Computational lambda-calculus and monads

Eugenio Moggi*
LFCS
Dept. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
em@lfcs.ed.ac.uk

October 1988

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[1] is introduced, that may jeopardise the applicability of theoretical results to real situations. In this paper we introduce a new calculus based on a categorical semantics for *computations*. This calculus provides a correct basis for proving equivalence of programs, independent from any specific computational model.

1 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 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.

*On leave from Università di Pisa. Research partially supported by the Joint Collaboration Contract # ST2J-0374-C(EDB) of the EEC

[1] programs are identified with total functions from *values* to *values*

- 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 $\approx$ and the set Th of formulas valid in the intended model respectively), and they (especially the operational approach) deal with programming languages on a rather case-by-case basis.

On the other hand, the logical approach gives a logical consequence relation $\vdash$ ($Ax \vdash A$ 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 (this is my main contribution).

2. We show that w.l.o.g. one may consider only monads over a topos (because of certain properties of the Yoneda embedding), and therefore one can use higher order intuitionistic logic.

3. We investigate how datatypes, in particular products, relates to computations (previous work by category-theorists is particularly useful here).

At the end we get a formal system, the computational lambda-calculus (λ_c-calculus for short), similar to $PP\lambda$ (see [GMW79]) for proving **equivalence** and **existence** of programs, which is sound and complete w.r.t. the categorical semantics of computations. The methodology outlined above is inspired by [Sco80][2], in particular the view that "category theory comes, logically, before the λ-calculus" led us to consider a categorical semantics of computations first, rather than trying to *hack* directly on the rules of $\beta\eta$-conversion to get a *correct* calculus.

[2] "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*".

1.1 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]) and side-effects (see [FFKD86, MT89]).

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.

The denotational approach may suggest important principles, e.g. fix-point induction (see [Sco93, 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.

The approach adopted in this paper generalises the one followed in [Ros86, Mog86] to obtain the λ_p-calculus, i.e. the calculus for reasoning about *partial* computations (or equivalently, about partial functions). In fact, the λ_p-calculus (like the λ-calculus) amounts to a particular λ_c-*theory*.

A type theoretic approach to partial functions and computations is attempted in [CS87, CS88] by introducing a new type constructor $\bar{A}$, whose intuitive meaning is the set of *computations* of type A. However, Constable and Smith do not adequately capture the general axioms for (partial) computations as we (and [Ros86]) do, since they lack a general notion of model and rely only on domain- and recursion-theoretic intuition.

2 A categorical semantics of computations

The basic idea behind the semantics of programs described below is that a program denotes a morphism from A (the object of values of type A) to TB (the object of computations of type B). There are many possible choices for TB corresponding to different notions of computations, for instance in the category of sets the set of partial computations (of type B) is the lifting $B + \{\perp\}$ and the set of non-deterministic computations is the powerset $\mathcal{P}(B)$. Rather than focus on specific notions of computations, we will try to identify the general properties that the object TB of computations must have.

Definition 2.1
A **computational model** *is a monad* (T, η, μ) *over a category* $\mathcal{C}$*, i.e. a functor*

$T: \mathcal{C} \to \mathcal{C}$ and two natural transformations $\eta: \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu: T^2 \dot{\to} T$ s.t.

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

which satisfies also an extra **equalizing requirement***: $\eta_A: A \to TA$ is an equalizer of $\eta_T A$ and $T(\eta_A)$, i.e. for any $f: B \to TA$ s.t. $f; \eta_{TA} = f; T(\eta_A)$ there exists a unique $m: B \to A$ s.t. $f = m; \eta_A$[3].*

Remark 2.2 Intuitively $\eta_A: A \to TA$ gives the inclusion of values into computations, while $\mu_A: T^2A \to TA$ flatten a *computation of a computation* into a computation. However, it is the **equalizing requirement** which ensures that η_A is a (strong) mono rather than an arbitrary morphism.

According to the view of "programs as functions from values to computations" the natural category for interpreting programs is not $\mathcal{C}$, but the Kleisli category.

Definition 2.3 (see [Mac71])
Given a monad (T, η, μ) over $\mathcal{C}$, the **Kleisli category** *$\mathcal{C}_T$, is the category s.t. :*

- *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 $A \stackrel{\eta_A}{\to} TA$*
- *the composition of $f \in \mathcal{C}_T(A, B)$ and $g \in \mathcal{C}_T(B, C)$ in $\mathcal{C}_T$ is $A \stackrel{f}{\to} TB \stackrel{Tg}{\to} T^2C \stackrel{\mu_C}{\to} TC$*

Remark 2.4 Our view of programs corresponds to call-by-value parameter passing, but there is an alternative view of "programs as functions from computations to computations" corresponding to call-by-name (see [Plo75] and Section 5). In any case, the fundamental issue is that there is a subset of the computations, the values, which has special properties and should not be forgotten. By taking call-by-value we can stress better the importance of values. Moreover, call-by-name can be more easily *represented* in call-by-value than the other way around.

Before going into the details of the interpretation we consider some examples of computational models over the category of sets.

Example 2.5 non-deterministic computations:

---

[3] The other property for being an equalizer, namely $\eta_A; \eta_{TA} = \eta_A; T(\eta_A)$, follows from the naturality of η

- $T(\_)$ is the covariant powerset functor, i.e. $T(A) = \mathcal{P}(A)$ and $T(f)(X)$ is the image of X along f
- η_A is the singleton map $a \mapsto \{a\}$
- μ_A is the big union map $X \mapsto \cup X$

It is easy to check the equalizing requirement, in fact

$$\eta_{TA}: X \mapsto \{X\} \qquad T(\eta_A): X \mapsto \{\{x\} | x \in X\}$$

therefore $\eta_{TA}(X) = T(\eta_A)(X)$ iff X is a singleton.

Example 2.6 computations with side-effects:

- $T(\_)$ is the functor $(S \to (\_ \times S))$, where S is a nonempty set of *stores*. Intuitively a computation takes a store and returns a value together with the modified store.
- η_A is the map $a \mapsto (\lambda s: S.\langle a, s\rangle)$
- μ_A is the map $f \mapsto (\lambda s: S.\mathrm{eval}(fs))$, i.e. $\mu_A(f)$ is the computation that given a store s, first computes the pair computation-store $\langle f', s'\rangle = fs$ and then returns the pair value-store $\langle a, s''\rangle = f's'$.

One can verify for himself that other notions of computation (e.g. partial, probabilistic or non-deterministic with side-effects) fit in this general definition.

2.1 A simple language and its interpretation

The aim of this section is to focus on the crucial ideas of the interpretation, and the language has been oversimplified (for instance terms have exactly one free variable) in order to define its interpretation in any computational model without requiring any additional structure on it. However, richer languages, e.g. with product and functional types, will be considered in Section 3. The term language we introduce is parametric in a signature (i.e. a set of base types and unary function symbols), therefore its interpretation in a computational model (T, η, μ) over a category $\mathcal{C}$, is parametric in an interpretation of the symbols in the signature.

