Math. Struct. in Comp. Science (1994), *vol.* 00, *pp.* 1–000 Copyright © Cambridge University Press

Sound and complete axiomatisations of call-by-value control operators

MARTIN HOFMANN†

Department of Computer Science, University of Edinburgh
JCMB, KB, Mayfield Rd., Edinburgh EH9 3JZ, Scotland

Received October 1993, revised September 1994

We formulate a typed version of call-by-value λ-calculus containing variants of Felleisen's control operators $\mathcal{A}$ and $\mathcal{C}$ which provide explicit access to continuations and logically extend the propositions-as-types correspondence to classical propositional logic. We give an equational theory for this calculus which is shown to be sound and complete with respect to to a class of categorical models based on continuation-passing-style semantics.

1. Introduction

It has often been pointed out that functional programs must be given explicit access to their flow control if they are to compete with imperative programs in terms of efficiency. For example a search program which recursively traverses a tree must be given a possibility to exit the recursion once it has found the entry sought. Also in order to implement backtracking or coroutines (see (Appel 1992)) a means to suspend a computation and to resume it at a later stage is required.

Such facilities can be introduced into functional programs by means of *control operators* which provide explicit access to the *continuation* of a computation. The continuation is the context in which the computation is carried out. For example, the programming language Scheme (Rees & Clinger 1986) provides the operator *call/cc* (call with current continuation) which can be given the type

$$call/cc : ((\sigma \to \tau) \to \sigma) \to \sigma$$

So in order to compute a value of type σ it is enough to give it relative to some variable of type $\sigma \to \tau$ which should be thought of as the continuation for the value. In particular this continuation can be instantiated with some value which will then immediately become the result. For example, the expression

$$call/cc(\lambda k : int \to int.19 + (k\ 4))$$

will evaluate to 4. On the other hand, the continuation need not be used, so for example the value of

$$call/cc(\lambda k : int \to int.35)$$

† The author is supported by a European Union HCM fellowship; contract number ERBCHBICT930420

is 35. The continuation made available by $call/cc$ can also be stored in some global data structure (for an example see (Appel 1992)) or become part of the result as in

$$(call/cc(\lambda k : (int \to int) \to int.\lambda x : int.k(\lambda y : int.x))) \, 1965$$

This expression will evaluate to 1965, which can be seen by carefully following the control flow. The arbitrary type τ in the typing of the continuation argument k can be explained as follows. As soon as k is applied to some value the current computation will be aborted, so the hypothetical return value of k of type τ can never actually be used.

Notice that in the presence of $call/cc$ the evaluation order affects the result. According to whether call-by-name or call-by-value evaluation is chosen the expression

$$call/cc(\lambda k : int \to int.(\lambda x.34)\,(k\,35))$$

will evaluate to 34 or 35. Both alternatives make sense, but in this article we shall fix the call-by-value evaluation order.

$call/cc$ is not the only possible control operator. There are other equivalent control operators, for example Standard ML's *callcc* of type

$$callcc : ((\sigma \to 0) \to \sigma) \to \sigma$$

where 0 is the empty type. In fact this is a slightly modified version of SML's *callcc*, since we have replaced the type constructor *cont* by $(- \to 0)$. See also Section 8. A variant of *callcc* is Felleisen's (untyped) $\mathcal{C}$ operator which can be assigned the type

$$\mathcal{C} : ((\sigma \to 0) \to 0) \to \sigma$$

In the case of *callcc* and $\mathcal{C}$ the codomain of the continuation argument even becomes the empty type. Again this is acceptable since no continuation will ever return a value but always leads to immediate abortion of the current computation. All control operators can be expressed in terms of each other, but the $\mathcal{C}$ operator is technically the simplest one, and so we shall base our investigations on it.

Another approach to control operators is via classical logic. The well-known Curry-Howard isomorphism relates proofs in intuitionistic propositional logic to terms of the simply-typed λ-calculus or equivalently to simple functional programs. Moreover, proofs in intuitionistic first-order Peano arithmetic can be translated into terms of a simply-typed λ-calculus containing a type of natural numbers and a primitive recursion combinator (Gödel's System T). This translation is such that the program obtained from a proof of a Π^0_2-statement computes precisely the function described by that statement.

If one wants to extend this procedure to proofs in *classical* Peano arithmetic one needs terms which realise the classical axioms. The $call/cc$-operator is one such, since its type corresponds to Peirce's law. Indeed, Murthy (1991), following ideas of Griffin (1990), has shown how programs with control operators can be extracted from classical proofs.

Programs involving control operators can be given a clear semantics by translating them into *continuation passing style*. Under this translation the meaning of a term of type σ becomes an element of the function space

$$(\sigma \to \rho) \to \rho$$

for some type of "final results" ρ. Such a function is understood as a mapping from the continuation for some value — this continuation has type $\sigma \to \rho$ — to the final result of type ρ. Since in this translation the continuation for some expression is made visible, it is possible to give an ordinary λ-term interpreting *call/cc*. This is explained in more detail in (Felleisen *et al.* 1987) or (Reynolds 1972).

In (Filinski 1989) abstract categorical models for λ-calculi with control operators are given and it is shown that the usual continuation-passing-style semantics forms an instance.

Felleisen *et al.* (1987) introduce an untyped λ-calculus including the control operator $\mathcal{C}$ described above, and give rewriting rules which allow to evaluate programs directly without translation into continuation-passing-style.

These rules, however, are not complete wrt. continuation-passing-style semantics. Not all equations that can be proven in the semantics follow directly from the rewrite rules. The aim of this work is to fill this gap. We describe a typed version of Felleisen's calculus and give a set of equations which is shown to be sound and complete for (a categorical version of) continuation-passing-style semantics.

After the completion of the present work Sabry and Felleisen (1993) independently came up with a refined version of the calculus described in (Felleisen *et al.* 1987) which is complete w.r.t. the cps-translation. The main difference to the work reported here is that they work in an untyped framework and that their completeness proof is entirely syntactic and therefore more complicated than ours, but also more elementary. The main idea in loc. cit. is the definition of an inverse to the cps-translation. The axioms then arise as the inverses of ordinary reductions in the cps-translation.

Axioms similar to the ones presented here also appear in (Talcott 1992). The question of completeness, however, is not addressed there.

The article is organised as follows. In the first section we introduce a typed variant of Plotkin's call-by-value λ-calculus (Plotkin 1975) which contains the $\mathcal{C}$ operator mentioned above and an elimination operator $\mathcal{A}$ for the empty type 0 used in the typing of $\mathcal{C}$.

The next section gives an equational theory for this calculus. It contains restricted forms of β- and η-equality and axioms describing the operators $\mathcal{C}$ and $\mathcal{A}$. We try to motivate the axioms from the computational intuition and also compare them to Felleisen's rewrite rules. We derive further identities, and show the necessity of various restrictions on the axioms. We also compare our axioms to Felleisen's incomplete set of axioms and the recent complete extension.

In Section 4 we define a continuation-passing-style interpretation of the calculus in (nearly) cartesian closed categories. This model construction is strongly related to the one presented by Agapiev and Moggi (1991). The only difference is that we do not require full cartesian closure for the modelling category but only the existence of those exponentials which actually appear in the semantic equations. The equational theory is shown to be sound for this class of models.

Section 5 contains the main result of this work. We show that the equational axiomatisation for the calculus is complete with respect to the continuation-passing-style semantics. The proof of this proceeds by exhibiting an interpretation of the calculus in a syntactic category formed out of the *values* (variables and abstractions) in the calculus.

A similar technique has also been used by Moggi (1991) to establish completeness of a simpler calculus.

In Section 6 we use these results to deduce complete axiomatisations for the other control operators *callcc* and Scheme's *call/cc*. In the last section we sketch some applications of our work and give directions for further research.

The article is not entirely self-contained; some very basic knowledge of category theory will be assumed. A good reference is for example (Barr & Wells 1990).

2. A call-by-value λ-calculus with control operators

We now come to the formal definition of the calculus which we will call $\lambda_{\mathcal{C}}$. Let a collection of base types be given. Then the types of $\lambda_{\mathcal{C}}$ are defined by the grammar

$$\sigma, \tau ::= 0 \mid \beta \mid \sigma \multimap \tau$$

where β ranges over the base types. The preterms are given by

$$M, N ::= x \mid \mathcal{A}_\sigma(M) \mid \mathcal{C}_\sigma(M) \mid (M\ N) \mid \lambda x : \sigma.M$$

where x ranges over an infinite set of variables and σ is a type.

A context is a list of variable declarations of the form $x : \sigma$ without duplications. The following rules assign types to certain preterms.

$$\Gamma \vdash x : \sigma \qquad \text{if } x : \sigma \text{ appears in } \Gamma$$

$$\frac{\Gamma \vdash M : \sigma \multimap \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\ N) : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma.M : \sigma \multimap \tau} \qquad \text{if } x \text{ does not appear in } \Gamma$$

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \mathcal{A}_\sigma(M) : \sigma}$$

$$\frac{\Gamma \vdash M : (\sigma \multimap 0) \multimap 0}{\Gamma \vdash \mathcal{C}_\sigma(M) : \sigma}$$

From now on a *term* will mean an α-equivalence class of judgements $\Gamma \vdash M : \sigma$. So for example

$$x : \sigma \vdash \lambda y : \tau.x$$

and

$$z : \sigma \vdash \lambda w : \tau.z$$

refer to the same term. If $\Gamma, x : \sigma \vdash M : \tau$ and $\Gamma \vdash N : \sigma$ then we define the syntactic substitution $\Gamma \vdash M[x := N] : \sigma$ in the usual way by induction on the structure of M where by suitable renaming we ensure that no unwanted variable captures can occur.

