Computational lambda-calculus and monads

Eugenio Moggi*
Lab. for Found. of Comp. Sci.
University of Edinburgh
EH9 3JZ Edinburgh, UK
On leave from Univ. di Pisa

Abstract

The λ-calculus is considered an useful mathematical tool in the study of programming languages. However, if one uses $\beta\eta$-conversion to prove equivalence of programs, then a gross simplification[1] is introduced. We give a calculus based on a categorical semantics for *computations*, which provides a correct basis for proving equivalence of programs, independent from any specific computational model.

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. [10]). 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 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 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.

2. We consider how the categorical semantics should be extended to interpret λ-calculus.

At the end we get a formal system, the computational lambda-calculus (λ_c-calculus for short), for proving **equivalence** of programs, which is sound and complete w.r.t. the categorical semantics of computations.

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

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

The methodology outlined above is inspired by [13][2], and it is followed in [11, 8] 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.

A type theoretic approach to partial functions and computations is attempted in [1] 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 they lack a general notion of model and rely instead on operational, domain- and recursion-theoretic intuition.

1 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).

This 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 [10]). In any case, the real issue is that the notions of value and computation should not be confused. 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.

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 $B + \{\bot\}$ and the set of non-deterministic computations is the powerset $\mathcal{P}(B)$. Rather than focus on specific notions of computations, we will identify the general properties that the object TB of computations must have. The basic requirement is that programs should form a category, and the obvious choice for it is the Kleisli category for a monad.

Definition 1.1 *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 equations hold:*

- $\mu_{TA}; \mu_A = T(\mu_A); \mu_A$
- $\eta_{TA}; \mu_A = \mathrm{id}_{TA} = T(\eta_A); \mu_A$

A **computational model** *is a monad (T, η, μ) satisfying the* **mono requirement***: η_A is a mono for every $A \in \mathcal{C}$.*

There is an alternative description of a monad (see [7]), which is easier to justify computationally.

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

- $\eta_A^* = \mathrm{id}_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Every Kleisli triple $(T, \eta, \_^*)$ corresponds to a monad (T, η, μ) where $T(f: A \to B) = (f; \eta_B)^*$ and $\mu_A = \mathrm{id}_{TA}^*$.

Intuitively η_A is the *inclusion* of values into computations 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. The equations for Kleisli triples say that programs form a category, the **Kleisli category** $\mathcal{C}_T$, where the set $\mathcal{C}_T(A, B)$ of morphisms from A to B is $\mathcal{C}(A, TB)$, the identity over A is η_A and composition of f followed by g is $f; g^*$. Although the mono requirement is very natural there are cases in which it seems appropriate to drop it, for instance: it may not be satisfied by the monad of continuations.

Before going into more details we consider some examples of monads over the category of sets.

Example 1.3 Non-deterministic computations:

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

Computations with side-effects:

- $T(\_)$ is the functor $(\_ \times S)^S$, where S is a nonempty set of *stores*. Intuitively a computation takes a store and returns a value together with the modified store.

[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*".

- $\eta_A(a)$ is $(\lambda s\colon S.\langle a, s\rangle)$
- $\mu_A(f)$ is $(\lambda s\colon S.\mathrm{eval}(fs))$, i.e. 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 $\langle a, s''\rangle = f's'$.

Continuations:

- $T(\_)$ is the functor $R^{R^{(-)}}$, where R is a nonempty set of *results*. Intuitively a computation takes a continuation and returns a result.
- $\eta_A(a)$ is $(\lambda k\colon R^A.ka)$
- $\mu_A(f)$ is $(\lambda k\colon R^A.f(\lambda h\colon R^{R^A}.hk))$

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

1.1 A simple language

We introduce a programming language (with existence and equivalence assertions), where programs denote morphisms in the Kleisli category $\mathcal{C}_T$ corresponding to a computational model (T, η, μ) over a category $\mathcal{C}$. The language is oversimplified (for instance terms have exactly one free variable) in order to define its interpretation in any computational model. The additional structure required to interpret λ-terms will be introduced incrementally (see Section 2), after computations have been understood and axiomatized in *isolation*.

