Partial Continuations as the Difference of Continuations
A Duumvirate of Control Operators*

Luc Moreau[1] and Christian Queinnec[2]

[1] Institut Montefiore B28, Service d'Informatique, Université de Liège, Sart-Tilman, 4000 Liège, Belgium – Email: `moreau@montefiore.ulg.ac.be`
[2] École Polytechnique (URA 1437) & INRIA-Rocquencourt France – Email: `queinnec@polytechnique.fr`

Abstract. We define a partial continuation as the difference of two continuations. We exhibit, in a single framework, several design choices and their impact on semantics. The ability of partial continuations to manipulate stack frames blurs the nature of dynamic extent; therefore, we introduce a new concept of *prefixal extent* that characterises the time during which a partial continuation can be reified. We propose two equivalent formal semantics for partial continuations: a context-rewriting system and a cps translation. Two new and realistic examples illustrate both the interest of partial continuations and the expressiveness of our choices.

1 Introduction

The mathematical concept of continuation was introduced by Strachey and Wadsworth [18] [16] to give a semantics to control operators such as `goto`. The continuation of an expression represents the rest of the computation to be performed after evaluation of this expression. Some programming languages, such as Scheme and SML of New-Jersey, provide the control operator `call/cc`, which gives the programmer the possibility to reify the current continuation into a first-class object. Invoking a reified continuation on a value v consists in resuming the computation where the continuation was captured, with v the value of the `call/cc` expression. The invocation of a first-class continuation has an abortive effect since the control is transferred from the current continuation to the captured continuation. First-class continuations can be used to program coroutines, exceptions, ... [8].

More recently, the concept of partial continuation was successively introduced and studied by Felleisen, Friedman, Wand, Duba, Merrill [6], [7], [4], Danvy and Filinski [3], Hieb and Dybvig [9], [10] Queinnec and Serpette [15]. Although their propositions differ, they nevertheless share a common idea: a partial continuation represents a part of the rest of the computation. Partial continuations can also be reified into first-class objects but, unlike continuations, when a partial continuation is invoked, control is returned to its invocation point after its termination. Therefore, partial continuations have the behaviour of functions and are composable.

Two operators are introduced to delimit the part of the computation that a partial continuation represents. The first of them can be regarded as a marker, while the second operator reifies the partial continuation between its invocation point and a mark. The operators #, `reset`, `spawn`, and `splitter` belong to the former category, while $\mathcal{F}$, `shift`, and `call/pc` belong to the latter. Although the notion of partial continuation comes from the simple idea of reifying a part of the current continuation, the solutions proposed in [7] and [3] introduce a debate on the "dynamic-ness" of

* This paper will appear in the proceedings of PLILP '94.

control operators. Indeed, during invocation of a partial continuation, some marks might be copied. Therefore, which mark should a control operator refer to, the most recent one, i.e. the one that was just copied, or the oldest one, i.e. the one that existed before the invocation of a partial continuation?

As Murthy [12] writes

> Now, naturally, one might ask: is there a way to "jump over" the delimiters? This would lead us to invent a new control-operator, which could jump over the prompts, and grab all of the evaluation context, to the top of the program. And after that, we might want to delimit the action of this operator, too. And so on. ...

In other words, a language should provide the programmer with control operators able to capture partial continuations between any two points, whatever the marks appearing inside. For this purpose, hierarchies of control operators were introduced by Sitaram and Felleisen [17] and Danvy and Filinski [3]. Control operators are now indexed by a number which is their *control level*, and an operator of level m is able to reify the partial continuation delimited by the first mark of level n if $n \geq m$. Although this solution provides the possibility to reify any part of the computation, it lacks of intuition: which intuition should a programmer rely on to choose a mark of level n or a mark of level $n+1$? Quantitative notions like control level are not easy to apprehend and are not easy to deal with in everyday programming. On the other hand, Hieb and Dybvig [9] and Queinnec and Serpette [15] propose pairs of control operators (a marker and a reifier) that are, respectively, parameterised by a label or a mark. These proposals are able to reify partial continuations between any two points but introduce debates on the extent during which partial continuations can be reified.

