdash_{pl} e_2 \equiv_{\tau_1} e_3}{x{:}\tau \vdash_{pl} e_2 \equiv_{\tau_1} e_3}$$

$$\text{congr}\ \frac{x{:}\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2}{x{:}\tau \vdash_{pl} \mathrm{p}(e_1) \equiv_{\tau_2} \mathrm{p}(e_2)}\ \mathrm{p}{:}\tau_1 \rightharpoonup \tau_2$$

$$\text{E.}x\ \frac{\vdash_{pl} \tau\ \text{type}}{x{:}\tau \vdash_{pl} x \downarrow_{\tau}}$$

$$\text{E.congr}\ \frac{x{:}\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2 \qquad x{:}\tau \vdash_{pl} e_1 \downarrow_{\tau_1}}{x{:}\tau \vdash_{pl} e_2 \downarrow_{\tau_1}}$$

$$\text{subst}\ \frac{x{:}\tau \vdash_{pl} e \downarrow_{\tau_1} \qquad x{:}\tau_1 \vdash_{pl} \phi}{x{:}\tau \vdash_{pl} [e/x]\phi}$$

Table 6: General Inference Rules

- a partial computation exists iff it terminates;
- a non-deterministic computation exists iff it gives exactly one result;
- a computation with side-effects exists iff it does not change the store.

Programs can be translated into terms of the metalanguage via a translation $\_^{\circ}$ s.t. for every well-formed program $x{:}\tau_1 \vdash_{pl} e{:}\tau_2$ the term $x{:}\tau_1 \vdash_{ml} e^{\circ}{:}T\tau_2$ is well-formed and $[\![x{:}\tau_1 \vdash_{pl} e{:}\tau_2]\!] = [\![x{:}\tau_1 \vdash_{ml} e^{\circ}{:}T\tau_2]\!]$ (the proof of these properties is left to the reader).

Definition 2.7 *Given a signature Σ for the programming language, let Σ° be the signature for the metalanguage with the same base types and a function $\mathrm{p}{:}\tau_1 \to T\tau_2$ for each command $\mathrm{p}{:}\tau_1 \rightharpoonup \tau_2$ in Σ. The translation $\_^{\circ}$ from programs over Σ to terms over Σ° is defined by induction on raw programs:*

- $x^{\circ} \stackrel{\Delta}{\equiv} [x]_T$
- $(\mathrm{let}\ x_1 \Leftarrow e_1\ \mathrm{in}\ e_2)^{\circ} \stackrel{\Delta}{\equiv} (\mathrm{let}_T\ x_1 \Leftarrow e_1{}^{\circ}\ \mathrm{in}\ e_2{}^{\circ})$
- $\mathrm{p}(e_1)^{\circ} \stackrel{\Delta}{\equiv} (\mathrm{let}_T\ x \Leftarrow e_1{}^{\circ}\ \mathrm{in}\ \mathrm{p}(x))$
- $[e]^{\circ} \stackrel{\Delta}{\equiv} [e^{\circ}]_T$
- $\mu(e)^{\circ} \stackrel{\Delta}{\equiv} (\mathrm{let}_T\ x \Leftarrow e^{\circ}\ \mathrm{in}\ x)$

The inference rules for deriving equivalence and existence assertions of the simple programming language can be partitioned as follows:

- general rules (see Table 6) for terms denoting computations, but with variables ranging over values; these rules replace those of Table 2 for many sorted monadic equational logic
- rules capturing the properties of type- and term-constructors (see Table 7) after interpretation of the programming language; these rules replace the additional rules for the metalanguage given in Table 4.

$$[\_].\xi \quad \frac{x{:}\,\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2}{x{:}\,\tau \vdash_{pl} [e_1] \equiv_{T\tau_1} [e_2]}$$

$$\mathrm{E}.[\_] \quad \frac{x{:}\,\tau \vdash_{pl} e_1 : \tau_1}{x{:}\,\tau \vdash_{pl} [e_1] \downarrow_{T\tau_1}}$$

$$\mu.\xi \quad \frac{x{:}\,\tau \vdash_{pl} e_1 \equiv_{T\tau_1} e_2}{x{:}\,\tau \vdash_{pl} \mu(e_1) \equiv_{\tau_1} \mu(e_2)}$$

$$\mu.\beta \quad \frac{x{:}\,\tau \vdash_{pl} e_1 : \tau_1}{x{:}\,\tau \vdash \mu([e_1]) \equiv_{\tau_1} e_1}$$

$$\mu.\eta \quad \frac{x{:}\,\tau \vdash_{pl} e_1 \downarrow_{T\tau_1}}{x{:}\,\tau \vdash [\mu(e_1)] \equiv_{T\tau_1} e_1}$$

$$\mathrm{let}.\xi \quad \frac{x{:}\,\tau \vdash_{pl} e_1 \equiv_{\tau_1} e_2 \qquad x'{:}\,\tau_1 \vdash_{pl} e_1' \equiv_{\tau_2} e_2'}{x{:}\,\tau \vdash_{pl} (\mathrm{let}\; x' \Leftarrow e_1 \;\mathrm{in}\; e_1') \equiv_{\tau_2} (\mathrm{let}\; x' \Leftarrow e_2 \;\mathrm{in}\; e_2')}$$

$$\mathrm{unit} \quad \frac{x{:}\,\tau \vdash_{pl} e_1 : \tau_1}{x{:}\,\tau \vdash_{pl} (\mathrm{let}\; x_1 \Leftarrow e_1 \;\mathrm{in}\; x_1) \equiv_{\tau_1} e_1}$$

$$\mathrm{ass} \quad \frac{x{:}\,\tau \vdash_{pl} e_1 : \tau_1 \qquad x_1 : \tau_1 \vdash_{pl} e_2 : \tau_2 \qquad x_2 : \tau_2 \vdash_{pl} e_3 : \tau_3}{x{:}\,\tau \vdash_{pl} (\mathrm{let}\; x_2 \Leftarrow (\mathrm{let}\; x_1 \Leftarrow e_1 \;\mathrm{in}\; e_2) \;\mathrm{in}\; e_3) \equiv_{\tau_3} (\mathrm{let}\; x_1 \Leftarrow e_1 \;\mathrm{in}\; (\mathrm{let}\; x_2 \Leftarrow e_2 \;\mathrm{in}\; e_3))}$$

$$\mathrm{let}.\beta \quad \frac{x{:}\,\tau \vdash_{pl} e_1 \downarrow_{\tau_1} \qquad x_1 : \tau_1 \vdash_{pl} e_2 : \tau_2}{x{:}\,\tau \vdash_{pl} (\mathrm{let}\; x_1 \Leftarrow e_1 \;\mathrm{in}\; e_2) \equiv_{\tau_2} [e_1/x_1]e_2}$$

$$\mathrm{let.p} \quad \frac{x{:}\,\tau \vdash_{pl} e_1 : \tau_1}{x{:}\,\tau \vdash_{pl} \mathrm{p}(e_1) \equiv_{\tau_1} (\mathrm{let}\; x_1 \Leftarrow e_1 \;\mathrm{in}\; \mathrm{p}(x_1))} \quad \mathrm{p}{:}\,\tau_1 \rightharpoonup \tau_2$$

Table 7: Inference Rules of the Simple Programming Language

Soundness and completeness of the formal consequence relation w.r.t. interpretation of the simple programming language in categories with a monad satisfying the mono requirement is established in the usual way (see Section 2.1). The only step which differs is how to view a theory $\mathcal{T}$ of the simple programming language (i.e. a set of equivalence and existence assertions closed w.r.t. the inference rules) as a category $\mathcal{F}(\mathcal{T})$ with the required structure.

Definition 2.8 *Given a theory $\mathcal{T}$ of the simple programming language, the category $\mathcal{F}(\mathcal{T})$ is defined as follows:*

- *objects are types τ,*
- *morphisms from τ_1 to τ_2 are equivalence classes $[x\colon\tau_1 \vdash_{pl} e\colon\tau_2]_{\mathcal{T}}$ of existing programs $x\colon\tau_1 \vdash_{pl} e\downarrow_{\tau_2} \in \mathcal{T}$ w.r.t. the equivalence relation induced by the theory $\mathcal{T}$, i.e.*

$$(x\colon\tau_1 \vdash_{pl} e_1\colon\tau_2) \equiv (x\colon\tau_1 \vdash_{pl} e_2\colon\tau_2) \iff (x\colon\tau_1 \vdash_{pl} e_1 \equiv_{\tau_2} e_2) \in \mathcal{T}$$

- *composition is substitution, i.e.*

$$[x\colon\tau_1 \vdash_{pl} e_1\colon\tau_2]_{\mathcal{T}};[x\colon\tau_2 \vdash_{pl} e_2\colon\tau_3]_{\mathcal{T}} = [x\colon\tau_1 \vdash_{pl} [e_1/x]e_2\colon\tau_3]_{\mathcal{T}}$$

- *identity over τ is $[x\colon\tau \vdash_{pl} x\colon\tau]_{\mathcal{T}}$.*

In order for composition in $\mathcal{F}(\mathcal{T})$ to be well-defined, it is essential to consider only equivalence classes of existing programs, since the simple programming language satisfies only a restricted form of substitutivity.