Two final remarks on notation are in order. Although all terms are typed and the typing

is indeed required, we shall freely suppress typing information if this (in our opinion) increases readibility. Similarly we shall occasionally omit contexts although both types and contexts form an integral part of a term.

3. An equational theory

We now give an equational theory for $\lambda_{\mathcal{C}}$ which extends Plotkin's account (Plotkin 1975) of the interconvertibility relation in the call-by-value λ-calculus (λ_v). We add structural equations which make composition defined in terms of abstraction associative. Furthermore we add equations which characterise the computational behaviour of the operators $\mathcal{A}$ and $\mathcal{C}$. As in Plotkin's λ_v-calculus we require the notion of *value*. A value is a "fully evaluated term"; more precisely:

— every variable is a value

— every abstraction (*i.e.* term of the form $\lambda x.M$) is a value

— if V is a value then so is $\mathcal{A}_\sigma(V)$

Definition 1. The relation $=_{\mathcal{C}}$ between terms is defined to be the least congruence with respect to the term constructors including the following equations. Let U, V range over values and M, N range over arbitrary terms.

$$
\begin{array}{ll}
(\lambda x : \sigma.M)\, V =_{\mathcal{C}} M[x := V] & \textsc{Beta-V} \\
\lambda x : \sigma.V\, x =_{\mathcal{C}} V & \textsc{Eta-V} \\
 & \\
(\lambda x : \sigma.U\, (V\, x))\, M =_{\mathcal{C}} U\, (V\, M) & \textsc{Ass} \\
(\lambda f : \sigma \rightharpoonup \tau.f\, N)\, M =_{\mathcal{C}} M\, N & \textsc{App} \\
 & \\
V\, \mathcal{A}_\sigma(M) =_{\mathcal{C}} \mathcal{A}_\tau(M) & \mathcal{A}\textsc{-Abs} \\
\mathcal{A}_0(M) =_{\mathcal{C}} M & \mathcal{A}_0\textsc{-Id} \\
 & \\
V\, \mathcal{C}_\sigma(M) =_{\mathcal{C}} \mathcal{C}_\tau(\lambda\kappa : \tau \rightharpoonup 0.M\, (\lambda x : \sigma.\kappa\, (V\, x))) & \mathcal{C}\textsc{-Nat} \\
\mathcal{C}_\sigma(\lambda\kappa : \sigma \rightharpoonup 0.\kappa\, M) =_{\mathcal{C}} M & \mathcal{C}\textsc{-App}
\end{array}
$$

In all rules except Beta-V the variables are supposed to be fresh.

To reduce clutter we have omitted both contexts and typings in the equations. They are supposed to hold in any context, and for each type for which they make sense, in particular this means that the types of the lhs and the rhs must agree. So for example the first equation is understood as follows. If $\Gamma, x : \sigma \vdash M : \tau$ is a term and $\Gamma \vdash V : \sigma$ is a value then

$$\Gamma \vdash (\lambda x : \sigma.M)\, V =_{\mathcal{C}} M[x := V] : \tau$$

A few comments concerning these equations are in order. The first is that our aim is to give a complete axiomatisation of the models we describe in Section 4, so an equation is justified by being true in any model. We attempt, however, to motivate the rules directly and to compare them to other equations known from the literature.

3.1. *Application and abstraction*

The rules Eta-V and Beta-V are the usual ones for call-by-value λ-calculus as introduced in (Plotkin 1975). They establish a bijective correspondence between *terms* in context $\Gamma, x{:}\sigma$ of type τ and *values* of type $\sigma \multimap \tau$ in context Γ. So the restriction that V be a value in rules $\mathcal{C}$-Nat and $\mathcal{A}$-Abs only ensures that V arises from a term with a free variable. In particular if V is a value of function type then this does not imply that V maps values to values. The rule Ass implies that composition defined by $f \circ g := \lambda x.f(gx)$ is associative. It corresponds to one of Moggi's let-axioms (Moggi 1991) when "let" is expanded into its call-by-value definition in terms of abstraction and application. The rule App seems to be new. However, it is equivalent to Felleisen's (untyped) $\mathcal{C}_R$-rule as we show below in Section 3.5.

The rules for the $\mathcal{A}$ operator express that 0 is the empty type. One may ask why the "type of results" is chosen to be 0. This question is, however, ill-posed. Scheme's *call/cc* operator allows an arbitrary type τ for the codomain of the continuation argument, so in particular 0 may be chosen. On the other hand 0 suffices since *call/cc* can be expressed in terms of its particular instance $\tau := 0$. So if there is an empty type 0 then it may well be chosen for the codomain of continuations. Under call-by-value the existence of an empty type is no problem, and indeed in the call-by-value language SML we can define the empty type by

```
datatype 0 = c of 0;
fun A(c x) = A(x);
```

Krivine (1992) and Filinski (1989) try to argue intuitively why the result type should be empty. They reason that every value must be consumed by some context be it the surrounding operating system, so that the final results must have

empty type since no context will consume these. We prefer, however, to argue pragmatically as above.

The rule that values are closed under $\mathcal{A}_\sigma$ may seem unnatural. Its main purpose is to endow the category $\mathcal{V}$ of contexts and values to be defined below in Section 5 with an initial object. We explain there how this can be avoided if so desired.

3.2. $\mathcal{C}$ *operator*

The first equation $\mathcal{C}$-Nat expresses that in $\mathcal{C}(M)$ the argument to M will become the current continuation. If $\kappa : \tau \multimap 0$ is the continuation for $V\ \mathcal{C}(M)$ then $\lambda x : \sigma.\kappa\,(V\ x) = \kappa \circ V$ is the continuation for $\mathcal{C}(M)$. So if we want to express $(V\ \mathcal{C}(M))$ as an expression of the form $\mathcal{C}(-)$ then we must instantiate M with this continuation, whence we obtain the right-hand side of equation $\mathcal{C}$-Nat.

The rule $\mathcal{C}$-App is another intuitive consequence of the idea that $\mathcal{C}$ makes the current continuation visible. If one merely instantiates this continuation with some term M then this term is the result.

3.3. *Constants*

We do not explicitly allow for constants; they may, however, be simulated by working in a suitably extended context, *e.g.* $\Gamma = \mathit{zero}{:}N, \mathit{succ}{:}N \rightharpoonup N$ to account for natural numbers. Then, however, $(\mathit{succ}\ \mathit{zero})$ is not a value, so that we cannot deduce certain equations which we expect to hold. Completeness for a system with constants and more generally with constants and equational axioms is an important issue, but goes beyond the scope of this article.

Proposition 2. The following equations hold in $\lambda_{\mathcal{C}}$ for M, N ranging over terms and U, V ranging over values and all variables fresh:

$$(\lambda x : 0.\mathcal{A}_\sigma(x))\, M \ =_{\mathcal{C}}\ \mathcal{A}_\sigma(M) \qquad \mathcal{A}\text{-Beta}$$

$$V \ =_{\mathcal{C}}\ \lambda x : 0.\mathcal{A}_\sigma(x) : 0 \rightharpoonup \sigma \qquad 0\text{-Ini}$$

$$V \ =_{\mathcal{C}}\ \lambda x : 0.x : 0 \rightharpoonup 0 \qquad 0\text{-Endo}$$

$$(\lambda x.x)\, M \ =_{\mathcal{C}}\ M \qquad \text{Ident}$$

$$\mathcal{C}_\sigma(M) \ =_{\mathcal{C}}\ \mathcal{C}_\sigma(\lambda\kappa.M\kappa) \qquad \mathcal{C}\text{-Eta}$$

$$\mathcal{C}_0(M) \ =_{\mathcal{C}}\ M\ (\lambda x : 0.x) \qquad \mathcal{C}_0\text{-End}$$

$$(\lambda x : (\sigma \rightharpoonup 0) \rightharpoonup 0.\mathcal{C}_\sigma(x))\, M \ =_{\mathcal{C}}\ \mathcal{C}_\sigma(M) \qquad \mathcal{C}\text{-Beta}$$

$$V\ ((\lambda f : \sigma \rightharpoonup \tau.f\ N)\, M) \ =_{\mathcal{C}}\ (\lambda f.V\ (f\ N))\, M \qquad \text{Conv}$$

$$\mathcal{A}_\sigma(M) \ =_{\mathcal{C}}\ \mathcal{C}_\sigma(\lambda\kappa : \sigma \rightharpoonup 0.M) \qquad \mathcal{A}\text{-}\mathcal{C}$$

$$\mathcal{C}_{\sigma\rightharpoonup\tau}(M)\, N \ =_{\mathcal{C}}\ \mathcal{C}_\sigma(\lambda\kappa.M\ (\lambda f.\kappa\ (f\ N))) \qquad \mathcal{C}\text{-Nat-R}$$

Proof. All these equations follow by rewriting the axioms, so they are stable under extensions of the calculus. The proofs are not completely straightforward, though, so we include them here.