The programming language is parametric in a signature (i.e. a set of base types and unary command symbols), therefore its interpretation in a computational model is parametric in an interpretation of the symbols in the signature. To stress the fact that the interpretation is in $\mathcal{C}_T$ (rather than $\mathcal{C}$), we use $\tau_1 \rightharpoonup \tau_2$ (instead of $\tau_1 \to \tau_2$) as arities and $\_ \equiv \_\colon \tau$ (instead of $\_ = \_\colon T\tau$) as equality of computations of type τ.

- Given an interpretation $[\![A]\!]$ for any base type A, i.e. an object of $\mathcal{C}_T$, then the interpretation of a type $\tau ::= A \mid T\tau$ is an object $[\![\tau]\!]$ of $\mathcal{C}_T$ defined in the obvious way, $[\![T\tau]\!] = T[\![\tau]\!]$.
- Given an interpretation $[\![\mathrm{p}]\!]$ for any unary command p of arity $\tau_1 \rightharpoonup \tau_2$, i.e. a morphism from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$ in $\mathcal{C}_T$, then the interpretation of a well-formed program $x\colon\tau \vdash e\colon\tau'$ is a morphism $[\![x\colon\tau \vdash e\colon\tau']\!]$ in $\mathcal{C}_T$ from $[\![\tau]\!]$ to $[\![\tau']\!]$ defined by induction on the derivation of $x\colon\tau \vdash e\colon\tau'$ (see Table 1).
- On top of the programming language we consider equivalence and existence assertions (see Table 2).

Remark 1.4 The let-constructor is very important semantically, since it corresponds to composition in the Kleisli category $\mathcal{C}_T$. While substitution corresponds to composition in $\mathcal{C}$. In the λ-calculus $(\mathrm{let}\, x{=}e \,\mathrm{in}\, e')$ is usually treated as syntactic sugar for $(\lambda x.e')e$, and this can be done also in the λ_c-calculus. 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, $(\mathrm{let}\, x{=}e \,\mathrm{in}\, e')$ cannot be *reduced* to the more basic substitution (i.e. $e'[x\colon= e]$) without collapsing $\mathcal{C}_T$ to $\mathcal{C}$.

The existence assertion $e\downarrow$ means that e denotes a value and it generalizes the existence predicate used in the logic of partial terms/elements, for instance:

- 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.

2 Extending the language

In this section we describe the additional structure required to interpret λ-terms in a computational model. It is well-known that λ-terms can be interpreted in a cartesian closed categories (ccc), so one expects that a monad over a ccc would suffice, however, there are two problems:

- the interpretation of $(\mathrm{let}\, x{=}e \,\mathrm{in}\, e')$, when e' has other free variables beside x, and
- the interpretation of functional types.

Example 2.1 To show why the interpretation of the let-constructor is problematic, 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. Suppose that $g_2\colon\tau_1 \to T\tau_2$ and $g\colon\tau_1 \times \tau_2 \to T\tau$ are the interpretations of $x_1\colon\tau_1 \vdash e_2\colon\tau_2$ and $x_1\colon\tau_1, x_2\colon\tau_2 \vdash e\colon\tau$ respectively. 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 $\langle \mathrm{id}_{\tau_1}, g_2\rangle; g^*$. But in $\langle \mathrm{id}_{\tau_1}, g_2\rangle; g^*$ 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 $T(\tau_1 \times \tau_2)$.

The problem is that the monad and cartesian products alone do not give us the ability to transform a pair value-computation (or computation-computation) into a computation of a pair. What is needed is a morphism $\mathrm{t}_{A,B}$ from $A \times TB$ to $T(A \times B)$, so that $x_1\colon \tau_1 \vdash (\mathrm{let}\, x_2{=}e_2 \,\mathrm{in}\, e)\colon T\tau$ will be interpreted by $\langle \mathrm{id}_{\tau_1}, g_2\rangle; \mathrm{t}_{\tau_1,\tau_2}; g^*$.