Proposition 2.9 *Every theory $\mathcal{T}$ of the simple programming language, viewed as a category $\mathcal{F}(\mathcal{T})$, is equipped with a Kleisli triple $(T, \eta, \_^*)$ satisfying the mono requirement:*

- $T(\tau) = T\tau$,
- $\eta_\tau = [x\colon\tau \vdash_{pl} [x]\colon T\tau]_{\mathcal{T}}$,
- $([x\colon\tau_1 \vdash_{pl} e\colon T\tau_2]_{\mathcal{T}})^* = [x'\colon T\tau_1 \vdash_{pl} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,\mu(e))]\colon T\tau_2]_{\mathcal{T}}$.

Proof We have to show that the three axioms for Kleisli triples are valid. The validity of each axiom amounts to the derivability of an existence and equivalence assertion. For instance, $\eta_\tau^* = \mathrm{id}_{T\tau}$ is valid provided $x'\colon T\tau \vdash_{pl} x' \downarrow_{T\tau}$ and $x'\colon T\tau \vdash_{pl} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,\mu([x]))] \equiv_{T\tau} x'$ are derivable. The existence assertion follows immediately from (E.x), while the equivalence is derived as follows:

- $x'\colon T\tau \vdash_{pl} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,\mu([x]))] \equiv_{T\tau} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,x)]$ by ($\mu.\beta$), (refl) and (let.ξ)
- $x'\colon T\tau \vdash_{pl} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,x)] \equiv_{T\tau} [\mu(x')]$ by (unit) and (let.ξ)
- $x'\colon T\tau \vdash_{pl} [\mu(x')] \equiv_{T\tau} x'$ by (E.x) and ($\mu.\eta$)
- $x'\colon T\tau \vdash_{pl} [(\mathrm{let}\, x{\Leftarrow}\mu(x')\,\mathrm{in}\,\mu([x]))] \equiv_{T\tau} x'$ by (trans).

We leave to the reader the derivation of the existence and equivalence assertions corresponding to the other axioms for Kleisli triples, and prove instead the mono requirement i.e. that $f_1;\eta_\tau = f_2;\eta_\tau$ implies $f_1 = f_2$. Let f_i be $[x\colon\tau' \vdash_{pl} e_i\colon\tau]_{\mathcal{T}}$, we have to derive $x\colon\tau' \vdash_{pl} e_1 \equiv_\tau e_2$ from $x\colon\tau' \vdash_{pl} [e_1] \equiv_{T\tau} [e_2]$ (and $x\colon\tau' \vdash_{pl} e_i \downarrow_\tau$) :

- $x\colon\tau' \vdash_{pl} \mu([e_1]) \equiv_\tau \mu([e_2])$ by the first assumption and ($\mu.\xi$)
- $x\colon\tau' \vdash_{pl} \mu([e_i]) \equiv_\tau e_i$ by ($\mu.\beta$)
- $x\colon\tau' \vdash_{pl} e_1 \equiv_\tau e_2$ by (trans).

∎

Remark 2.10 One can show that the canonical interpretation of a program $x{:}\,\tau_1 \vdash_{pl} e{:}\,\tau_2$ in the category $\mathcal{F}(\mathcal{T})$ is the morphism $[x{:}\,\tau_1 \vdash_{pl} [e]{:}\,T\tau_2]_{\mathcal{T}}$. This interpretation establishes a one-one correspondence between morphisms from τ_1 to $T\tau_2$ in the category $\mathcal{F}(\mathcal{T})$, i.e. morphisms from τ_1 to τ_2 in the Kleisli category, and equivalence classes of programs $x{:}\,\tau_1 \vdash_{pl} e{:}\,\tau_2$ (not necessarely existing). The inverse correspondence maps a morphism $[x{:}\,\tau_1 \vdash_{pl} e'{:}\,T\tau_2]_{\mathcal{T}}$ to the equivalence class of $x{:}\,\tau_1 \vdash_{pl} \mu(e'){:}\,\tau_2$. Indeed, $x{:}\,\tau_1 \vdash_{pl} e \equiv_{\tau_2} \mu([e])$ and $x{:}\,\tau_1 \vdash_{pl} e' \equiv_{\tau_2} [\mu(e')]$ are derivable provided $x{:}\,\tau_1 \vdash_{pl} e' \downarrow_{T\tau_2}$.

3 Extending the simple metalanguage

So far we have considered only languages and formal systems for *monadic terms* $x{:}\,\tau_1 \vdash e{:}\,\tau_2$, having exactly one free variable (occurring once). In this section we want to extend these languages (and formal systems) by allowing *algebraic terms* $x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n \vdash e{:}\,\tau$, having a finite number of free variables (occurring finitely many times) and investigate how this affects the interpretation and the structure on theories viewed as categories. For convenience in relating theories and categories with additional structure, we also allow types to be closed w.r.t. finite products[4], in particular a typing context $x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n$ can be identified with a type. In general, the interpretation of an algebraic term $x_1{:}\,\tau_1, \ldots, x_n{:}\,\tau_n \vdash e{:}\,\tau$ in a category (with finite products) is a morphism from $([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!])$ to $[\![\tau]\!]$.

The extension of monadic equational logic to algebraic terms is equational logic, whose theories correspond to categories with finite products. We will introduce the *metalanguage*, i.e. the extension of the simple metalanguage described in Section 2.2 to algebraic terms, and show that its theories correspond to categories with finite products and a *strong monad*, i.e. a monad and a natural transformation $\mathrm{t}_{A,B}{:}\, A\times TB \to T(A\times B)$. Intuitively $\mathrm{t}_{A,B}$ transforms a pair value-computation into a computation of a pair of values, as follows

$$a{:}\,A, c{:}\,TB \quad \overset{\mathrm{t}_{A,B}}{\longmapsto} \quad (\mathrm{let}\, y{\Leftarrow}c\, \mathrm{in}\, [\langle a, y\rangle]){:}\,T(A\times B)$$

Remark 3.1 To understand why a category with finite products and a monad is not enough to interpret the metalanguage (and where the natural transformation t is needed), one has to look at the interpretation of a let-expression

$$\text{let}\ \frac{\Gamma \vdash_{ml} e_1{:}\,T\tau_1 \qquad \Gamma, x{:}\,\tau_1 \vdash_{ml} e_2{:}\,T\tau_2}{\Gamma \vdash_{ml} (\mathrm{let}_T\, x{\Leftarrow}e_1\, \mathrm{in}\, e_2){:}\,T\tau_2}$$

where Γ is a typing context. Let $g_1{:}\,c \to Tc_1$ and $g_2{:}\,c\times c_1 \to Tc_2$ be the interpretations of $\Gamma \vdash_{ml} e_1{:}\,T\tau_1$ and $\Gamma, x{:}\,\tau_1 \vdash_{ml} e_2{:}\,T\tau_2$ respectively, where c is the interpretation of the typing context Γ and c_i is the interpretation of the type τ_i, then the interpretation of $\Gamma \vdash_{ml} (\mathrm{let}_T\, x{\Leftarrow}e_1\, \mathrm{in}\, e_2){:}\,T\tau_2$ ought to be a morphism $g{:}\,c \to Tc_2$. If (T, η, μ) is the *identity monad*, i.e. T is the identity functor over $\mathcal{C}$ and η and μ are the identity natural transformation over T, then computations get identified with values. In this case $(\mathrm{let}_T\, x{\Leftarrow}e_1\, \mathrm{in}\, e_2)$ can be replaced by $[e_1/x]e_2$, so g is simply $\langle \mathrm{id}_c, g_1\rangle; g_2{:}\,c \to c_2$. In the general case Table 3 suggests that $\_;\_$ above is indeed composition in the Kleisli category, therefore $\langle \mathrm{id}_c, g_1\rangle; g_2$ should be replaced by $\langle \mathrm{id}_c, g_1\rangle; {g_2}^*$. But in $\langle \mathrm{id}_c, g_1\rangle; {g_2}^*$ there is a type mismatch, since the codomain of $\langle \mathrm{id}_c, g_1\rangle$ is $c\times Tc_1$, while the domain of Tg_1 is $T(c\times c_1)$. The natural transformation $\mathrm{t}_{A,B}{:}\, A\times TB \to T(A\times B)$ mediates between these two objects, so that g can be defined as $\langle \mathrm{id}_c, g_1\rangle; \mathrm{t}_{c,c_1}; {g_2}^*$.

[4]If the metalanguage does not have finite products, we conjecture that its theories would no longer correspond to categories with finite products and a strong monad (even by taking as objects contexts and/or the Karoubi envelope, used in [Sco80] to associate a cartesian closed category to an untyped λ-theory), but instead to *multicategories* with a Kleisli triple. We felt the greater generality (of not having products in the metalanguage) was not worth the mathematical complications.