- Given an interpretation $[\![A]\!]$ for any base type A, i.e. an object of the Kleisli category $\mathcal{C}_T$, then the interpretation of a type $\tau ::= A \mid T\tau$ is an object $[\![\tau]\!]$ of $\mathcal{C}_T$ defined in the obvious way, namely $[\![T\tau]\!] = T[\![\tau]\!]$.
- Given an interpretation $[\![f]\!]$ for any unary function symbol f of arity $\tau_1 \to \tau_2$, i.e. a morphism from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$ in $\mathcal{C}_T$, then the interpretation of a well-formed term $x{:}\,\tau \vdash e{:}\,\tau'$ is a morphism $[\![x{:}\,\tau \vdash e{:}\,\tau']\!]$ from $[\![\tau]\!]$ to $[\![\tau']\!]$ in $\mathcal{C}_T$ defined by induction on the derivation of $x{:}\,\tau \vdash e{:}\,\tau'$ (see Table 1).
- On top of the term language we consider two atomic predicates: equivalence and existence (see Table 2).

RULE	SYNTAX	SEMANTICS
var	$\dfrac{}{x\colon\tau \vdash x\colon\tau}$	$= \eta_{[\![\tau]\!]}$
let	$\dfrac{x\colon\tau \vdash e_1\colon\tau_1 \quad x_1\colon\tau_1 \vdash e_2\colon\tau_2}{x\colon\tau \vdash (\mathrm{let}\, x_1{=}e_1\, \mathrm{in}\, e_2)\colon\tau_2}$	$= g_1$ $= g_2$ $= g_1; Tg_2; \mu_{[\![\tau_2]\!]}$ i.e. $g_1; g_2$ in the Kleisli category
$f\colon\tau_1 \to \tau_2$	$\dfrac{x\colon\tau \vdash e_1\colon\tau_1}{x\colon\tau \vdash f(e_1)\colon\tau_2}$	$= g_1$ $= g_1; T[\![f]\!]; \mu_{[\![\tau_2]\!]}$
$[\_]$	$\dfrac{x\colon\tau \vdash e\colon\tau'}{x\colon\tau \vdash [e]\colon T\tau'}$	$= g$ $= g; \eta_{T[\![\tau']\!]}$
μ	$\dfrac{x\colon\tau \vdash e\colon T\tau'}{x\colon\tau \vdash \mu(e)\colon\tau'}$	$= g$ $= g; \mu_{[\![\tau']\!]}$

Table 1: Terms and their interpretation

RULE	SYNTAX	SEMANTICS
eq	$\dfrac{x\colon\tau_1 \vdash e_1\colon\tau_2 \quad x\colon\tau_1 \vdash e_2\colon\tau_2}{x\colon\tau_1 \vdash e_1 = e_2\colon\tau_2}$	$= g_1$ $= g_2$ $\iff g_1 = g_2$
ex	$\dfrac{x\colon\tau_1 \vdash e\colon\tau_2}{x\colon\tau_1 \vdash e \downarrow \tau_2}$	$= g$ $\iff g$ factors through $\eta_{[\![\tau_2]\!]}$ i.e. there exists (unique) h s.t. $g = h; \eta_{[\![\tau_2]\!]}$

Table 2: Atomic assertions and their interpretation

Remark 2.7 The let-constructor is very important semantically, since it corresponds to composition in the Kleisli category $\mathcal{C}_T$. While substitution (of a variable with an expression denoting a value) corresponds to composition in $\mathcal{C}$.

In the λ-calculus ($\mathsf{let}\, x{=}e \,\mathsf{in}\, e'$) is usually treated as syntactic sugar for $(\lambda x.e')e$, and this can be done also in the λ_c-calculus (because of (β) in Table 8). However, we think that this is not the right way to proceed, because it amounts to *understanding* the let-constructor, which makes sense in any computational model, in terms of constructors that make sense only in λ_c*-models*. On the other hand, ($\mathsf{let}\, x{=}e \,\mathsf{in}\, e'$) cannot be *reduced* to the more basic substitution (i.e. $e'[x{:=}\,e]$) without collapsing $\mathcal{C}_T$ to $\mathcal{C}$.

Remark 2.8 The existence of e does not simply means that the computation denoted by e terminates (as, say, in the logic of partial terms), but something stronger, namely that e denotes a value. For instance:

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

According to the paradigm of Categorical Logic, formulas should be interpreted by subobjects. This can be achieved by interpreting the binary predicate $\_ \equiv \_ \colon \tau$, i.e. equality of computations of type τ, by the diagonal $\Delta_{T[\![\tau]\!]}$ and the unary predicate $\_ \downarrow \tau$, i.e. existence of computations of type τ, by $\eta_{[\![\tau]\!]}$, which is a mono because of the equalizing requirement.

2.2 Embedding of a computational model in a topos

We show that any computational model (T, η, μ) over a small category $\mathcal{C}$ can be lifted to a computational model $(\hat{T}, \hat{\eta}, \hat{\mu})$ over the topos $\hat{\mathcal{C}}$ of presheaves (i.e. the functor category $\mathbf{Set}^{\mathcal{C}^{op}}$), and that such a lifting commutes with the Yoneda embedding Y of $\mathcal{C}$ into $\hat{\mathcal{C}}$, i.e.

$$\hat{T}(\mathrm{Y}\_) = \mathrm{Y}(T\_) \quad , \quad \hat{\eta}_{\mathrm{Y}\_} = \mathrm{Y}(\eta\_) \quad , \quad \hat{\mu}_{\mathrm{Y}\_} = \mathrm{Y}(\mu\_)$$

As pointed out in [Sco80] such an embedding enable us to switch from the equational (and rather inexpressive) calculus of an arbitrary computational model to the intuitionistic higher-order logic of (a computational model over) a topos.

The monad $(\hat{T}, \hat{\eta}, \hat{\mu})$ is defined by using the **Yoneda embedding** $\mathrm{Y}\colon \mathcal{C} \to \hat{\mathcal{C}}$ and $\mathrm{Lan}_{\mathrm{Y}}$, i.e. the left adjoint to $\mathrm{Y}; \_ \colon \hat{\mathcal{C}}^{\hat{\mathcal{C}}} \to \hat{\mathcal{C}}^{\mathcal{C}}$ mapping any $F\colon \mathcal{C} \to \hat{\mathcal{C}}$ to its **left Kan extension**[4] along Y (see [Mac71]), namely:

$$\hat{T} = \mathrm{Lan}(T; \mathrm{Y}) \quad , \quad \hat{\eta} = \mathrm{Lan}(\eta; \mathrm{Y}) \quad , \quad \hat{\mu} = \mathrm{Lan}(\mu; \mathrm{Y})$$

The commutativity with the Yoneda embedding (stated above) and the fact that Y induces a full and faithful embedding Y of $\mathcal{C}_T$ into $\hat{\mathcal{C}}_{\hat{T}}$ follow from some well-known properties of Y and $\mathrm{Lan}_{\mathrm{Y}}$, summarised in the following lemma:

Lemma 2.9 *If* $\mathcal{C}$ *is small category, then* $\mathrm{Y}\colon \mathcal{C} \to \hat{\mathcal{C}}$ *and* $\mathrm{Lan}_{\mathrm{Y}}\colon \hat{\mathcal{C}}^{\mathcal{C}} \to \hat{\mathcal{C}}^{\hat{\mathcal{C}}}$ *are full and faithful. Moreover* $\mathrm{Y}; \mathrm{Lan}_{\mathrm{Y}} F = F$ *for every* $F\colon \mathcal{C} \to \hat{\mathcal{C}}$.

