J. Functional Programming **8** (6): 543–572, November 1998. Printed in the United Kingdom

Classical logic, continuation semantics and abstract machines

Th. STREICHER
Fachbereich 4 Mathematik, TU Darmstadt,
Schlossgartenstr. 7, 64289 Darmstadt, Germany
(*e-mail:* `streicher@mathematik.tu-darmstadt.de`)

B. REUS
Institut für Informatik, Ludwig-Maximilians-Universität,
Oettingenstr. 67, D-80538 München, Germany
(*e-mail:* `reus@informatik.uni-muenchen.de`)

Abstract

One of the goals of this paper is to demonstrate that denotational semantics is useful for operational issues like implementation of functional languages by abstract machines. This is exemplified in a tutorial way by studying the case of extensional untyped call-by-name λ-calculus with Felleisen's control operator $\mathscr{C}$. We derive the transition rules for an abstract machine from a continuation semantics which appears as a generalization of the $\neg\neg$-translation known from logic. The resulting abstract machine appears as an extension of Krivine's machine implementing head reduction. Though the result, namely Krivine's machine, is well known our method of deriving it from continuation semantics is new and applicable to other languages (as e.g. call-by-value variants). Further new results are that Scott's D_∞-models are all instances of continuation models. Moreover, we extend our continuation semantics to Parigot's $\lambda\mu$-calculus from which we derive an extension of Krivine's machine for $\lambda\mu$-calculus. The relation between continuation semantics and the abstract machines is made precise by proving computational adequacy results employing an elegant method introduced by Pitts.

Capsule Review

In this paper the authors employ a "proof-relevant" version of a double negation translation (due to Krivine and Girard) giving rise to a so-called "category of negative domains" $\mathscr{N}_R$ which is the full subcategory of the category of predomains on objects of the form R^X for some predomain X where R is a domain (i.e. a predomain with $\perp$) chosen in advance. The idea behind this is that a "classical proposition" is simply an intuitionistic negation of an intuitionistic proposition.

The category $\mathscr{N}_R$ is a well-pointed cartesian closed category with least fixpoints. Moreover, it allows to interpret "control features" as Felleisen's control operator $\mathscr{C}$ and Parigot's $\lambda\mu$-calculus (which can be seen as calculi whose terms represent proofs of classical (propositional) logic). Moreover, in $\mathscr{N}_R$ one may find models of the untyped λ-calculus with control, namely the negative domain R^C where C is the solution of the domain equation $C \cong R^C \times C$. It is observed in the paper that D. Scott's D_∞-models of untyped λ-calculus are subsumed amongst these by putting $D = R$.

The main point made in this paper is that, when unfolding the interpretation of untyped λ-calculus in R^C, the ensuing semantic equations correspond to the transition rules of Krivine's machine for computing weak head normal forms of λ-terms. This extends to λ-calculi with

control. This observation is made precise by extending denotational semantics to the abstract machines and proving correctness and computational adequacy (for the case where R is the two element lattice).

The paper gives a *reconstruction* of well-known operational results (Krivine's machine) based on *continuation semantics* where continuations constitute the denotational analogue of evaluation contexts. But, moreover, these results are extended to Parigot's $\lambda\mu$-calculus providing a machine for it that appears to be new in the literature (but has been found independently by Ph. deGroote by purely proof-theoretic methods).

1 Introduction and motivation

Continuation-passing-style (*cps*) translations of call-by-value λ-calculus were introduced originally by Fischer (1972) and Reynolds (1972) in the early 1970s. From its very beginning *continuations* were thought of as analogues of the operational notion of *evaluation context*. An early study of the cps-translation for λ-calculi can be found in Plotkin (1975). In *loc.cit.* Plotkin had already introduced a call-by-name variant of the cps-translation which was later taken up again by, for example, Okasaki *et al.* (1994), where this call-by-name cps-translation has been reformulated on a semantical level as an appropriate continuation semantics. A semantic version of the call-by-value cps-translation has been studied as a special instance of Moggi's computational monads, the so-called continuation monad (Moggi, 1991).

The relation between cps-translation and abstract machines for call-by-value λ-calculus with control was studied by Felleisen and his colleagues starting from the mid-1980s (Felleisen, 1986; Felleisen and Friedman, 1986). Over the years, this method has been developed to an engineering tool for compiler construction (Appel, 1992). Besides this, in a sequence of papers, Felleisen and his collaborators have studied equational axiomatizations of the cps-translation of call-by-value λ-calculus with control (Sabry and Felleisen, 1992).

All the above-mentioned cps-translations and continuation semantics comprise a notion of *value* even for the call-by-name variants. Consequently, these cps-translations and continuation semantics do not validate the η-rule. In any case, Lafont (1991) introduced an elegant $\neg\neg$-translation of classical propositional logic to the $\neg,\wedge$-fragment of intuitionistic propositional logic based on previous work by Girard and Krivine. It is different from Gödel's and Kolmogoroff's $\neg\neg$-translations which correspond to a call-by-name cps-translation with values and a call-by-value one, respectively. As constructive logic has a *proof semantics* corresponding to a (model of a) simple functional language, such a translation of classical to constructive logic gives rise to a *proof semantics for classical logic*. It was made clear by Lafont in *loc.cit.* that also his $\neg\neg$-translation can be understood as a cps-translation of call-by-name λ-calculus with control to a particular fragment of λ-calculus corresponding to the $\neg,\wedge$-fragment of intuitionistic logic. A semantic analogue of Lafont's new cps-translation was studied and extended to *PCF* with control (and input/output) by Lafont *et al.* (1993). The distinguishing feature of this cps-translation and the corresponding continuation semantics is that it does not admit a basic notion of value but, instead, a basic notion of continuation.

Continuation semantics à la Lafont gives rise to a cartesian closed category, the category of 'negated domains'.

This category $\mathscr{N}_R$ appears as the full subcategory of the category of domains and continuous functions on objects of the form R^A where A is a predomain and R is some fixed domain of 'responses'. This domain $R \cong R^1$ is the meaning of the proposition $\bot$. Interpreting λ-calculus in $\mathscr{N}_R$ the denotation of a λ-term is an object of R^C mapping elements of C – so-called 'continuations' – to 'responses' or 'answers', i.e. elements of R. Accordingly, elements of R^C are called 'denotations'.

Due to the isomorphism $(R^B)^{(R^A)} \cong R^{R^A \times B}$ we get that the predomain of continuations for the exponential $(R^B)^{(R^A)}$ is $R^A \times B$. This means that a continuation for a function f from R^A to R^B is a pair $\langle d, k \rangle$ where $d \in R^A$ is an argument for f and $k \in B$ is a continuation for $f(d)$.

Due to this simple construction of function spaces in $\mathscr{N}_R$ we get that $\neg R^A \cong R^{R^A}$ as $\neg R^A$ is defined as $R^A \Rightarrow R^1$ which is $R^{R^A \times 1} \cong R^{R^A}$. Moreover, the canonical map from $R^{R^{R^A}}$ to R^A sending Φ to $\lambda a{:}A.\,\Phi(\lambda f{:}R^A.\,f(a)) \in R^A$ provides an interpretation of the classical proof principle $\neg\neg P \Rightarrow P$ (*reductio ad absurdum*). It is (a variant of) this interpretation of *reductio ad absurdum* which will be assigned as meaning to the control operator $\mathscr{C}$ originally introduced by Felleisen (1986). The idea to understand the control operator $\mathscr{C}$ as a proof of *reductio ad absurdum* via the principle of propositions-as-types was first introduced by Griffin (1990).

To interpret untyped λ-calculus in $\mathscr{N}_R$ one has to exhibit a so-called *reflexive object* in $\mathscr{N}_R$ i.e. a C with $R^C \cong R^{R^C \times C}$. For this purpose, it suffices to provide a domain C with $C = R^C \times C$. Reflexive objects in $\mathscr{N}_R$ of this form will be called *continuation models* of untyped λ-calculus. It turns out that these – up to isomorphism – coincide with Scott's D_∞-models.

The point we try to make in this paper is that the category of negated domains arises from fairly simple 'logical' considerations without any *a priori* operational motivation. Furthermore, it turns out that the interpretation of λ-calculus in the category of negated domains extends easily to an interpretation of an untyped version of Parigot's $\lambda\mu$-calculus[1] (cf. Parigot (1992)), where continuations can be referred to by continuation variables that can be bound by μ-abstraction. Accordingly, one has more freedom in expressing control structure than by Felleisen's control operator $\mathscr{C}$.

Parigot's 'labelling' $[\alpha]M$ is interpreted as application of the meaning of M – an element of $D = R^C$ – to the continuation bound to α thus giving rise to an element of R. Parigot's μ-abstraction $\mu\alpha.\,t$ is interpreted as functional abstraction over the continuation variable α on the level of continuation semantics. As objects of $C = D \times C$ can be considered as (lazy) stacks of denotations it is natural to extend the $\lambda\mu$-calculus by allowing more general *continuation terms* than mere continuation variables namely stack expressions of the form $M_1 :: \ldots M_n :: \alpha$. Using this semantically motivated extension we obtain a simplification of Parigot's equational theory getting rid of his 'mixed substitution' and replacing it by ordinary substitution of continuation terms for continuation variables.

[1] Parigot's $\lambda\mu$-calculus, however, was invented for the purpose of representing proofs in a natural deduction formulation of classical propositional logic by terms.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

As mentioned before, Felleisen and others have used cps-translation for deriving abstract machines for the call-by-value λ-calculus with control. In this paper, we use continuation semantics à la Lafont for deriving Krivine's machine. It turns out that the semantic equations of the interpretation of λ-calculus in the category of negated domains are in 1-1-correspondence with the transition rules of Krivine's machine, the world's simplest machine interpreting λ-calculus. All this extends easily to λ-calculus with control and also $\lambda\mu$-calculus.

This way the partial correctness of Krivine's machine follows easily from the way it is derived from continuation semantics. The correspondence is given by identifying expressions of the form $[\![M]\!]\, e\, k$, i.e. the meaning of term M in environment e applied to continuation k, with the states of Krivine's machine, i.e. expressions of the form $\langle\, [M, env],\, S\,\rangle$ where env is an environment assigning closures to variables and S is a stack of closures.

For a moderate extension of Krivine's machine (computing head normal forms and not only weak head normal forms) we can prove computational adequacy in a very semantic way employing the technique of 'inclusive predicates'. This goes back to Reynolds and was simplified and extended to untyped languages by Pitts (1994) using recent methods arising from Freyd's category-theoretic analysis of recursive domain equations (Freyd, 1992).

For the case of $\lambda\mu$-calculus a similar machine has been obtained by P. de Groote via purely syntactic methods in de Groote (1996) which, however, seems to be more complicated.

A different relation between denotational semantics and implementations of functional languages has been investigated by Jeffrey (1994). There, it has been shown that the initial/terminal solution of $D = [D \to D]_\perp$ provides a fully abstract model for concurrent graph reduction for an untyped lazy λ-calculus with recursive declarations and a parallel convergence tester. The models, presented in this paper, are not fully abstract for operational semantics as given by our abstract machines. This could be achieved, however, by extending them in such a way that they implement a parallel convergence tester as well. In contrast to our work, Jeffrey starts with a given operational semantics and proves that the (obvious) Scott model for it is actually fully abstract, whereas we derive operational semantics from a denotational semantics, namely a continuation semantics arising from a generalization of the $\neg\neg$–translation of classical to intuitionistic logic.

2 The category $\mathcal{N}_R$ of negated domains

In this section we describe a category of 'negated domains' originally introduced by Lafont *et al.* (1993), where terms of λ-calculi with control will be assigned their meaning.

Ordinary 'direct' semantics lives in the category $\mathscr{P}$ of (pre)domains and Scott continuous functions. In our context, a *predomain* is simply a partial order having suprema of all directed subsets but not necessarily a least element. A function between predomains is *Scott continuous* iff it preserves suprema of directed sets. A *domain* is a predomain that has also a least element, called *bottom element*. The corresponding full subcategory of domains will be referred to as $\mathscr{D}$. Notice that a continuous function between domains need not preserve bottom elements. If it does it is called *strict*. We write $\mathscr{D}_\perp$ for the category of domains with strict maps as morphisms.

We present a similarly general framework for continuation semantics: the *category of negated domains* $\mathscr{N}_R$ which is parameterized by an arbitrary *domain* R of *responses*. We assume R to have a least element in order to guarantee that $\mathscr{N}_R$ has a (least) fixpoint operator.