Definition 3.2 *A* **strong monad** *over a category $\mathcal{C}$ with (explicitly given) finite products is a monad (T, η, μ) together with a natural transformation $\mathrm{t}_{A,B}$ from $A \times TB$ to $T(A \times B)$ s.t.*

$$
\begin{tikzcd}
TA & \\
1 \times TA \arrow[u, "r_{TA}"] \arrow[r, "\mathrm{t}_{1,A}"] & T(1 \times A) \arrow[ul, "Tr_A"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
(A \times B) \times TC \arrow[r, "\mathrm{t}_{A \times B,C}"] \arrow[d, "\alpha_{A,B,TC}"'] & T((A \times B) \times C) \arrow[dr, "T\alpha_{A,B,C}"'] & \\
A \times (B \times TC) \arrow[r, "\mathrm{id}_A \times \mathrm{t}_{B,C}"] & A \times T(B \times C) \arrow[r, "\mathrm{t}_{A,B \times C}"] & T(A \times (B \times C))
\end{tikzcd}
$$

$$
\begin{tikzcd}
A \times B \arrow[d, "\mathrm{id}_A \times \eta_B"'] \arrow[dr, "\eta_{A \times B}"'] & & \\
A \times TB \arrow[r, "\mathrm{t}_{A,B}"] & T(A \times B) & \\
A \times T^2B \arrow[u, "\mathrm{id}_A \times \mu_B"] \arrow[r, "\mathrm{t}_{A,TB}"] & T(A \times TB) \arrow[r, "T\mathrm{t}_{A,B}"] & T^2(A \times B) \arrow[ul, "\mu_{A \times B}"']
\end{tikzcd}
$$

where r and α are the natural isomorphisms

$$r_A : (1 \times A) \to A \quad , \quad \alpha_{A,B,C} : (A \times B) \times C \to A \times (B \times C)$$

Remark 3.3 The diagrams above are taken from [Koc72], where a characterisation of strong monads is given in terms of $\mathcal{C}$-enriched categories (see [Kel82]). Kock fixes a commutative monoidal closed category $\mathcal{C}$ (in particular a cartesian closed category), and in this setup he establishes a one-one correspondence between *strengths* $\mathrm{st}_{A,B} : B^A \to (TB)^{TA}$ and *tensorial strengths* $\mathrm{t}_{A,B} : A \otimes TB \to T(A \otimes B)$ for a endofunctor T over $\mathcal{C}$ (see Theorem 1.3 in [Koc72]). Intuitively a strength $\mathrm{st}_{A,B}$ *internalises* the action of T on morphisms from A to B, and more precisely it makes (T, st) a $\mathcal{C}$-enriched endofunctor on $\mathcal{C}$ enriched over itself (i.e. the hom-object $\mathcal{C}(A, B)$ is B^A). In this setting the diagrams of Definition 3.2 have the following meaning:

- the first two diagrams are (1.7) and (1.8) in [Koc72], saying that t is a tensorial strength of T. So T can be made into a $\mathcal{C}$-enriched endofunctor.
- the last two diagrams say that $\eta : \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu : T^2 \dot{\to} T$ are $\mathcal{C}$-enriched natural transformations, where $\mathrm{Id}_{\mathcal{C}}$, T and T^2 are enriched in the obvious way (see Remark 1.4 in [Koc72]).

There is another purely categorical characterisation of strong monads, suggested to us by G. Plotkin, in terms of $\mathcal{C}$-indexed categories (see [JP78]). Both characterisations are instances of a general methodological principle for studying programming languages (or logics) categorically (see [Mog89b]):

> when studying a complex language the 2-category **Cat** of small categories, functors and natural transformations may not be adequate; however, one may replace **Cat** with a different 2-category, whose objects captures better some *fundamental structure* of the language, while *less fundamental structure* can be modelled by *2-categorical concepts*.

Monads are a 2-categorical concept, so we expect notions of computations for a complex language to be modelled by monads in a suitable 2-category.

The first characterisation takes a commutative monoidal closed structure on $\mathcal{C}$ (used in [Laf88, See87] to model a fragment of *linear logic*), so that $\mathcal{C}$ can be enriched over itself. Then a strong monad over a cartesian closed category $\mathcal{C}$ is just a monad over $\mathcal{C}$ in the 2-category of $\mathcal{C}$-enriched categories.

The second characterisation takes a class $\mathcal{D}$ of display maps over $\mathcal{C}$ (used in [HP87] to model *dependent types*), and defines a $\mathcal{C}$-indexed category $\mathcal{C}/_{\mathcal{D}}\_$. Then a strong monad over a category $\mathcal{C}$ with finite products amounts to a monad over $\mathcal{C}/_{\mathcal{D}}\_$ in the 2-category of $\mathcal{C}$-indexed categories, where $\mathcal{D}$ is the class of first projections (corresponding to constant type dependency).

In general the natural transformation t has to be given explicitly as part of the additional structure. However, t is uniquely determined (but it may not exists) by T and the cartesian structure on $\mathcal{C}$, when $\mathcal{C}$ has enough points.

Proposition 3.4 (Uniqueness) *If (T, η, μ) is a monad over a category $\mathcal{C}$ with finite products and enough points (i.e. $\forall h: 1 \to A. h; f = h; g$ implies $f = g$ for any $f, g: A \to B$), then $(T, \eta, \mu, \mathrm{t})$ is a strong monad over $\mathcal{C}$ if and only if $\mathrm{t}_{A,B}$ is the unique family of morphisms s.t. for all points $a: 1 \to A$ and $b: 1 \to TB$*

$$\langle a, b\rangle; \mathrm{t}_{A,B} = b; T(\langle !_B; a, \mathrm{id}_B\rangle)$$

where $!_B: B \to 1$ is the unique morphism from B to the terminal object.

Proof Note that there is at most one $\mathrm{t}_{A,B}$ s.t. $\langle a, b\rangle; \mathrm{t}_{A,B} = b; T(\langle !_B; a, \mathrm{id}_B\rangle)$ for all points $a: 1 \to A$ and $b: 1 \to TB$, because $\mathcal{C}$ has enough points.

First we show that if $(T, \eta, \mu, \mathrm{t})$ is a strong monad, then $\mathrm{t}_{A,B}$ satisfies the equation above. By naturality of t and by the first diagram in Definition 3.2 the following diagram commutes

$$\begin{tikzcd}
1 \arrow[r, "\langle a, b\rangle"] \arrow[dr, "\langle \mathrm{id}_1, b\rangle"'] & A \times TB \arrow[r, "\mathrm{t}_{A,B}"] & T(A \times B) \\
 & 1 \times TB \arrow[u, "a \times \mathrm{id}_{TB}"'] \arrow[r, "\mathrm{t}_{1,B}"] \arrow[dr, "r_{TB}"] & T(1 \times B) \arrow[u, "T(a \times \mathrm{id}_B)"'] \arrow[d, "Tr_B"'] \\
 & & TB
\end{tikzcd}$$

Since r_B is an isomorphism (with inverse $\langle !_B, \mathrm{id}_B\rangle$), then the two composite morphisms $\langle a, b\rangle; \mathrm{t}_{A,B}$ and $\langle \mathrm{id}_1, b\rangle; r_{TB}; T(r_B^{-1}); T(a \times \mathrm{id}_B)$ from 1 to $T(A \times B)$ must coincide. But the second composition can be rewritten as $b; T(\langle !_B; a, \mathrm{id}_B\rangle)$.

Second we have to show that if t is the unique family of morphisms satisfying the equation above, then $(T, \eta, \mu, \mathrm{t})$ is a strong monad. This amount to prove that t is a natural transformation and that the three diagrams in Definition 3.2 commute. The proof is a tedious diagram chasing, which relies on $\mathcal{C}$ having enough points. For instance, to prove that $\mathrm{t}_{1,A}; Tr_A = r_{TA}$ it is enough to show that $\langle \mathrm{id}_1, a\rangle; \mathrm{t}_{1,A}; Tr_A = \langle \mathrm{id}_1, a\rangle; r_{TA}$ for all points $a: 1 \to A$. ∎

Example 3.5 We go through the monads given in Example 1.4 and show that they have a tensorial strength.

- **partiality** $TA = A_\perp (= A + \{\perp\})$
  $\mathrm{t}_{A,B}(a, \perp) = \perp$ and $\mathrm{t}_{A,B}(a, b) = \langle a, b\rangle$ (when $b \in B$)

- **nondeterminism** $TA = \mathcal{P}_{fin}(A)$
  $\mathrm{t}_{A,B}(a, c) = \{\langle a, b\rangle | b \in c\}$

- **side-effects** $TA = (A \times S)^S$
  $\mathrm{t}_{A,B}(a,c) = (\lambda s{:}\,S.(\mathrm{let}\,\langle b,s'\rangle = c(s)\,\mathrm{in}\,\langle\langle a,b\rangle,s'\rangle))$

- **exceptions** $TA = (A + E)$
  $\mathrm{t}_{A,B}(a,\mathrm{inr}(e)) = \mathrm{inr}(e)$ (when $e \in E$) and
  $\mathrm{t}_{A,B}(a,\mathrm{inl}(b)) = \mathrm{inl}(\langle a,b\rangle)$ (when $b \in B$)

- **continuations** $TA = R^{(R^A)}$
  $\mathrm{t}_{A,B}(a,c) = (\lambda k{:}\,R^{A\times B}.c(\lambda b{:}\,B.k(\langle a,b\rangle)))$

- **interactive input** $TA = (\mu\gamma.A + \gamma^U)$
  $\mathrm{t}_{A,B}(a,c)$ is the tree obtained by replacing leaves of c labelled by b with the leaf labelled by $\langle a,b\rangle$

- **interactive output** $TA = (\mu\gamma.A + (U \times \gamma))$
  $\mathrm{t}_{A,B}(a,\langle s,b\rangle) = \langle s,\langle a,b\rangle\rangle$.