Similarly for interpreting $x\colon \tau \vdash \mathrm{p}(e_1, e_2)\colon \tau'$, we need a morphism $\psi_{A,B}\colon TA \times TB \to T(A \times B)$, which given a pair of computations returns a computation computing a pair, so that, when $g_i\colon \tau \to T\tau_i$ is the interpretation of $x\colon \tau \vdash e_i\colon \tau_i$, then $[\![x\colon \tau \vdash \mathrm{p}(e_1, e_2)\colon \tau']\!]$ is $\langle g_1, g_2\rangle; \psi_{\tau_1,\tau_2}; [\![\mathrm{p}]\!]^*$.

Definition 2.2 *A* **strong monad** *over a category $\mathcal{C}$ with 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.*

$$\mathrm{t}_{1,A}; T(r_A) = r_{TA}$$

$$\mathrm{t}_{A\times B,C}; T(\alpha_{A,B,C}) = \alpha_{A,B,TC}; (\mathrm{id}_A \times \mathrm{t}_{B,C}); \mathrm{t}_{A,B\times C}$$

$$(\mathrm{id}_A \times \eta_B); \mathrm{t}_{A,B} = \eta_{A\times B}$$

$$(\mathrm{id}_A \times \mu_B); \mathrm{t}_{A,B} = \mathrm{t}_{A,TB}; T(\mathrm{t}_{A,B}); \mu_{A\times B}$$

where r and α are the natural isomorphisms

- $r_A\colon 1 \times A \to A$
- $\alpha_{A,B,C}\colon (A \times B) \times C \to A \times (B \times C)$

Remark 2.3 The natural transformation t with the above properties is not the result of some *ad hoc* considerations, instead it can be obtained via the following general principle:

> when interpreting a complex language the 2-category **Cat** of small categories, functors and natural transformations may not be adequate and one may have to use a different 2-category which captures better some *fundamental* structures underlying the language.

Since monads and adjunctions are 2-category concepts, the most natural way to model computations (and datatypes) for more complex languages is simply by monads (and adjunctions) in a suitable 2-category. Following this general principle we can give two explanations for t, one based on enriched categories (see [4]) and the other on indexed categories (see [3]).

The first explanation takes as fundamental a commutative monoidal structure on $\mathcal{C}$, which models the tensor product of *linear logic* (see [6, 14]). If $\mathcal{C}$ is a monoidal closed category, in particular a ccc, then it can be enriched over itself by taking $\mathcal{C}(A, B)$ to be the object B^A. The equations for t are taken from [5], where a one-one correspondence is established between *functorial* and *tensorial strengths*[3]:

- the first two equations say that t is a **tensorial strength** of T, so that T is a $\mathcal{C}$-enriched functor.
- the last two equations say that η and μ are natural transformations between $\mathcal{C}$-enriched functors, namely $\eta\colon \mathrm{Id}_{\mathcal{C}} \dot{\to} T$ and $\mu\colon T^2 \dot{\to} T$.

So a strong monad is just a monad over $\mathcal{C}$ enriched over itself in the 2-category of $\mathcal{C}$-enriched categories.

The second explanation was suggested to us by G. Plotkin, and takes as fundamental structure a class $\mathcal{D}$ of display maps over $\mathcal{C}$, which models *dependent types* (see [2]), and induces 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 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.

Proposition 2.4 *If (T, η, μ) is a monad over a category $\mathcal{C}$ with finite products and enough points (i.e. for any $f, g\colon A \to B$ if $h; f = h; g$ for every points $h\colon 1 \to A$, then $f = g$), and $\mathrm{t}_{A,B}$ is a family of morphisms s.t. for all points $a\colon 1 \to A$ and $b\colon 1 \to TB$*

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

where $!_B$ is the unique morphism from B to the terminal object 1, then $(T, \eta, \mu, \mathrm{t})$ is a strong monad over $\mathcal{C}$.