$\mathcal{A}$-Beta

$$\begin{array}{lll} (\lambda x : 0.\mathcal{A}_\sigma(x))\, M & =_{\mathcal{C}} & \text{by } \mathcal{A}_0\text{-Id} \\ (\lambda x : 0.\mathcal{A}_\sigma(x))\, \mathcal{A}_0(M) & =_{\mathcal{C}} & \text{by } \mathcal{A}\text{-Abs} \\ \mathcal{A}_\sigma(M) & & \end{array}$$

0-Ini

$$\begin{array}{lll} V : 0 \rightharpoonup \sigma & =_{\mathcal{C}} & \text{by Eta-V and } \mathcal{A}_0\text{-Id} \\ \lambda x{:}0.V\ \mathcal{A}_0(x) & =_{\mathcal{C}} & \text{by } \mathcal{A}\text{-Abs} \\ \lambda x{:}0\mathcal{A}_\sigma(x) & & \end{array}$$

0-Endo is an immediate consequence of 0-Ini.

IDENT

$$\begin{array}{lll}
(\lambda x.x)\, M & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\
(\lambda x.x)\, \mathcal{C}(\lambda\kappa.\kappa\, M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-Nat} \\
\mathcal{C}(\lambda k.(\lambda\kappa.\kappa\, M)(\lambda x.\kappa\, ((\lambda x.x)x))) & =_{\mathcal{C}} & \text{by Beta-V} \\
\mathcal{C}(\lambda\kappa.\kappa\, M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\
M & &
\end{array}$$

$\mathcal{C}$-ETA

$$\begin{array}{lll}
\mathcal{C}_\sigma(M) & =_{\mathcal{C}} & \text{by Ident} \\
(\lambda x:\sigma.x)\, \mathcal{C}_\sigma(M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-Nat,Beta-V} \\
\mathcal{C}_\sigma(\lambda\kappa.M\kappa) & &
\end{array}$$

$\mathcal{C}_0$-END

$$\begin{array}{lll}
\mathcal{C}_0(M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-Eta} \\
\mathcal{C}_0(\lambda\kappa:0\rightharpoonup 0.M\,\kappa) & =_{\mathcal{C}} & \text{by 0-Endo and Ident} \\
\mathcal{C}_0(\lambda\kappa.\kappa\, (M\, (\lambda x:0.x))) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\
M\, (\lambda x:0.x) & &
\end{array}$$

$\mathcal{C}$-BETA

$$\begin{array}{lll}
(\lambda x:(\sigma\rightharpoonup 0)\rightharpoonup 0.\mathcal{C}_\sigma(x))\, M & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\
\mathcal{C}_\sigma(\lambda\kappa.\kappa((\lambda x.\mathcal{C}_\sigma(x))\, M)) & =_{\mathcal{C}} & \text{by Ass and Beta-V} \\
\mathcal{C}_\sigma(\lambda\kappa.(\lambda x.\kappa\, \mathcal{C}_\sigma(x))\, M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-Nat, Beta-V, Eta-V} \\
\mathcal{C}_\sigma(\lambda\kappa.(\lambda x.\mathcal{C}_0(\lambda k:0\rightharpoonup 0.x\,\kappa)\, M)) & =_{\mathcal{C}} & \text{by } \mathcal{C}_0\text{-End} \\
\mathcal{C}_\sigma(\lambda\kappa.(\lambda x.x\,\kappa)\, M) & =_{\mathcal{C}} & \text{by App} \\
\mathcal{C}_\sigma(\lambda\kappa.M\,\kappa) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-Eta} \\
\mathcal{C}_\sigma(M) & &
\end{array}$$

CONV

$$\begin{array}{lll}
V\, ((\lambda f:\sigma\rightharpoonup\tau.f\, N)\, M) & =_{\mathcal{C}} & \text{by Ass} \\
(\lambda g:\sigma\rightharpoonup\tau.V\, ((\lambda f.f\, N)\, g))\, M & =_{\mathcal{C}} & \text{by Beta-V} \\
(\lambda g.V\, (g\, N))\, M & &
\end{array}$$

$\mathcal{A}$-$\mathcal{C}$

$$\begin{array}{lll}
\mathcal{A}_\sigma(M) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\
\mathcal{C}_\sigma(\lambda\kappa.\kappa\, \mathcal{A}_\sigma(M)) & =_{\mathcal{C}} & \text{by } \mathcal{A}\text{-Abs} \\
\mathcal{C}_\sigma(\lambda\kappa.\mathcal{A}_0(M)) & =_{\mathcal{C}} & \text{by } \mathcal{A}_0\text{-Id} \\
\mathcal{C}_\sigma(\lambda\kappa.M) & &
\end{array}$$

$\mathcal{C}$-NAT-R is an immediate consequence of $\mathcal{C}$-NAT and APP. This completes the proof of Proposition 2 □

Remark 3. One might use the identity $\mathcal{A}$-$\mathcal{C}$ in order to eliminate $\mathcal{A}$ altogether. One would then have to add $\mathcal{C}_0$-END as an axiom and postulate that $\mathcal{C}_\sigma(\lambda\kappa.V)$ is a value if V is. This seems a bit unnatural and so we prefer to keep $\mathcal{A}$. We also remark that $\mathcal{C}_0$-END

resembles one of Krivine's (Krivine 1992) axioms for $\mathcal{C}$. He has $\mathcal{C}_0$-End for arbitrary type σ, not only for the particular case $\mathcal{C}_0$. This is possible because he only allows weak head β-reduction, *i.e.* no rewriting underneath an abstraction. In our calculus Krivine's rule would be unsound.

3.4. *Inconsistency by unrestricted β-equality*

Let M and N be arbitrary terms of some type σ. Then using unrestricted β-equality we can compute as follows

$$\begin{array}{lll} M & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\ \mathcal{C}(\lambda\kappa : \sigma \rightharpoonup 0.\kappa\, M) & =_{\mathcal{C}} & \text{by } \mathcal{A}_0\text{-Id} \\ \mathcal{C}(\lambda\kappa.(\lambda x : 0.\kappa\, N)\ (\kappa\, M)) & =_{\mathcal{C}} & \text{by unrestricted Beta-V} \\ \mathcal{C}(\lambda\kappa.\kappa\, N) & =_{\mathcal{C}} & \text{by } \mathcal{C}\text{-App} \\ N & & \end{array}$$

so all terms will be equated.

3.5. *A comparison with Felleisen's calculus*

In (Felleisen *et al.* 1987) an untyped variant of $\lambda_{\mathcal{C}}$ is introduced which is not complete wrt. continuation-passing-style semantics. It consists of untyped versions of Beta-V, Eta-V, $\mathcal{C}$-Nat, $\mathcal{C}$-Nat-R (called $\mathcal{C}_L$, and $\mathcal{C}_R$ there), $\mathcal{A}$-Abs, $\mathcal{A}_0$-Id, $\mathcal{C}_0$-End. Since $\mathcal{A}_0$-Id and $\mathcal{C}_0$-End become unsound without types they are restricted to occurrences at the root of a term, *i.e.* they may not be applied inside a term. Moreover, they have an additional instance of the $\mathcal{A}$ operator in the right-hand sides of $\mathcal{C}$-Nat and $\mathcal{C}$-Nat-R which in the typed calculus is of type 0, hence equal to the identity. Our rule App can be deduced from $\mathcal{C}$-Nat-R by rewriting both sides of App using $\mathcal{C}$-App on the two respective root terms, *i.e.* $\lambda f{:}\sigma \rightharpoonup \tau.f\ N$ and M and then using $\mathcal{C}$-Nat-R.

In the recent complete version (Sabry & Felleisen 1993) an axiom similar to Ass is added to obtain completeness.

4. Models

We now describe a notion of model for the calculus. The only difference between our model and the one given in (Agapiev & Moggi 1991) is that we do not require the modelling category to be cartesian closed, but only require those exponentials to exist which are needed for the interpretation. This has the consequence that no continuation monad on the whole category can be defined. In spirit, however, our model construction works via the Kleisli category for the continuation monad R^{R^-} introduced in (Moggi 1991). The reader familiar with this field should keep this intuition in mind.

For the development which follows we need the notion of exponential object in a category with binary products. We use the standard syntax π, π', $\langle -, - \rangle$ for projections and pairing for binary products and $f \times g$ for the morphism $\langle f \circ \pi, g \circ \pi' \rangle$. Among the many possible equivalent definitions of exponentials we pick the following:

Definition 4. Let $\mathcal{K}$ be a category with binary products and A, B be objects in $\mathcal{K}$. An *exponential* of B by A consists of an object B^A, and for each object C a bijection

$$cur_{C,A,B} : \mathcal{K}(C \times A, B) \to \mathcal{K}(C, B^A)$$

natural in C. This means that if $f : C \times A \to B$ and $s : D \to C$ then

$$cur_{C,A,B}(f) \circ s = cur_{C',A,B}(f \circ (s \times \mathrm{id}_A))$$

Following Lambek and Scott (1985) we introduce an informal λ-calculus to define morphisms in a category with certain exponentials. A morphism from X to Y is written as a term with a free variable of type X or alternatively if $X = X_1 \times \cdots \times X_n$ a term of type Y with n free variables of types X_1 through X_n. Accordingly, composition is rendered by substitution. If the exponential B^A exists and if $c{:}C, a{:}A \vdash f(c,a) : B$ denotes a morphism from $C \times A$ to B then its image under the bijection $cur_{C,A,B}$ is written $c{:}C \vdash \underline{\lambda} a{:}A.f(a,c) : B^A$. The formation of $\underline{\lambda} a{:}A.f(a)$ is allowed only if the corresponding exponential exists. Conversely, if $c{:}C \vdash f : B^A$ and $c{:}C \vdash g : A$ then we write $c{:}C \vdash f\,!\,g : B$ for the image of f under cur^{-1} composed with the pairing of g and the identity on A. The function cur^{-1} itself arises as the special case $c{:}C, a{:}A \vdash f\,!\,a : B$. The two equations specifying that cur is a bijection then arise as β and η-rules

$$c{:}C \vdash \underline{\lambda} a{:}A.f(a)\,!\,g = f(g) : B$$
$$c{:}C \vdash \underline{\lambda} a{:}A.f\,!\,a = f : B^A$$

The naturality condition on cur is implicit in this notation because it becomes a syntactic identity in the $\underline{\lambda}$-notation which would not be well-defined in the absence of naturality.