Remark 3.6 The tensorial strength t induces a natural transformation $\psi_{A,B}$ from $TA \times TB$ to $T(A \times B)$, namely

$$\psi_{A,B} = c_{TA,TB};\mathrm{t}_{TB,A};(c_{TB,A};\mathrm{t}_{A,B})^*$$

where c is the natural isomorphism $c_{A,B}{:}\,A \times B \to B \times A$.

The morphism $\psi_{A,B}$ has the correct domain and codomain to interpret the pairing of a computation of type A with one of type B, obtained by first evaluating the first argument and then the second, namely

$$c_1{:}\,TA, c_2{:}\,TB \quad \overset{\psi_{A,B}}{\longmapsto} \quad (\mathrm{let}\, x{\Leftarrow}c_1\,\mathrm{in}\,(\mathrm{let}\, y{\Leftarrow}c_2\,\mathrm{in}\,[\langle x,y\rangle])){:}\,T(A \times B)$$

There is also a dual notion of pairing, $\tilde{\psi}_{A,B} = c_{TA,TB};\psi_{B,A};Tc_{B,A}$ (see [Koc72]), which amounts to first evaluating the second argument and then the first.

3.1 Interpretation and formal system

We are now in a position to give the metalanguage for algebraic terms, its interpretation and inference rules.

Definition 3.7 (metalanguage) *An interpretation $[\![\_]\!]$ of the metalanguage in a category $\mathcal{C}$ with terminal object $!_A{:}\,A \to 1$, binary products $\pi_i^{A_1,A_2}{:}\,A_1 \times A_2 \to A_i$ and a strong monad (T,η,μ,t) is parametric in an interpretation of the symbols in the signature and is defined by induction on the derivation of well-formedness for types (see Table 8), terms and equations (see Table 9).*

Finite products $\pi_i^{A_1,\ldots,A_n}{:}\,A_1 \times \ldots \times A_n \to A_i$ used to interpret contexts and variables are defined by induction on n:

0 $A_1 \times \ldots \times A_0 \stackrel{\Delta}{=} 1$

$n+1$ $A_1 \times \ldots \times A_{n+1} \stackrel{\Delta}{=} (A_1 \times \ldots \times A_n) \times A_{n+1}$

- $\pi_{n+1}^{A_1,\ldots,A_{n+1}} = \pi_2^{(A_1\times\ldots\times A_n),A_{n+1}}$
- $\pi_i^{A_1,\ldots,A_{n+1}} = \pi_1^{(A_1\times\ldots\times A_n),A_{n+1}};\pi_i^{A_1,\ldots,A_n}$

The inference rules for the metalanguage (see Table 10) are divided into three groups:

- *general rules for many sorted equational logic*
- *rules for finite products*
- *rules for T*

RULE	SYNTAX	SEMANTICS
A	$\dfrac{}{\vdash_{ml} \text{A type}}$	$= [\![\text{A}]\!]$
T	$\dfrac{\vdash_{ml} \tau \text{ type}}{\vdash_{ml} T\tau \text{ type}}$	$\begin{array}{l} = c \\ = Tc \end{array}$
1	$\dfrac{}{\vdash_{ml} 1 \text{ type}}$	$= 1$
$\times$	$\dfrac{\begin{array}{l}\vdash_{ml} \tau_1 \text{ type} \\ \vdash_{ml} \tau_2 \text{ type}\end{array}}{\vdash_{ml} \tau_1 \times \tau_2 \text{ type}}$	$\begin{array}{l} = c_1 \\ = c_2 \\ = c_1 \times c_2 \end{array}$
$\emptyset$	$\dfrac{\vdash_{ml} \tau_i \text{ type} \quad (1 \leq i \leq n)}{x_1\colon\tau_1, \ldots, x_n\colon\tau_n \vdash}$	$\begin{array}{l} = c_i \\ = c_1 \times \ldots \times c_n \end{array}$

Table 8: Interpretation of types in the Metalanguage

Proposition 3.8 *Every theory $\mathcal{T}$ of the metalanguage, viewed as a category $\mathcal{F}(\mathcal{T})$, is equipped with finite products and a strong monad whose tensorial strength is*

$$\mathrm{t}_{\tau_1,\tau_2} = [x\colon \tau_1 \times T\tau_2 \vdash_{ml} (\mathrm{let}_T\, x_2 \Leftarrow \pi_2 x \,\mathrm{in}\, [\langle \pi_1 x, x_2\rangle]_T)\colon T(\tau_1 \times \tau_2)]_{\mathcal{T}}$$

Proof Similar to that of Proposition 2.5 ∎

Once we have a metalanguage for algebraic terms it is straightforward to add data-types characterised by *universal properties* and extend the categorical semantics accordingly[5]. For instance, if we want to have function spaces, then we simply require the category $\mathcal{C}$ (where the metalanguage is interpreted) to have exponentials B^A and add the inference rules for the simply typed λ-calculus (see Table 11) to those for the metalanguage. From a programming language perspective the situation is more delicate. For instance, the semantics of functional types should reflect the choice of *calling mechanism*[6]:

- in call-by-value a procedure of type $A \to B$ expects a value of type A and computes a result of type B, so the interpretation of $A \to B$ is $(TB)^A$;
- in call-by-name a procedure of type $A \to B$ expects a computation of type A, which is evaluated only when needed, and computes a result of type B, so the interpretation of $A \to B$ is $(TB)^{TA}$.

In both cases the only exponentials needed to interpret the functional types of a programming language are of the form $(TB)^A$. By analogy with partial cartesian closed categories (pccc), where only *p-exponentials* are required to exists (see [Mog86, Ros86]), we adopt the following definition of λ_c-model:

[5]The next difficult step in extending the metalanguage is the combination of dependent types and computations, which is currently under investigation.

[6]call-by-need does not have a simple categorical semantics, since the environment in which an expression is evaluated may itself undergo evaluation.

RULE	SYNTAX		SEMANTICS
$\mathrm{var_i}$	$\vdash_{ml} \tau_i$ type $\quad (1 \leq i \leq n)$	$=$	c_i
	$x_1\!:\tau_1, \ldots, x_n\!:\tau_n \vdash x_i\!:\tau_i$	$=$	$\pi_i^{c_1,\ldots,c_n}$
$*$			
	$\Gamma \vdash *\!:1$	$=$	$!_{[\![\Gamma]\!]}$
$\langle\rangle$	$\Gamma \vdash e_1\!:\tau_1$	$=$	g_1
	$\Gamma \vdash e_2\!:\tau_2$	$=$	g_2
	$\Gamma \vdash \langle e_1, e_2\rangle\!:\tau_1 \times \tau_2$	$=$	$\langle g_1, g_2\rangle$
π_i	$\Gamma \vdash e\!:\tau_1 \times \tau_2$	$=$	g
	$\Gamma \vdash \pi_i(e)\!:\tau_1$	$=$	$g; \pi_i^{[\![\tau_1]\!],[\![\tau_2]\!]}$
$\mathrm{f}\!:\tau_1 \to \tau_2$	$\Gamma \vdash_{ml} e_1\!:\tau_1$	$=$	g
	$\Gamma \vdash_{ml} \mathrm{f}(e_1)\!:\tau_2$	$=$	$g; [\![\mathrm{f}]\!]$
$[\_]_\mathrm{T}$	$\Gamma \vdash_{ml} e\!:\tau$	$=$	g
	$\Gamma \vdash_{ml} [e]_T\!:T\tau$	$=$	$g; \eta_{[\![\tau]\!]}$
let	$\Gamma \vdash_{ml} e_1\!:T\tau_1$	$=$	g_1
	$\Gamma, x\!:\tau_1 \vdash_{ml} e_2\!:T\tau_2$	$=$	g_2
	$\Gamma \vdash_{ml} (\mathrm{let}_T\, x\Leftarrow e_1 \,\mathrm{in}\, e_2)\!:T\tau_2$	$=$	$\langle \mathrm{id}_{[\![\Gamma]\!]}, g_1\rangle; \mathrm{t}_{[\![\Gamma]\!],[\![\tau_1]\!]}; g_2^*$
eq	$\Gamma \vdash_{ml} e_1\!:\tau$	$=$	g_1
	$\Gamma \vdash_{ml} e_2\!:\tau$	$=$	g_2
	$\Gamma \vdash_{ml} e_1 =_\tau e_2$	$\iff$	$g_1 = g_2$

Table 9: Interpretation of terms in the Metalanguage

$$\text{refl}\ \frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash e =_\tau e}$$

$$\text{symm}\ \frac{\Gamma \vdash e_1 =_\tau e_2}{\Gamma \vdash e_2 =_\tau e_1}$$

$$\text{trans}\ \frac{\Gamma \vdash e_1 =_\tau e_2 \qquad \Gamma \vdash e_2 =_\tau e_3}{\Gamma \vdash e_2 =_\tau e_3}$$

$$\text{congr}\ \frac{\Gamma \vdash e_1 =_{\tau_1} e_2}{\Gamma \vdash \mathrm{f}(e_1) =_{\tau_2} \mathrm{f}(e_2)}\ \mathrm{f}\colon \tau_1 \to \tau_2$$

$$\text{subst}\ \frac{\Gamma \vdash e\colon \tau \qquad \Gamma, x\colon \tau \vdash \phi}{\Gamma \vdash [e/x]\phi}$$