[4] the left adjoint Lan_Y exists because **Set** is small cocomplete

3 Extending the language

In this section we discuss how to interpret terms with any finite number of variables (instead of exactly one as in Table 1) and how datatypes relate to computations. We will consider only product and functional types, because sum types are completely straightforward[5]. This will allow a comparison with cartesian closed categories (ccc) and partial cartesian closed categories (pccc).

The standard requirement on a category for interpreting terms with any finite number of variables is that it must have finite products, so that the interpretation $[\![f]\!]$ of a function symbol f of arity $\overline{\tau} \to \tau$ is a morphism from $[\![\times(\overline{\tau})]\!]$ (i.e. $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$) to $[\![\tau]\!]$ and similarly the interpretation of a well-formed term $x_1\colon\tau_1, \ldots, x_n\colon\tau_n \vdash e\colon\tau$ is a morphism from $[\![\times(\overline{\tau})]\!]$ to $[\![\tau]\!]$.

According to the view of "programs as functions from values to computations", products should be taken in $\mathcal{C}$, since a value of type $A \times B$ is a pair of values one of type A and the other of type B, even though the natural category for interpreting programs is $\mathcal{C}_T$. However, products are not enough to extend the interpretation to terms with more than one free variable, because we must be able to take a pair value-computation or computation-computation and turn it into a computation of a pair.

Example 3.1 Let $g_2\colon\tau_1 \to T\tau_2$ and and $g\colon\tau_1 \times \tau_2 \to T\tau$ be the interpretations of $x_1\colon\tau_1 \vdash e_2\colon\tau_2$ and $x_1\colon\tau_1, x_2\colon\tau_2 \vdash e\colon\tau$ respectivelly. The problem with terms having more than one free variable (and its solution) becomes apparent if we try to interpret $x_1\colon\tau_1 \vdash (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)\colon\tau$, when both x_1 and x_2 are free in e.

If T were $\mathrm{Id}_{\mathcal{C}}$, then $[\![x_1\colon\tau_1 \vdash (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)\colon\tau]\!]$ would be $\langle \mathrm{id}_{\tau_1}, g_2\rangle; g$. In the general case, Table 1 says that $\_;\_$ above is indeed composition in the Kleisli category, therefore $\langle \mathrm{id}_{\tau_1}, g_2\rangle; g$ becomes $\langle \mathrm{id}_{\tau_1}, g_2\rangle; Tg; \mu_\tau$. But in $\langle \mathrm{id}_{\tau_1}, g_2\rangle; Tg; \mu_\tau$ there is a type mismatch, since the codomain of $\langle \mathrm{id}_{\tau_1}, g_2\rangle$ is $\tau_1 \times T\tau_2$, while the domain of Tg is $T(\tau_1 \times \tau_2)$. To get around this we require T to have a tensorial strength $\mathrm{t}_{A,B}\colon A \times TB \to T(A \times B)$ (see below), so that $x_1\colon\tau_1 \vdash (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)\colon\tau$ will be interpreted by $\langle \mathrm{id}_{\tau_1}, g_2\rangle; \mathrm{t}_{\tau_1,\tau_2}; Tg; \mu_\tau$.

Similarly for interpreting $x\colon\tau \vdash f(e_1, e_2)\colon\tau'$, we need a natural transformation $\psi_{A,B}\colon(TA \times TB) \to T(A \times B)$ (see Definition 3.4), which given a pair of programs returns a program computing a pair. More precisely, let $g_i\colon\tau \to T\tau_i$ be the interpretation of $x\colon\tau \vdash e_i\colon\tau_i$, then $[\![x\colon\tau \vdash f(e_1, e_2)\colon\tau']\!]$ is $\langle g_1, g_2\rangle; \psi_{\tau_1,\tau_2}; T[\![f]\!]; \mu$.

Definition 3.2 *Let $\mathcal{C}$ be a category with finite products, and r_A, $\alpha_{A,B,C}$ and $c_{A,B}$ be the natural isomorphisms:*

$$(1 \times A) \xrightarrow{r} A \quad , \quad (A \times B) \times C \xrightarrow{\alpha} A \times (B \times C) \quad , \quad (A \times B) \xrightarrow{c} (B \times A)$$

A **computational cartesian model** *over $\mathcal{C}$ is a computational model (T, η, μ) over $\mathcal{C}$ together with a* **tensorial strength** $\mathrm{t}_{A,B}\colon(A \times TB) \to T(A \times B)$ *of T, i.e. a natural*

[5] coproducts are preserved by the inclusion of $\mathcal{C}$ into the Kleisli category $\mathcal{C}_T$

transformation s.t.

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

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

satisfying the following diagrams:

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

Remark 3.3 In general the tensorial strength t has to be given as an extra parameter for models. 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, i.e. if $f, g: A \to B$, then $f = g \longleftrightarrow (\forall h: 1 \to A.h; f = h; g)$.

The diagrams above are not new, they are all in [Koc70b], where a one-one correspondence is established between *functorial* and *tensorial strengths*[6]:

- the first two diagrams, saying that t is a tensorial strength of T, are (1.7) and (1.8) in [Koc70b]. By Theorem 1.3 in [Koc70b] t induces a functorial strength of T making T a $\mathcal{C}$-enriched (also called strong) functor.
- the last two diagrams say that η and μ are natural transformations between suitable $\mathcal{C}$-enriched functors, namely $\eta: \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu: T^2 \dot{\to} T$ (see Remark 1.5 in [Koc70b]).

[6]If V is a monoidal closed category, then a **functorial strength** of an endofunctor T on V is a natural transformation $\mathrm{st}_{A,B}: B^A \to TB^{TA}$ making T a V-enriched functor. Intuitively st *internalizes* the action of T on morphisms.

Definition 3.4 *The tensorial strength* t *induces a* **monoidal structure**, *i.e. a natural transformation* $\psi_{A,B}\colon (TA \times TB) \to T(A \times B)$ *and a map* $\psi_1\colon 1 \to T1$

$$\psi_{A,B} = c_{TA,TB}; \mathrm{t}_{TB,A}; T(c_{TB,A}; \mathrm{t}_{A,B}); \mu_{A\times B} \quad , \quad \psi_1 = \eta_1$$

satisfying certain diagrams (see [EK66]).

The morphism $\psi_{A,B}\colon (TA \times TB) \to T(A \times B)$ 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), while the morphism ψ_1 interprets the computation of $\langle\rangle$ (the empty tuple). There is also a dual notion of pairing, namely $\tilde{\psi}_{A,B} = c_{A,B}; \psi_{B,A}; Tc_{B,A}$, which amounts to first evaluating the second argument and then the first (see (2.1) and (2.2) at page 14 in [Koc70b]).