Before giving the precise definition of $\mathscr{N}_R$ we provide some motivation considering the semantics of classical proofs.

In the 1930s, Kurt Gödel showed how classical logic can be translated into intuitionistic logic by his famous 'double negation translation' explained, for example, in Troelstra and van Dalen (1988). Though this can be done syntactically, we prefer to explain Gödel's double negation translation in terms of *truth value semantics*.

Let A be a *Heyting algebra*, i.e. a lattice together with a binary operation $\rightarrow$: $A \times A \rightarrow A$ such that for all $a, b, c \in A$ we have $c \leqslant a \rightarrow b$ iff $c \wedge a \leqslant b$. Notice that the operation $\rightarrow$ is determined uniquely already by the lattice structure of A. Now for all $r \in A$ (including the least element of lattice A)

$$A^r = \{a \rightarrow r \mid a \in A\}$$

is a *Boolean algebra* w.r.t. the partial order inherited from A. The *Boolean negation* of $a \in A^r$ is given by $a \rightarrow r$. Notice that infima and $\rightarrow$ are inherited from A but r is the least element in A^r and the supremum of a and b in A^r is given by $((a \rightarrow r) \wedge (b \rightarrow r)) \rightarrow r$.

The definition of $\mathscr{N}_R$ is motivated by lifting this simple construction from truth values semantics, i.e. Heyting algebras, to *proof semantics*, i.e. cartesian closed categories with finite coproducts. This way one obtains a *proof semantics for classical logic* as will be shown subsequently.

Definition 2.1

The category $\mathscr{N}_R$ of *negated domains* is defined as follows. The objects of $\mathscr{N}_R$ are the objects of $\mathscr{P}$ and $\mathscr{N}_R(A, B) = \mathscr{P}(R^A, R^B)$, i.e. a morphism in $\mathscr{N}_R$ from A to B is a morphism in $\mathscr{P}$ from R^A to R^B. Composition of morphisms in $\mathscr{N}_R$ is inherited from $\mathscr{P}$.

Thus, the category $\mathscr{N}_R$ is *equivalent* to the full subcategory of $\mathscr{P}$ on powers of R. As R has a least element by assumption any of its powers has a least element, too. Therefore, $\mathscr{N}_R$ is equivalent actually to a full subcategory of the category of domains and all continuous functions.

Next we show that the category $\mathscr{N}_R$ is still well-behaved in the sense that it has enough structure to interpret functional programs.

Theorem 2.1

For any domain R the category $\mathscr{N}_R$ is cartesian closed and admits a least fixpoint operator.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

Proof
As the category of predomains has categorical sums we have the isomorphisms $R^A \times R^B \cong R^{A+B}$. Therefore, $\mathscr{N}_R$ has cartesian products. The terminal object in $\mathscr{N}_R$ is given by the empty predomain 0 as $R^0 \cong 1$ contains precisely one element. Due to the isomorphism $(R^B)^{(R^A)} \cong R^{R^A \times B}$ we get that $\mathscr{N}_R$ is also closed under function spaces. For any predomain A the predomain R^A has a least element $\perp_{R^A} = \lambda x{:}A.\perp_R$. Thus, any $f \in \mathscr{N}_R(A, A) = \mathscr{P}(R^A, R^A)$ has the least fixpoint $\bigsqcup_{n\in\mathbb{N}} f^n(\perp_{R^A})$. □

Remark 2.1
Notice that for the existence of cartesian products in $\mathscr{N}_R$ it is essential to have predomains instead of only domains because the category of domains and continuous functions lacks sums in the categorical sense.

Theorem 2.1 suggests notation as fixed in the following definition.

Definition 2.2
In $\mathscr{N}_R$ we write cartesian product as $A \wedge B := A + B$ and function space (exponentiation) as $A \Rightarrow B := R^A \times B$.

Next we show how to interpret 'classical negation' in $\mathscr{N}_R$.

Definition 2.3
We write $\perp$ ('falsity') for the terminal predomain $1 = \{\star\}$ considered as an object on $\mathscr{N}_R$. For any A in $\mathscr{N}_R$ let $\neg A := A \Rightarrow \perp$ which abbreviates $R^A \times 1$.

Next we show that this notion of negation actually behaves classically, i.e. for any A in $\mathscr{N}_R$ there is a morphism $\mathscr{C}_A : \neg\neg A \to A$ in $\mathscr{N}_R$ corresponding to *reductio ad absurdum* distinguishing classical logic from intuitionistic logic.

Theorem 2.2
For any A in $\mathscr{N}_R$ let $\eta_A : A \to \neg\neg A$ and $\mathscr{C}_A : \neg\neg A \to A$ be the *morphisms in* $\mathscr{N}_R$ such that

$$\eta_A(a)\langle f, \star\rangle = f\langle a, \star\rangle$$

for all $a \in R^A$ and $f \in R^{\neg A}$ and

$$\mathscr{C}_A(f)(k) = f\,\langle(\lambda\langle a, h\rangle{:}\neg A.\, a(k)), \star\rangle$$

for all $f \in R^{\neg\neg A}$ and $k \in A$.
Then $\mathscr{C}_A \circ \eta_A = id_A$.

Proof
For $d \in R^A$ and $k \in A$ we have

$$\begin{aligned}
(\mathscr{C}_A \circ \eta_A)(d)(k) = & \\
&= \eta_A(d)\langle(\lambda\langle f, h\rangle{:}\neg A.\, f(k)),\, \star\rangle = \\
&= (\lambda\langle f, h\rangle{:}\neg A.\, f(k))\langle d, \star\rangle = \\
&= d(k) = \\
&= id_A(d)(k)
\end{aligned}$$

Thus, by extensionality of $\mathscr{N}_R$-morphisms we have $\mathscr{C}_A \circ \eta_A = id_A$. □

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

For any domain R the category $\mathcal{N}_R$ provides a 'proof semantics for classical logic', i.e. a λ-calculus with a distinguished type $\bot$ representing the proposition *falsity* such that for any type A there is a morphism $\mathscr{C}_A : \neg\neg A \to A$ in $\mathcal{N}_R$ corresponding to the classically valid principle of *reductio ad absurdum*.

Remark 2.2
If we had decided to define $\neg A$ as R^A then $\eta_A : A \to \neg\neg A$ and $\mathscr{C}_A : \neg\neg A \to A$ could have been defined more easily as the following morphisms in $\mathscr{P}$

$$\eta_A = \varepsilon_{R^A} \text{ and } \mathscr{C}_A = R^{\varepsilon_A}$$

where for any X in $\mathscr{P}$ the $\mathscr{P}$-morphism $\varepsilon_X : X \to R^{R^X}$ is defined as $\varepsilon_X(x)(p) = p(x)$. Straightforward computation shows that

$$R^{\varepsilon_A} \circ \varepsilon_{R^A} = id_{R^A}.$$

This observation should make transparent the idea behind our 'official' definition of η and $\mathscr{C}$ which appears as slightly more complicated only because – for reasons of uniformity – we insist on defining $\neg A$ as $A \Rightarrow \bot$.

If we had chosen R to be the empty predomain 0 then the resulting category $\mathcal{N}_R$ would be rather trivial. As 0^A is empty if A is non-empty and 0^A contains precisely one element, otherwise, the category $\mathcal{N}_R$ were equivalent to the 2-element Boolean lattice Σ. The case $R = 1$ leads to the same problem as for any A we have $1 \cong 1^A$. Thus, for obtaining a nontrivial category of negated domains a minimal choice is $R = \Sigma$.

We conclude this section by showing that Theorem 2.2 cannot be improved to the extent that $\eta_A \circ \mathscr{C}_A = id_{\neg\neg A}$ for all A.

The underlying reason for this phenomenon is the following quite general fact (originally observed by Joyal for the special case where R is initial).

Theorem 2.3
Let $\mathscr{C}$ be a cartesian closed category together with a distinguished object R such that $A \cong R^{R^A}$ for all A in $\mathscr{C}$. Then R is subterminal, i.e. R is a subobject of 1, and $\mathscr{C}$ is a preorder, i.e. all parallel arrows in $\mathscr{C}$ are equal. Thus, $\mathscr{C}$ is equivalent to a Boolean lattice.

Proof
If $\varepsilon_1 : 1 \to R^{R^1}$ were an isomorphism then this would give rise to the following 1-1-correspondence

$$\mathscr{C}(A, 1) \cong \mathscr{C}(A, R^{R^1}) \cong \mathscr{C}(A, R^R) \cong \mathscr{C}(A \times R, R)$$

for all A in $\mathscr{C}$.

Instantiating A by R itself we get that there exists precisely one map $R \times R \to R$. Thus, both projections $\pi_i : R \times R \to R$ are equal and, therefore, for any A there is at most one map $A \to R$. Thus, $R \to 1$ is a monomorphism, i.e. R is subterminal.

Now for any objects A and B in $\mathscr{C}$ we have that

$$\mathscr{C}(A, B) \cong \mathscr{C}(A, R^{R^B}) \cong \mathscr{C}(A \times R^B, R).$$

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

As there is at most one map $A \times R^B \to R$ since R is subterminal there also exists at most one map from A to B. So $\mathscr{C}$ is a preorder and, therefore, equivalent to a Boolean lattice. $\square$

The theorem shows that any model of classical logic where any proposition A is isomorphic to $\neg\neg A$ is already equivalent to a Boolean lattice where all proofs of a proposition are equal. But, the categories $\mathscr{N}_R$ introduced above provide models of classical logic where for any proposition A there are maps $A \to \neg\neg A$ and $\neg\neg A \to A$ establishing the *logical equivalence* of the propositions A and $\neg\neg A$ although they are not isomorphic. This is in accordance with traditional classical logic which only postulates the logical equivalence of A and $\neg\neg A$ but not that they are isomorphic[2].

3 Continuation semantics for λ-calculi with control features

3.1 The pure λ-calculus

According to Scott (1980), a model of the extensional λ-calculus is given by a *reflexive* object D in a cartesian closed category where an object D is called reflexive iff D is isomorphic to $D^D = [D \to D]$, the type of functions from D to D (in the sense of the ambient cartesian closed category).

In a category of negated domains $\mathscr{N}_R$ an object C is reflexive iff $R^C \cong R^{R^C \times C}$ in the category $\mathscr{P}$ of predomains. Thus, for obtaining a reflexive object in $\mathscr{N}_R$ it suffices to find a solution of the domain equation

$$C = R^C \times C$$

in the category $\mathscr{D}$ of domains. It is clearly sufficient to look for solutions in $\mathscr{D}$ and, furthermore, in $\mathscr{P}$ there do not exist solutions which are simultaneously initial and terminal (Plotkin, 1983; Freyd, 1992).

The initial/terminal solution of this domain equation gives rise to a continuous isomorphism

$$\mathsf{con} : (R^C \times C) \to C$$

(called 'constructor') with inverse

$$\mathsf{dec} := \mathsf{con}^{-1} : C \to (R^C \times C)$$

(called 'destructor') which in turn – by applying the contravariant functor $R^{(-)}$ – gives rise to the (mutually inverse) pair of continuous isomorphisms

$$R^{\mathsf{dec}} : R^{R^C \times C} \to R^C \text{ and } R^{\mathsf{con}} : R^C \to R^{R^C \times C}.$$

establishing that $C \cong R^C \times C$ in $\mathscr{N}_R$. As $R^{R^C \times C} \cong (R^C)^{R^C}$ in $\mathscr{D}$ we get a solution to the domain equation $D = D^D$ in $\mathscr{D}$ by taking $D = R^C$.

Convention. We assume that con (and dec) are actually identities, i.e. that we have an initial terminal solution of the domain equation $C = R^C \times C$ *up to equality*

[2] That propositions are isomorphic cannot even be expressed in traditional logic due to the absence of proof objects and equalities between them.

(which can always be achieved by choosing an appropriate isomorphic variant of the functor $\times$). This assumption will facilitate subsequent computations as con and dec being identities will allow us to omit them.

Surprisingly, it will turn out that $D = R^C$ is isomorphic to the D_∞-model of the extensional λ-calculus as constructed by Dana Scott in 1969 (cf. Barendregt, 1984, § 18.3) by instantiating the D of D_∞ by the domain R of responses. Thus all known *non-syntactic* models of extensional[3] λ-calculus turn out as being isomorphic to *continuation models*, i.e. as solutions to the domain equation $D = [D \to D]$ in $\mathcal{N}_R$, and, therefore, allow one to interpret control operators like Felleisen's $\mathscr{C}$ as we shall see in the next section.