The four following points are the original contribution of this paper.

1. We suggest a new definition for partial continuations: we regard a partial continuation as the *difference of two continuations* if one is the prefix of the other. We provide two operators: `marker` names the current continuation, while `call/pc` subtracts a named continuation from the current one and reifies their difference into a partial continuation. Since a hierarchy is not required any longer, our two operators, `marker` and `call/pc`, are solely given the power to exercise control, they form our *duumvirate* of control operators.
2. We rely on intuitive stack frames manipulations to explain our control operators in Section 2. In this single framework, we show different semantical choices and their impact. There are two issues. First, are continuation names copied when a partial continuation is reified? Second, what is the extent during which partial continuations can be reified with respect to a named continuation? As opposed to the current literature, we discuss the possible choices and select the most expressive one in Section 3. Although dynamic extent has some interesting properties as far as implementation is concerned, we observe that the ability of partial continuations to manipulate stack frames blurs the nature of dynamic extent: we therefore introduce a new concept of extent that we name *prefixal extent*.
3. We present two formal semantics. The first one is a reduction system (Section 4); its most salient feature is its conciseness. The second semantics is a cps translation (Section 5) in which the connection between stack frames and cps is explicit. The two semantics were proved equivalent, and both formalise the different aforementioned choices.
4. In Section 6, we introduce two new examples that illustrate the interest of partial continuations and the choices that we propose. In particular, prefixal extent is emphasized by the latter example.

2 Partial Continuations as a Difference of Continuations

Let us first consider the program in Figure 1, where first-class continuations are only used as downward continuations. One of the first-class continuations `k1`, `k2`, or `k3` is invoked by `(k`i` v)` in the extent of the `call/cc` by which it was reified. Before evaluating the expression `(k`i` v)`, the control stack can be *symbolically* represented as in Figure 1. The top and base of the stack are indicated by the `top` and `base` pointers respectively, and the stack is *conceptually* divided into four regions by the marks α_1, α_2, and α_3. Each variable `k`i is bound to a continuation that corresponds to a portion of the stack from the `base` pointer to the α_i mark. Evaluating `(k`i` v)` consists in removing the portion of the stack from the `top` pointer to the α_i mark.

```
(f0 (call/cc
     (lambda (k1)
       (f1 (call/cc
            (lambda (k2)
              (f2 (call/cc
                   (lambda (k3)
                     (f3 (ki v))))))))))
                       ; i = 1,2, or 3
```

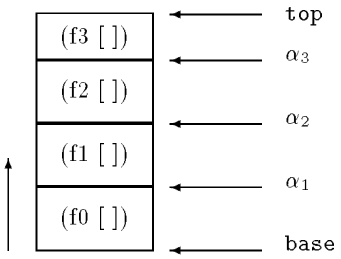

Fig. 1. Program 1 and stack representation

However, first-class continuations are not restricted to a downward use, and `call/cc` is not aimed at leaving marks on the stack. So, let us call `marker` a new control operator that is intended to leave a mark with a *new* name on the control stack. Such marks divide the control stack into regions that we call *control blocks*, and the purpose of a mark is to give a name to the block just below it. We call the *current block* the top block: it is delimited by the top pointer and the previous mark left or the base pointer if no mark was left. Pushing a new mark on the current block consists in naming the current block. Moreover, like `call/cc`, `marker` requires one argument, called a receiver, which must be a unary function. Besides naming the current block, `marker` also reifies the names of the current continuation and applies its receiver to the sequence of names of the current continuation, i.e. the sequence of names of all marks left in the current continuation.

According to Figure 2, before evaluating `(marker (lambda (k3) ...))`, two `marker` expressions were already evaluated, having given names α_1 and α_2 to the two first control blocks. Just after invocation of the third `marker`, the old current block is given the new name α_3, and the application of the receiver `(lambda (k3) ...)` to the sequence of names $\langle \alpha_1, \alpha_2, \alpha_3 \rangle$ starts a new current block, pointed by the `top` pointer.

When a value v is returned to a mark α, this mark is removed from the stack, the block that was named α becomes the current block, and the value v is now returned to the current block. Therefore, marks are transparent to normal returns.