The categorical interpretation of functional types in a computational model resembles that of partial function spaces in a pccc (see [Ros86, Mog86]):

Definition 3.5 *Let $\mathcal{C}$ be a category with finite products. A* λ_c**-model** *over $\mathcal{C}$ is a computational cartesian model $(T, \eta, \mu, \mathrm{t})$ over $\mathcal{C}$ together with a family of universal arrows* $\mathrm{eval}^T_{A,B}\colon (B^A_T \times A) \to TB$ *(in $\mathcal{C}$) s.t. for any $f\colon (C \times A) \to TB$ there exists a unique $h\colon C \to B^A_T$ (denoted by $\Lambda^T_{A,B,C}(f)$) making the following diagram commute*

$$\begin{tikzcd}
B^A_T \times A \arrow[r, "\mathrm{eval}^T_{A,B}"] & TB \\
C \times A \arrow[u, "h \times \mathrm{id}_A"] \arrow[ur, "f"'] &
\end{tikzcd}$$

A more suggestive way of saying the same thing is that there is a natural isomorphism $\mathcal{C}_T(C \times A, B) \cong \mathcal{C}(C, B^A_T)$, where A, B and C vary over $\mathcal{C}^{op}$, $\mathcal{C}_T$ and $\mathcal{C}$ respectively.

The simple language introduced in Section 2.1 and its interpretation can be extended according to the additional structure available in a cartesian computational model $(T, \eta, \mu, \mathrm{t})$ on a category $\mathcal{C}$ with finite products:

- there is a new type 1, interpreted by the terminal object of $\mathcal{C}$, and a new type constructor $\tau_1 \times \tau_2$ interpreted by the product of $[\![\tau_1]\!]$ and $[\![\tau_2]\!]$ in $\mathcal{C}$
- the interpretation of a well-formed term $\Gamma \vdash e\colon \tau$, where Γ is a sequence $x_1\colon \tau_1, \ldots, x_n\colon \tau_n$, is a morphism from $[\![\Gamma]\!]$ (i.e. $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$) to $[\![\tau]\!]$ in $\mathcal{C}_T$ (see Table 3)[7].

In a λ_c-model the interpretation can be extended to functional types and λ-terms, namely: the type $\tau_1 \rightharpoonup \tau_2$ is interpreted by $[\![\tau_2]\!]^{[\![\tau_1]\!]}_T$, while abstraction and application are interpreted as in Table 4.

[7] We do not have to consider nonunary functions explicitly, because in a language with products they can be treated as unary functions from a product type.

RULE	SYNTAX		SEMANTICS
var	$\dfrac{}{x_1\colon\tau_1,\ldots,x_n\colon\tau_n \vdash x_i\colon\tau_i}$	$=$	$\pi^n_i;\eta_{[\![\tau_i]\!]}$
let	$\Gamma\vdash e_1\colon\tau_1$	$=$	g_1
	$\Gamma,x_1\colon\tau_1\vdash e_2\colon\tau_2$	$=$	g_2
	$\Gamma\vdash(\mathrm{let}\,x_1{=}e_1\,\mathrm{in}\,e_2)\colon\tau_2$	$=$	$\langle\mathrm{id}_{[\![\Gamma]\!]},g_1\rangle;\mathrm{t}_{[\![\Gamma]\!],[\![\tau_1]\!]};Tg_2;\mu_{[\![\tau_2]\!]}$
$*$	$\dfrac{}{\Gamma\vdash *\colon 1}$	$=$	$!_{[\![\Gamma]\!]};\eta_1$
			where $!_A$ is the only morphism from A to 1
$\langle\rangle$	$\Gamma\vdash e_1\colon\tau_1$	$=$	g_1
	$\Gamma\vdash e_2\colon\tau_2$	$=$	g_2
	$\Gamma\vdash\langle e_1,e_2\rangle\colon\tau_1\times\tau_2$	$=$	$\langle g_1,g_2\rangle;\psi_{[\![\tau_1]\!],[\![\tau_2]\!]}$
π_i	$\Gamma\vdash e\colon\tau_1\times\tau_2$	$=$	g
	$\Gamma\vdash\pi_i(e)\colon\tau_1$	$=$	$g;T(\pi_i)$

Table 3: Terms and their interpretation

RULE	SYNTAX		SEMANTICS
λ	$\Gamma,x_1\colon\tau_1\vdash e_2\colon\tau_2$	$=$	g
	$\Gamma\vdash(\lambda x_1\colon\tau_1.e_2)\colon\tau_1\rightharpoonup\tau_2$	$=$	$\Lambda^T_{[\![\tau_1]\!],[\![\tau_2]\!],[\![\Gamma]\!]}(g);\eta_{[\![\tau_1\rightharpoonup\tau_2]\!]}$
app	$\Gamma\vdash e_1\colon\tau_1$	$=$	g_1
	$\Gamma\vdash e\colon\tau_1\rightharpoonup\tau_2$	$=$	g
	$\Gamma\vdash e(e_1)\colon\tau_2$	$=$	$\langle g,g_1\rangle;\mathrm{app}_{[\![\tau_1]\!],[\![\tau_2]\!]}$
	where $\mathrm{app}_{A,B}\colon T(B^A_T)\times TA\to TB$ is $\psi_{B^A_T,A};T(\mathrm{eval}^T_{A,B});\mu_B$		

Table 4: λ-terms and their interpretation

3.1 Examples

In this section we show few general ways of constructing computational models from simpler ones. Each of them amounts to *adding* a new *feature* to computations.

Example 3.6 Let $(T, \eta, \mu, \mathrm{t})$ be a cartesian computational model on a topos (for simplicity **Set**), then the following are cartesian computational models:

- Let S be inhabited (i.e. $1 \lhd S$), then the model $(T_S, \eta^S, \mu^S, \mathrm{t}^S)$ of T-computations **with side-effects** in S is

  $T_S(\_) = (\_ \times S)^S_T$
  $\eta^S_A = \Lambda^T_{S,(A\times S),A}(\eta_{A\times S})$
  $\mu^S_A = \Lambda^T_{S,(A\times S),(T^2_S A)}(\mathrm{eval}^T_{S,(T_S A\times S)}; T(\mathrm{eval}^T_{S,(A\times S)}); \mu_{A\times S})$
  $\mathrm{t}^S_{A,B} = \Lambda^T_{S,(A\times S),(A\times T_S B)}(\alpha_{A,T_S B,S}; (\mathrm{id}_A \times \mathrm{eval}^T_{S,(B\times S)}); \mathrm{t}_{A,B\times S}; T(\alpha^{-1}_{A,B,S}))$

- the model $(T_E, \eta^E, \mu^E, \mathrm{t}^E)$ of T-computations **with exceptions** in E is

  $T_E(\_) = T(\_ + E)$
  $\eta^E_A = \mathrm{in}_1; \eta_{A+E}$
  $\mu^E_A = T([\mathrm{id}_{T(A+E)}, \mathrm{in}_2; \eta_{A+E}]); \mu_{A+E}$
  $\mathrm{t}^E_{A,B} = \mathrm{t}_{A,B+E}; T(d_{A,B,E}; [\mathrm{id}_{A\times B}, \pi_2])$

  where $A \stackrel{\mathrm{in}_1}{\to} A + B \stackrel{\mathrm{in}_2}{\leftarrow} B$ is a coproduct diagram,
  $[f, g]: A + B \to C$ is the mediating morphism of $f: A \to C$ and $g: B \to C$, i.e. the unique $h: A + B \to C$ s.t. $f = \mathrm{in}_1; h$ and $g = \mathrm{in}_2; h$,
  $d_{A,B,C}$ is the natural isomorphism $A \times (B + C) \stackrel{d}{\to} (A \times B) + (A \times C)$ expressing commutativity of coproducts w.r.t. products[8]

These constructions provide basic building blocks, that can be combined together for instance:

- $T_{E\,S}(\_) = T((\_ \times S) + E)^S$ and $T_{S\,E}(\_) = T((\_ + E) \times S)^S$ combine side-effects and exceptions. In the former the store is lost, when an exception is raised, while in the latter it is retained.