Theorem 3.1
Let C be the initial/terminal solution of the equation $C = R^C \times C$ in $\mathscr{D}$. Then for $D := R^C$ the continuous functions

$$\mathsf{eval} : D \to D^D \quad \text{and} \quad \mathsf{abst} : D^D \to D$$

defined as

$$\mathsf{eval}(d)(d')(k) = d\,\langle d', k\rangle \quad \text{and} \quad \mathsf{abst}(f)\,\langle d, k\rangle = f(d)(k)$$

constitute an isomorphism pair.

Furthermore, D is isomorphic to the R_∞-model, i.e. the D_∞-model with $D = R$.

Proof
First we show that $\mathsf{abst} \circ \mathsf{eval} = id_D$ and $\mathsf{eval} \circ \mathsf{abst} = id_{D^D}$:

$$\begin{aligned}
&(\mathsf{abst} \circ \mathsf{eval})(d)\langle d', k\rangle = \\
&\qquad = \mathsf{abst}(\mathsf{eval}(d))\langle d', k\rangle = \mathsf{eval}(d)(d')(k) \\
&\qquad = d\,\langle d', k\rangle
\end{aligned}$$

$$\begin{aligned}
&(\mathsf{eval} \circ \mathsf{abst})(f)(d)(k) = \\
&\qquad = \mathsf{eval}(\mathsf{abst}(f))(d)(k) = \mathsf{abst}(f)\langle d, k\rangle = \\
&\qquad = f(d)(k)\ .
\end{aligned}$$

Next we will show that $D = R^C$ is isomorphic to R_∞. This will be done by exhibiting an isomorphism between the ω-diagrams of embedding/projection pairs whose inverse limits are $D = R^C$ and R_∞, respectively.

First, remember that the initial/terminal solution to the recursive domain equation $C = R^C \times C$ is constructed as the inverse limit of the sequence of embedding/projection pairs

$$(i_n : C_n \to C_{n+1}\,,\ q_n : C_{n+1} \to C_n)_{n \in \mathbb{N}}$$

which is defined by primitive recursion as follows

$$\begin{array}{lll}
C_0 := \{\bot\}, & C_{n+1} := R^{C_n} \times C_n & \\
i_0 : C_0 \to C_1, & q_0 : C_1 \to C_0 & \text{are the unique } \textit{strict} \text{ maps} \\
i_{n+1} := R^{q_n} \times i_n & q_{n+1} := R^{i_n} \times q_n\ . &
\end{array}$$

[3] Here *extensional* means that the η-rule $\lambda y.\, xy = x$ is valid in the model.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

Next remember that R_∞ is defined as the inverse limit of the sequence of embedding/projection pairs

$$(e_n : R_n \to R_{n+1}\,,\, p_n : R_{n+1} \to R_n)_{n\in\mathbb{N}}$$

which is defined by primitive recursion as follows

$$\begin{array}{ll} R_0 := R & R_{n+1} := R_n^{R_n} \\ e_0 : R_0 \to R_1 : r \mapsto \lambda x{:}R.\,r & e_{n+1} := {e_n}^{p_n} \\ p_0 : R_1 \to R_0 : f \mapsto f(\bot) & p_{n+1} = {p_n}^{e_n} \quad . \end{array}$$

To prove that $D = R^C$ and R_∞ are isomorphic it is sufficient to show that the sequences of embedding/projection pairs

$$(R^{q_n}, R^{i_n})_{n\in\mathbb{N}} \quad \text{and} \quad (e_n, p_n)_{n\in\mathbb{N}}$$

are isomorphic because then their inverse limits are isomorphic, too. For this purpose we define a sequence of isomorphism pairs

$$(f_n : R_n \to R^{C_n}\,,\, g_n : R^{C_n} \to R_n)_{n\in\mathbb{N}}$$

such that for all $n \in \mathbb{N}$

$$f_{n+1} \circ e_n = R^{q_n} \circ f_n$$

Such a sequence can be defined recursively as follows:

$$\begin{array}{ll} f_0 : R_0 \to R^{C_0} : r \mapsto \lambda x{:}C_0.\,r & f_{n+1} := \mathit{uncurry} \circ {f_n}^{g_n} \\ g_0 : R^{C_0} \to R_0 : h \mapsto h(\bot) & g_{n+1} = {g_n}^{f_n} \circ \mathit{curry} \quad . \end{array}$$

The required properties can be proved by straightforward, but tedious induction. Despite the technicality of the induction proof, intuitively, the key point is that the conditions above are satisfied for $n = 0$. The rest follows from the fact that $(R^Y)^{R^X}$ and $R^{R^X \times Y}$ are isomorphic naturally in X and Y. $\square$

For the reflexive object $D = R^C$, where the required isomorphism is given by eval and abst of the previous Theorem 3.1, we can define the interpretation of the extensional untyped λ-calculus according to the general pattern described by Scott (1980).

Definition 3.1

The interpretation function $[\![\_]\!] : \mathsf{Term} \to (\mathsf{Var} \to D) \to D$ is defined by structural recursion as follows

$$\begin{array}{l} [\![x]\!]\, e := e(x) \\ [\![\lambda\, x.\, M]\!]\, e := \mathsf{abst}(\lambda d{:}D.[\![M]\!]\, e[x := d]) \\ [\![M\, N]\!]\, e := \mathsf{eval}([\![M]\!]\, e)([\![N]\!]\, e)\ , \end{array}$$

where abst and eval are defined as in Theorem 3.1.

By unfolding the definitions of abst and eval in the previous definition we get the following more explicit definition of the interpretation function.

Theorem 3.2
The interpretation function $[\![\_]\!] : \mathsf{Term} \to (\mathsf{Var} \to D) \to D$ of Definition 3.1 can be defined equivalently by the following equations

$$
\begin{aligned}
&[\![x]\!]\, e\, k = e(x)(k)\\
&[\![\lambda x. M]\!]\, e\, \langle d, k\rangle = [\![M]\!]\, e[x := d]\, k\\
&[\![M\, N]\!]\, e\, k = [\![M]\!]\, e\, \langle [\![N]\!]\, e, k\rangle\ .
\end{aligned}
$$

Proof
The first equation is immediate. The remaining two equations can be proved by unfolding the definitions of abst and eval from Theorem 3.1 and exploiting the fact that any object of type C is necessarily of the form $\langle d, k\rangle$.

$$
\begin{aligned}
[\![\lambda x. M]\!]\, e\, \langle d, k\rangle &= \mathsf{abst}(\lambda d : D.\, [\![M]\!]\, e[x := d])\langle d, k\rangle =\\
&= (\lambda d{:}D.\, [\![M]\!]\, e[x := d])(d)(k) =\\
&= [\![M]\!]\, e[x := d]\, k
\end{aligned}
$$

$$
\begin{aligned}
[\![M\, N]\!]\, e\, k &= \mathsf{eval}([\![M]\!]\, e)([\![N]\!]\, e)(k) =\\
&= [\![M]\!]\, e\, \langle [\![N]\!]\, e, k\rangle\ .
\end{aligned}
$$

□

The simplest continuation model of the extensional λ-calculus is Σ_∞, i.e. D_∞ for $D = \Sigma$, where $\Sigma = \{\bot, \top\}$ is the domain containing only two different elements, also known as Sierpinski space. This Σ corresponds to the space of observations where one can only observe termination represented by $\top$ and non-termination or divergence represented by $\bot$.

A famous result of Wadsworth (cf. Barendregt, 1984, Theorem 19.2.4) establishes a useful equivalence between interpretations in the Σ_∞-model and operational properties of λ-terms: for a *closed* term M the process of *head reduction* terminates iff the interpretation of M in Σ_∞ is different from $\bot$.

Theorem 3.3
Let $\Sigma = \{\bot, \top\}$ and C be the initial/terminal solution of $C = \Sigma^C \times C$ in $\mathscr{D}_\bot$. Let $D = \Sigma^C$, $\top_D = \lambda k{:}C.\top$ and $\mathsf{stop} \in C$ be the greatest element in C, i.e. $\mathsf{stop} = \langle \top_D, \mathsf{stop}\rangle$. Then for arbitrary λ-terms M the following are equivalent:

(i) M has a head normal form, i.e. the process of head reduction terminates
(ii) $[\![M]\!]\, e_\top\, \mathsf{stop} = \top$

where $e_\top(x) = \top_D$ for all variables x, i.e. $e_\top$ is the environment that maps any variable to the maximal element in D.

Proof
Wadsworth's Theorem 19.2.4 states that a *closed* term M is unsolvable, i.e. the process of head reduction diverges, iff the interpretation of M in Σ_∞ is $\bot$. From this and our Theorem 3.1 it follows immediately that a closed term M has a head normal form iff its interpretation in Σ_∞ is different from $\bot$, i.e. $[\![M]\!]\, e_\top\, \mathsf{stop} = \top$ (we have that $[\![M]\!]\, e = [\![M]\!]\, e_\top$ for all environments e as M is closed by assumption).

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

We now extend this result to *open* terms M. Obviously, an open term M has a head normal form iff its 'λ-closure' $\lambda\vec{x}.M$ has a head normal form (where $\vec{x}$ is the list of all variables free in M). Thus, from the consideration above it follows that M has a head normal form iff $[\![\lambda\vec{x}.M]\!]\, e_\top\, \mathsf{stop} = \top$. From the second semantic equation of Theorem 3.2 we get that for an arbitrary term N we have

$$[\![\lambda x.N]\!]\, e_\top\, \mathsf{stop} = [\![N]\!]\, e_\top[x := \top]\, \mathsf{stop} = [\![N]\!]\, e_\top\, \mathsf{stop}$$

as $\mathsf{stop} = \langle \top, \mathsf{stop}\rangle$. Thus, by induction on the length of $\vec{x}$ we get that

$$[\![\lambda\,\vec{x}.M]\!]\, e_\top\, \mathsf{stop} = [\![M]\!]\, e_\top\, \mathsf{stop}$$

and, therefore, M has a head normal form iff $[\![M]\!]\, e_\top\, \mathsf{stop} = \top$. □

This result will be crucial for proving a computational adequacy result for an extension of Krivine's machine computing head normal forms instead of weak head normal forms.

3.2 The $\lambda\mathscr{C}$-calculus

3.2.1 Continuation semantics

The syntax of the $\lambda\mathscr{C}$-calculus is that of the untyped λ-calculus together with a new unary operator $\mathscr{C}$, called *Felleisen's Control Operator*, which was introduced originally in (Felleisen and Friedman, 1986) for call-by-value λ-calculus.

Later we will interpret (the call-by-name version of) $\mathscr{C}$ as an *untyped* analogue of the operator $\mathscr{C}$ introduced in Theorem 2.2 above. The equations governing the use of $\mathscr{C}$ will be derived from its semantics (c.f. Theorem 3.5).

But first we define terms and evaluation contexts of $\lambda\mathscr{C}$-calculus.

Definition 3.2
The terms and evaluation contexts of the $\lambda\mathscr{C}$-calculus are defined as follows :

(Term) $\quad M ::= x \mid \lambda x.M \mid MM \mid \mathscr{C}M$
(EvCont) $\quad E ::= [] \mid E\,M$

The fragment without $\mathscr{C}$ is known as the ordinary untyped λ-calculus.

Notice that an evaluation context is always of the form $[]M_1...M_n$, i.e. given by a list of arguments.

The conversion or rewrite rules of the $\lambda\mathscr{C}$-calculus are intentionally not stated here but will be extracted from a careful examination of the subsequently given *continuation semantics* which we consider as more fundamental.

Next we will give an interpretation of Felleisen's control operator $\mathscr{C}$ in $D = R^C$ where C is the initial/terminal solution of $C = R^C \times C$ in $\mathscr{D}_\perp$.

Recall (Theorem 2.2) that in $\mathscr{N}_R$ for every object A there is a morphism $\mathscr{C}_A$: $((A \Rightarrow \perp) \Rightarrow \perp) \to A$ with

$$\mathscr{C}_A(d)(k) = d\,(\langle\, \lambda\,\langle d', h\rangle : A \Rightarrow \perp.\ d'(k)\ ,\ \star\,\rangle)$$

for all $d : ((A \Rightarrow \perp) \Rightarrow \perp) \to R$ and $k \in A$.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

The definition of $\mathscr{C}_A : ((A \Rightarrow \bot) \Rightarrow \bot) \to A$ can be generalized by replacing $\bot$ by an arbitrary *non-empty* predomain B. For any $b \in B$ and all objects A of $\mathscr{N}_R$ there is a morphism $\mathscr{C}_A^b : ((A \Rightarrow B) \Rightarrow B) \to A$ with

$$\mathscr{C}_A^b(d)(k) = d\,(\langle\, \lambda\langle d', h\rangle : A \Rightarrow B.\ d'(k)\ ,\ b\,\rangle)$$

for all $d : ((A \Rightarrow B) \Rightarrow B) \to R$ and $k \in A$. Again, as in Theorem 2.2 by straightforward computation one can show that $\mathscr{C}_A^b \circ \eta_A^B = id_{R^A}$ for the morphism $\eta_A^B : A \to (A \Rightarrow B) \Rightarrow B$ in $\mathscr{N}_R$ with

$$\eta_A^B(a)\langle d, y\rangle = d\langle a, y\rangle$$

for all $d : (A \Rightarrow B) \to R$ and $y \in B$.