According to this description, a continuation is composed of several control blocks with associated names. We call $\mathcal{N}(c)$ (names of c) the sequence of block names of continuation c. We can now give two definitions related to continuations.

Definition 2.1 (Prefix) *A continuation c_1 is a prefix of a continuation c_2 if the sequence of block names $\mathcal{N}(c_1)$ is a prefix of the sequence of block names $\mathcal{N}(c_2)$.*

```
(f0 (marker
     (lambda (k1)
       (f1 (marker
            (lambda (k2)
              (f2 (marker
                   (lambda (k3)
                     (f3 (call/pc k1 (lambda (pc) ...))))))))))))
```

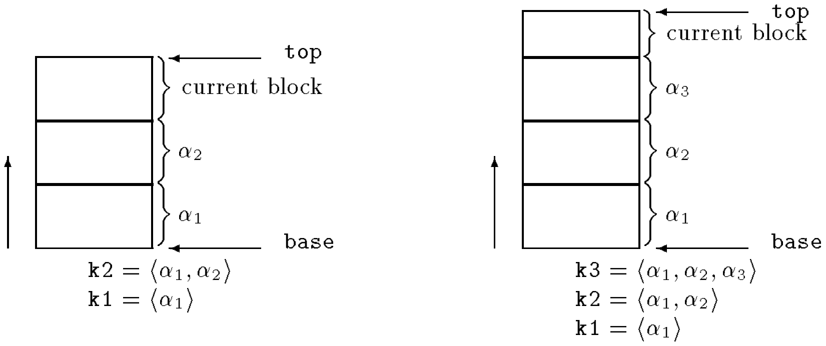

Fig. 2. Stack representation before and after evaluating `(marker (lambda (k3) ...))`

Definition 2.2 (Difference of two continuations) *The difference of two continuations c_1 and c_2 is defined, if c_2 is a prefix of c_1, by the sequence of blocks s from c_1 such that $\mathcal{N}(c_1) = append(\mathcal{N}(c_2), \mathcal{N}(s))$.*

We can now define our notion of partial continuation.

Definition 2.3 (Partial continuation) *A partial continuation is a sequence of control blocks obtained by the difference of two continuations.*

Let us introduce a second control operator `call/pc` that is able to reify partial continuations into first-class objects; `call/pc` requires two arguments: a sequence of block names and a receiver. The receiver is a unary function that will be applied to the reified partial continuation. Figure 2 represents the stack before evaluating the expression `(call/pc k1 (lambda (pc) ...))` appearing in the program. The behaviour of `call/pc` is specified by the five following actions.

- A new mark is pushed on the stack, giving the current block the new name α_4; so, the current continuation is composed of the blocks $\alpha_1, \alpha_2, \alpha_3$, and α_4.
- The partial continuation is computed by difference between the current continuation and the continuation whose sequence of block names is the value of the first argument of `call/pc` (here `k1`). This defines a unique partial continuation which, here, is the sequence of blocks named $\langle \alpha_2, \alpha_3, \alpha_4 \rangle$.
- This partial continuation is reified into a first-class object.
- The computation is aborted to the continuation with names `k1`: the portion of the stack from the `top` pointer to the mark α_1 is removed; the mark α_1 is left as it is because a mark is intended to name the block below it.
- The receiver is applied to the reified object above the mark α_1, starting a new control block.

After these steps, we can now represent the stack as in Figure 3. The current block is above the block α_1 and the captured partial continuation is composed of blocks α_2, α_3, and α_4.

```
(f0 (marker
     (lambda (k1)
      (f1 (marker
           (lambda (k2)
             (f2 (marker
                  (lambda (k3)
                    (f3 (call/pc k1
                         (lambda (pc)
                          (f4 (pc v))))
                    ))))))))))
```

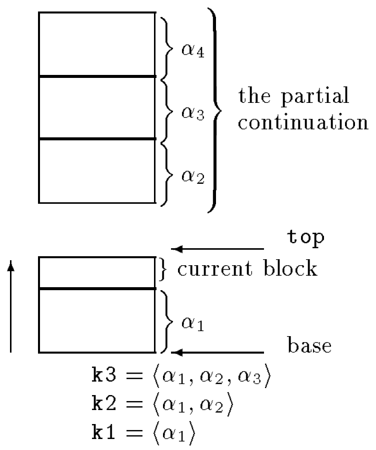

Fig. 3. Stack representation after application of the receiver to the partial continuation