- If T is the monad of R-**continuations**[9], i.e. $T(\_) = R^{R^{(\_)}}$, then the monad $T_S(A) = R^{S\times(R^{A\times S})}$ combines continuation and side-effects as done when giving the denotational semantics of imperative languages with goto.

Monad-morphisms provide a simple tool for relating two computational models:

Definition 3.7
Given two cartesian computational models $(T, \eta^T, \mu^T, \mathrm{t}^T)$ *and* $(S, \eta^S, \mu^S, \mathrm{t}^S)$ *over the*

[8] which holds in cartesian closed categories, but not in general

[9] It is not clear what properties R must have in order for the monad T to satisfy the equalizing requirement. Intuitively one expects that the category $\mathcal{C}$ must have **enough** R**-observations**, i.e. $f = g \longleftrightarrow (\forall h: B \to R. f; h = g; h)$ for any $f, g: A \to B$

same category, a **monad-morphism** *from the first to the second model is a natural transformation* $\sigma: T \dot{\to} S$ *s.t. :*

$$\begin{tikzcd} A \arrow[r, "\eta^T_A"] \arrow[d, "\mathrm{id}_A"'] & TA \arrow[d, "\sigma_A"'] & T^2A \arrow[l, "\mu^T_A"'] \arrow[d, "\sigma^2_A"] \\ A \arrow[r, "\eta^S_A"'] & TA & T^2A \arrow[l, "\mu^S_A"] \end{tikzcd} \qquad \begin{tikzcd} A \times TB \arrow[r, "\mathrm{t}^T_{A,B}"] \arrow[d, "A \times \sigma_B"'] & T(A \times B) \arrow[d, "\sigma_{A \times B}"] \\ A \times SB \arrow[r, "\mathrm{t}^S_{A,B}"'] & S(A \times B) \end{tikzcd}$$

where σ^2 *is the horizontal composition, i.e.* $\sigma^2_A = T(\sigma_A); \sigma_{SA} = \sigma_{TA}; S(\sigma_A)$.

Example 3.8 For each of the computational model constructions defined above there is a monad morphism from T to it, namely:

- $\sigma^S: T \dot{\to} T_S$ is the natural transformation s.t. σ^S_A is $\Lambda^T_{S, A\times S, T(A)}(\mathrm{t}_{A,S})$
- $\sigma^E: T \dot{\to} T_E$ is the natural transformation s.t. σ^E_A is $T(\mathrm{in}_1^{A,E})$

Monad-morphisms are not adequate for relating λ_c-models, because the natural transformation σ cannot be *extended* to functional types. Instead, one can use a notion of *logical relation* between λ_c-models (see [Mog88] for various notions of logical relation between λ_p-models).

4 The λ_c-calculus

In this section we present a formal system, the λ_c**-calculus**, based on many sorted intuitionistic logic with two atomic predicates, existence and equivalence.

We claim that the formal system is **sound** and **complete** w.r.t. λ_c-models (over toposes). Soundness amounts to showing that the inference rules are admissible in any λ_c-model, while completeness amounts to showing that any λ_c-theory has an *initial model* (given by a term-model construction).

The inference rules of the λ_c-calculus are for deriving sequents $\Gamma.\Delta \vdash A$, where Γ is a sequence of type assignments $x: \tau$, Δ is a set of formulas and A is a formula s.t. the free variables $\mathrm{FV}(\Delta, A)$ of Δ and A are included in the declared variables $\mathrm{DV}(\Gamma)$ of Γ. The intuitive meaning of $\Gamma.\Delta \vdash A$ is: "for all variables in Γ, if all formulas in Δ are true, then A is true". We have intentionally left the set of formulas unspecified, since it depends on what class of models one is interested in. There is a minimal and maximal choice for the set of formulas:

- if the language has to be interpreted in any λ_c-model, then only atomic formulas (including $e \equiv e': \tau$ and $e \downarrow \tau$) are allowed
- if the language has to be interpreted only in λ_c-model over a topos, then all higher order formulas are allowed.

The inference rules are partitioned as follows:

- general rules for (higher order) intuitionistic logic, where variables range over values, while terms denotes computations (see Table 5 for the most relevant rules)[10]
- the basic inference rules for computational models (see Table 6)
- the inference rules for product types (see Table 7)
- the inference rules for functional types (see Table 8)

Remark 4.1 A comparison among λ_c-, λ_v- and λ_p-calculus shows that:

- the λ_v-calculus proves less equivalences between λ-terms, e.g. $(\lambda x.x)(yz) \equiv (yz)$ is provable in the λ_c- but not in the λ_v-calculus
- the λ_p-calculus proves more equivalences between λ-terms, e.g. $(\lambda x.yz)(yz) \equiv (yz)$ is provable in the λ_p- but not in the λ_c-calculus, because y can be a procedure, which modifies the store (e.g. by increasing the value contained in a local static variable) each time it is executed.
- a λ-term e **has a value** in the λ_c-calculus, i.e. e is provably equivalent to some **value** (either a variable or a λ-abstraction), iff e has a value in the λ_v-calculus (λ_p-calculus)

5 Untyped λ_c-models

It is well-known that a categorical model for the untyped λ-calculus is a reflexive object $D^D \cong D$ in a cartesian closed category (see [Sco80, Bar82]). In a λ_c-model there are two analogs for a reflexive object: $V_T^V \cong V$ and $N_T^{TN} \cong N$ (see [Ong88] for similar definitions in the context of partial cartesian closed categories).

In the first case we have a *model* of call-by-value. In fact the elements of V correspond to functions from values to computations (as V_T^V stands for V^{TV}), and therefore an element can be applied to a computation e only after e has been *evaluated*. In the second case we have a *model* of call-by-name, since the elements of N correspond to functions from computations to computations.

The call-by-value and call-by-name interpretations are defined by induction on the derivation of the untyped λ-term $x_1, \ldots, x_n \vdash e$ (with let):

- Let $G: V_T^V \to V$ be an isomorphism with inverse F, then the call-by-value interpretation of $x_1, \ldots, x_n \vdash e$ is a morphism from V^n to TV (see Table 9), because free variables range over values.

[10]The general rules of sequent calculus (in [Sza69]), more precisely those for substitution and quantifiers, have to be modified slightly, because variables range over values. These modifications are similar to those introduced in the logic of partial terms (see Section 2.4 in [Mog88]).