Now we turn to the definition of the intended semantics of $\lambda_{\mathcal{C}}$.

Definition 5. A $\lambda_{\mathcal{C}}$-category is a triple $(\mathcal{K}, \mathcal{B}, R)$ where

— $\mathcal{K}$ is a cartesian category, *i.e.* a category with terminal object (1) and binary products ($\times$), which has an initial object (0).
— $\mathcal{B}$ is a collection of $\mathcal{K}$-objects, called *type objects* containing the initial object 0.
— R is a type object
— For any type object A the exponential of R by A exists and is a type object.
— For any two type objects A and B the exponential of R^{R^B} by A exists and is a type object.

The type objects of a $\lambda_{\mathcal{C}}$-category will be used to interpret the types of a $\lambda_{\mathcal{C}}$-calculus. Contexts will be interpreted by certain cartesian products of type objects. However, a term $\Gamma \vdash M : \sigma$ will not be interpreted as a morphism from the interpretation of Γ to the one of σ (denoted $[\![\sigma]\!]$) but to the object $R^{R^{[\![\sigma]\!]}}$. This means that the interpretation of a term is not an actual element, but a function which maps a continuation for it (an element of $R^{[\![\sigma]\!]}$) to the "final result" (in R). This is why the control operators can be interpreted in such a category. Moggi (1991) calls such elements "computations" of type $[\![\sigma]\!]$.

One may organise a $\lambda_{\mathcal{C}}$-category $\mathcal{K}$ as a *constant comprehension category* over $\mathcal{K}$ in the sense of Jacobs (1991, p. 94) the fibres of which would be the type objects. It is then possible to define a fibred continuation monad even in the absence of arbitrary

exponentials in $\mathcal{K}$. For simplicity we prefer to stick to the more elementary view presented here.

We use the informal $\underline{\lambda}$-calculus described above to denote morphisms in a $\lambda_{\mathcal{C}}$-category. To that end we augment it by a term $x{:}0 \vdash ?_A(x)$ corresponding to the unique morphism from the initial object 0 to object A. To increase readability we use $\underline{\lambda}$ and $^{\mathsf{I}}$ only for the exponentials of the form R^A and use $\underline{\underline{\lambda}}$ and $^{\mathsf{II}}$ for the exponentials of the form $(R^{R^B})^A$. We thus obtain the following derived typing rules for the $\underline{\lambda}$-calculus.

$$\frac{\Gamma, a{:}A \vdash f(a) : R}{\Gamma \vdash \underline{\lambda}a{:}A.f(a) : R^A} \qquad \frac{\Gamma \vdash f : R^A \qquad \Gamma \vdash g : A}{\Gamma \vdash f^{\mathsf{I}} g : R}$$

$$\frac{\Gamma, a{:}A \vdash f(x) : R^{R^B}}{\Gamma \vdash \underline{\underline{\lambda}}x{:}\sigma.f(x) : (R^{R^B})^A} \qquad \frac{\Gamma \vdash f{:}(R^{R^B})^A \qquad \Gamma \vdash g : A}{\Gamma \vdash f^{\mathsf{II}} g : R^{R^B}}$$

$$\frac{\Gamma \vdash f : 0}{\Gamma \vdash ?_A(f) : A}$$

Before we describe the interpretation in detail we define a few auxiliary morphisms which in addition should explain the informal $\underline{\lambda}$-notation.

For any type object A we define the canonical map $\eta_A : A \to R^{R^A}$ by

$$\eta_A(a : A) := \underline{\lambda}\kappa : R^A.\kappa^{\mathsf{I}} a$$

In order to get a definition of η_A without using the $\underline{\lambda}$-calculus we would start with the identity morphism id_{R^A} and apply cur^{-1} giving the so-called evaluation map $ev_{A,R} = cur^{-1}_{R^A,A,R}(\mathrm{id}_{R^A}) : R^A \times A \to R$. We then compose $ev_{A,R}$ with the "twist-map" $\langle \pi', \pi \rangle : A \times R^A \to R^A \times A$ and finally apply $cur_{A,A,R}$ to obtain η_A. We see that the $\underline{\lambda}$-calculus notation glosses over unnecessary detail.

The exponentials of the form $(R^{R^B})^A$ will be used to interpret the arrow types of $\lambda_{\mathcal{C}}$. For typographical reasons we shall henceforth abbreviate this object by $A \Rightarrow B$. In order to interpret application we need a morphism which applies a "computation" of this type, this is an element of $R^{R^{A \Rightarrow B}}$, to a computation of type A, *i.e.* an element of type R^{R^A}. So we define a morphism

$$app_{A,B} : R^{R^{A \Rightarrow B}} \times R^{R^A} \to R^{R^B}$$

by

$$app_{A,B}(\langle F : R^{R^{A \Rightarrow B}}, X : R^{R^A} \rangle) := \underline{\lambda}\kappa : R^B.F^{\mathsf{I}}(\underline{\lambda}f : A \Rightarrow B.X^{\mathsf{I}}(\underline{\lambda}a : A.f^{\mathsf{II}} a^{\mathsf{I}} \kappa))$$

corresponding to the usual definition of call-by-value application in continuation passing style (Felleisen *et al.* 1987). The idea behind *app* is that F is applied to the continuation for it which in turn is obtained by applying X to its continuation relative to the variable f. Next we need a morphism which interprets the $\mathcal{C}$ operator.

$$C_A : R^{R^{(A \Rightarrow 0) \Rightarrow 0}} \to R^{R^A}$$

It is defined by

$$C_A(\Phi : R^{R^{(A \Rightarrow 0) \Rightarrow 0}}) := \underline{\lambda}\kappa : R^A.\Phi^{\mathsf{I}}(\underline{\lambda}\phi : (A \Rightarrow 0) \Rightarrow 0. \\ \phi^{\mathsf{II}}(\underline{\underline{\lambda}}a : A.\underline{\lambda}q : R^0.\kappa^{\mathsf{I}} a)^{\mathsf{I}}(\underline{\lambda}x : 0.?_R(x)))$$

Intuitively C passes the current continuation (κ) to its argument (Φ). Finally we need a morphism $A_A : R^{R^0} \to R^{R^A}$ interpreting the $\mathcal{A}$ operator. It is defined by

$$A_A(X : R^{R^0}) := \underline{\lambda} k : R^A . X^{\mathsf{l}}(\underline{\lambda} x : 0 . k^{\mathsf{l}}(?_A(x)))$$

or equivalently since 0 is initial by

$$A_A(X : R^{R^0}) := \underline{\lambda} k : R^A . X^{\mathsf{l}}(\underline{\lambda} x : 0 . ?_R(x))$$

For the reader familiar with computational monads (Moggi 1991) we should add that our morphisms app, C, and A are in fact the *lifted* versions of the actual Kleisli morphisms with domains $(A \Rightarrow B) \times A$, $(A \Rightarrow 0) \Rightarrow 0$, and 0, respectively. Additionally app has been precomposed with a tensorial strength; otherwise it would have domain $R^{R^{(A \Rightarrow B) \times A}}$. This enables us to do without formally introducing Kleisli composition.

Definition 6. An *interpretation* of a $\lambda_{\mathcal{C}}$-calculus in a $\lambda_{\mathcal{C}}$-category $(\mathcal{K}, \mathcal{B}, R)$ consists of

— A type object $[\![\sigma]\!]$ for each type σ
— An object $[\![\Gamma]\!]$ for each context Γ
— A morphism $[\![\Gamma \vdash M : \tau]\!] : [\![\Gamma]\!] \to R^{R^{[\![\tau]\!]}}$ for each term $\Gamma \vdash M : \tau$

in such a way that

— $[\![\text{empty context}]\!] = 1$
— $[\![\Gamma, x : \sigma]\!] = [\![\Gamma]\!] \times [\![\sigma]\!]$
— $[\![0]\!] = 0$
— $[\![\sigma \multimap \tau]\!] = [\![\sigma]\!] \Rightarrow [\![\tau]\!]$ $\quad (= (R^{R^{[\![\tau]\!]}})^{[\![\sigma]\!]})$

and

— $[\![\Gamma \vdash x : \sigma]\!] = \eta_{[\![\sigma]\!]} \circ \pi_x$ $\qquad$ π_x is the projection corresponding to x in $[\![\Gamma]\!]$.
— $[\![\Gamma \vdash \lambda x : \sigma . M : \sigma \multimap \tau]\!] = \eta_{[\![\sigma]\!] \Rightarrow [\![\tau]\!]} \circ cur_{[\![\Gamma]\!], [\![\sigma]\!], [\![\tau]\!]}([\![M]\!])$
— $[\![\Gamma \vdash (M\ N) : \tau]\!] = app_{[\![\sigma]\!], [\![\tau]\!]} \circ \langle [\![M]\!], [\![N]\!] \rangle$
— $[\![\Gamma \vdash \mathcal{A}_\sigma(M) : \sigma]\!] = A_{[\![\sigma]\!]} \circ [\![M]\!]$
— $[\![\Gamma \vdash \mathcal{C}_\sigma(M) : \sigma]\!] = C_{[\![\sigma]\!]} \circ [\![M]\!]$

These clauses soundly interpret the equational theory from the last section.