Inference Rules of Many Sorted Equational Logic

$$1.\eta\ \ \Gamma \vdash * =_1 x$$

$$\langle\rangle.\xi\ \frac{\Gamma \vdash e_1 =_{\tau_1} e_1' \qquad \Gamma \vdash e_2 =_{\tau_2} e_2'}{\Gamma \vdash \langle e_1, e_2\rangle =_{\tau_1\times\tau_2} \langle e_1', e_2'\rangle}$$

$$\times.\beta\ \frac{\Gamma \vdash e_1\colon \tau_1 \qquad \Gamma \vdash e_2\colon \tau_2}{\Gamma \vdash \pi_i(\langle e_1, e_2\rangle) =_{\tau_i} e_i}$$

$$\times.\eta\ \frac{\Gamma \vdash e\colon \tau_1 \times \tau_2}{\Gamma \vdash \langle \pi_1(e), \pi_2(e)\rangle =_{\tau_1\times\tau_2} e}$$

rules for product types

$$[\_].\xi\ \frac{\Gamma \vdash_{ml} e_1 =_\tau e_2}{\Gamma \vdash_{ml} [e_1]_T =_{T\tau} [e_2]_T}$$

$$\text{let}.\xi\ \frac{\Gamma \vdash_{ml} e_1 =_{T\tau_1} e_2 \qquad \Gamma, x\colon \tau_1 \vdash_{ml} e_1' =_{T\tau_2} e_2'}{\Gamma \vdash_{ml} (\mathrm{let}_T\, x \Leftarrow e_1 \,\mathrm{in}\, e_1') =_{T\tau_2} (\mathrm{let}_T\, x \Leftarrow e_2 \,\mathrm{in}\, e_2')}$$

$$\text{ass}\ \frac{\Gamma \vdash_{ml} e_1\colon T\tau_1 \qquad \Gamma, x_1\colon \tau_1 \vdash_{ml} e_2\colon T\tau_2 \qquad \Gamma, x_2\colon \tau_2 \vdash_{ml} e_3\colon T\tau_3}{\Gamma \vdash_{ml} (\mathrm{let}_T\, x_2 \Leftarrow (\mathrm{let}_T\, x_1 \Leftarrow e_1 \,\mathrm{in}\, e_2) \,\mathrm{in}\, e_3) =_{T\tau_3} (\mathrm{let}_T\, x_1 \Leftarrow e_1 \,\mathrm{in}\, (\mathrm{let}_T\, x_2 \Leftarrow e_2 \,\mathrm{in}\, e_3))}$$

$$\mathrm{T}.\beta\ \frac{\Gamma \vdash_{ml} e_1\colon \tau_1 \qquad \Gamma, x_1\colon \tau_1 \vdash_{ml} e_2\colon T\tau_2}{\Gamma \vdash_{ml} (\mathrm{let}_T\, x_1 \Leftarrow [e_1]_T \,\mathrm{in}\, e_2) =_{T\tau_2} [e_1/x_1]e_2}$$

$$\mathrm{T}.\eta\ \frac{\Gamma \vdash_{ml} e_1\colon T\tau_1}{\Gamma \vdash_{ml} (\mathrm{let}_T\, x_1 \Leftarrow e_1 \,\mathrm{in}\, [x_1]_T) =_{T\tau_1} e_1}$$

Table 10: Inference Rules of the Metalanguage

$$\text{app}.\xi\ \frac{\Gamma \vdash e_1 =_{\tau_1} e_1' \qquad \Gamma \vdash e =_{\tau_1\to\tau_2} e'}{\Gamma \vdash e e_1 =_{\tau_2} e' e_1'}$$

$$\lambda.\xi\ \frac{\Gamma, x\colon \tau_1 \vdash e_1 =_{\tau_2} e_2}{\Gamma \vdash (\lambda x\colon \tau_1.e_1) =_{\tau_1\to\tau_2} (\lambda x\colon \tau_1.e_2)}$$

$$\to.\beta\ \frac{\Gamma \vdash e_1\colon \tau_1 \qquad \Gamma, x\colon \tau_1 \vdash e_2\colon \tau_2}{\Gamma \vdash (\lambda x\colon \tau_1.e_2)e_1 =_{\tau_2} [e_1/x]e_2}$$

$$\to.\eta\ \frac{\Gamma \vdash e\colon \tau_1 \to \tau_2}{\Gamma \vdash (\lambda x\colon \tau_1.ex) =_{\tau_1\to\tau_2} e}\ x \notin \mathrm{DV}(\Gamma)$$

Table 11: rules for function spaces

Definition 3.9 *A* λ_c**-model** *is a category* $\mathcal{C}$ *with finite products, a strong monad* $(T, \eta, \mu, \mathrm{t})$ *satisfying the mono requirement (i.e.* η_A *mono for every* $A \in \mathcal{C}$*) and* T**-exponential** $(TB)^A$ *for every* $A, B \in \mathcal{C}$*.*

Remark 3.10 The definition of λ_c-model generalises that of pccc, in the sense that every pccc can be viewed as a λ_c-model. By analogy with p-exponentials, a T-exponential can be defined by giving an isomorphism $\mathcal{C}_T(C \times A, B) \cong \mathcal{C}(C, (TB)^A)$ natural in $C \in \mathcal{C}$. We refer to [Mog89c] for the interpretation of a call-by-value programming language in a λ_c-model and the corresponding formal system, the λ_c-calculus.

4 Strong monads over a topos

In this section we show that, as far as monads or strong monads are concerned, we can assume w.l.o.g. that they are over a topos (see Theorem 4.9). The proof of Theorem 4.9 involves non-elementary notions from Category Theory, and we postpone it after discussing some applications, with particular emphasis on further extensions of the metalanguage and on conservative extension results.

Let us take as formal system for toposes the type theory described in [LS86], this is a many sorted intuitionistic higher order logic with equality and with a set of types satisfying the following closure properties[7]:

- the terminal object 1, the natural number object N and the subobject classifier Ω are types
- if A is a type, then the power object PA is a type
- if A and B are types, then the binary product $A \times B$ and the function space $A \to B$ are types
- if A is a type and $\phi: A \to \Omega$ is a predicate, then $\{x \in A | \phi(x)\}$ is a type.

Notation 4.1 We introduce some notational conventions for formal systems:

- ML_T is the metalanguage for algebraic terms, whose set of types is closed under terminal object, binary products and TA;
- $\lambda\mathrm{ML}_T$ is the extension of ML_T with function spaces $A \to B$ (interpreted as exponentials);
- HML_T is the type theory described above (see [LS86]) extended with objects of computations TA;
- PL is the programming language for algebraic terms (see [Mog89c]);
- λ_cPL is the extension of PL with function spaces $A \rightharpoonup B$ (interpreted as T-exponentials), called λ_c-calculus in [Mog89c].

Definition 4.2 *We say that a formal system* $(L_2, \vdash_2)$*, where* $\vdash_2 \subseteq \mathcal{P}(L_2) \times L_2$ *is a formal consequence relation*[8] *over* L_2*, is a* **conservative extension** *of* $(L_1, \vdash_1)$ *provided* $L_1 \subseteq L_2$ *and* $\vdash_1$ *is the restriction of* $\vdash_2$ *to* $\mathcal{P}(L_1) \times L_1$*.*

Theorem 4.3 HML_T *is a conservative extension of* ML_T *and* $\lambda\mathrm{ML}_T$*. In particular* $\lambda\mathrm{ML}_T$ *is a conservative extension of* ML_T*.*

---

[7] Lambek and Scott do not require closure under function spaces and subsets $\{x \in A | \phi(x)\}$.

[8] For instance, in the case of ML_T the elements of L are well-formed equality judgements $\Gamma \vdash_{ml} e_1 =_\tau e_2$ and $P \vdash C$ iff there exists a derivation of C, where all assumptions are in P.

Proof The first result follows from Theorem 4.9, which implies that for every model $\mathcal{C}$ of ML_T the Yoneda embedding maps the interpretation of an ML_T-term in $\mathcal{C}$ to its interpretation in $\hat{\mathcal{C}}$, and the faithfulness of the Yoneda embedding, which implies that two ML_T-terms have the same interpretation in $\mathcal{C}$ iff they have the same interpretation in $\hat{\mathcal{C}}$. The second result follows, because the Yoneda embedding preserves function spaces. The third conservative extension result follows immediately from the first two. ∎

The above result means that we can think of computations naively in terms of sets and functions, provided we treat them intuitionistically, and can use the full apparatus of higher-order (intuitionistic) logic instead of the less expressive many sorted equational logic.