Remark 2.5 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

- $c_{A,B}\colon A \times B \to B \times A$

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). There is also

[3] A **functorial strength** for an endofunctor T is a natural transformation $\mathrm{st}_{A,B}\colon B^A \to (TB)^{TA}$ which *internalizes* the action of T on morphisms.

a dual notion of pairing, $\tilde{\psi}_{A,B} = c_{A,B}; \psi_{B,A}; Tc_{B,A}$ (see [5]), which amounts to first evaluating the second argument and then the first.

The reason why a functional type $A \rightarrow B$ in a programming language (like ML) cannot be interpreted by the exponential B^A (as done in a ccc) is fairly obvious; in fact the application of a functional procedure to an argument requires some computation to be performed before producing a result. By analogy with partial cartesian closed categories (see [8, 11]), we will interpret functional types by exponentials of the form $(TB)^A$.

Definition 2.6 *A* λ_c**-model** *over a category* $\mathcal{C}$ *with finite products is a strong monad* $(T, \eta, \mu, \mathrm{t})$ *together with a* T**-exponential** *for every pair* $\langle A, B\rangle$ *of objects in* $\mathcal{C}$*, i.e. a pair*

$$\langle (TB)^A, \mathrm{eval}_{A,TB}: ((TB)^A \times A) \rightarrow TB\rangle$$

satisfying the universal property that for any object C *and* $f: (C \times A) \rightarrow TB$ *there exists a unique* $h: C \rightarrow (TB)^A$*, denoted by* $\Lambda_{A,TB,C}(f)$*, s.t.*

$$f = (\Lambda_{A,TB,C}(f) \times \mathrm{id}_A); \mathrm{eval}_{A,TB}$$

Like p-exponentials, a T-exponential $(TB)^A$ can be equivalently defined by giving a natural isomorphism $\mathcal{C}_T(C \times A, B) \cong \mathcal{C}(C, (TB)^A)$, where C varies over $\mathcal{C}$.

The programming language introduced in Section 1.1 and its interpretation can be extended according to the additional structure available in a λ_c-model as follows:

- there is a new type 1, interpreted by the terminal object of $\mathcal{C}$, and two new type constructors $\tau_1 \times \tau_2$ and $\tau_1 \rightharpoonup \tau_2$ interpreted by the product $[\![\tau_1]\!] \times [\![\tau_2]\!]$ and the T-exponent $(T[\![\tau_2]\!])^{[\![\tau_1]\!]}$ respectively

- the interpretation of a well-formed program $\Gamma \vdash e: \tau$, where Γ is a sequence $x_1: \tau_1, \ldots, x_n: \tau_n$, is a morphism in $\mathcal{C}_T$ from $[\![\Gamma]\!]$ (i.e. $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$) to $[\![\tau]\!]$ (see Table 3)[4].

3 The λ_c-calculus

In this section we introduce a formal system, the λ_c**-calculus**, with two basic judgements: existence ($\Gamma \vdash e \downarrow \tau$) and equivalence ($\Gamma \vdash e_1 \equiv e_2: \tau$).

[4]In a language with products nonunary commands can be treated as unary commands from a product type.

We claim that the formal system is **sound** and **complete** w.r.t. interpretation in λ_c-models. 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 partitioned as follows:

- general rules for terms denoting computations, but with variables ranging over values (see Table 4)[5]

- the inference rules for let-constructor and types of computations (see Table 5)

- the inference rules for product and functional types (see Table 6)

Remark 3.1 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. So all three calculi are *correct* w.r.t. call-by-value operational equivalence.

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 2.3), while the λ_c-calculus is a straightforward fallout, which is easier to understand and relate to other calculi.

This semantics of computations corroborates the view that (constructive) proofs and programs are

[5]The general rules of sequent calculus, more precisely those for substitution and quantifiers, have to be modified slightly, because variables range over values and types can be empty. These modifications are similar to those introduced in the logic of partial terms (see Section 2.4 in [9]).

rather unrelated, although both of them can be understood in terms of functions. For instance, various logical modalities (like possibility and necessity in modal logic or why not and of course of linear logic) are modelled by monads or comonads which cannot 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.