3 Reifying a Partial Continuation

In Section 2, we defined a partial continuation as a sequence of blocks obtained by difference of two continuations. We have not yet precisely specified how such a partial continuation was reified into a first-class object. Several behaviours can be considered according to the way a reified partial continuation preserves its structure of control blocks. Let us examine the different choices and their impact on semantics.

Let us now consider the program of Figure 3, where `call/pc` is applied to a receiver that invokes the reified partial continuation `pc`. Two different approaches can be adopted.

1. Since we have defined a partial continuation as a sequence of control blocks separated by marks, we can consider that the structure of the blocks is preserved during the reification, i.e. marks are conserved. This case is illustrated in figure 4, where applying a partial continuation consists in *concatenating* the current continuation with the partial continuation. However, we can again consider two cases depending on whether we want to keep the structure of the block that is current at invocation time.
   (a) At invocation time, a new mark (here α_5) is introduced between the current block and the first block of the partial continuation.
   (b) At invocation time, no marker is introduced between the current block and the first block of the partial continuation so that they both appear under the mark α_2; since a mark is intended to name the block below it, both blocks seem to be merged in a single block named by α_2.

   After installing the partial continuation `pc`, the stack is composed of block names $\alpha_1, \alpha_5, \alpha_2, \alpha_3, \alpha_4$ in the first case and $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ in the second case. Afterwards, the value `v` is returned and the mark α_4 is removed; hence, the block that was previously called α_4 becomes the current block. But, in both cases, the variables `k1`, `k2`, and `k3` are still bound to $\langle\alpha_1\rangle$, $\langle\alpha_1, \alpha_2\rangle$, and $\langle\alpha_1, \alpha_2, \alpha_3\rangle$ respectively. We can conclude that $\langle\alpha_1\rangle$ is still a prefix of the names of the current continuation, but neither $\langle\alpha_1, \alpha_2\rangle$ nor $\langle\alpha_1, \alpha_2, \alpha_3\rangle$ is a prefix of the names of the current continuation in the first case. Consequently, after invocation of `pc`, it is allowed to capture a partial continuation with respect to `k2` and `k3` in the second case, but this is forbidden in the first case.

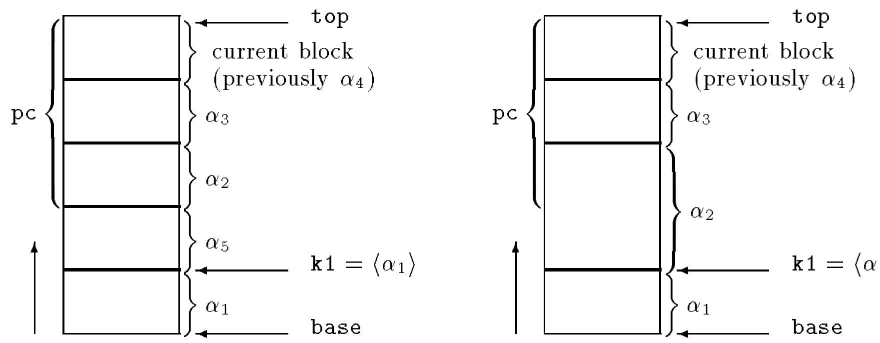

Fig. 4. Stack representation after invocation of a partial continuation preserving marks

Remark. Just before reifying a partial continuation, a newly created mark (α_4 in the running example) was pushed on the stack. As soon as this partial continuation is invoked, a value is returned to this mark, and this mark is removed from the stack. Therefore, this mark can never be reified and is somehow useless. However, we thought that for the purpose of explanation it was more uniform to consider named blocks rather than a sequence of named blocks and one unnamed block.