Proposition 7. (Soundness) If $\Gamma \vdash M =_{\mathcal{C}} N : \sigma$ is derivable in $\lambda_{\mathcal{C}}$ then $[\![\Gamma \vdash M]\!] = [\![\Gamma \vdash N : \sigma]\!]$ in any interpretation.

Proof. By a straightforward but laborious induction on the term structure. The induction hypothesis includes that if $\Gamma \vdash V : \sigma$ is a value then $[\![V]\!] = \eta_\sigma \circ f$ for some morphism $f : [\![\Gamma]\!] \to [\![\sigma]\!]$. □

Clearly an interpretation is uniquely determined by its restriction to the base types. We could thus alternatively define an interpretation as an assignment of type objects to base types and in a second step define an interpretation function using the equations in Def. 6 as inductive clauses.

Remark 8. Using η we can transform elements of type A into elements of type R^{R^A}. The converse is in general impossible, consider *e.g.* the case where $R = 0$. In the special case, however, where $A = R$ the morphism η_A is a split mono with left inverse given by application to the identity. Since the interpretation of $\lambda_{\mathcal{C}}$ is uniform in R we can always

choose R to be some particular type of results. For example if $M : \beta$ then by choosing $R = [\![\beta]\!]$ we can obtain an element of $[\![\beta]\!]$ from $[\![M]\!]$ by composing with the left inverse to $\eta_{[\![\beta]\!]}$. This is known as the "(Harvey)-Friedman-trick" (Murthy 1991) and forms the heart of program extraction from classical proofs.

4.1. *Examples of models*

The most familiar example of a $\lambda_{\mathcal{C}}$-category is the category of sets and functions where the exponential is just the ordinary set of functions. The object R can be chosen arbitrarily; a possible choice which comes to mind is the set of strings on some alphabet with the intention that the final result of any computation is an output to the screen. Of course any other cartesian closed category with initial object, such as the category of ω-sets or various categories of predomains, forms a $\lambda_{\mathcal{C}}$-category, too. Finally we have the free model which corresponds to the informal $\underline{\lambda}$-calculus we introduced above to describe the various interpreting morphisms. Every equation derivable in this model (using full $\beta\eta$-equality) will hold in any $\lambda_{\mathcal{C}}$-category. So reasoning in this $\underline{\lambda}$-calculus provides a way of reasoning about $\lambda_{\mathcal{C}}$, by interpreting the two sides of a conjectured $\lambda_{\mathcal{C}}$-equation in the free model. However, in the next section we shall show that the $\lambda_{\mathcal{C}}$-axioms given in Section 1 are actually complete so that this translation becomes unnecessary.

5. Completeness of $\lambda_{\mathcal{C}}$

Theorem 9. (Completeness) If $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$ in every interpretation then $\Gamma \vdash M =_{\mathcal{C}} N : \sigma$ is derivable in $\lambda_{\mathcal{C}}$.

We postpone the proof until we shall have established the lemmas 10 and 11 below, and only sketch its structure here. From a given $\lambda_{\mathcal{C}}$-calculus we construct a syntactic category the morphisms of which will (roughly) be the *values* of the calculus. In Lemma 10 we show that this category is a $\lambda_{\mathcal{C}}$-category as defined above. We then show that the assignment which maps a term $\Gamma \vdash M : \sigma$ to the value $\Gamma \vdash \lambda\kappa{:}\sigma \multimap 0.\kappa\, M$ is an interpretation in the sense of Definition 6. So if two terms are equal in every interpretation then in particular the values assigned to them will be equal. Using $\mathcal{C}$ we can deduce from this that the terms themselves are equal.

This technique also works for free monads with $\mathcal{C}$ replaced by μ and $\lambda\kappa.\kappa\, x$ replaced by η. This is the idea of Moggi's (1991) proof of completeness of (the equational axioms of) his "simple programming language" (a calculus with first-order functions and a generic monad) with respect to interpretations in Kleisli categories.

The difficulty here consists of demonstrating that the category of values indeed supports the required structure and that composition with η constitutes an interpretation. It is not clear, how this can be done for other monads than the continuation monad, *e.g.* for continuations together with side effects.

5.1. *The category of values*

Let a $\lambda_{\mathcal{C}}$-calculus be given. We define its category of values, denoted $\mathcal{V}$, as follows. The objects of $\mathcal{V}$ are the contexts, *i.e.* lists of variable declarations. A morphism from $\Gamma = (x_1 : \sigma_1, \ldots, x_m : \sigma_m)$ to $(y_1 : \tau_1, \ldots, y_n : \tau_n)$ is an n-tuple of $=_{\mathcal{C}}$ -equivalence classes of values $(V_1, \ldots, V_n)$ such that $\Gamma \vdash V_i : \tau_i$. Composition is defined by simultaneous substitution. Observe that since we are dealing with values this notion of composition coincides with the usual one in terms of abstraction and application. The category of values is cartesian with terminal object given by the empty context and binary products given by juxtaposition of contexts. In view of our convention on α-equivalence we shall often identify contexts with lists of types and contexts of length one with types.

Lemma 10. The category of values $\mathcal{V}$ forms a $\lambda_{\mathcal{C}}$-category with type objects the types of $\lambda_{\mathcal{C}}$ and the following settings:

0 (initial object)	$:=$	0
R	$:=$	$(0 \multimap 0) \multimap 0$
R^σ	$:=$	$\sigma \multimap 0$
$(R^{R^\tau})^\sigma$	$:=$	$\sigma \multimap \tau$
$cur_{\Gamma,\sigma,R}(\Gamma, x{:}\sigma \vdash V : R)$	$:=$	$\Gamma \vdash \lambda x{:}\sigma.V\,(\lambda x{:}0.x) : \sigma \multimap 0$
$cur_{\Gamma,\sigma,R^{R^\tau}}(\Gamma, x{:}\sigma \vdash V : (\tau \multimap 0) \multimap 0)$	$:=$	$\Gamma \vdash \lambda x{:}\sigma.\mathcal{C}_\sigma(V) : \sigma \multimap \tau$

Proof. First we convince ourselves that these settings are type-correct by comparing the definition of $\mathcal{V}$ and the typing rules for $\lambda_{\mathcal{C}}$. As for the verifications we must show that 0 is indeed an initial object and that the two "cur"-functions are isomorphisms between the respective homsets and natural in Γ.

If $\Gamma = (\rho_1, \ldots, \rho_n)$ then the unique morphism $?_\Gamma : 0 \to \Gamma$ is the m-tuple of values $x{:}0 \vdash \mathcal{A}_{\rho_i}(x) : \rho_i$ for $i = 1 \ldots m$. Uniqueness is immediate from 0-Ini and Beta-V.

Now let σ, τ be types (type objects). For the required inverse to $cur_{\Gamma,\sigma,R}$ let $\Gamma \vdash V : \sigma \multimap 0$ be a value. Its "uncurried" version $cur^{-1}_{\Gamma,\sigma,R}(V)$ is

$$\Gamma, x : \sigma \vdash \lambda\kappa : 0 \multimap 0.\kappa\,(V\,x) : (0 \multimap 0) \multimap 0$$

To see that these two are inverse to each other let $\Gamma, x : \sigma \vdash V : (0 \multimap 0) \multimap 0$ be a value. We must show that

$$\Gamma, x : \sigma \vdash \lambda\kappa : 0 \multimap 0.\kappa\,(V\,(\lambda x : 0.x)) = V : (0 \multimap 0) \multimap 0$$

Now with rule 0-Endo we have $\kappa = \lambda x : 0.x$. So the left-hand side becomes $\lambda\kappa : 0 \multimap 0.V\,\kappa = V$ with rule Eta-V, exploiting the fact that V is a value. The other direction is immediate from Beta-V and Eta-V. Naturality of cur is actually a syntactic identity because by definition substitution commutes with the abstraction and application in $\lambda_{\mathcal{C}}$.