We write $\_[x{:}=e]$ for the substitution of x with e in $\_$.

$$\text{E}.x \quad \frac{}{\Gamma.\Delta \vdash x \downarrow \tau}$$

$$\text{subst} \quad \frac{\Gamma.\Delta \vdash e \downarrow \tau \qquad \Gamma, x{:}\,\tau.\Delta \vdash A}{\Gamma.\Delta \vdash A[x{:}=e]}$$

$\equiv$ is an equivalence relation

$$\text{congr} \quad \frac{\Gamma.\Delta \vdash e_1 = e_2{:}\,\tau \qquad \Gamma.\Delta \vdash A[x{:}=e_1]}{\Gamma.\Delta \vdash A[x{:}=e_2]}$$

Table 5: General rules

We write $(\text{let}\,\overline{x}{=}\overline{e}\,\text{in}\,e)$ for $(\text{let}\,x_1{=}e_1\,\text{in}\,(\ldots(\text{let}\,x_n{=}e_n\,\text{in}\,e)\ldots))$, where n is the lenght of the sequence $\overline{x}$ (and $\overline{e}$). In particular, $(\text{let}\,\emptyset{=}\emptyset\,\text{in}\,e)$ stands for e.

$$\text{id} \quad \frac{}{\Gamma.\Delta \vdash (\text{let}\,x{=}e\,\text{in}\,x) = e{:}\,\tau}$$

$$\text{comp} \quad \frac{}{\Gamma.\Delta \vdash (\text{let}\,x_2{=}(\text{let}\,x_1{=}e_1\,\text{in}\,e_2)\,\text{in}\,e) = (\text{let}\,x_1{=}e_1\,\text{in}\,(\text{let}\,x_2{=}e_2\,\text{in}\,e)){:}\,\tau} \quad x_1 \notin \text{FV}(e)$$

$$\text{let}.\xi \quad \frac{\Gamma.\Delta \vdash e_1 = e_1'{:}\,\tau \qquad \Gamma, x{:}\,\tau.\Delta \vdash e_2 = e_2'{:}\,\tau'}{\Gamma.\Delta \vdash (\text{let}\,x{=}e_1\,\text{in}\,e_2) = (\text{let}\,x{=}e_1'\,\text{in}\,e_2'){:}\,\tau'}$$

$$\text{let}.\beta \quad \frac{}{\Gamma.\Delta \vdash (\text{let}\,x_1{=}x_2\,\text{in}\,e) = e[x_1{:}=x_2]{:}\,\tau}$$

$$\text{let}.f \quad \frac{}{\Gamma.\Delta \vdash f(\overline{e}) = (\text{let}\,\overline{x}{=}\overline{e}\,\text{in}\,f(\overline{x})){:}\,\tau}$$

$$\text{E}.[\_] \quad \frac{}{\Gamma.\Delta \vdash [e] \downarrow T\tau}$$

$$T.\xi \quad \frac{\Gamma.\Delta \vdash e = e'{:}\,\tau}{\Gamma.\Delta \vdash [e] = [e']{:}\,T\tau}$$

$$\text{let}.\mu \quad \frac{}{\Gamma.\Delta \vdash \mu(e) = (\text{let}\,x{=}e\,\text{in}\,\mu(x)){:}\,\tau}$$

$$T.\beta \quad \frac{}{\Gamma.\Delta \vdash \mu([e]) = e{:}\,\tau}$$

$$T.\eta \quad \frac{}{\Gamma.\Delta \vdash [\mu(x)] = x{:}\,T\tau}$$

$\Gamma.\Delta \vdash e \downarrow \tau$ and $\Gamma.\Delta \vdash [e] = (\text{let}\,x{=}e\,\text{in}\,[x]){:}\,\tau$ are interderivable

Table 6: rules for let and computational types

$$\mathrm{E.}* \quad \frac{}{\Gamma.\Delta \vdash * \downarrow 1}$$

$$1.\eta \quad \frac{}{\Gamma.\Delta \vdash * = x\colon 1}$$

$$\mathrm{E.}\langle\_\rangle \quad \frac{}{\Gamma.\Delta \vdash \langle x_1, x_2\rangle \downarrow \tau_1 \times \tau_2}$$

$$\mathrm{let.}\langle\_\rangle \quad \frac{}{\Gamma.\Delta \vdash \langle e_1, e_2\rangle = (\mathrm{let}\, x_1, x_2{=}e_1, e_2 \,\mathrm{in}\, \langle x_1, x_2\rangle)\colon \tau_1 \times \tau_2}$$

$$\mathrm{E.}\pi_i \quad \frac{}{\Gamma.\Delta \vdash \pi_i(x) \downarrow \tau_i}$$

$$\mathrm{let.}\pi_i \quad \frac{}{\Gamma.\Delta \vdash \pi_i(e_1, e_2) = (\mathrm{let}\, x_1, x_2{=}e_1, e_2 \,\mathrm{in}\, \pi_i(x_1, x_2))\colon \tau_i}$$

$$\times.\beta \quad \frac{}{\Gamma.\Delta \vdash \pi_i(\langle x_1, x_2\rangle) = x_i\colon \tau_i}$$

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

Table 7: rules for unit and product types

$$\xi \quad \frac{\Gamma, x\colon\tau.\Delta \vdash e = e'\colon \tau'}{\Gamma.\Delta \vdash (\lambda x\colon\tau.e) = (\lambda x\colon\tau.e')\colon \tau \rightharpoonup \tau'} \quad x \notin \mathrm{FV}(\Delta)$$

$$\mathrm{E.}\lambda \quad \frac{}{\Gamma.\Delta \vdash (\lambda x\colon\tau_1.e) \downarrow \tau_1 \rightharpoonup \tau_2}$$

$$\mathrm{let.app} \quad \frac{}{\Gamma.\Delta \vdash e(e_1) = (\mathrm{let}\, x, x_1{=}e, e_1 \,\mathrm{in}\, x(x_1))\colon \tau_2}$$

$$\beta \quad \frac{}{\Gamma.\Delta \vdash (\lambda x_1\colon\tau_1.e_2)(x_1) = e_2\colon \tau_2}$$

$$\eta \quad \frac{}{\Gamma.\Delta \vdash (\lambda x_1\colon\tau_1.x(x_1)) = x\colon \tau_1 \rightharpoonup \tau_2}$$

Table 8: rules for functional types

Application call-by-value $\mathrm{app}_v: TV \times TV \to TV$ is *strict* in both arguments:

$$\mathrm{app}_v = \psi_{V,V}; T((F \times \mathrm{id}_V); \mathrm{eval}^T_{V,V}); \mu_V$$

- Let $G: N_T^{TN} \to N$ be an isomorphism with inverse F, then the call-by-name interpretation of $x_1, \ldots, x_n \vdash e$ is a morphism from $(TN)^n$ to TN (see Table 10), because free variables range over computations.

  Application call-by-name $\mathrm{app}_n: TN \times TN \to TN$ is *strict* in the first argument but *lazy* on the second:

$$\mathrm{app}_n = c_{TN,TN}; \mathrm{t}_{TN,N}; T(c_{TN,N}); T((F \times \mathrm{id}_{TN}); \mathrm{eval}^T_{TN,N}); \mu_N$$

Remark 5.1 In call-by-value $(\mathrm{let}\, x{=}e \,\mathrm{in}\, e')$ is equivalent to $(\lambda x.e')(e)$, but in call-by-name there is no way of expressing $(\mathrm{let}\, x{=}e \,\mathrm{in}\, e')$ in terms of application and abstraction only, because e is evaluated before binding its value to x (see [Ong88] for an analysis of call-by-name for partial computations).