Obviously, $\mathscr{C}_A^b$ depends upon b. Moreover, if $b_1 \sqsubseteq b_2$ then $\mathscr{C}_A^{b_1} \sqsubseteq \mathscr{C}_A^{b_2}$. Therefore, if B happens to have a greatest element $\top$ then it is natural[4] to choose this for b as $\mathscr{C}_A^\top$ is the *greatest* element in $\{\, \mathscr{C}_A^b \mid b \in B\,\}$ w.r.t. the domain ordering $\sqsubseteq$.

Now having generalized $\mathscr{C}_A$ to $\mathscr{C}_A^b$ whenever $b \in B$ we are ready to interpret Felleisen's control operator $\mathscr{C}$ in a type-free setting, namely as $\mathscr{C}_C^c$ for some $c \in C$ where C is the initial/terminal solution of $C = R^C \times C$.

If R has a greatest element $\top$ then C has a greatest element stop which is characterized uniquely by the equation

$$\mathsf{stop} = \langle \lambda k{:}C.\,\top, \mathsf{stop}\rangle$$

and in this case $\mathscr{C}$ will be interpreted as $\mathscr{C}_C^{\mathsf{stop}}$. This convention applies in particular when $R = \Sigma$.

Definition 3.3
Let $D = R^C$ where C is the initial/terminal solution of $C = R^C \times C$. The interpretation function $[\![\_]\!] : \mathsf{Term} \to (\mathsf{Var} \to D) \to D$ (where Term denotes the set of $\lambda\mathscr{C}$-terms) is defined by structural recursion as follows :

$$[\![x]\!]\, e = e(x)$$
$$[\![\lambda x.\, M]\!]\, e\,\langle d, k\rangle = [\![M]\!]\, e[x := d]\, k$$
$$[\![M\, N]\!]\, e\, k = [\![M]\!]\, e\,\langle [\![N]\!]\, e, k\rangle$$
$$[\![\mathscr{C}\, M]\!]\, e\, k = [\![M]\!]\, e\,\langle \mathsf{ret}(k), \mathsf{stop}\rangle$$

where $\mathsf{ret}(k) = \lambda\langle d, h\rangle.\, d(k) \in D$ and $\mathsf{stop} \in C$.

Convention. If R contains a greatest element $\top$ then stop will always be the greatest object in C characterized uniquely by the equation $\mathsf{stop} = \langle \top_{R^C}, \mathsf{stop}\rangle$ where $\top_{R^C} = \lambda k{:}C.\,\top_R$ is the greatest element in R^C.

Since C is defined recursively as $C = R^C \times C$ one may consider a *continuation*, i.e. a $k \in C$, as an *infinite list* of *denotations*, i.e. elements of $D = R^C$. Such infinite lists of denotations can be interpreted as *denotational versions* of *call-by-name evaluation contexts*. Under this correspondence between denotational and operational notions, the semantic equation for $\mathscr{C}$ expresses that, to evaluate $\mathscr{C}\, M$ in an evaluation context

[4] Given two objects or programs satisfying a specification one will certainly prefer the one which terminates more often.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

represented by k one simply applies (the meaning of) M to $\mathsf{ret}(k)$ in the *empty evaluation context* represented by the continuation stop. The denotation $\mathsf{ret}(k)$ is used only implicitly in the $\lambda\mathscr{C}$-calculus. In the subsequently introduced $\lambda\mu$-calculus, however, it will appear as the denotation of a term of the extended language (provided k is denotable by a term). The behaviour of denotation $\mathsf{ret}(k)$ can be explained as follows: when applying the denotation $\mathsf{ret}(k)$ to a denotation d w.r.t. a continuation h then the result is $d(k)$, i.e. the current continuation h is forgotten and the argument d is evaluated w.r.t. the 'returned' continuation k.

These intuitive explanations will get precise when we study equational laws of $\lambda\mathscr{C}$-calculus and Krivine's machine. But first, we consider some examples showing the use and expressivity of the control operator $\mathscr{C}$.

To illustrate the expressivity of Felleisen's $\mathscr{C}$ we briefly show how to define some simple (and well-known) $\lambda\mathscr{C}$-terms implementing some derived control operators analogously to those found as primitives in realistic call-by-value functional languages as SCHEME and NJ-SML. Due to the importance of these call-by-value languages there is a large amount of syntactically oriented work investigating Felleisen's $\mathscr{C}$ and its expressivity for a call-by-value version of $\lambda\mathscr{C}$-calculus (Felleisen, 1986; Felleisen *et al*., 1987; Griffin, 1990; Felleisen and Hieb, 1992; Sabry and Felleisen, 1992).

First, we state a lemma which is technically useful for many computations and explains in which sense $\mathscr{C}$ is an inverse to 'double negation'.

Lemma 3.4
For any term M we have

(1) $[\![\mathscr{C}(\lambda f.\, f\, M)]\!]\, e\, k = [\![M]\!]\, e[f := \mathsf{ret}(k)]\, k$
(2) $[\![\mathscr{C}(\lambda f.\, f\, M)]\!]\, e\, k = [\![M]\!]\, e\, k \qquad \text{if } f \notin FV(M)\,.$

Notice that point (2) of Lemma 3.4 says that $\mathscr{C}$ can be reformulated as $\mathscr{C}(\eta\, M) = M$ where η stands for the 'double negation' operator $\lambda x.\, \lambda f.\, f\, x$. That means that using $\mathscr{C}$ one can 'unpack' terms which have been 'encapsulated' by the 'double negation' operator η.

Below we briefly sketch how other control operators known from the literature can be expressed in terms of $\mathscr{C}$ by giving a syntactic definition and the corresponding semantic equation.

Abort operator

$$\mathscr{A} M := \mathscr{C}(\lambda f.\, M) \quad \text{with} \quad f \notin FV(M).$$

Its semantics can be computed as

$$[\![\mathscr{A}\, M]\!]\, e\, k = [\![\mathscr{C}(\lambda f.\, M)]\!]\, e\, k = [\![M]\!]\, e[f := \mathsf{ret}(k)]\, \mathsf{stop} = [\![M]\!]\, e\, \mathsf{stop}$$

demonstrating that evaluation of $\mathscr{A} M$ in context k amounts to forgetting the current context and evaluating M in the empty context represented by stop.

Error-handling

$$\mathbf{handle\ err\ in\ M\ by\ N} := \mathscr{C}(\lambda \mathbf{f}.\, \mathbf{f}((\lambda \mathbf{err}.\, \mathbf{M})(\mathbf{f\, N})))$$

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

where f is a fresh variable. The semantics of this construct can be computed as follows:

$$
\begin{aligned}
&[\![\mathbf{handle\,err\,in\,M\,by\,N}]\!]\, e\, k = \\
&\qquad = [\![\mathscr{C}(\lambda f.\, f((\lambda err.\, M)(f\, N)))]\!]\, e\, k = \text{(Lemma 3.4(1))} \\
&\qquad = [\![(\lambda err.\, M)(f\, N)]\!]\, e[f := \mathsf{ret}(k)]\, k = \\
&\qquad = [\![M]\!]\, e[err := [\![f\, N]\!]\, e[f := \mathsf{ret}(k)]]\, k = \\
&= [\![M]\!]\, e[err := \lambda h.\, [\![N]\!]\, e\, k]\, k
\end{aligned}
$$

Intuitively, the evaluation of **handle err in M by N** in context k is as follows: one evaluates expression M in context k but whenever during that process one has to evaluate the expression *err* w.r.t. a (new) context h then this context h is forgotten and expression N is evaluated instead w.r.t. the old context k. Note that no **raise** construct is necessary as opposed to the call-by-value case.

Call with current continuation

$$\mathbf{call/cc\,M} := \mathscr{C}(\lambda \mathbf{f}.\, \mathbf{f}(\mathbf{M\,f})) \quad \text{with} \quad \mathbf{f} \notin \mathbf{FV}(\mathbf{M}).$$

This yields the following semantic equation.

$$[\![\mathbf{call/cc\,M}]\!]\, e\, k = [\![M]\!]\, e\, \langle \mathsf{ret}(k), k\rangle.$$

Notice that the difference between **call/cc** and $\mathscr{C}$ is that – although $[\![M]\!]\, e$ in both cases is applied to $\mathsf{ret}(k)$ – the continuations w.r.t. which the applications are evaluated are different: in case of **call/cc** the continuation is the *current continuation* k whereas in the case of $\mathscr{C}$ it is stop representing the empty evaluation context.

Taking $\mathscr{A}$ and **call/cc** as basic control operators together with their defining semantic equations

$$[\![\mathscr{A} M]\!]\, e\, k = [\![M]\!]\, e\, \mathsf{stop}$$

$$[\![\mathbf{call/ccM}]\!]\, e\, k = [\![M]\!]\, e\, \langle \mathsf{ret}(k), k\rangle$$

then one can verify that

$$[\![\mathbf{call/cc}(\lambda \mathbf{x}.\, \mathscr{A}(\mathbf{M\,x})]\!]\, e\, k = [\![M]\!]\, e\, \langle \mathsf{ret}(k), \mathsf{stop}\rangle$$

for all terms M with $x \notin FV(M)$. Thus, Felleisen's $\mathscr{C}$ is definable from $\mathscr{A}$ and **call/cc**.

3.2.2 Some useful laws of $\lambda\mathscr{C}$-calculus

In this subsection we will derive some equational laws for the $\lambda\mathscr{C}$-calculus. These laws will turn out as analogous to the ones stated by Felleisen *et al.* (1987) and Felleisen and Hieb (1992) for the call-by-value variant of the $\lambda\mathscr{C}$-calculus.

Theorem 3.5

In any continuation model for the $\lambda\mathscr{C}$-calculus the following equalities are true for all terms M, N and evaluation contexts E:

$$\begin{array}{lll}
(\beta) & (\lambda x.M)\,N = M[N/x] & \\
(\eta) & \lambda x.(M\,x) = M & \text{if } x \notin FV(M) \\
(\mathscr{C}1) & \mathscr{C}(\lambda f.f\,M) = M & \text{if } f \notin FV(M) \\
(\mathscr{C}2) & \mathscr{C}(\lambda f.\mathscr{C}M) = \mathscr{C}(\lambda f.\,M(\lambda x.\mathscr{A}x)) & \\
(\mathscr{C}3) & E[\mathscr{C}M] = \mathscr{C}(\lambda f.\,M(\lambda x.f\,E[x])) & \text{with } f \notin FV(M) \cup FV(E)
\end{array}$$

Proof

The Substitution Lemma, i.e. $[\![M[N/x]]\!]\,e = [\![M]\!]\,e[x := [\![N]\!]\,e]$ provided N is free for x in M, can be proved straightforwardly by induction on the structure of M. Using this basic fact we can show the validity of the rules (β) and (η).

The equation $(\mathscr{C}1)$ follows immediately from Lemma 3.4 (2).