We turn to the exponential $(R^{R^\tau})^\sigma = \sigma \multimap \tau$. If $\Gamma \vdash V : \sigma \multimap \tau$ is a $\mathcal{V}$-morphism from Γ to $\sigma \Rightarrow \tau$ then

$$\Gamma, x : \sigma \vdash \lambda\kappa : \tau \multimap 0.\kappa\,(V\,x) : (\tau \multimap 0) \multimap 0$$

is a $\mathcal{V}$-morphism from $\Gamma, x{:}\sigma = \Gamma \times \sigma$ to $R^{R^\tau} = (\tau \multimap 0) \multimap 0$. To see that this assignment forms an inverse to $cur_{\Gamma,\sigma,R^{R^\tau}}$ let $\Gamma \vdash V : \sigma \multimap \tau$ be a value. We have

$$\lambda x : \sigma.\mathcal{C}_\sigma(\lambda\kappa : \tau \multimap 0.\kappa\,(V\,x)) = \lambda x : \sigma.V\,x = V$$

by rules $\mathcal{C}$-App and Eta-V. Conversely, let $\Gamma, x : \sigma \vdash V : (\tau \rightharpoonup 0) \rightharpoonup 0$ then

$$\begin{array}{lcr}
\lambda\kappa : \tau \rightharpoonup 0.\kappa\,((\lambda x : \sigma.\mathcal{C}_\tau(V))\,x) & = & \text{by Beta-V } (x \text{ is free in } V) \\
\lambda\kappa : \tau \rightharpoonup 0.\kappa\,\mathcal{C}_\tau(V) & = & \text{by } \mathcal{C}\text{-Nat applied to } \kappa \\
\lambda\kappa : \tau \rightharpoonup 0.\mathcal{C}_0(\lambda q : 0 \rightharpoonup 0.V\,(\lambda y : \tau.q\,(\kappa\,y))) & = & q = \mathrm{id}_0 \text{ by } \mathcal{A}_0\text{-Id} \\
\lambda\kappa : \tau \rightharpoonup 0.\mathcal{C}_0(\lambda q : 0 \rightharpoonup 0.q\,(V\,\kappa)) & = & \text{by } \mathcal{C}\text{-App} \\
\lambda\kappa : \tau \rightharpoonup 0.V\,\kappa & = & \text{by Eta-V} \\
V &&
\end{array}$$

Again, naturality of the isomorphism is a syntactic identity. □

We remark that this proof is the only place where we use that values are closed under $\mathcal{A}$. We could restrict the values to variables and abstractions but would then have to augment the morphisms in $\mathcal{V}$ by terms of the form $\mathcal{A}_\sigma(x)$ for x a variable. At any rate, the terms $\mathcal{A}_\sigma(V)$ *behave* like values in that Beta-V and Eta-V hold for them, simply because in the presence of a value V of type 0 any equation holds by $M = (\lambda x{:}0.M)V = (\lambda x{:}0.N)V = N$ where the second step is an instance of 0-Ini.

For the development which follows it is useful to elaborate the interpretation of the $\underline{\lambda}$-calculus in $\mathcal{V}$ which is implicit in Lemma 10 and the definition of $\mathcal{V}$. We have

$$\begin{array}{lcr}
\underline{\lambda}x{:}\sigma.V & = & \lambda x{:}\sigma.V\,(\lambda x{:}0.x) \\
U\,\mathrm{I}\,V & = & \lambda\kappa{:}0 \rightharpoonup 0.\kappa\,(U\;V) \\
\underline{\underline{\lambda}}x{:}\sigma.V & = & \lambda x{:}\sigma.\mathcal{C}(V) \\
U\,\mathrm{II}\,V & = & \lambda\kappa{:}\tau \rightharpoonup 0.\kappa\,(U\;V)
\end{array}$$

Our next step consists of interpreting the $\lambda_{\mathcal{C}}$-calculus in this category.

Lemma 11. The assignment

$$\begin{array}{l}
[\![\sigma]\!] := \sigma \\
[\![\Gamma]\!] := \Gamma \\
[\![\Gamma \vdash M : \sigma]\!] := \Gamma \vdash \lambda\kappa : \sigma \rightharpoonup 0.\kappa\,M : (\sigma \rightharpoonup 0) \rightharpoonup 0
\end{array}$$

is an interpretation.

Proof. We know from Lemma 10 that $(\sigma \rightharpoonup 0) \rightharpoonup 0 = R^{R^\sigma}$ and $\sigma \rightharpoonup \tau = (R^{R^\tau})^\sigma$ in $\mathcal{V}$. So the typing requirements are satisfied. It remains to show that $[\![-]\!]$ obeys the equations given in the definition of an interpretation. To facilitate this proof we first exhibit the canonical morphisms used to define the equations. To obtain the embedding η_σ we calculate as follows.

$$\begin{array}{lcr}
\eta_\sigma(x{:}\sigma) & = & \text{by definition} \\
\underline{\lambda}\kappa{:}R^\sigma.\kappa\,\mathrm{I}\,x & = & \text{by definition} \\
\lambda\kappa{:}\sigma \rightharpoonup 0.(\lambda k{:}0 \rightharpoonup 0.k\,(\kappa\,x))\,(\lambda x{:}0.x) & = & \text{by Beta-V and Ident} \\
\lambda\kappa{:}\sigma \rightharpoonup 0.\kappa\,x &&
\end{array}$$

The morphism $app : R^{R^{\sigma \Rightarrow \tau}} \times R^{R^\sigma} \to R^{R^\tau}$ arises in the following way by expanding the definitions and performing Beta-V-steps. Nnotice how κ moves in front of the innermost

application.

$$\begin{array}{ll} app_{\sigma,\tau}(F:((\sigma \multimap \tau) \multimap 0) \multimap 0, X:(\sigma \multimap 0) \multimap 0) & = \\ \underline{\lambda}\kappa{:}R^{\tau}.F^{\mathsf{I}}(\underline{\lambda}f{:}\sigma \Rightarrow \tau.X^{\mathsf{I}}(\underline{\lambda}x{:}\sigma.f^{\mathsf{II}}x^{\mathsf{I}}\kappa)) & = \\ \lambda\kappa{:}\tau \multimap 0.F\ (\underline{\lambda}f{:}\sigma \Rightarrow \tau.X^{\mathsf{I}}(\underline{\lambda}x{:}\sigma.f^{\mathsf{II}}x^{\mathsf{I}}\kappa)) & = \\ \lambda\kappa{:}\tau \multimap 0.F\ (\lambda f{:}\sigma \multimap \tau.X\ (\lambda x{:}\sigma.(f^{\mathsf{II}}x)\ \kappa)) & = \\ \lambda\kappa{:}\tau \multimap 0.F\ (\lambda f{:}\sigma \multimap \tau.X\ (\lambda x{:}\sigma.\kappa\ (f\ x))) & \end{array}$$

Similarly we derive the following expression for the morphism A_σ interpreting $\mathcal{A}$.

$$x{:}(0 \multimap 0) \multimap 0 \vdash \lambda k{:}\sigma \multimap 0.k\ \mathcal{A}_\sigma(\mathcal{C}_0(x))$$

which by $\mathcal{C}_0$-End, $\mathcal{A}_0$-Id, and $\mathcal{A}$-Abs is equal to

$$x : (0 \multimap 0) \multimap 0 \vdash \lambda k : \sigma \multimap 0.x\ (\lambda x : 0.x)$$

Finally the morphism C implementing $\mathcal{C}$ is given by

$$F : (((\sigma \multimap 0) \multimap 0) \multimap 0) \multimap 0 \lambda\kappa.F\ (\lambda f : (\sigma \multimap 0) \multimap 0.mf\ \kappa) : (\sigma \multimap 0) \multimap 0$$

We now consider the equations in order.

Variables. If $x : \sigma$ appears in Γ then

$$\begin{array}{ll} [\![x]\!] & = \\ \lambda\kappa : \sigma \multimap 0.\kappa\ x & = \\ \eta_\sigma(\pi_x) & \end{array}$$

because $\pi_x := \Gamma \vdash x : \sigma$ is the product projection corresponding to x in the category $\mathcal{V}$.

Abstraction. Let $\Gamma, x : \sigma \vdash M : \tau$.

$$\begin{array}{lll} [\![\lambda x : \sigma.M]\!] & = & \\ \lambda\kappa : (\sigma \multimap \tau) \multimap 0.\kappa\ (\lambda x : \sigma.M) & = & \text{by } \mathcal{C}\text{-App} \\ \lambda\kappa.\kappa\ (\lambda x.\mathcal{C}_\tau(\lambda k : \tau \multimap 0.k\ M)) & = & \\ \lambda\kappa.\kappa\ (\lambda x.\mathcal{C}_\tau([\![M]\!])) & = & \\ \eta_{\sigma\multimap\tau}(\lambda x.\mathcal{C}([\![M]\!])) & & \end{array}$$

which is the required expression since $\lambda x.\mathcal{C}_\sigma([\![M]\!])$ is the abstraction of $[\![M]\!]$ wrt. x in $\mathcal{V}$.