Before giving a conservative extension result for the programming language, we have to express the mono requirement, equivalence and existence in HML_T. The idea is to extend the translation from PL-terms to ML_T-terms given in Definition 2.7 and exploit the increased expressiveness of HML_T over ML_T to axiomatise the mono requirement and translate existence and equivalence assertions (see Remark 2.1):

- the **mono requirement** for τ, i.e. η_τ is mono, is axiomatised by

  $$\text{mono.}\tau \quad (\forall x, y{:}\,\tau.[x]_T =_{T\tau} [y]_T \to x =_\tau y)$$

- the **equalising requirement** for τ, i.e. η_τ is the equaliser of $T(\eta_\tau)$ and $\eta_{T\tau}$, is axiomatised by (mono.τ) and the axiom

  $$\text{eqls.}\tau \quad (\forall x{:}\,T\tau.[x]_T =_{T^2\tau} (\mathrm{let}_T\, y \Leftarrow x \,\mathrm{in}\, [[y]_T]_T) \to (\exists! y{:}\,\tau.x =_{T\tau} [y]_T))$$

- the **translation** $\_^\circ$ is extended to assertions and functional types as follows:
  - $(e_1 \equiv_\tau e_2)^\circ \stackrel{\Delta}{\equiv} e_1{}^\circ =_{T\tau} e_2{}^\circ$
  - $(e_1 \downarrow_\tau)^\circ \stackrel{\Delta}{\equiv} (\exists! x{:}\,\tau.e_1{}^\circ =_{T\tau} [x]_T)$
  - $(\tau_1 \rightharpoonup \tau_2)^\circ \stackrel{\Delta}{\equiv} \tau_1{}^\circ \to T\tau_2{}^\circ$

Theorem 4.4 $\mathrm{HML}_T + \{(\text{mono.}\tau) \mid \tau$ *type of* PL$\}$ *(i.e. τ is built using only base types, 1, TA, and $A \times B$) is a conservative extension of* PL *(after translation). Similarly,* $\mathrm{HML}_T + \{(\text{mono.}\tau) \mid \tau$ *type of* λ_cPL$\}$ *(i.e. τ is built using only base types, 1, TA, $A \times B$ and $A \to B$) is a conservative extension of* λ_cPL *(after translation).*

Proof The proof proceeds as in the previous theorem. The only additional step is to show that for every type τ of PL (or λ_cPL) the axiom (mono.τ) holds in $\hat{\mathcal{C}}$, under the assumption that $\mathcal{C}$ satisfies the mono requirement. Let c be the interpretation of τ in $\mathcal{C}$ (therefore $\mathrm{Y}c$ is the interpretation of τ in $\hat{\mathcal{C}}$), then the axiom (mono.τ) holds in $\hat{\mathcal{C}}$ provided $\hat{\eta}_{\mathrm{Y}c}$ is a mono. η_c is mono (by the mono requirement), so $\hat{\eta}_{\mathrm{Y}c} = \mathrm{Y}(\eta_c)$ is mono (as Y preserves monos). ∎

In the theorem above only types from the programming language have to satisfy the mono requirement. Indeed, $\mathrm{HML}_T + \{(\text{mono.}\tau) \mid \tau$ type of $\mathrm{HML}_T\}$ is not a conservative extension of PL (or λ_cPL).

Lemma 4.5 *If (T, η, μ) is a monad over a topos $\mathcal{C}$ satisfying the mono requirement, then it satisfies also the equalising requirement.*

Proof See Lemma 6 on page 110 of [BW85]. ∎

In other words, for any type τ the axiom (eqls.τ) is derivable in HML_T from the set of axioms $\{(\text{mono.}\tau) \mid \tau$ type of $\mathrm{HML}_T\}$. In general, when $\mathcal{C}$ is not a topos, the mono requirement does not entail the equalising requirement; one can easily define strong monads (over an Heyting algebra) that satisfy the mono but not the equalising requirement (just take $T(A) = A \vee B$, for some element $B \neq \bot$ of the Heyting algebra). In terms of formal consequence relation this means that in HML_T + mono requirement the existence assertion $\Gamma \vdash_{pl} e \downarrow_\tau$ is derivable from $\Gamma \vdash_{pl} [e] \equiv_{T\tau} (\mathrm{let}\, x \Leftarrow e \,\mathrm{in}\, [x])$, while such derivation is not possible in λ_cPL. We do not know whether HML_T + equalising requirement is a conservative extension of PL + equalising requirement, or whether λ_cPL is a conservative extension of PL.

A language which combines computations and higher order logic, like HML_T, seems to be the ideal framework for program logics that go beyond proving equivalence of programs, like Hoare's logic for partial correctness of imperative languages. In HML_T (as well as ML_T and PL) one can describe a programming language by introducing additional constant and axioms. In $\lambda\mathrm{ML}_T$ or λ_cPL such constants correspond to program-constructors, for instance:

- $lookup: L \to TU$, which given a location $l \in L$ produces the value of such location in the current store, and $update: L \times U \to T1$, which changes the current store by assigning to $l \in L$ the value $u \in U$;
- $if: Bool \times TA \times TA \to TA$ and $while: T(Bool) \times T1 \to T1$;
- $new: 1 \to TL$, which returns a newly created location;
- $read: 1 \to TU$, which computes a value by reading it from the input, and $write: U \to T1$, which writes a value $u \in U$ on the output.

In HML_T one can describe also a program logic, by adding constants $p: TA \to \Omega$ corresponding to properties of computations.

Example 4.6 Let T be the monad for non-deterministic computations (see Example 1.4), then we can define a predicate $may: A \times TA \to \Omega$ such that $may(a, c)$ is true iff the value a is a possible outcome of the computation c (i.e. $a \in c$). However, there is a more uniform way of defining the may predicate for any type. Let $\Diamond: T\Omega \to \Omega$ be the predicate such that $\Diamond(X) = \top$ iff $\top \in X$, where Ω is the set $\{\bot, \top\}$ (note that $\Diamond(\_) = may(\top, \_)$). Then, $may(a, c)$ can be defined as $\Diamond(\mathrm{let}_T\, x \Leftarrow c \,\mathrm{in}\, [a =_\tau x]_T)$.

The previous example suggests that predicates defined *uniformly* on computations of any type can be better described in terms of *modal operators* $\gamma: T\Omega \to \Omega$, relating a computation of truth values to a truth value. This possibility has not been investigated in depth, so we will give only a tentative definition.

Definition 4.7 *If (T, η, μ) is a monad over a topos $\mathcal{C}$, then a T-***modal operator** *is a T-algebra $\gamma: T\Omega \to \Omega$, i.e.*

$$\begin{tikzcd}
T^2\Omega \arrow[r, "\mu_\Omega"] \arrow[d, "T\gamma"'] & T\Omega \arrow[d, "\gamma"] & \Omega \arrow[l, "\eta_\Omega"'] \arrow[dl, "\mathrm{id}_{T\Omega}"] \\
T\Omega \arrow[r, "\gamma"'] & \Omega &
\end{tikzcd}$$

where Ω is the subobject classifier in $\mathcal{C}$.

The commutativity of the two diagrams above can be expressed in the metalanguage:

- $x: \Omega \vdash \gamma([x]_T) \longleftrightarrow x$
- $c: T^2\Omega \vdash \gamma(\mathrm{let}\, x \Leftarrow c \,\mathrm{in}\, x) \longleftrightarrow \gamma(\mathrm{let}\, x \Leftarrow c \,\mathrm{in}\, [\gamma(x)]_T)$

We consider some examples and *non-examples* of modal operators.

Example 4.8 For the monad T of non-deterministic computations (see Example 1.4) there are only two modal operators $\Box$ and $\Diamond$:

- $\Box(X) = \bot$ iff $\bot \in X$;

- $\Diamond(X) = \top$ iff $\top \in X$.

Given a nondeterministic computation e of type τ and a predicate $A(x)$ over τ, i.e. a term of type Ω, then $\Box(\mathrm{let}_T\, x \Leftarrow e \,\mathrm{in}\, [A(x)]_T)$ is true iff all possible results of e satisfy $A(x)$.

For the monad T of computations with side-effects (see Example 1.4) there is an operator $\Box: (\Omega \times S)^S \to \Omega$ that can be used to express Hoare's triples:

- $\Box f = \top$ iff for all $s \in S$ there exists $s' \in S$ s.t. $fs = \langle \top, s' \rangle$

  this operator does not satisfy the second equivalence, as only one direction is valid, namely $c: T^2\Omega \vdash \gamma(\mathrm{let}\, x \Leftarrow c \,\mathrm{in}\, [\gamma(x)]_T) \to \gamma(\mathrm{let}\, x \Leftarrow c \,\mathrm{in}\, x)$

Let $P: U \to \Omega$ and $Q: U \times U \to \Omega$ be predicates over storable values, $e \in T1$ a computation of type 1 and $x, y \in L$ locations. The intended meaning of the triple $\{P(x)\}e\{Q(x,y)\}$ is "if in the initial state the content u of x satisfies $P(u)$, then in the final state (i.e. after executing e) the content v of y satisfies $Q(u,v)$". This intended meaning can be expressed formally in terms of the modal operator $\Box$ and the program-constructors *lookup* and *update* as follows:

$$\forall u: U.P(u) \to \Box(\mathrm{let}_T\, v \Leftarrow (\mathit{update}(x,u); e; \mathit{lookup}(y)) \,\mathrm{in}\, [Q(u,v)]_T)$$

where $\_;\_: TA \times TB \to TB$ is the derived operation $e_1; e_2 \stackrel{\Delta}{\equiv} (\mathrm{let}_T\, x \Leftarrow e_1 \,\mathrm{in}\, e_2)$ with x not free in e_2.