Our work is just an example of what can be achieved in the study of programming languages by using a category-theoretic methodology, which free us from the irrelevant detail of syntax and focus our mind on the important structures underlying programming languages. We believe that there is a great potential to be exploited here. The λ_c-calculus open also the possibility to develop a new Logic of Computable Functions (see [12]), based on an abstract semantic of computations rather than domain theory, for studying axiomatically different notions of computation and their relations.

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. Discussions with R. Amadio, R. Burstall, J.Y. Girard, R. Harper, F. Honsell, Y. Lafont, 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

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

[2] 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.

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

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

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

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

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

[8] 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.

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

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

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

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

[13] 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.

[14] 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.

RULE	SYNTAX		SEMANTICS
var			
	$x\colon\tau \vdash x\colon\tau$	$=$	$\eta_{[\![\tau]\!]}$
let			
	$x\colon\tau \vdash e_1\colon\tau_1$	$=$	g_1
	$x_1\colon\tau_1 \vdash e_2\colon\tau_2$	$=$	g_2
	$x\colon\tau \vdash (\mathrm{let}\, x_1{=}e_1\, \mathrm{in}\, e_2)\colon\tau_2$	$=$	$g_1; g_2^*$
$\mathrm{p}\colon\tau_1 \rightharpoonup \tau_2$			
	$x\colon\tau \vdash e_1\colon\tau_1$	$=$	g_1
	$x\colon\tau \vdash \mathrm{p}(e_1)\colon\tau_2$	$=$	$g_1; \mathrm{p}^*$
$[\_]$			
	$x\colon\tau \vdash e\colon\tau'$	$=$	g
	$x\colon\tau \vdash [e]\colon T\tau'$	$=$	$g; \eta_{T[\![\tau']\!]}$
μ			
	$x\colon\tau \vdash e\colon T\tau'$	$=$	g
	$x\colon\tau \vdash \mu(e)\colon\tau'$	$=$	$g; \mu_{[\![\tau']\!]}$

Table 1: Programs and their interpretation

RULE	SYNTAX		SEMANTICS
eq			
	$x\colon\tau_1 \vdash e_1\colon\tau_2$	$=$	g_1
	$x\colon\tau_1 \vdash e_2\colon\tau_2$	$=$	g_2
	$x\colon\tau_1 \vdash e_1 \equiv e_2\colon\tau_2$	$\iff$	$g_1 = g_2$
ex			
	$x\colon\tau_1 \vdash e\colon\tau_2$	$=$	g
	$x\colon\tau_1 \vdash e \downarrow \tau_2$	$\iff$	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

RULE	SYNTAX		SEMANTICS
var	$x_1 \colon \tau_1, \ldots, x_n \colon \tau_n \vdash x_i \colon \tau_i$	$=$	$\pi_i^n ; \eta_{[\![\tau_i]\!]}$
let	$\Gamma \vdash e_1 \colon \tau_1$	$=$	g_1
	$\Gamma, x_1 \colon \tau_1 \vdash e_2 \colon \tau_2$	$=$	g_2
	$\Gamma \vdash (\mathrm{let}\, x_1 {=} e_1 \,\mathrm{in}\, e_2) \colon \tau_2$	$=$	$\langle \mathrm{id}_{[\![\Gamma]\!]}, g_1 \rangle ; \mathrm{t}_{[\![\Gamma]\!], [\![\tau_1]\!]} ; g_2^*$
$*$	$\Gamma \vdash * \colon 1$	$=$	$!_{[\![\Gamma]\!]} ; \eta_1$
$\langle\rangle$	$\Gamma \vdash e_1 \colon \tau_1$	$=$	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
	$\Gamma \vdash \pi_i(e) \colon \tau_1$	$=$	$g ; T(\pi_i)$
λ	$\Gamma, x_1 \colon \tau_1 \vdash e_2 \colon \tau_2$	$=$	g
	$\Gamma \vdash (\lambda x_1 \colon \tau_1 . e_2) \colon \tau_1 \rightharpoonup \tau_2$	$=$	$\Lambda_{[\![\tau_1]\!], T[\![\tau_2]\!], [\![\Gamma]\!]}(g) ; \eta_{[\![\tau_1 \rightharpoonup \tau_2]\!]}$
app	$\Gamma \vdash e_1 \colon \tau_1$	$=$	g_1
	$\Gamma \vdash e \colon \tau_1 \rightharpoonup \tau_2$	$=$	g
	$\Gamma \vdash e(e_1) \colon \tau_2$	$=$	$\langle g, g_1 \rangle ; \psi_{(T[\![\tau_2]\!])^{[\![\tau_1]\!]}, [\![\tau_1]\!]} ; (\mathrm{eval}_{[\![\tau_1]\!], T[\![\tau_2]\!]})^*$