Application. Let $\Gamma \vdash M : \sigma \multimap \tau$ and $\Gamma \vdash N : \sigma$.

$$\begin{array}{lll} [\![M\ N]\!] & = & \text{by definition of } [\![.]\!] \\ \lambda\kappa : \tau \multimap 0.\kappa\ (M\ N) & = & \text{by App} \\ \lambda\kappa.\kappa\ ((\lambda m.m\ N)\ M) & = & \text{by Conv} \\ \lambda\kappa.(\lambda m.\kappa\ (m\ N))\ M & = & \text{by Beta-V} \\ \lambda\kappa.(\lambda k.k\ M)\ (\lambda m.\kappa\ (m\ N)) & = & \text{by Ass} \\ \lambda\kappa.(\lambda k.k\ M)\ (\lambda m.(\lambda n.(\kappa\ (m\ n)))\ N) & = & \text{by Beta-V} \\ \lambda\kappa.(\lambda k.k\ M)\ (\lambda m.(\lambda l.l\ N)\ (\lambda n.(\kappa\ (m\ n)))) & = & \text{by definition of } app. \\ app([\![M]\!], [\![N]\!]) & & \end{array}$$

$\mathcal{A}$ operator. Let $\Gamma \vdash M : 0$.

$$\begin{array}{lcl}
[\![\mathcal{A}_\sigma(M)]\!] & = & \text{by definition} \\
\lambda\kappa : \sigma \rightharpoonup 0.\kappa\,(\mathcal{A}_\sigma(M)) & = & \text{by } \mathcal{A}\text{-Abs} \\
\lambda\kappa.\mathcal{A}_0(M) & = & \text{by } \mathcal{A}_0\text{-Id} \\
\lambda\kappa.M & = & \text{by Ident} \\
\lambda\kappa.(\lambda x : 0.x)\,M & = & \text{by Beta-V} \\
\lambda\kappa.(\lambda k.k\ M)(\lambda x : 0.x) & = & \\
A_\sigma([\![M]\!]) & &
\end{array}$$

$\mathcal{C}$ operator. If $\Gamma \vdash M : (\sigma \rightharpoonup 0) \rightharpoonup 0$ then

$$\begin{array}{lcl}
[\![\mathcal{C}_\sigma(M)]\!] & = & \\
\lambda\kappa : \sigma \rightharpoonup 0.\kappa\,\mathcal{C}(M) & = & \text{by } \mathcal{C}\text{-Nat} \\
\lambda\kappa.\mathcal{C}_0(\lambda k : 0 \rightharpoonup 0.M\ (\lambda x : \sigma.k\ (\kappa\ x))) & = & \text{by 0-Endo and Ident} \\
\lambda\kappa.\mathcal{C}_0(\lambda k.k\ (M\ (\lambda x : \sigma.\kappa\ x))) & = & \text{by } \mathcal{C}\text{-App and Eta-V} \\
\lambda\kappa.M\ (\kappa) & = & \text{by App} \\
\lambda\kappa.(\lambda m : (\sigma \rightharpoonup 0) \rightharpoonup 0.m\ \kappa)\ M & = & \text{by Beta-V} \\
\lambda\kappa.(\lambda k : ((\sigma \rightharpoonup 0) \rightharpoonup 0) \rightharpoonup 0.k\ M)\ (\lambda m.m\ \kappa) & = & \\
\lambda\kappa.[\![M]\!]\ (\lambda m.m\ \kappa) & = & \\
C([\![M]\!]) & &
\end{array}$$

This completes the proof of Lemma 11 □

5.2. *Proof of Theorem 9*

Let $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$ be arbitrary terms of $\lambda_\mathcal{C}$. If $[\![M]\!] = [\![N]\!]$ for any interpretation of $\lambda_\mathcal{C}$ then in particular

$$\lambda\kappa : \sigma \rightharpoonup 0.\kappa\ M =_\mathcal{C} \lambda\kappa : \sigma \rightharpoonup 0.\kappa\ N$$

using the interpretation defined in Lemma 11. Applying $\mathcal{C}$ to both sides of this identity yields $M =_\mathcal{C} N$ using $\mathcal{C}$-App. □

6. Other control operators

6.1. *Axiomatisation of callcc*

Instead of $\mathcal{C}$ we may also use the *callcc*-operator

$$callcc_\sigma : ((\sigma \rightharpoonup 0) \rightharpoonup \sigma) \rightharpoonup \sigma$$

which provides explicit access to the current continuation. $\mathcal{C}$ and *callcc* are expressible in terms of each other. If $\Gamma \vdash M : (\sigma \rightharpoonup 0) \rightharpoonup \sigma$ then define

$$callcc_\sigma(M) \quad := \quad \mathcal{C}_\sigma(\lambda\kappa : \sigma \rightharpoonup 0.\kappa\,(M\ \kappa))$$

Conversely, if we consider *callcc* as primitive then if $\Gamma \vdash M : (\sigma \rightharpoonup 0) \rightharpoonup 0$ we can define

$$\mathcal{C}_\sigma(M) := callcc_\sigma(\lambda\kappa : \sigma \rightharpoonup 0.\mathcal{A}_\sigma(M\ \kappa))$$

For *callcc* defined in terms of $\mathcal{C}$ the following two derived rules hold for V ranging over values and M ranging over terms.

$$\begin{array}{ll} callcc_\sigma(\lambda\kappa:\sigma\multimap 0.V\ (\kappa\ M)) =_{\mathcal{C}} M & callcc\text{-}\textsc{App} \\ V\ callcc_\sigma(M) =_{\mathcal{C}} callcc_\tau \lambda\kappa:\tau\multimap 0.V\ (M\ (\lambda x:\sigma.\kappa\ (V\ x))) & callcc\text{-}\textsc{Nat} \end{array}$$

These equations in turn permit to derive the axioms for $\mathcal{C}$ when expressed in terms of *callcc* whence we obtain

Proposition 12. The calculus λ_{callcc} obtained from $\lambda_{\mathcal{C}}$ by omitting $\mathcal{C}$ and the corresponding equality rules and adding *callcc* together with the axioms *callcc*-App and *callcc*-Nat is sound and complete for equality in $\lambda_{\mathcal{C}}$-models with

$$[\![\Gamma\vdash callcc_\sigma(M)]\!] = callcc_{[\![\sigma]\!]}\circ[\![M]\!]$$

where

$$callcc_A(\Phi:R^{R^{(A\Rightarrow 0)\Rightarrow A}}) := \underline{\lambda}k:R^A.\Phi^{\mathsf{l}}(\underline{\lambda}\phi{:}(A\Rightarrow 0)\Rightarrow A.\phi^{\mathsf{l}}(\underline{\lambda}a:A.\underline{\lambda}q:R^0.k^{\mathsf{l}}a)^{\mathsf{l}}k)$$

Proof. By simple equality reasoning □

Instead of *callcc* as defined above one may also introduce a combinator corresponding to Peirce's law.

$$\frac{\Gamma\vdash M:(\sigma\multimap\tau)\multimap\sigma}{\Gamma\vdash call/cc_{\sigma,\tau}(M):\sigma}$$

It can be expressed in terms of *callcc* by

$$call/cc_{\sigma,\tau}(M) := callcc_\sigma(\lambda\kappa:\sigma\multimap 0.M\ (\lambda x:\sigma.\mathcal{A}_\tau(\kappa\ x)))$$

Using this translation the following three rules can be derived for *call/cc* which can again be shown to be complete. As usual U, V range over values and M ranges over terms.

$$\begin{array}{l} call/cc_{\sigma,\tau}(\lambda\kappa:\sigma\multimap\tau.V\ (\kappa\ M)) =_{\mathcal{C}} M \\ call/cc_{\sigma,\tau}(M) =_{\mathcal{C}} call/cc_{\sigma,\tau'}(\lambda\kappa:\sigma\multimap\tau'.M\ (\lambda x:\sigma V\ (\kappa\ x))) \\ V\ call/cc_{\sigma,\tau}(M) =_{\mathcal{C}} call/cc_{\sigma',\tau}(\lambda\kappa:\sigma'\multimap\tau.V\ (M\ (\lambda x:\sigma.\kappa\ (V\ x)))) \end{array}$$

Although the type of *call/cc* does not involve the empty type 0 its presence (together with $\mathcal{A}$) is still required for the translation into $\lambda_{\mathcal{C}}$ and hence the completeness proof. We do not know whether the axioms for *call/cc* are complete if 0 and $\mathcal{A}$ are not part of the calculus.

7. Completeness for cartesian closed models

As noted in Section 4 every cartesian closed category with an initial object gives rise to a $\lambda_{\mathcal{C}}$-category. Our aim is to show that in these models not more equations hold than in arbitrary models.

Proposition 13. If $[\![\Gamma\vdash M:\sigma]\!] = [\![\Gamma\vdash N:\sigma]\!]$ in every interpretation in a cartesian closed $\lambda_{\mathcal{C}}$-category, then $\Gamma\vdash M =_{\mathcal{C}} N:\sigma$ is derivable in $\lambda_{\mathcal{C}}$.

To prove this we make use of the following lemma which was pointed out to the author by Alex Simpson. Its proof uses slightly more involved categorical machinery; the required

material may be found in Chapters 0.2 and II.9 of (Lambek & Scott 1985) except for the well-known fact that the Yoneda embedding preserves products and exponentials, which follows by routine calculations.

Lemma 14. Let $\mathcal{K}$ be a category with an initial object 0 such that every morphism into 0 is an isomorphism. Then there exists a cartesian closed category $\tilde{\mathcal{K}}$ with an initial object and a full and faithful functor $\mathcal{Y} : \mathcal{K} \to \tilde{\mathcal{K}}$ which preserves the initial object and all products and exponentials which exist in $\mathcal{K}$.