Finally, we state the main theorem and outline its proof. In doing so we assume that the reader is familiar with non-elementary concepts from Category Theory.

Theorem 4.9 *Let $\mathcal{C}$ be a small category, $\hat{\mathcal{C}}$ the topos of presheaves over $\mathcal{C}$ and Y the Yoneda embedding of $\mathcal{C}$ into $\hat{\mathcal{C}}$. Then for every monad (T, η, μ) over $\mathcal{C}$, there exists a monad $(\hat{T}, \hat{\eta}, \hat{\mu})$ over $\hat{\mathcal{C}}$ such that the following diagram commutes*[9]

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "T"] \arrow[d, "\mathrm{Y}"'] & \mathcal{C} \arrow[d, "\mathrm{Y}"] \\
\hat{\mathcal{C}} \arrow[r, "\hat{T}"'] & \hat{\mathcal{C}}
\end{tikzcd}$$

and for all $a \in \mathcal{C}$ the following equations hold

$$\hat{\eta}_{\mathrm{Y}a} = \mathrm{Y}(\eta_a) \quad , \quad \hat{\mu}_{\mathrm{Y}a} = \mathrm{Y}(\mu_a)$$

Moreover, for every strong monad $(T, \eta, \mu, \mathrm{t})$ over $\mathcal{C}$, there exists a natural transformation $\hat{\mathrm{t}}$ such that $(\hat{T}, \hat{\eta}, \hat{\mu}, \hat{\mathrm{t}})$ is a strong monad over $\hat{\mathcal{C}}$ and for all $a, b \in \mathcal{C}$ the following equation holds

$$\hat{\mathrm{t}}_{\mathrm{Y}a, \mathrm{Y}b} = \mathrm{Y}(\mathrm{t}_{a,b})$$

where we have implicitly assume that the Yoneda embedding preserves finite products on the nose, i.e. the following diagrams commute

$$\begin{tikzcd}
1 \arrow[r, "1"] \arrow[dr, "1"'] & \mathcal{C} \arrow[d, "\mathrm{Y}"] & \mathcal{C} \times \mathcal{C} \arrow[l, "\times"'] \arrow[d, "\mathrm{Y} \times \mathrm{Y}"] \\
 & \hat{\mathcal{C}} & \hat{\mathcal{C}} \times \hat{\mathcal{C}} \arrow[l, "\times"]
\end{tikzcd}$$

[9] This is a simplifying assumption. For our purposes it would be enough to have a natural isomorphism $\sigma: T; \mathrm{Y} \dot{\to} \mathrm{Y}; \hat{T}$, but then the remaining equations have to be patched. For instance, the equation relating η and $\hat{\eta}$ would become $\hat{\eta}_{\mathrm{Y}a} = \mathrm{Y}(\eta_a); \sigma_a$.

and for all $a, b \in \mathcal{C}$. the following equations hold

$$!_{\mathrm{Y}a} = \mathrm{Y}(!_a) \quad , \quad \pi_i^{\mathrm{Y}a,\mathrm{Y}b} = \mathrm{Y}(\pi_i^{a,b})$$

Definition 4.10 ([Mac71]) *Let $T: \mathcal{C} \to \mathcal{D}$ be a functor between two small categories and $\mathcal{A}$ a cocomplete category. Then, the* **left Kan extension** $\mathrm{L}_T^{\mathcal{A}}: \mathcal{A}^{\mathcal{C}} \to \mathcal{A}^{\mathcal{D}}$ *is the left adjoint of $\mathcal{A}^T$ and can be defined as follows:*

$$\mathrm{L}_T^{\mathcal{A}}(F)(d) = \mathrm{Colim}_{T\downarrow d}^{\mathcal{A}}(\pi; F)$$

where $F: \mathcal{C} \to \mathcal{A}$, $d \in \mathcal{D}$, $T \downarrow d$ is the comma category whose objects are pairs $\langle c \in \mathcal{C}, f: Tc \to d\rangle$, $\pi: T \downarrow d \to \mathcal{C}$ is the projection functor (mapping a pair $\langle c, f\rangle$ to c) and $\mathrm{Colim}_I^{\mathcal{A}}: \mathcal{A}^I \to \mathcal{A}$ (with I small category) is a functor mapping an I-diagram in $\mathcal{A}$ to its colimit.

The following proposition is a 2-categorical reformulation of Theorem 1.3.10 of [MR77]. For the sake of simplicity, we use the strict notions of 2-functor and 2-natural transformation, although we should have used pseudo-functors and pseudo-natural transformations.

Proposition 4.11 *Let* **Cat** *be the 2-category of small categories,* **CAT** *the 2-category of locally small categories and $\_: \mathbf{Cat} \to \mathbf{CAT}$ the inclusion 2-functor. Then, the following $\hat{\_}: \mathbf{Cat} \to \mathbf{CAT}$ is a 2-functor*

- *if $\mathcal{C}$ is a small category, then $\hat{\mathcal{C}}$ is the topos of presheaves $\mathbf{Set}^{\mathcal{C}^{op}}$*
- *if $T: \mathcal{C} \to \mathcal{D}$ is a functor, then $\hat{T}$ is the left Kan extension $\mathrm{L}_{T^{op}}^{\mathbf{Set}}$*
- *if $\sigma: S \dot{\to} T: \mathcal{C} \to \mathcal{D}$ is a natural transformation and $F \in \hat{\mathcal{C}}$, then $\hat{\sigma}_F$ is the natural transformation corresponding to $\mathrm{id}_{\hat{T}F}$ via the following sequence of steps*

$$\begin{tikzcd}
\hat{\mathcal{C}}(F, T^{op}; \hat{T}F) \arrow[d, maps to, "\hat{\mathcal{C}}(F, \sigma^{op}; \hat{T}F)"] & \hat{\mathcal{D}}(\hat{T}F, \hat{T}F) \arrow[l, maps to, "\sim"'] \\
\hat{\mathcal{C}}(F, S^{op}; \hat{T}F) \arrow[r, maps to, "\sim"] & \hat{\mathcal{D}}(\hat{S}F, \hat{T}F)
\end{tikzcd}$$

Moreover, $\mathrm{Y}: \_ \dot{\to} \hat{\_}$ is a 2-natural transformation.

Since monads are a 2-categorical concept (see [Str72]), the 2-functor $\hat{\_}$ maps monads in **Cat** to monads in **CAT**. Then, the statement of Theorem 4.9 about lifting of monads follows immediately from Proposition 4.11. It remains to define the lifting $\hat{\mathrm{t}}$ of a tensorial strength t for a monad (T, η, μ) over a small category $\mathcal{C}$.

Proposition 4.12 *If $\mathcal{C}$ is a small category with finite products and T is an endofunctor over $\mathcal{C}$, then for every natural transformation $\mathrm{t}_{a,b}: a \times Tb \to T(a \times b)$ there exists a unique natural transformation $\hat{\mathrm{t}}_{F,G}: F \times \hat{T}G \to \hat{T}(F \times G)$ s.t. $\hat{\mathrm{t}}_{\mathrm{Y}a,\mathrm{Y}b} = \mathrm{Y}(\mathrm{t}_{a,b})$ for all $a, b \in \mathcal{C}$.*

Proof Every $F \in \hat{\mathcal{C}}$ is isomorphic to the colimit $\mathrm{Colim}_{\mathrm{Y}\downarrow F}^{\hat{\mathcal{C}}}(\pi; \mathrm{Y})$ (shortly $\mathrm{Colim}_i \mathrm{Y}i$), where Y is the Yoneda embedding of $\mathcal{C}$ into $\hat{\mathcal{C}}$. Similarly G is isomorphic to $\mathrm{Colim}_j \mathrm{Y}j$. Both functors $(\_ \times \hat{T}\_)$ and $\hat{T}(\_ \times \_)$ from $\hat{\mathcal{C}} \times \hat{\mathcal{C}}$ to $\hat{\mathcal{C}}$ preserves colimits (as $\hat{T}$ and $\_ \times F$ are left adjoints) and commute with the Yoneda embedding (as $\mathrm{Y}(a \times b) = \mathrm{Y}a \times \mathrm{Y}b$ and $\hat{T}(\mathrm{Y}a) = \mathrm{Y}(Ta)$). Therefore, $F \times \hat{T}G$ and $\hat{T}(F \times G)$ are isomorphic to the colimits $\mathrm{Colim}_{i,j} \mathrm{Y}i \times \hat{T}(\mathrm{Y}j)$ and $\mathrm{Colim}_{i,j} \hat{T}(\mathrm{Y}i \times \mathrm{Y}j)$ respectively. Let $\hat{\mathrm{t}}$ be the natural transformation we are looking for, then

$$\begin{tikzcd}
\mathrm{Y}i \times \hat{T}(\mathrm{Y}j) \arrow[r, "\mathrm{Y}(\mathrm{t}_{i,j})"] \arrow[d, "f \times \hat{T}g"'] & \hat{T}(\mathrm{Y}i \times \mathrm{Y}j) \arrow[d, "\hat{T}(f \times g)"] \\
F \times \hat{T}(G) \arrow[r, "\hat{\mathrm{t}}_{F,G}"'] & \hat{T}(F \times G)
\end{tikzcd}$$

for all $f: \mathrm{Y}i \to F$ and $g: \mathrm{Y}j \to g$ (by naturality of $\hat{\mathrm{t}}$ and $\hat{\mathrm{t}}_{\mathrm{Y}i,\mathrm{Y}j} = \mathrm{Y}(\mathrm{t}_{i,j})$). But there exists exactly one morphism $\hat{\mathrm{t}}_{F,G}$ making the diagram above commute, as $\langle \mathrm{t}_{i,j} | i, j \rangle$ is a morphism between diagrams in $\hat{\mathcal{C}}$ of the same shape, and these diagrams have colimit cones $\langle f \times \hat{T} g | f, g \rangle$ and $\langle \hat{T}(f \times g) | f, g \rangle$ respectively. ∎