The equation $(\mathscr{C}2)$ is valid as

$$\begin{aligned}
&[\![\mathscr{C}(\lambda f.\mathscr{C}M)]\!]\,e\,k = \\
&\qquad = [\![\lambda f.\mathscr{C}M]\!]\,\langle\,\mathsf{ret}(k),\mathsf{stop}\,\rangle = \\
&\qquad = [\![\mathscr{C}M]\!]\,e[f := \mathsf{ret}(k)]\,e\,\mathsf{stop} = \\
&= [\![M]\!]\,e[f := \mathsf{ret}(k)]\,\langle\,\mathsf{ret}(\mathsf{stop}),\mathsf{stop}\,\rangle
\end{aligned}$$

$$\begin{aligned}
&[\![\mathscr{C}(\lambda f.\,M(\lambda x.\mathscr{A}x)]\!]\,e\,k = \\
&\qquad = [\![\lambda f.\,M(\lambda x.\mathscr{A}x)]\!]\,e\,\langle\,\mathsf{ret}(k),\mathsf{stop}\,\rangle = \\
&\qquad = [\![M(\lambda x.\mathscr{A}x)]\!]\,e[f := \mathsf{ret}(k)]\,\mathsf{stop} = \\
&= [\![M]\!]\,e[f := \mathsf{ret}(k)]\,\langle\,[\![\lambda x.\mathscr{A}x)]\!]\,e,\mathsf{stop}\,\rangle
\end{aligned}$$

and

$$\begin{aligned}
&[\![\lambda x.\mathscr{A}x)]\!]\,e\,\langle\,d,k\,\rangle = \\
&\qquad = [\![\mathscr{A}x]\!]\,e[x := d]\,k = \\
&\qquad = [\![x]\!]\,e[x := d]\,\mathsf{stop} = d\,\mathsf{stop} = \\
&= \mathsf{ret}(\mathsf{stop})\,\langle\,d,k\,\rangle \qquad .
\end{aligned}$$

For proving the equations $(\mathscr{C}3)$ assume that $E \equiv [\,]\ P_1 \ldots P_n$ and for a continuation $k \in C$ and an environment e let $k_{E,e} := \langle [\![P_1]\!]\,e, \ldots, \langle [\![P_n]\!]\,e, k\rangle \ldots\rangle$.

$$[\![E[\mathscr{C}M]]\!]\,e\,k = [\![\mathscr{C}M]\!]\,e\,k_{E,e} = [\![M]\!]\,e\,\langle\,\mathsf{ret}(k_{E,e}),\mathsf{stop}\,\rangle$$

and

$$\begin{aligned}
&[\![\mathscr{C}(\lambda f.\,M(\lambda x.f\,E[x]))]\!]\,e\,k \\
&\qquad = [\![\lambda f.\,M(\lambda x.f\,E[x])]\!]\,e\,\langle\,\mathsf{ret}(k),\mathsf{stop}\,\rangle = \\
&\qquad = [\![M(\lambda x.f\,E[x])]\!]\,e[f := \mathsf{ret}(k)]\,\mathsf{stop} = \\
&= [\![M]\!]\,e\,\langle\,[\![\lambda x.f\,E[x]]\!]\,e[f := \mathsf{ret}(k)],\mathsf{stop}\,\rangle\ .
\end{aligned}$$

It remains to show that $\mathsf{ret}(k_{E,e}) = [\![\lambda x.f\,E[x]]\!]\,e[f := \mathsf{ret}(k)]$:

$$\begin{aligned}
&[\![\lambda x.f\,E[x]]\!]\,e[f := \mathsf{ret}(k)]\langle\,d,h\,\rangle = \\
&\qquad = [\![f\,E[x]]\!]\,e[f := \mathsf{ret}(k)][x := d]\,h = \\
&\qquad = \mathsf{ret}(k)\langle\,[\![E[x]]\!]\,e[x := d],h\,\rangle =
\end{aligned}$$

$$
\begin{aligned}
&\quad = [\![E[x]]\!]\, e[x := d]\, k = \\
&\quad = d\, k_{E,e[x:=d]} = \text{(as } x \text{ is not free in } E) \\
&\quad = d\, k_{E,e} = \\
&= \mathsf{ret}(k_{E,e})\langle d, h\rangle
\end{aligned}
$$

which finishes the proof. □

Remark 3.1
One might be inclined to postulate

$$E[\mathscr{C}M] = M\,(\lambda x.\, E[x])$$

as an intuitive explanation of the meaning of $\mathscr{C}$. It is, however, inconsistent as $\mathscr{C}(\lambda f.\, \lambda x.\, x) = (\lambda f.\, \lambda x.\, x)(\lambda x.\, x) = \lambda x.\, x$ and, therefore, for all terms M we have $M = (\lambda x.\, x)\, M = \mathscr{C}(\lambda f.\, \lambda x.\, x)\, M = (\lambda f.\, \lambda x.\, x)\,(\lambda x.\, x\, M) = \lambda x.\, x$, i.e. all terms M are equal to $\lambda x.\, x$.

3.3 The $\lambda\mu$-Calculus

In this section we will use our continuation semantics for interpreting an *untyped variant* of Parigot's $\lambda\mu$-calculus. The typed $\lambda\mu$-calculus has been introduced by Parigot (1992) in a purely syntactical way, to give a proof term assignment for classical logic formulated in natural deduction style. Here we will not further investigate the logical aspects of the $\lambda\mu$-calculus, but rather demonstrate that it is a flexible language for expressing general control operators.

The untyped $\lambda\mu$-calculus is an *extension* of the ordinary λ-calculus by two new syntactic categories: *continuation expressions* and *R-terms*. The underlying intuition is that ordinary terms denote elements of D, i.e. denotations, R-terms denote elements in R, i.e. responses, and continuation expressions denote elements in C, i.e. continuations. Thus the untyped $\lambda\mu$-calculus allows to refer explicitly to semantic objects like responses and continuations which in $\lambda\mathscr{C}$-calculus can be referred to only in an indirect way.

First we give the syntax of the untyped $\lambda\mu$-calculus in BNF-form.

Definition 3.4
Let Var and CVar be two disjoint infinite sets of *(object) variables* and *continuation variables*, respectively. We will use $x, y, z \ldots$ as meta-variables for object variables and $\alpha, \beta, \gamma \ldots$ as meta-variables for continuation variables.

$$
\begin{array}{ll}
\text{(Term)} & M ::= x \mid \lambda x.M \mid MM \mid \mu\alpha.\, t \\
\text{(Cont)} & C ::= \alpha \mid M :: C \\
(R\text{-Term}) & t ::= [C]M
\end{array}
$$

The $\lambda\mu$-calculus is an extension of the ordinary λ-calculus. Therefore, we may extend our continuation semantics for the λ-calculus (as given in Theorem 3.2) to the full $\lambda\mu$-calculus.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

Definition 3.5

Let $D = R^C$ where C is the initial/terminal solution of $C = R^C \times C$. Let Env be the set of environments, i.e. functions mapping object variables to elements of D and continuations variables to elements of C. The interpretation functions

$$\begin{array}{l} [\![\_]\!]_D : \mathsf{Term} \to Env \to D \\ [\![\_]\!]_C : \mathsf{Cont} \to Env \to C \\ [\![\_]\!]_R : R\text{–}\mathsf{Term} \to Env \to R \end{array}$$

are defined by structural recursion as follows :

$$\begin{array}{rcl} [\![x]\!]_D\, e & = & e(x) \\ [\![\lambda\, x.\, M]\!]_D\, e\, \langle d, k\rangle & = & [\![M]\!]_D\, e[x := d]\, k \\ [\![M\, N]\!]_D\, e\, k & = & [\![M]\!]_D\, e\, \langle [\![N]\!]_D\, e, k\rangle \\ [\![\mu\alpha.\, t]\!]_D\, e\, k & = & [\![t]\!]_R\, e[\alpha := k] \\ & & \\ [\![\alpha]\!]_C\, e & = & e(\alpha) \\ [\![M :: C]\!]_C\, e & = & \langle [\![M]\!]_D\, e, [\![C]\!]_C\, e\rangle \\ & & \\ [\![[C]M]\!]_R\, e & = & [\![M]\!]_D\, e\, ([\![C]\!]_C\, e) \end{array}$$

Convention. We will omit the subscripts of the interpretation functions defined above as they can be read off from the term between the semantic brackets.

The idea of ‘continuations as objects’ is illustrated by the following example

$$[\![\mu\alpha.[\beta]M]\!]\, e\, k = [\![M]\!]\, e[\alpha := k]\, e(\beta)$$

swapping continuations.

Notice that Parigot’s original formulation of the $\lambda\mu$-calculus – besides being typed rather than untyped – does not have continuation terms but only continuation variables. In our extended syntax the general form of continuation expressions is $M_1 :: \ldots :: M_n :: \alpha$, i.e. continuation expressions are stacks of ordinary terms whose bottom is a continuation variable. Due to this extension, we can express the substitution $[M :: \beta/\alpha]$ directly instead of introducing it as a new primitive called ‘mixed substitution’ in Parigot (1992). Thus, by admitting these more general continuation expressions we get a considerable simplification of the equational presentation of $\lambda\mu$-calculus.

Theorem 3.6

The continuation model for the untyped $\lambda\mu$-calculus validates the following equational axioms.

$$\begin{array}{lll} (\beta) & (\lambda x.M)\, N = M[N/x] & \\ (\eta) & \lambda x.\, (M\, x) = M & \text{where } x \text{ not free in } M \\ (\beta_{\text{cont}}) & [C]\, \mu\alpha.t = t[C/\alpha] & \\ (\eta_{\text{cont}}) & \mu\alpha.\, [\alpha]M = M & \text{where } \alpha \text{ not free in } M \\ (\text{Swap}) & [C](MN) = [N :: C]M & \end{array}$$

Proof

The verifications of (β) and (η) are as in the proof of Theorem 3.5.

The remaining equations follow from the semantic equations of Definition 3.5 and a Substitution Lemma for continuation variables which says that for arbitrary expressions A and arbitrary continuation expressions C

$$[\![A[C/\alpha]]\!]\, e = [\![A]\!]\, e[\alpha := [\![C]\!]\, e]$$

for all $e \in \mathit{Env}$.

For (β_{cont}) consider

$$[\![[C]\,\mu\alpha.t]\!]\, e = [\![\mu\alpha.t]\!]\, e\,([\![C]\!]\, e) = [\![t]\!]\, e[\alpha := [\![C]\!]\, e] = [\![t[C/\alpha]]\!]\, e \quad .$$

For (η_{cont}) consider

$$[\![\mu\alpha.\,[\alpha]M]\!]\, e\, k = [\![[\alpha]M]\!]\, e[\alpha := k] = [\![M]\!]\, e[\alpha := k]\,(e[\alpha := k](\alpha)) = [\![M]\!]\, e\, k \quad .$$

For (Swap) consider

$$\begin{aligned}&[\![[C](MN)]\!]\, e = \\ &\qquad [\![MN]\!]\, e\,([\![C]\!]\, e) = [\![M]\!]\, e\,\langle [\![N]\!]\, e, [\![C]\!]\, e\rangle = [\![M]\!]\, e\,([\![N :: C]\!]\, e) = \\ &[\![[N :: C]M]\!]\, e\ .\end{aligned}$$

□

The usual control operators can now be expressed in the $\lambda\mu$-calculus as

$$\mathscr{C} \equiv \lambda f.\, \mu\alpha.\, [\sigma] f(\lambda x.\, \mu\beta.\, [\alpha]x)$$

$$\mathbf{call/cc} \equiv \lambda \mathbf{f}.\, \mu\alpha.\, [\alpha]\mathbf{f}(\lambda \mathbf{x}.\, \mu\beta.\, [\alpha]\mathbf{x})$$

$$\mathscr{A} \equiv \lambda f.\, \mu\alpha.\, [\sigma] f$$

where σ is a distinguished unbound continuation variable whose intended meaning is the distinguished continuation stop considered previously (for $\lambda\mathscr{C}$-calculus).

Notice that the $\lambda\mu$-terms above are almost identical with the semantic equations for these control operators in our previous continuation semantics for $\lambda\mathscr{C}$-calculus. This demonstrates that $\lambda\mu$-calculus reflects more closely the underlying semantics than $\lambda\mathscr{C}$-calculus.

We now discuss the equation (Swap) and explain why it is crucial for simplifying the previous axiomatisations given in Parigot (1992) and Ong and Ritter (1994). The rule (*Swap*) does not appear in *loc.cit.* as its right hand side is not even part of his syntax. Using the rule (Swap) we can derive in our extended calculus the equation

$$(\mu\alpha.t)M = \mu\beta.\,[\beta]((\mu\alpha.\,t)M) = \mu\beta.\,[M :: \beta](\mu\alpha.\,t) = \mu\beta.\,(t[M :: \beta/\alpha])$$

employing *ordinary* substitution of continuation expressions for continuation variables. This was impossible in Parigot's original calculus where continuation variables were the only form of continuation expressions. Using the equation above we can derive the so-called (ζ)-rule

$$\mu\alpha.t = \lambda x.(\mu\alpha.t)x = \lambda x.\mu\beta.t[x :: \beta/\alpha]$$

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

which plays an essential role in Ong's treatment of $\lambda\mu$-calculus (Ong and Ritter, 1994; Hofmann and Streicher, 1997).