Proof. We let $\tilde{\mathcal{K}}$ be the subcategory of the functor category $\hat{\mathcal{K}} := \mathbf{Sets}^{\mathcal{K}^{op}}$ consisting of those functors F for which $F(0)$ is a singleton set. Notice that this implies that $F(X)$ is a singleton for X *isomorphic* to 0, since every functor preserves isomorphisms. Since $\mathcal{K}(0, X)$ is a singleton set, the Yoneda embedding sending an object X in $\mathcal{K}$ to the representable functor $\mathcal{K}(-, X)$ in $\hat{\mathcal{K}}$ restricts to a functor from $\mathcal{K}$ to $\tilde{\mathcal{K}}$. This gives the embedding $\mathcal{Y}$ of $\mathcal{K}$ into $\tilde{\mathcal{K}}$. It is known (and it actually follows by straightforward calculations) that the Yoneda embedding and hence the functor $\mathcal{Y}$ is full and faithful and preserves all limits and exponentials that exist in $\mathcal{K}$. We must show that $\mathcal{Y}(0)$ is initial and that $\tilde{\mathcal{K}}$ is closed under products and exponentials. Recall that the functor category $\hat{\mathcal{K}}$ is cartesian closed. Let F be an object in $\tilde{\mathcal{K}}$. We define the natural transformation $?(F) : \mathcal{Y}(0) \to F$ by letting $?(F)_X$ be the unique function between singleton sets if X is isomorphic to 0 and the empty function if X is not isomorphic to 0. Here we use the assumption that $\mathcal{Y}(0)(X) = \mathcal{K}(X, 0)$ is empty for X not isomorphic to 0. It is clear that this natural transformation is the only one from $\mathcal{Y}(0)$ to F; so $\mathcal{Y}(0)$ is initial.
Now since products are taken pointwise in $\hat{\mathcal{K}}$ it follows immediately that $\tilde{\mathcal{K}}$ is closed under products. It remains to show that $\mathcal{Y}(0)$ is also closed under exponentiation. Let F, G be objects in $\tilde{\mathcal{K}}$, *i.e.* two functors from $\mathcal{K}^{op}$ to **Sets** such that $F(0)$ and $G(0)$ are singleton sets. Their exponential in $\mathbf{Sets}^{\mathcal{K}^{op}}$ is the functor sending X to the set of natural transformations from the (pointwise) product $\mathcal{Y}(X) \times F$ to G. So we must show that there is exactly one natural transformation from $\mathcal{Y}(0) \times F$ to G. This follows using the argument in the proof of initiality of $\mathcal{Y}(0)$ above. □

Remark 15. The category $\tilde{\mathcal{K}}$ is the category of *sheaves* for a suitable Grothendieck topology on $\mathcal{K}$. This gives a more elegant and shorter, but less elementary proof of the above lemma.

In more logical terms the above lemma states that adding higher-order functions to some typed equational theory is a conservative extension and that already existing function types and the empty type are not changed by this addition. It is not clear what conditions other more complicated data types have to satisfy so as to be preserved by this extension.

Moggi (1991) uses a similar technique to establish conservativity of higher-order logic over the equational theory of monads. Since he does not need to preserve the initial object the ordinary Yoneda embedding into $\mathbf{Sets}^{\mathcal{K}^{op}}$ does the job.

7.1. *Proof of Proposition 13*

Our aim is to apply the above construction to the category $\mathcal{V}$ of values introduced in Section 9. Let $\Gamma \vdash V : 0$ be a $\mathcal{V}$-morphism from some context $\Gamma = (x_1 : \sigma_1, \ldots, x_n : \sigma_n)$

into the initial object 0. We must show that this is an inverse to the unique morphism from $x : 0$ to Γ given by $(\mathcal{A}_{\sigma_1}(x), \ldots, \mathcal{A}_{\sigma_n}(x))$.

Since 0 has no nontrivial endomorphisms this boils down to showing that

$$\Gamma \vdash \mathcal{A}_{\sigma_i}(V) = x_i : \sigma_i$$

for each declaration $x_i : \sigma_i$ in Γ. Since V is a value we can BETA-V-expand x_i to $(\lambda y : 0.x_i)\, V$. Now the left part being a value of type $0 \multimap \sigma_i$ equals $\lambda y : 0.\mathcal{A}_{\sigma_i}(y)$. Thus by BETA-V-contraction we obtain the r.h.s. .

So the category of values satisfies the requirements of Lemma 14 and we obtain a full and faithful embedding $\mathcal{Y}$ into a cartesian closed category. Since this embedding preserves the cartesian closed structure and the initial object, the composition of the interpretation $[\![M]\!] := \lambda k.k\, M$ with $\mathcal{Y}$ is an interpretation again. So if $[\![\Gamma \vdash M : \sigma]\!] = [\![\Gamma \vdash N : \sigma]\!]$ in every interpretation $[\![-]\!]$ in a cartesian closed category then in particular

$$\mathcal{Y}(\Gamma \vdash \lambda\kappa : \sigma \multimap 0.\kappa\, M) = \mathcal{Y}(\Gamma \vdash \lambda\kappa : \sigma \multimap 0.\kappa\, N)$$

Since $\mathcal{Y}$ is faithful this implies that $\lambda\kappa.\kappa\, M =_{\mathcal{C}} \lambda\kappa.\kappa\, N$ so $M =_{\mathcal{C}} N$ by $\mathcal{C}$-APP. □

8. Conclusions and directions for further research

We have given a complete axiomatisation of various control operators with respect to their interpretation in call-by-value continuation passing style. This means that in order to reason about a program involving control operators it is no longer necessary to translate into cps — it can directly be described using the axioms. Moreover, in various examples reasoning with the axioms appears easier and more intuitive than using the cps translation. We do not know, however, whether the equations we give can be directed so as to yield a calculus in which programs with control operators can be executed directly. It seems that for mere execution not all of the axioms are needed, since the calculus described in (Felleisen *et al.* 1987) only contains a subset of our equations and is proven to be adequate with respect to a certain machine model.

Quite a number of directions for further research suggest themselves. We have neither allowed constants nor additional equations (δ-equations) describing their behaviour. This means that so far our calculus can not even in principle be used as a programming language. Preliminary work suggests that as far as inductive types like the natural numbers or lists are to be included, completeness can be carried over.

The completeness proof is in a certain sense modular, namely the category of values is defined without mention to the particular feature one is interested in, here the control operators. One may thus try to carry out a similar programme for λ-calculi with more refined features like side-effects or addition of exceptions and hope to find the right set of axioms as one tries to identify the necessary structure in the category of values. Indeed, the rule APP was found in this way.

For various reasons one may object against the heavy use of the empty type in the present development. A possible solution which has also been adopted for the continuation facility in New Jersey ML (Appel 1992) consists of introducing a new type operator *cont* which associates to each type σ the type $cont(\sigma)$ of σ-continuations. One may

then introduce a $\mathcal{C}$ operator of type $cont(cont(\sigma)) \rightharpoonup \sigma$ or a *callcc*-operator of type $(cont(\sigma) \rightharpoonup \sigma) \rightharpoonup \sigma$. Moreover, one needs a facility to invoke continuations: $throw : cont(\sigma) \rightharpoonup \sigma \rightharpoonup \tau$. Finally one needs a means to apply functions to continuations. If $f : \sigma \rightharpoonup \tau$ and $k : cont(\tau)$ then $capp(f,k) : cont(\sigma)$. This *capp* operator can be defined in terms of *callcc* and *throw* as

$$capp(f,k) = callcc(\lambda\kappa{:}cont(cont(\sigma)).throw\ k\ f(callcc(\lambda q{:}cont(\sigma).throw\ \kappa\ q)))$$

but in view of an axiomatisation it appears better to introduce it as a primitive. We conjecture that for this calculus a complete axiomatisation can be given along the lines of this article provided the calculus contains a unit type (1) which is a terminal object for the values. Then in the category of values we can put $R := cont(1)$ and prove that $cont(\sigma)$ and $\sigma \rightharpoonup \tau$ are the exponentials R^σ and $(R^{R^\tau})^\sigma$, respectively.

Another interesting task would be an axiomatisation of *call by name* continuation passing style. Here unrestricted β and η rules will hold, however the axioms for the control operator will undergo certain restrictions.

One may also try to apply the present framework to Felleisen's untyped calculi using cartesian closed categories with reflexive objects (Lambek & Scott 1985, Ch. I.15)

Finally, it remains to be seen whether axiomatisations like the one proposed here provide useful for reasoning about programs used in practice. We have successfully used the axioms to verify a simple program from (Appel 1992) which uses continuations as an imperative abort facility. Unfortunately, it seems to be the case that the more interesting applications of control operators make use either of references or of recursive datatypes, which both are not yet fully understood logically.

Acknowledgement

Matthias Felleisen and Thomas Streicher made helpful remarks on earlier versions; an anonymous referee suggested various improvements of presentation and notation.

References

Boris Agapiev and Eugenio Moggi. Declarative Continuations and Monads. unpublished draft, July 1991.

Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

Michael Barr and Charles Wells. *Category Theory for Computing Science.* International Series in Computer Science. Prentice Hall, 1990.

Matthias Felleisen *et al.* A Syntactic Theory of Sequential Control. *TCS*, 52:205–237, 1987.

Andrzej Filinski. Declarative Comtinuations and Categorical Duality. Computer Science Department, University of Copenhagen, DIKU Report 89/11, August 1989. Master's thesis.

Timothy Griffin. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 47–58, 1990.

Bart Jacobs. *Categorical Type Theory.* PhD thesis, University of Nijmegen, 1991.

Jean-Louis Krivine. Classical Logic, Storage Operators, and Second Order Lambda-Calculus. unpublished note.

Joachim Lambek and Philip Scott. *Introduction to Higher-Order Categorical Logic.* Cambridge University Press, 1985.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

Chetan Murthy. An evaluation semantics for classical proofs. In *Proceedings of the sixth IEEE Symposium on Logic in Computer Science (LICS)*, 1991.

Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

J. Rees and W. Clinger. Revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, 1986.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740, 1972.

Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358, 1993.

Carolyn Talcott. A theory for program and data specification. *Theor. Comput. Sci.*, 10(1), 1992.