We think that it is desirable (and very natural) for a programming language to have a let, which forces evaluation of an expression. We conjecture that the $\lambda\beta$-calculus (i.e. Plotkin's call-by-name λ-calculus) proves exactly those equivalences between untyped λ-terms without let that are true in any model of call-by-name $N_T^{TN} \cong N$[11].

6 Reduction

The *syntactic aspects* of the λ_c-calculus can be studied according to the same pattern used for the λ-calculus and the λ_v-calculus (see Chapter 3 of [Bar84] and [Plo75]). For simplicity we consider only untyped λ-terms with let-constructor.

In order to define the notions of reduction we need to distinguish between two kind of terms: values and nonvalues. The notion of value is that introduced in [Plo75] and gives a sufficient (syntactic) criteria for a term to denote a value.

Definition 6.1 (Basics)

- Terms, Values *and* NonValues *are the sets defined by the following bnfs*

$$e \in \mathrm{Terms}{::=}\, v | nv$$

$$v \in \mathrm{Values}{::=}\, x | (\lambda x.e)$$

$$nv \in \mathrm{NonValues}{::=}\, (\mathrm{let}\, x{=}e \,\mathrm{in}\, e') | e(e')$$

- *A binary relation* $\to$ *over* Terms, *is* **compatible** *iff for all* $M \to N$ *and* $P \in$ Terms

[11] This is obviously true if we allow $N_T^{TN} \lhd N$.

RULE	SYNTAX		SEMANTICS
var	$\overline{}$		
	$x_1,\ldots,x_n \vdash x_i$	$=$	$\pi_i^n;\eta_V$
let	$\overline{x} \vdash e_1$	$=$	g_1
	$\overline{x}, x \vdash e_2$	$=$	g_2
	$\overline{x} \vdash (\mathrm{let}\, x{=}e_1 \,\mathrm{in}\, e_2)$	$=$	$\langle \mathrm{id}_{V^n}, g_1\rangle; \mathrm{t}_{V^n,V}; Tg_2; \mu_V$
λ	$\overline{x}, x \vdash e$	$=$	g
	$\overline{x} \vdash (\lambda x.e)$	$=$	$\Lambda^T_{V,V,V^n}(g); G; \eta_{V_T^V}$
app	$\overline{x} \vdash e_1$	$=$	g_1
	$\overline{x} \vdash e$	$=$	g
	$\overline{x} \vdash e(e_1)$	$=$	$\langle g, g_1\rangle; \mathrm{app}_v$

Table 9: call-by-value interpretation

RULE	SYNTAX		SEMANTICS
var	$\overline{}$		
	$x_1,\ldots,x_n \vdash x_i$	$=$	π_i^n
let	$\overline{x} \vdash e_1$	$=$	g_1
	$\overline{x}, x \vdash e_2$	$=$	g_2
	$\overline{x} \vdash (\mathrm{let}\, x{=}e_1 \,\mathrm{in}\, e_2)$	$=$	$\langle \mathrm{id}_{(TN)^n}, g_1\rangle; \mathrm{t}_{(TN)^n,N}; T(\mathrm{id}_{(TN)^n} \times \eta_N); Tg_2; \mu_N$
λ	$\overline{x}, x \vdash e$	$=$	g
	$\overline{x} \vdash (\lambda x.e)$	$=$	$\Lambda^T_{TN,N,(TN)^n}(g); G; \eta_{N_T^{TN}}$
app	$\overline{x} \vdash e_1$	$=$	g_1
	$\overline{x} \vdash e$	$=$	g
	$\overline{x} \vdash e(e_1)$	$=$	$\langle g, g_1\rangle; \mathrm{app}_n$

Table 10: call-by-name interpretation

- $(\lambda x.M) \to (\lambda x.N)$
- $M(P) \to N(P)$ *and* $P(M) \to P(N)$
- $(\mathrm{let}\, x{=}M \,\mathrm{in}\, P) \to (\mathrm{let}\, x{=}N \,\mathrm{in}\, P)$ *and* $(\mathrm{let}\, x{=}P \,\mathrm{in}\, M) \to (\mathrm{let}\, x{=}P \,\mathrm{in}\, N)$

- *a* **notion of reduction** R, *i.e. a binary relation over* Terms, *induces the following binary relations over* Terms
  - **one-step** R**-reduction** $\to_R$, *i.e. the compatible closure of* R
  - R**-reduction** $\Rightarrow_R$, *i.e. the reflexive and transitive closure of* $\to_R$
  - R**-convertibility** $=_R$, *i.e. the symmetric and transitive closure of* $\Rightarrow_R$

We introduce three notions of reductions: *let*, β_v and η_v. The notion β_v was first introduced in [Plo75] as the call-by-value analog of β, while *let* is a new notion, which gives to the λ_c-calculus extra power w.r.t. the λ_v-calculus.

Definition 6.2 (Notions of reduction)

- β_v *is the notion of reduction* $>$ *s.t.* $(\lambda x.e)v > e[x{:=}\,v]$
- η_v *is the notion of reduction* $>$ *s.t.* $(\lambda x.v(x)) > v$ *if* $x \notin FV(v)$
- *let is the notion of reduction* $>$ *defined by the following clauses:*

$$\begin{array}{rl}
id & (\mathrm{let}\, x{=}e \,\mathrm{in}\, x) > e \\
comp & (\mathrm{let}\, x_2{=}(\mathrm{let}\, x_1{=}e_1 \,\mathrm{in}\, e_2) \,\mathrm{in}\, e) > (\mathrm{let}\, x_1{=}e_1 \,\mathrm{in}\, (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)) \\
let_v & (\mathrm{let}\, x{=}v \,\mathrm{in}\, e) > e[x{:=}\,v] \\
let.1 & nv(e) > (\mathrm{let}\, x{=}nv \,\mathrm{in}\, x(e)) \\
let.2 & v(nv) > (\mathrm{let}\, x{=}nv \,\mathrm{in}\, v(x))
\end{array}$$

Remark 6.3 The last two clauses of *let* together with β_v provide mutually exclusive clauses for reducing an application $e_1(e_2)$, namely:

- if $e_1 \in$ NonValues, then $e_1(e_2) > (\mathrm{let}\, x{=}e_1 \,\mathrm{in}\, x(e_2))$ by *let*.1
- else if $e_2 \in$ NonValues, then $e_1(e_2) > (\mathrm{let}\, x{=}e_2 \,\mathrm{in}\, e_1(x))$ by *let*.2
- else if e_1 is $(\lambda x.e)$, then $e_1(e_2) > e[x{:=}\,e_2]$ by β_v
- else we can only try to reduce the subterm e_2

The clause *let*.2 is particularly important in conjunction with β_v, since it reduces a β-redex $(\lambda x.e)(nv)$, which is not a β_v-redex, to a β_v-redex in the body of a let.