Table 3: Interpretation in a λ_c-model

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

E.x $\Gamma \vdash x \downarrow \tau$

subst $\dfrac{\Gamma \vdash e \downarrow \tau \qquad \Gamma, x \colon \tau \vdash A}{\Gamma \vdash A[x \colon = e]}$

$\equiv$ is an congruence relation

Table 4: 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.

unit	$\Gamma \vdash (\text{let}\,x{=}e\,\text{in}\,x) \equiv e\colon \tau$	
ass	$\Gamma \vdash (\text{let}\,x_2{=}(\text{let}\,x_1{=}e_1\,\text{in}\,e_2)\,\text{in}\,e) \equiv (\text{let}\,x_1{=}e_1\,\text{in}\,(\text{let}\,x_2{=}e_2\,\text{in}\,e))\colon \tau$	$x_1 \notin \mathrm{FV}(e)$
let.β	$\Gamma \vdash (\text{let}\,x_1{=}x_2\,\text{in}\,e) \equiv e[x_1 := x_2]\colon \tau$	
let.p	$\Gamma \vdash \mathrm{p}(\overline{e}) \equiv (\text{let}\,\overline{x}{=}\overline{e}\,\text{in}\,\mathrm{p}(\overline{x}))\colon \tau$	

E.[_]	$\Gamma \vdash [e] \downarrow T\tau$
$T.\beta$	$\Gamma \vdash \mu([e]) \equiv e\colon \tau$
$T.\eta$	$\Gamma \vdash [\mu(x)] \equiv x\colon T\tau$

Table 5: rules for let and computational types

E.$*$	$\Gamma \vdash * \downarrow 1$
1.η	$\Gamma \vdash * \equiv x\colon 1$

E.$\langle \_ \rangle$	$\Gamma \vdash \langle x_1, x_2\rangle \downarrow \tau_1 \times \tau_2$
let.$\langle \_ \rangle$	$\Gamma \vdash \langle e_1, e_2\rangle \equiv (\text{let}\,x_1, x_2{=}e_1, e_2\,\text{in}\,\langle x_1, x_2\rangle)\colon \tau_1 \times \tau_2$
E.π_i	$\Gamma \vdash \pi_i(x) \downarrow \tau_i$
$\times.\beta$	$\Gamma \vdash \pi_i(\langle x_1, x_2\rangle) \equiv x_i\colon \tau_i$
$\times.\eta$	$\Gamma \vdash \langle \pi_1(x), \pi_2(x)\rangle \equiv x\colon \tau_1 \times \tau_2$

E.λ	$\Gamma \vdash (\lambda x\colon \tau_1.e) \downarrow \tau_1 \rightharpoonup \tau_2$
β	$\Gamma \vdash (\lambda x_1\colon \tau_1.e_2)(x_1) \equiv e_2\colon \tau_2$
η	$\Gamma \vdash (\lambda x_1\colon \tau_1.x(x_1)) \equiv x\colon \tau_1 \rightharpoonup \tau_2$

Table 6: rules for product and functional types