When trying to use the equations of Theorem 3.6 in order to obtain a deterministic rewrite strategy for $\lambda\mu$-calculus it is not clear how to orient the equation (Swap) due to its apparent symmetry.

But for giving a rewrite semantics to $\lambda\mu$-calculus by η_{cont} it suffices to give reduction rules for R-terms, i.e. expressions of the form $[C]M$. In order to have a deterministic evaluation strategy the rule used to rewrite an R-term $[C]M$ should depend only upon the shape of M.

If $M \equiv M_1 M_2$ then by applying (Swap) in the direction left-to-right $[C]M$ reduces to $[M_2 :: C]M_1$.

If $M \equiv \mu\alpha.\,t$ then by applying (β_{cont}) in the direction left-to-right $[C]M$ reduces to $t[C/\alpha]$.

Using the equation (Swap) in the direction right-to-left we get

$$[N :: C](\lambda x.\,M) = [C]((\lambda x.\,M)N) = [C](M[N/x])$$

which – when read from left to right – tells us what to do in case of functional abstractions.

Summarizing, we have the following three rewrite rules allowing one to reduce R-terms :

$$[C](MN) \to [N :: C]M$$

$$[N :: C](\lambda x.\,M) \to [C](M[N/x])$$

$$[C](\mu\alpha.t) \to t[C/\alpha].$$

The first two rules correspond to the transition rules of Krivine's machine for pure λ-calculus, which will be introduced in the next section. The third rule provides a transition rule suitable for an extension of Krivine's machine to $\lambda\mu$-calculus.

Though the rewrite system above contains the key ideas of Krivine's machine, it is still different from it in the respect that the formulation of the rules employs substitution as a basic operation, e.g. the second rule is essentially the β-rule of ordinary λ-calculus. The pragmatic superiority of Krivine's machine is that it avoids substitution as a basic operation (which might be quite costy as the size of terms may explode) and, instead of terms, manipulates so-called closures, i.e. terms together with an environment. Substitution will only be performed when actually needed, i.e. when applied to a term that is already a variable. This will be achieved by a further transition rule of Krivine's machine.

4 From continuation semantics to abstract machines

The aim of this section is to give a *rational reconstruction* of the operational semantics of λ-calculi with control features by deriving abstract machines from their continuation semantics.

Usually, these machines compute only weak head normal forms. It is straightforward to extend them to machines computing head normal forms and for these we can prove computational adequacy w.r.t. our continuation semantics.

4.1 The $\lambda\mathscr{C}$-calculus

In this section we will derive an abstract machine for $\lambda\mathscr{C}$-calculus based on its continuation semantics as introduced in section 3.2 by turning the semantic equations into transition rules. Our abstract machine for $\lambda\mathscr{C}$-calculus will be an extension of Krivine's machine for pure untyped λ-calculus (Abadi *et al.*, 1991).

4.1.1 Krivine's machine

Any semantic equation of Definition 3.3 is of the form

$$[\![M]\!]\, e\, k = [\![M']\!]\, e'\, k'$$

where M, M' are terms, e, e' are environments and k, k' are continuations. Expressions of the form $[\![M]\!]\, e$ denote elements of D and can be considered simply as pairs of terms and environments, traditionally called *closures*. In the presence of control operator $\mathscr{C}$, closures may also be of the form $\mathsf{ret}(k)$ where k is a continuation. Continuation expressions are of the form stop or $\langle c, k \rangle$, where c is a closure and k is a continuation expression. Thus continuation expressions are simply *stacks of closures* (with stop as empty stack).

This suggests to define a machine whose states are pairs whose first component is a closure and whose second component is a stack of closures. As already noted above, a closure is a pair of a term and an environment binding finitely many variables to closures. The rewrite rules of the machine operating on states will mimic the semantic equations of Definition 3.3. We will relate the machine arising this way to the continuation semantics by defining interpretation functions mapping *closures* to elements of D, *stacks* to elements of C, *environments* to functions from Var to D and *states* to elements of Σ.

We first give a definition of Krivine's machine.

Definition 4.1

If A and B are sets then $A \to_{\mathrm{fin}} B$ denotes the set of finite partial functions from A to B. For any $e \in A \to_{\mathrm{fin}} B$ we write $\mathsf{dom}(e)$ for the finite subset of A where e is defined.

The sets Term of terms, Env of environments, Clos of closures, Stack of stacks (of closures) and State of machine states are defined inductively as follows:

$$\begin{array}{llll}
(\mathsf{Term}) & M & ::= & x \mid \lambda x.\, M \mid M\, M \mid \mathscr{C} M \\
(\mathsf{Env}) & env & \in & \mathsf{Var} \to_{\mathrm{fin}} \mathsf{Clos} \\
(\mathsf{Clos}) & c & ::= & [\, M, env\,] \mid \mathsf{ret}(S) \\
(\mathsf{Stack}) & S & ::= & \mathsf{stop} \mid \langle\, c, S\,\rangle \\
(\mathsf{State}) & \sigma & ::= & \langle\, c, S\,\rangle
\end{array}$$

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

The binary transition relation $\to$ on State is given by the following transition rules:

$$
\begin{array}{llll}
\text{(Var)} & \langle\, [x, env],\, S\,\rangle & \to & \langle\, env(x), S\,\rangle \quad \text{if } x \in \mathsf{dom}(env) \\
\text{(Fun)} & \langle\, [\lambda x.\, M, env],\, \langle\, c, S\,\rangle\,\rangle & \to & \langle\, [M, env[x := c]],\, S\,\rangle \\
\text{(App)} & \langle\, [M\,N, env],\, S\,\rangle & \to & \langle\, [M, env],\, \langle\, [N, env],\, S\,\rangle\,\rangle \\
(\mathscr{C}) & \langle\, [\mathscr{C}M, env],\, S\,\rangle & \to & \langle\, [M, env],\, \langle\, \mathsf{ret}(S), \mathsf{stop}\,\rangle\,\rangle \\
\text{(Ret)} & \langle\, \mathsf{ret}(S), \langle\, c, S'\,\rangle\,\rangle & \to & \langle\, c, S\,\rangle
\end{array}
$$

We write trans for the partial function whose graph is $\to$. Notice that $\to$ is deterministic, i.e. if $\sigma \to \sigma_1$ and $\sigma \to \sigma_2$ then σ_1 and σ_2 are equal. Let Eval be the partial function associating with any state σ the state $\mathsf{trans}^n(\sigma)$ where $\mathsf{trans}^{n+1}(\sigma)$ is undefined and $\mathsf{trans}^i(\sigma)$ is defined for all $i \leqslant n$. If $\mathsf{trans}^n(\sigma)$ is defined for all n then $\mathsf{Eval}(\sigma)$ is undefined. A state σ is final iff $\mathsf{trans}(\sigma)$ is undefined. Obviously, a state σ is final iff it is of one of the following forms:

(i) $\langle\, [x, env],\, S\,\rangle$ with $x \notin \mathsf{dom}(env)$
(ii) $\langle\, [\lambda x.\, M, env],\, \mathsf{stop}\,\rangle$
(iii) $\langle\, \mathsf{ret}(S), \mathsf{stop}\,\rangle$

Thus, final states are *either* a head variable followed by a list of closures (case (i)) or the stack is empty and the first component is a function definition either of the form $[\,\lambda x.\, M, env\,]$ (case (ii)) or of the form $\mathsf{ret}(S)$ (case (iii)).

Next we define the denotational semantics of Krivine's Machine (KM).

Definition 4.2
The interpretation functions

$$
\begin{array}{l}
[\![\_]\!]_{\mathrm{State}} : \mathsf{State} \to \Sigma \\
[\![\_]\!]_{\mathrm{Clos}} : \mathsf{Clos} \to D \\
[\![\_]\!]_{\mathrm{Env}} : \mathsf{Env} \to \mathsf{Var} \to D \\
[\![\_]\!]_{\mathrm{Stack}} : \mathsf{Stack} \to C
\end{array}
$$

are given by the following semantic equations

$$
\begin{array}{lcl}
[\![\langle\, c, S\,\rangle]\!]_{\mathrm{State}} & = & [\![c]\!]_{\mathrm{Clos}}([\![S]\!]_{\mathrm{Stack}}) \\
\\
[\![\langle\, c, S\,\rangle]\!]_{\mathrm{Stack}} & = & \langle [\![c]\!]_{\mathrm{Clos}}, [\![S]\!]_{\mathrm{Stack}} \rangle \\
[\![\mathsf{stop}]\!]_{\mathrm{Stack}} & = & \top_C := \langle \top_D, [\![\mathsf{stop}]\!]_{\mathrm{Stack}} \rangle \\
\\
[\![env]\!]_{\mathrm{Env}}(x) & = & \textbf{if } x \in \mathsf{dom}(env) \textbf{ then } [\![env(x)]\!]_{\mathrm{Clos}} \textbf{ else } \top_D \\
\\
[\![[\, M, env\,]]\!]_{\mathrm{Clos}} & = & [\![M]\!]\, [\![env]\!]_{\mathrm{Env}} \\
[\![\mathsf{ret}(S)]\!]_{\mathrm{Clos}} \langle d, k \rangle & = & d\, [\![S]\!]_{\mathrm{Stack}}
\end{array}
$$

where $\top_D = \lambda k.\top$ (recall that $R = \Sigma$) and for a term M its semantics $[\![M]\!]$ is defined as in Definition 3.3.

The next theorem states the correctness of Krivine's machine.

Theorem 4.1

(1) For all terms M it holds that

$$[\![\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = [\![M]\!]\, e_\top\, \top_C$$

where ϵ is the empty environment and $e_\top(x) = \top_D$ for all variables x.

(2) The relation $\to$ preserves semantics of states, i.e. for all states σ, σ' it holds that

$$\sigma \to \sigma' \quad \text{implies} \quad [\![\sigma]\!]_{\mathrm{State}} = [\![\sigma']\!]_{\mathrm{State}}.$$

Proof

(1) follows immediately from the semantic equations of Definition 4.2. (2) is proved by straightforward case analysis on $\sigma \to \sigma'$ employing the semantic equations of Definition 3.3 and Definition 4.2. $\square$

This is a rather minimal form of correctness stating only that the transitions of the machine preserve the semantics of states. Nevertheless, it might happen for a term M that $\mathsf{Eval}(\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle)$ is undefined, i.e. the machine started with initial state $\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle$ never halts, although $[\![\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = \top$, i.e. 'semantically' it should terminate.

Actually, one would like that

$$[\![\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = \top \quad \text{iff} \quad \mathsf{Eval}(\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle) \text{ is defined}$$

i.e. that the machine started with initial state $\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle$ eventually halts if and only if it halts 'semantically', i.e. $[\![\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = \top$. Such an equivalence is commonly called *computational adequacy* because it says that operational and semantical notions of termination are equivalent, i.e. the *denotational semantics is adequate w.r.t. the operational behaviour*.

The implication from left to right will be proved in section 4.1.2 in Theorem 4.4(ii). But the reverse direction cannot be true in general for the following reason. The term $\Omega \equiv (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$ does not have a head normal form and therefore $\lambda x.\, \Omega$ does not have a head normal form either. Therefore, by Theorem 3.3 $[\![\langle\, [\lambda x.\, \Omega, \epsilon],\ \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = \bot$ though $\langle\, [\lambda x.\, \Omega, \epsilon],\ \mathsf{stop}\,\rangle$ is already a final state and thus $\mathsf{Eval}(\langle\, [\lambda x.\, \Omega, \epsilon],\ \mathsf{stop}\,\rangle)$ is defined.

The reason for this 'failure' is that Krivine's machine does not compute head normal forms but *weak* head normal forms.

Theorem 4.2

A term M has a weak head normal form iff $\mathsf{Eval}(\langle\, [M,\epsilon],\ \mathsf{stop}\,\rangle)$ is defined where ϵ is the empty environment.

Proof

For a precise proof one has to introduce a λ-calculus with *explicit substitution* as done in detail in Abadi *et al.* (1991) and Curien (1991). Here we only give an intuitive relation between reduction steps in Krivine's machine and steps of the weak head reduction.