Remark 4.13 If T is a monad of partial computations, i.e. it is induced by a dominion $\mathcal{M}$ on $\mathcal{C}$ s.t. $\mathrm{P}(\mathcal{C}, \mathcal{M})(a, b) \cong \mathcal{C}(a, Tb)$, then the lifting $\hat{T}$ is the monad of partial computations induced by the dominion $\hat{\mathcal{M}}$ on $\hat{\mathcal{C}}$, obtained by lifting $\mathcal{M}$ to the topos of presheaves, as described in [Ros86]. For other monads, however, the lifting is not the *expected* one. For instance, if T is the monad of side-effects $(\_ \times S)^S$, then $\hat{T}$ is not (in general) the endofunctor $(\_ \times \mathrm{Y}S)^{\mathrm{Y}S}$ on the topos of presheaves.

Conclusions and further research

The main contribution of this paper is the category-theoretic semantics of computations and the general principle for extending it to more complex languages (see Remark 3.3 and Section 4), while the formal systems presented are a straightforward fallout, easy to understand and relate to other calculi.

Our work is just an example of what can be achieved in the study of programming languages by using a category-theoretic methodology, which avoids irrelevant syntactic detail and focus instead on the important structures underlying programming languages. We believe that there is a great potential to be exploited here. Indeed, in [Mog89b] we give a categorical account of phase distinction and program modules, that could lead to the introduction of higher order modules in programming languages like ADA or ML (see [HMM90]), while in [Mog89a] we propose a "modular approach" to Denotational Semantics based on the idea of monad-constructor (i.e. an endofunctor on the category of monads over a category $\mathcal{C}$).

The metalanguage open also the possibility to develop a new Logic of Computable Functions (see [Sco69]), based on an abstract semantic of computations rather than domain theory, for studying axiomatically different notions of computation and their relations. Some recent work by Crole and Pitts (see [CP90]) has consider an extension of the metalanguage equipped with a logic for inductive predicates, which goes beyond equational reasoning. A more ambitious goal would be to try exploiting the capabilities offered by higher-order logic in order to give a uniform account of various program logics, based on the idea of "T-modal operator" (see Definition 4.7).

The semantics of computations corroborates the view that (constructive) proofs and programs are rather unrelated, although both of them can be understood in terms of functions. Indeed, monads (and comonads) used to model logical modalities, e.g. possibility and necessity in modal logic or why not and of course of linear logic, usually do not have a tensorial strength. In general, one should expect types suggested by logic to provide a more fine-grained type system without changing the *nature* of computations.

We have identified monads as important to model notions of computations, but *computational monads* seem to have additional properties, e.g. they have a tensorial stregth and may satisfy the mono requirement. It is likely that there are other properties of computational monads still to be identified, and there is no reason to believe that such properties have to be found in the literature on monads.

Acknowledgements

I have to thank many people for advice, suggestions and criticisms, in particular: R. Amadio, R. Burstall, M. Felleisen, R. Harper, F. Honsell, M. Hyland, B. Jay, A. Kock, Y. Lafont, G. Longo, R. Milner, A. Pitts, G. Plotkin, J. Power and C. Talcott.

References

[BW85] M. Barr and C. Wells. *Toposes, Triples and Theories.* Springer Verlag, 1985.

[CP90] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations. In *4th LICS Conf.* IEEE, 1990.

[CS87] R.L. Constable and S.F. Smith. Partial objects in constructive type theory. In *2nd LICS Conf.* IEEE, 1987.

[CS88] R.L. Constable and S.F. Smith. Computational foundations of basic recursive function theory. In *3rd LICS Conf.* IEEE, 1988.

[FF89] M. Felleisen and D.P. Friedman. A syntactic theory of sequential state. *Theoretical Computer Science*, 69(3), 1989.

[FFKD86] M. Felleisen, D.P. Friedman, E. Kohlbecker, and B. Duba. Reasoning with continuations. In *1st LICS Conf.* IEEE, 1986.

[Fou77] M.P. Fourman. The logic of topoi. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic*. North Holland, 1977.

[GMW79] M.J.C. Gordon, R. Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[GS89] C. Gunter and S. Scott. Semantic domains. Technical Report MS-CIS-89-16, Dept. of Comp. and Inf. Science, Univ. of Pennsylvania, 1989. to appear in North Holland Handbook of Theoretical Computer Science.

[HMM90] R. Harper, J. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *17th POPL*. ACM, 1990.

[HP87] J.M.E. Hyland and A.M. Pitts. The theory of constructions: Categorical semantics and topos-theoretic models. In *Proc. AMS Conf. on Categories in Comp. Sci. and Logic (Boulder 1987)*, 1987.

[JP78] P.T. Johnstone and R. Pare, editors. *Indexed Categories and their Applications*, volume 661 of *Lecture Notes in Mathematics*. Springer Verlag, 1978.

[Kel82] G.M. Kelly. *Basic Concepts of Enriched Category Theory*. Cambridge University Press, 1982.

[Koc72] A. Kock. Strong functors and monoidal monads. *Archiv der Mathematik*, 23, 1972.

[KR77] A. Kock and G.E. Reyes. Doctrines in categorical logic. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic*. North Holland, 1977.

[Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59, 1988.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[Mac71] S. MacLane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Man76] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[Mas88] I.A. Mason. Verification of programs that destructively manipulate data. *Science of Computer Programming*, 10, 1988.

[Mog86] E. Moggi. Categories of partial morphisms and the partial lambda-calculus. In *Proceedings Workshop on Category Theory and Computer Programming, Guildford 1985*, volume 240 of *Lecture Notes in Computer Science*. Springer Verlag, 1986.

[Mog88] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988.

[Mog89a] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh Univ., Dept. of Comp. Sci., 1989. Lecture Notes for course CS 359, Stanford Univ.

[Mog89b] E. Moggi. A category-theoretic account of program modules. In *Proceedings of the Conference on Category Theory and Computer Science, Manchester, UK, Sept. 1989*, volume 389 of *Lecture Notes in Computer Science.* Springer Verlag, 1989.

[Mog89c] E. Moggi. Computational lambda-calculus and monads. In *4th LICS Conf.* IEEE, 1989.

[Mos89] P. Mosses. Denotational semantics. Technical Report MS-CIS-89-16, Dept. of Comp. and Inf. Science, Univ. of Pennsylvania, 1989. to appear in North Holland Handbook of Theoretical Computer Science.

[MR77] M. Makkai and G. Reyes. *First Order Categorical Logic.* Springer Verlag, 1977.

[MT89a] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *16th Colloquium on Automata, Languages and Programming.* EATCS, 1989.

[MT89b] I. Mason and C. Talcott. A sound and complete axiomatization of operational equivalence of programs with memory. In *POPL 89.* ACM, 1989.

[Plo75] G.D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1, 1975.

[Plo85] G.D. Plotkin. Denotational semantics with partial functions. Lecture Notes at C.S.L.I. Summer School, 1985.

[Ros86] G. Rosolini. *Continuity and Effectiveness in Topoi.* PhD thesis, University of Oxford, 1986.

[Sch86] D.A. Schmidt. *Denotational Semantics: a Methodology for Language Development.* Allyn & Bacon, 1986.

[Sco69] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Oxford notes, 1969.

[Sco79] D.S. Scott. Identity and existence in intuitionistic logic. In M.P. Fourman, C.J. Mulvey, and D.S. Scott, editors, *Applications of Sheaves*, volume 753 of *Lecture Notes in Mathematics.* Springer Verlag, 1979.

[Sco80] D.S. Scott. Relating theories of the λ-calculus. In R. Hindley and J. Seldin, editors, *To H.B. Curry: essays in Combinarory Logic, lambda calculus and Formalisms.* Academic Press, 1980.

[See87] R.A.G. Seely. Linear logic, $*$-autonomous categories and cofree coalgebras. In *Proc. AMS Conf. on Categories in Comp. Sci. and Logic (Boulder 1987)*, 1987.

[Sha84] K. Sharma. Syntactic aspects of the non-deterministic lambda calculus. Master's thesis, Washington State University, September 1984. available as internal report CS-84-127 of the comp. sci. dept.

[SP82] M. Smith and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

[Str72] R. Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2, 1972.