Example 6.4 We show how *let* and β_v combined together reduce $(\lambda x.x)(yz)$ to (yz), while β_v alone cannot:

- $(\lambda x.x)(yz) > (\mathrm{let}\, x{=}(yz) \,\mathrm{in}\, (\lambda x.x)(x))$ by *let*.2

- $(\mathrm{let}\, x{=}(yz)\, \mathrm{in}\, (\lambda x.x)(x)) > (\mathrm{let}\, x{=}(yz)\, \mathrm{in}\, x)$ by β_v
- $(\mathrm{let}\, x{=}(yz)\, \mathrm{in}\, x) > (yz)$ by *id*

It is easy to give a syntactic characterization of *let*- and *let*β_v-normal forms:

Proposition 6.5 *The set* NF *of let-normal forms is given by the following bnfs:*

$$e \in \mathrm{NF}{::=}\; v | v_1(v_2) | (\mathrm{let}\, x{=}v_1(v_2)\, \mathrm{in}\, e) \quad \textit{provided } e \textit{ is not } x$$

$$v \in \mathrm{NFValues}{::=}\; x | (\lambda x.e)$$

While the set β_vNF *of let*β_v*-normal forms is given by the following bnfs:*

$$e \in \beta_v\mathrm{NF}{::=}\; v | x_1(v_2) | (\mathrm{let}\, x{=}x_1(v_2)\, \mathrm{in}\, e) \quad \textit{provided } e \textit{ is not } x$$

$$v \in \beta_v\mathrm{NFValues}{::=}\; x | (\lambda x.e)$$

The following lemma is the basis for characterizing equivalence and existence in the λ_c-calculus in terms of reduction.

Lemma 6.6 (Normalization and Commutativity)

- *let-reduction is normalizing, i.e. every term reduces to a let-normal form.*
- *let-, β_v- and η_v-reduction commute with each other, i.e. if $M \Rightarrow_R M_1$ and $M \Rightarrow_S M_2$, then there exists M' s.t. $M_1 \Rightarrow_S M'$ and $M_2 \Rightarrow_R M'$, where R and S can be let, β_v or η_v.*
- *η_v-reduction can be postponed after let- and β_v-reduction, i.e. if $M \Rightarrow_{\eta_v} N$ and $N \Rightarrow_R Q$, then there exists P s.t. $M \Rightarrow_R P$ and $P \Rightarrow_{\eta_v} Q$, where R can be either let or β_v.*

Remark 6.7 Since *let*-conversion is decidable, one could consider terms up to *let*-conversion, and define β_v and η_v as notions of reduction on NF (the set of *let*-normal forms).

The study of equational presentation and reduction for the λ_p-calculus in Chapters 7 and 8 of [Mog88] is far more complicated than here, because a proper analog of *let*-reduction is lacking (although there is an analog of *let*-conversion). We think that these complications are due to the *non-equational* axiomatization of partial computations in the λ_c-calculus, in particular the axiom saying that two partial computations e_1 and e_2 are equivalent iff $(e_1 \downarrow \vee e_2 \downarrow) \rightarrow (e_1 \equiv e_2)$.

Theorem 6.8 (Syntactic characterization of λ_c-calculus)

- *two terms are provably equivalent in the λ_c-calculus iff they let$\beta_v\eta_v$-reduce to a common term*
- *a term can be proved to exist in the λ_c-calculus iff it let$\beta_v\eta_v$-reduces to a value.*

Conclusions and further research

In this paper we have presented an abstract approach to computations (based on category theory), which achieves the following objectives:

- it provides a general framework for reasoning about programs, rather than a collection of similar, but not clearly related, calculi based on an operational (or denotational) semantic;
- it improves calculi inspired by operational semantics (like the λ_v-calculus), by deriving more correct equivalences between programs.

A comparison between the categorical semantic of computations and that of linear logic based on monoidal closed categories (see [See87]) shows that they lead to *orthogonal* (and *compatible*) modifications of the notion of cartesian closed category. In fact, in the former the monad $\mathrm{Id}_{\mathcal{C}}$ is replaced by another monad T, while in the latter the cartesian product $\times$ is replaced by a tensor product $\otimes$. In our opinion this means that proof and program are rather unrelated notions, although both of them can be understood in terms of functions. Moreover, we expect categorical datatypes suggested by logic to provide a more fine-grained type system (e.g. the only procedures of a *linear functional type* are those where the formal parameter is *used exactly once*), but without changing the *qualitative nature* of computations (e.g. partial, nondeterministic, and so on), which is given by T. A different view is suggested in [Gir88], based on the paradigm: "proofs as actions".

The λ_c-calculus open the possibility to study axiomatically specific notions of computation, e.g. nondeterminism and parallelism, and their relations. For instance, an investigation of the relation between direct and continuation semantics might be carried out in full generality, without any commitment to a specific language. In the λ_c-calculus there is a very simple (and natural) definition of equality, namely $e_1 = e_2$ iff both e_1 and e_2 exist and they are equivalent, which can be safely used at compile time to check whether two program units share a common component, as required for checking a *sharing constrain* in ML (see [HMT87]). While up to now the correctness of a type-checking has to be proved by looking at the details of the operational semantics.

Acknowledgements

My thanks to M. Hyland, A. Kock (and other participants to the 1988 Category Theory Meeting in Sussex) for directing me towards the literature on monads relevant to the subject. Discussions with R. Amadio, R. Harper, F. Honsell, G. Longo, R. Milner, G. Plotkin provided useful criticisms and suggestions. Thanks also to M. Tofte and P. Taylor for suggesting improvements to an early draft.

References

[Bar82] H.P. Barendregt. The lambda calculus and its models. In G. Lolli, G. Longo, and A. Marcja, editors, *Logic Colloquium '82*. North Holland, 1982.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984. revised edition.

[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.

[EK66] S. Eilenberg and M.G. Kelly. Closed categories. In S. Eilenberg et al., editors, *Proc. Conf. Categorical Algebra (La Jolla 1965)*. Springer Verlag, 1966.

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

[Gir88] J.Y. Girard. Geometry of interaction, 1988.

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

[HMT87] R. Harper, R. Milner, and M. Tofte. A type discipline for program modules. In *Proc. of TAPSOFT 87 (Pisa 1987)*, volume 250 of *LNCS*. Springer Verlag, 1987.

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

[Koc70a] A. Kock. Monads on symmetric monoidal closed categories. *Archiv der Mathematik*, 21, 1970.

[Koc70b] A. Kock. Strong functors and monoidal monads. Various Publications Series 11, Aarhus Universitet, August 1970.

[Koc71] A. Kock. Bilinearity and cartesian closed monads. *Math. Scand.*, 29, 1971.

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

[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 *LNCS*. Springer Verlag, 1986.

[Mog88] E. Moggi. *The Partial Lambda-Calculus.* PhD thesis, University of Edinburgh, 1988. available as CST-53-88.

[MT89] I. Mason and C. Talcott. Programming, transforming, and proving with function abstractions and memories. In *16th Colloquium on Automata, Languages and Programming.* EATCS, 1989.

[Ong88] C.-H.L. Ong. *Lazy Lambda Calculus: An Investigation into the Foundations of Functional Programming.* PhD thesis, University of London, October 1988.

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

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

[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.

[Sco93] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. *Theoretical Computer Science*, 121, 1993.

[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.

[Sza69] E. Szabo, editor. *The Collected Papers of Gerhard Gentzen.* North Holland, 1969.