The reduction steps (Fun), (Ret) of Krivine's machine correspond to β-reduction steps in the process of weak head reduction. Reduction step $(\mathscr{C})$ of Krivine's

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

machine corresponds to step ($\mathscr{C}$) in the process of weak head reduction, where $\mathsf{ret}(S)$ corresponds to $\lambda x.\mathscr{C}(\lambda f.E[x])$ and S is the stack corresponding to evaluation context E. The rule (App) of Krivine's machine allows to store the current evaluation context on the stack. The rule (Var) handles substitution. It has to be noted that substitution is actually performed only when applied to a variable. The rule (App) distributes substitution over the components of an application term. A substitution applied to a λ-abstraction is never performed as we are only interested in (weak) head normal forms. □

4.1.2 Extended Krivine's machine and its computational adequacy

Now we define an extension of Krivine's machine computing head normal forms instead of only weak head normal forms. For this Extended Krivine's machine we prove a computational adequacy theorem which says that for every term M there is a terminating sequence of transitions, starting starting from initial state $\langle\, [M,\epsilon],\, \mathsf{stop}\,\rangle$ iff the denotation of this initial state equals $\top$.

Definition 4.3
Let $\rightarrow_h$ be the binary transition relation on State containing the relation $\rightarrow$ of Definition 4.1 augmented by the rules

$$
\begin{array}{lll}
\text{(Fun-h)} & \langle\, [\lambda x.\, M, env],\, \mathsf{stop}\,\rangle \rightarrow_h & \langle\, [M[y/x], env],\, \mathsf{stop}\,\rangle \\
\text{(Ret-h)} & \langle\, \mathsf{ret}(S), \mathsf{stop}\,\rangle \qquad\quad \rightarrow_h & \langle\, [y,\epsilon],\, S\,\rangle
\end{array}
$$

where in both cases y is a *fresh* variable (which in case of (Fun-h) in particular means that $y \notin \mathsf{dom}(env)$).

The resulting extension of Krivine's machine will be called the *Extended Krivine's machine*.

Let $\mathsf{trans\text{-}h}$ and $\mathsf{Eval\text{-}h}$ be defined analogously to Definition 4.1. A state σ is *h-final* iff $\mathsf{trans\text{-}h}(\sigma)$ is undefined. Obviously, a state σ is *h-final* iff it is of the form $\langle\, [x, env],\, S\,\rangle$ with $x \in \mathsf{Var}$ and $x \notin \mathsf{dom}(env)$.

Remark 4.1
The Extended Krivine's machine with $\rightarrow_h$ as transition relation is a modification which allows us to compute head normal forms, and not only weak head normal forms, since whenever computation reaches a state of the form $\langle\, [\lambda x.\, M, env],\, \mathsf{stop}\,\rangle$ or of the form $\langle\, \mathsf{ret}(S), \mathsf{stop}\,\rangle$ – both corresponding to a functional abstraction – then one introduces a fresh variable for the bound variable and proceeds with the computation.

The introduction of the fresh variable may be considered as a 'side effect' of a transition of the form (Fun-h) or (Ret-h). One could keep track of these side-effects by adding a further component to the state, namely a list of variables where in steps (Fun-h) and (Ret-h) the new fresh variable y is added to the list of variables, and in all other steps the list remains unchanged. Thus, when computation has finished one has a list of fresh variables corresponding to the λ-prefix of the head normal form, together with a head variable which can be read off from the final state, and a list of closures corresponding to the list of arguments for the head variable. To

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

compute normal forms by leftmost-outermost strategy, one could now apply the machine recursively to each of these closures in parallel.

We now prove the computational adequacy of the Extended Krivine's Machine.

Theorem 4.3
For all $\sigma \in \mathsf{State}$ it holds that $[\![\sigma]\!]_{\mathrm{State}} = \top$ iff $\sigma \in \mathsf{dom(Eval\text{-}h)}$, i.e. Extended Krivine's Machine stops when started with initial state σ.

Proof
First notice that there exist relations – so-called *inclusive predicates* –

$$
\begin{aligned}
&R_{\mathrm{State}} \subseteq \Sigma \times \mathsf{State}\\
&R_{\mathrm{Stack}} \subseteq C \times \mathsf{Stack}\\
&R_{\mathrm{Clos}} \subseteq D \times \mathsf{Clos}\\
&R_{\mathrm{Env}} \subseteq (\mathsf{Var} \to D) \times \mathsf{Env}\\
&R_{\mathrm{Term}} \subseteq ((\mathsf{Var} \to D) \to D) \times \mathsf{Term}
\end{aligned}
$$

satisfying the requirements:

$$
\begin{array}{lcl}
u\, R_{\mathrm{State}}\, \sigma & \Leftrightarrow & u = \top \ \text{ implies } \ \sigma \in \mathsf{dom(Eval\text{-}h)}\\
k\, R_{\mathrm{Stack}}\, \mathsf{stop} & & \text{always valid}\\
\langle d, k\rangle\, R_{\mathrm{Stack}}\, \langle c, S\rangle & \Leftrightarrow & d\, R_{\mathrm{Clos}}\, c \ \text{ and } \ k\, R_{\mathrm{Stack}}\, S\\
d\, R_{\mathrm{Clos}}\, c & \Leftrightarrow & \forall k\, R_{\mathrm{Stack}}\, S.\ d(k)\, R_{\mathrm{State}}\, \langle c, S\rangle\\
e\, R_{\mathrm{Env}}\, env & \Leftrightarrow & \forall x \in \mathsf{dom}(env).\, e(x)\, R_{\mathrm{Clos}}\, env(x)\\
f\, R_{\mathrm{Term}}\, M & \Leftrightarrow & \forall e\, R_{\mathrm{Env}}\, env.\, f(e)\, R_{\mathrm{Clos}}\, [M, env].
\end{array}
$$

An elegant *general* method for the construction of such inclusive predicates for the initial/terminal solution of an arbitrary domain equation and its associated language has been given by Pitts (1994), to which we refer for a proof. We do not repeat Pitt's argument here as for the purposes of our proof; we only need the *mere existence* of the required inclusive predicates.

But now, from the existence of the inclusive predicates and the required equivalences for them one shows by straightforward (simultaneous) structural induction that

$$
\begin{array}{ll}
[\![\sigma]\!]_{\mathrm{State}}\, R_{\mathrm{State}}\, \sigma & \text{for all } \sigma \in \mathsf{State}\\
[\![S]\!]_{\mathrm{Stack}}\, R_{\mathrm{Stack}}\, S & \text{for all } S \in \mathsf{Stack}\\
[\![c]\!]_{\mathrm{Clos}}\, R_{\mathrm{Clos}}\, c & \text{for all } c \in \mathsf{Clos}\\
[\![env]\!]_{\mathrm{Env}}\, R_{\mathrm{Env}}\, env & \text{for all } env \in \mathsf{Env}\\
[\![M]\!]\, R_{\mathrm{Term}}\, M & \text{for all terms } M \ .
\end{array}
$$

Thus, for all $\sigma \in \mathsf{State}$ we have $[\![\sigma]\!]_{\mathrm{State}}\, R_{\mathrm{State}}\, \sigma$, i.e. $\sigma \in \mathsf{dom(Eval\text{-}h)}$ if $[\![\sigma]\!]_{\mathrm{State}} = \top$, which proves the implication from left to right.

The implication from right to left follows from the facts that the transition relation $\to_h$ preserves denotations of states and that the denotation of final states is $\top$ (as if $\langle\, [x, env],\, S\,\rangle$ is a final state then $x \notin \mathsf{dom}(env)$ and, therefore, $[\![env]\!]_{\mathrm{Env}}(x) = \top$).
$\square$

As a consequence, we get the following theorem.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

Theorem 4.4
Let ϵ be the empty environment. Then for any term M

(i) $[\![M]\!]\, e_\top \neq \bot_D \Leftrightarrow \langle\, [M, \epsilon], \mathsf{stop}\,\rangle \in \mathsf{dom}(\mathsf{Eval\text{-}h})$
(ii) $[\![M]\!]\, e_\top \neq \bot_D \Rightarrow \langle\, [M, \epsilon], \mathsf{stop}\,\rangle \in \mathsf{dom}(\mathsf{Eval})$.

Proof
First notice that, by Theorem 4.1 (1), we have $[\![\langle\, [M, \epsilon], \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = [\![M]\!]\, e_\top\, \top_C$ and therefore $[\![M]\!]\, e_\top \neq \bot_D$ iff $[\![M]\!]\, e_\top\, \top_C = \top$ iff $[\![\langle\, [M, \epsilon], \mathsf{stop}\,\rangle]\!]_{\mathrm{State}} = \top$. Now by instantiating $\sigma = \langle\, [M, \epsilon], \mathsf{stop}\,\rangle$ claim (i) follows immediately from Theorem 4.3.

Claim (ii) follows from (i) by the fact that $\mathsf{dom}(\mathsf{Eval\text{-}h})$ is contained in $\mathsf{dom}(\mathsf{Eval})$ as $\rightarrow$ is a subrelation of $\rightarrow_h$. □

The reverse implication of (ii) in the above theorem does not hold as the containment of $\mathsf{dom}(\mathsf{Eval\text{-}h})$ in $\mathsf{dom}(\mathsf{Eval})$ is *proper* even for states of the form $\langle\, [M, \epsilon], \mathsf{stop}\,\rangle$ (e.g. when M has a weak head normal form but not a head normal form as is the case for $M = \lambda x.\,\Omega$).

4.2 The $\lambda\mu$-calculus

In this section we derive an abstract machine for $\lambda\mu$-calculus based on its continuation semantics as introduced in section 3.3. As for $\lambda\mathscr{C}$-calculus the method of derivation again will be to consider the semantic equations as transition rules of the abstract machine.

Our abstract machine for $\lambda\mu$-calculus will be an extension of Krivine's machine for the untyped λ-calculus without control operators. It will turn out that the distinguishing feature of $\lambda\mu$-calculus is that there are continuation variables which can be assigned continuations by environments. Thus, we have an extended notion of environment which assign *denotations* to *object* variables and *continuations* to *continuation variables*. We write Var for the set of object variables and CVar for the set of continuation variables.

We will not employ our extended syntax of section 3.3, but stick to Parigot's original language. The reason for this is that the extended language is only needed for formulating the rule (Swap) simplifying equational reasoning in $\lambda\mu$-calculus. Therefore, the only term formation rule besides those for pure untyped λ-calculus is the following: $\mu\alpha.\,[\beta]M$ is a term if M is a term and α, β are continuation variables.

4.2.1 Krivine's machine for $\lambda\mu$-calculus

Before giving the precise definition, we informally describe the components of Krivine's machine for $\lambda\mu$-calculus. Note that a similar machine has been found independently by de Groote (1996), *albeit by purely syntactical methods* which seem, however, to be more complicated than our semantic approach, in the authors' opinion.

States will be pairs $\langle\, c, S\,\rangle$ where c is a closure and S is a stack representing a continuation.

Due to the absence of $\mathscr{C}$ and ret, closures will now simply be pairs $[\,M, env\,]$ where M is a term and env is an environment.

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

An environment env will actually be a pair $\langle env_{\mathrm{ob}}, env_{\mathrm{cont}} \rangle$ where env_{ob} is a finite partial map sending *object variables* to *closures* and env_{cont} is a finite partial map sending *continuation variables* to *stacks*. For $x \in \mathsf{Var}$ and $\alpha \in \mathsf{CVar}$ we systematically write $env(x)$ and $env(\alpha)$ for $env_{\mathrm{ob}}(x)$ and $env_{\mathrm{cont}}(\alpha)$, respectively. Accordingly, we write $\mathsf{dom}(env)$ for the (finite) set of object and continuation variables on which env is defined.

Continuations representing stacks are expressions of the form $\langle c_1, \dots \langle c_n, \alpha \rangle \dots \rangle$, i.e. stacks of closures built on top of 'empty stacks' represented by unbound continuation variables.

Definition 4.4
The sets Term of terms, Env of environments, Clos of closures, Stack of stacks (of closures) and State of machine states are defined inductively as follows

$$
\begin{array}{llll}
(\mathsf{Term}) & M & ::= & x \mid \lambda x.\, M \mid M\, M \mid \mu\alpha.\, [\beta]M \\
(\mathsf{Env}) & env & \in & (\mathsf{Var} \to_{\mathrm{fin}} \mathsf{Clos}) \times (\mathsf{CVar} \to_{\mathrm{fin}} \mathsf{Stack}) \\
(\mathsf{Clos}) & c & ::= & [\, M, env\,] \\
(\mathsf{Stack}) & S & ::= & \alpha \mid \langle\, c, S\, \rangle \\
(\mathsf{State}) & \sigma & ::= & \langle\, c, S\, \rangle
\end{array}
$$

The binary transition relation $\to$ on State is given by the following transition rules

$$
\begin{array}{llll}
(\mathrm{Var}) & \langle\, [x, env],\, S\, \rangle & \to & \langle\, env(x), S\, \rangle \quad \text{if } x \in \mathsf{dom}(env) \\
(\mathrm{Fun}) & \langle\, [\lambda x.\, M, env],\, \langle\, c, S\, \rangle\, \rangle & \to & \langle\, [M, env[x := c]],\, S\, \rangle \\
(\mathrm{App}) & \langle\, [M\, N, env],\, S\, \rangle & \to & \langle\, [M, env],\, \langle\, [N, env],\, S\, \rangle\, \rangle \\
(\mu) & \langle\, [\mu\alpha.\, [\beta]M, env],\, S\, \rangle & \to & \langle\, [M, env[\alpha := S]],\, env[\alpha := S](\beta)\, \rangle
\end{array}
$$

where the last rule (μ) applies if and only if $\beta \in \mathsf{dom}(env[\alpha := S])$, i.e. $\beta \in \mathsf{dom}(env)$ or $\alpha \equiv \beta$.

The machine given by the above transition rules is called *Krivine's machine for $\lambda\mu$-calculus.*

Again we write trans and Eval for the partial transition function and the partial evaluation map, respectively, which are defined as usual.

A state σ is final iff $\mathsf{trans}(\sigma)$ is undefined, i.e. iff σ is of one of the following forms

(i) $\langle\, [x, env],\, S\, \rangle$ with $x \notin \mathsf{dom}(env)$
(ii) $\langle\, [\lambda x.\, M, env],\, \alpha\, \rangle$ for some $\alpha \in \mathsf{CVar}$
(iii) $\langle\, [\mu\alpha.\, [\beta]M, env],\, S\, \rangle$ with $\beta \notin \mathsf{dom}(env)$ and $\alpha \not\equiv \beta$.

To make the relation to continuation semantics precise we extend it to Krivine's machine for $\lambda\mu$-calculus.

Definition 4.5
The interpretation functions

$[\![\_]\!]_{\mathrm{State}} : \mathsf{State} \to \Sigma$
$[\![\_]\!]_{\mathrm{Clos}} : \mathsf{Clos} \to D$
$[\![\_]\!]_{\mathrm{Env}} : \mathsf{Env} \to \mathsf{Var} \to D$
$[\![\_]\!]_{\mathrm{Stack}} : \mathsf{Stack} \to C$

https://doi.org/10.1017/S0956796898003141 Published online by Cambridge University Press

are defined by the following semantic equations

$$
\begin{array}{lcl}
[\![\langle\, c, S \,\rangle]\!]_{\mathrm{State}} & = & [\![c]\!]_{\mathrm{Clos}}([\![S]\!]_{\mathrm{Stack}}) \\[1ex]
[\![\langle\, c, S \,\rangle]\!]_{\mathrm{Stack}} & = & \langle [\![c]\!]_{\mathrm{Clos}}, [\![S]\!]_{\mathrm{Stack}} \rangle \\
[\![\alpha]\!]_{\mathrm{Stack}} & = & \top_C := \langle \top_D, \top_C \rangle \\[1ex]
[\![env]\!]_{\mathrm{Env}}(x) & = & \textbf{if } x \in \mathsf{dom}(env) \textbf{ then } [\![env(x)]\!]_{\mathrm{Clos}} \textbf{ else } \top_D \\
[\![env]\!]_{\mathrm{Env}}(\alpha) & = & \textbf{if } \alpha \in \mathsf{dom}(env) \textbf{ then } [\![env(\alpha)]\!]_{\mathrm{Stack}} \textbf{ else } \top_C \\[1ex]
[\![[M, env]]\!]_{\mathrm{Clos}} & = & [\![M]\!]\, [\![env]\!]_{\mathrm{Env}}
\end{array}
$$

where $\top_D = \lambda k.\top$ (recall that $R = \Sigma$) and for a term M its semantics $[\![M]\!]$ is defined as in Definition 3.5.

Again we have that Krivine's machine for the $\lambda\mu$-calculus is correct w.r.t. its continuation semantics.

Theorem 4.5

(1) For all terms M it holds that

$$[\![\langle\, [M, \epsilon],\, \mathsf{stop} \,\rangle]\!]_{\mathrm{State}} = [\![M]\!]\, e_\top\, \top_C$$

where ϵ is the empty environment and $e_\top(x) = \top_D$ for all $x \in \mathsf{Var}$ and $e_\top(\alpha) = \top_C$ for all $\alpha \in \mathsf{CVar}$.

(2) If $\sigma \to \sigma'$ then $[\![\sigma]\!]_{\mathrm{State}} = [\![\sigma']\!]_{\mathrm{State}}$.

Proof

(1) follows immediately from the semantic equations of Definition 4.5. (2) is proved by straightforward case analysis on the transition rules employing the semantic equations of Definition 3.5 and Definition 4.5. □

4.2.2 *Extended Krivine's machine for $\lambda\mu$-calculus and its computational adequacy*

Again, to obtain computational adequacy, we have to extend Krivine's machine for $\lambda\mu$-calculus in a way that it reduces under λ- and μ-abstractions.

Definition 4.6

Let $\to_h$ be the binary transition relation on State containing the relation $\to$ of Definition 4.4 augmented by the rules

(Fun-h) $\langle\, [\lambda x.\, M, env],\, \alpha \,\rangle \to_h \langle\, [M[y/x], env],\, \alpha \,\rangle$ with y fresh

(μ-h) $\langle\, [\mu\alpha.\, [\beta]M, env],\, S \,\rangle \to_h \langle\, [M, env[\alpha := S]],\, \beta \,\rangle$ if $\beta \notin \mathsf{dom}(env[\alpha := S])$

The resulting extension of Krivine's machine for $\lambda\mu$-calculus will be called *Extended Krivine's machine for $\lambda\mu$-calculus.*

Again, we write $\mathsf{trans\text{-}h}$ and $\mathsf{Eval\text{-}h}$ for the transition and evaluation functions, respectively. A state σ is *h-final* iff $\mathsf{trans\text{-}h}(\sigma)$ is undefined. Obviously, a state σ is *h-final* iff it is of the form $\langle\, [x, env],\, S \,\rangle$ with $x \in \mathsf{Var}$ and $x \notin \mathsf{dom}(env)$.

We have computational adequacy of the Extended Krivine's Machine for $\lambda\mu$-calculus with respect to its continuation semantics.

Theorem 4.6
For all $\sigma \in \mathsf{State}$ it holds that $[\![\sigma]\!]_{\text{State}} = \top$ iff $\sigma \in \mathsf{dom}(\mathsf{Eval\text{-}h})$, i.e. Extended Krivine's Machine for $\lambda\mu$-calculus stops when started with initial state σ.

Proof
The proof is almost identical with the proof of Theorem 4.3. The only difference is that now

$$R_{\text{Env}} \subseteq Env \times \mathsf{Env}$$
$$R_{\text{Term}} \subseteq (Env \to D) \times \mathsf{Term}$$

with

$$\begin{array}{rcl} e\, R_{\text{Env}}\, env & \Leftrightarrow & \forall x \in \mathsf{dom}(env) \cap \mathsf{Var}.\, e(x)\, R_{\text{Clos}}\, env(x) \ \text{ and} \\ & & \forall \alpha \in \mathsf{dom}(env) \cap \mathsf{CVar}.\, e(\alpha)\, R_{\text{Stack}}\, env(\alpha) \\ f\, R_{\text{Term}}\, M & \Leftrightarrow & \forall e\, R_{\text{Env}}\, env.\, f(e)\, R_{\text{Clos}}\, [M, env] \end{array}$$

as new condition for R_{Env}. □

5 Conclusion

We have shown how continuation semantics arising from a simple semantics of classical logic allows one to explain the meaning of control features in call-by-name functional languages, and how one can read off an abstract machine from a continuation semantics. This has been exemplified for λ-calculus with Felleisen's control operator $\mathscr{C}$ and Parigot's $\lambda\mu$-calculus.

Moreover, employing Pitts' method for cooking up proofs of computational adequacy, we have established that our abstract machines compute head normal forms of terms whose denotation is different from $\perp$.

An analogous treatment is possible for call-by-value languages, but in this case one has to employ the opposite of $\mathscr{N}_R$ which is isomorphic to the Kleisli category for the continuation monad $R^{R^{(-)}}$. It would be nice if one could relate this duality on the level of semantics to a duality on the syntactical level. This might provide a deeper understanding of the relation between call-by-name and call-by-value languages.

Another strand of research is to extend the paradigm of deriving abstract machines from continuation semantics to more realistic languages with basic data types as booleans, integers, etc., and recursive types. For this purpose it might be appropriate to give a semantic reformulation of Andrew Appel's work on 'compiling with continuations' (Appel, 1992) by employing and extending the methods we have introduced in this paper.

References

Abadi, M., Cardelli, L., Curien, P.-L. and Levy, J. J. (1991) Explicit substitutions. *J. Functional Programming*, **1(4)**, 375–416.

Appel, A. (1992) *Compiling with Continuations*. Cambridge University Press.

Barendregt, H. P. (1984) *The Lambda Calculus*. North Holland.

Curien, P.-L. (1991) An abstract framework for environment machines. *Theor. Comput. Sci.*, **82**, 389–402.

de Groote, Ph. (1996) *An environment machine for the $\lambda\mu$-calculus.* submitted.

Felleisen, M. (1986) The calculi of λ_v-*cs* conversion: A syntactic theory of control and state in imperative higher order programming languages. *PhD thesis*, Indiana University.

Felleisen, M. and Friedman, D. P. (1986) Control operators, the SECD machine, and the λ-calculus. In: Wirsing, M. (ed), *Formal Descriptions of Programming Concepts III*, pp. 193–217. North Holland.

Felleisen, M. and Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, **103**, 235–271.

Felleisen, M., Friedman, D., Kohlbecker, E. and Duba, B. (1987) A syntactic theory of sequential control. *Theor. Comput. Sci.*, **52**, 205–237.

Fischer, M. (1972) Lambda calculus schemata. *Proc. ACM Conference on Proving Assertions About Programs*, pp. 104–109.

Freyd, P. (1992) Remarks on algebraically compact categories. *Applications of Categories in Computer Science*, vol. 177, Notes of the London Mathematical Society.

Griffin, T. H. (1990) A formula-as-types notion of control. *Conference Record of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 47–58. ACM Press.

Hofmann, M. and Streicher, Th. (1997) Continuation models are universal for lambda-mu calculus. *Proc. 12th IEEE Symposium on Logic in Computer Science (LICS), Warsaw, Poland*, pp. 387–397.

Jeffrey, A. (1994) A fully abstract semantics for concurrent graph reduction: Extended abstract. *Proc. 9th IEEE Symposium on Logic in Computer Science*. IEEE Press.

Lafont, Y. (1991) *Negation versus implication.* Draft.

Lafont, Y., Reus, B. and Streicher, T. (1993) Continuation semantics or expressing implication by negation. *Technical Report 93-21*, University of Munich.

Moggi, E. (1991) Notions of computation and monads. *Information & Computation*, **93**, 55–92.

Okasaki, C., Lee, P. and Tarditi, D. (1994) Call-by-need and continuation-passing style. *Lisp and Symbolic Computation*, **7**, 57–81.

Ong, C.-H. L. and Ritter, E. (1994) A generic strong normalization proof: application to the calculus of constructions (Extended abstract). *Computer Science Logic: 7th Workshop, CSL'93: Lecture Notes in Computer Science 832*, pp. 261–279. Swansea, UK.

Parigot, M. (1992) $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. *Proc. International Conference on Logic Programming and Automated Deduction: Lecture Notes in Computer Science 624*, pp. 190–201. Springer-Verlag.

Pitts, A. M. (1994) Computational adequacy via 'mixed' inductive definitions. *9th MFPS conference*, pp. 72–82. Springer-Verlag.

Plotkin, G. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theor. Comput. Sci.*, **1**, 125–159.

Plotkin, G. D. (1983) *Domains.* TEXversion edited by Y. Kashiwagi and H. Kondoh. Course notes of a lecture held 1983 in Pisa.

Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. *Proc. of the ACM Annual Conference*, pp. 717–740.

Sabry, A. and Felleisen, M. (1992) Reasoning about programs in continuation-passing style. *Proc. of ACM Conference on Lisp and Symolic Computation.*

Scott, D. S. (1980) Relating theories of lambda calculus. In: Hindley, J. R. and Seldin, J. P. (eds), *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pp. 403–450. Academic Press.

Troelstra, A. and van Dalen, D. (1988) *Constructivism in Mathematics.* North Holland.