2. Although a partial continuation was defined as a sequence of blocks, we can consider that the process of reification removes all marks between blocks. Therefore, all blocks in the partial continuation appear to be merged as a single block. This is illustrated in Figure 5, where again two cases can be considered w.r.t. the current block.
   (a) At invocation time, a new mark (here α_5) is introduced between the current block and the single block of the partial continuation.
   (b) At invocation time, no marker is introduced between the current block and the single block of the partial continuation so that they are merged into a unique block.

   After installing the partial continuation `pc`, the stack is composed of block names $\alpha_1, \alpha_5, \alpha_4$ in the first case and α_1, α_4 in the second case. Afterwards, the value `v` is returned and the mark α_4 is removed; hence, the block that was previously called α_4 becomes the current block. Consequently, both solutions are able to reify a partial continuation with respect to `k1` but none of these solutions is able to reify a partial continuation with respect to `k2` or `k3`. Indeed, immediately after invocation of the partial continuation, $\langle\alpha_1\rangle$ is a prefix of the names of the current continuation, but neither $\langle\alpha_1, \alpha_2\rangle$ nor $\langle\alpha_1, \alpha_2, \alpha_3\rangle$ is a prefix.

The *extent* of sequences of names is the property that essentially makes a distinction between these four solutions. We shall define the *extent of a sequence of names* as the period of time within which this sequence of names can be used to reify a partial continuation.

The notion of dynamic extent is related to the use of a stack; it is traditionally defined as the period of time during which a stack remains active, i.e. blocks are pushed and popped *above* a given stack. In a language without partial continuations, the dynamic extent can be *equivalently* defined as the period of time during which evaluation is concerned with blocks *above a given block*. But, the ability of partial continuations to manipulate blocks blurs this notion of dynamic extent because a block is no longer uniquely associated with a stack. Therefore, we refine this notion of extent for partial continuations.

Definition 3.1 (Dynamic extent of a sequence of names) *Let n^* be the sequence of names of the current continuation c obtained by the operator* marker. *We say that*

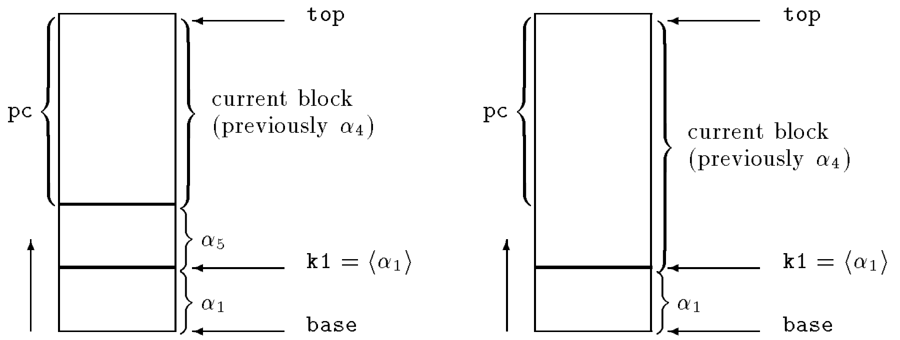

Fig. 5. Stack representation after invocation of a partial continuation without preservation of marks

n^* has a dynamic extent iff n^* cannot be used to reify a partial continuation after a value is returned to the current continuation c, or after the computation is aborted to a continuation c' that is a prefix of c.

In the first, third, and fourth solutions, sequences of names have a dynamic extent for two different reasons. In the first and third solutions, when a partial continuation is invoked, a new mark gives a name to the current block; so, the sequences of names captured in the partial continuation become obsolete because they are no longer prefix of the current continuation. In the third and fourth solutions, as soon as a partial continuation is captured with call/pc, all the marks captured in the partial continuation disappear.

However, note that sequences of names do not have a dynamic extent *per se*. As all first-class values, they have an unlimited extent, i.e. they can be referenced, stored, or returned even outside their dynamic extent, but it is the process of subtracting continuations that restricts the use of sequences of names w. r. t. call/pc.

On the contrary, in the second solution, sequences of names do not have a dynamic extent. We call *prefixal extent* the extent of sequences of names in the second solution.

Definition 3.2 (Prefixal Extent) *Let c be the continuation existing at the moment an object o is created. We say that o has a prefixal extent iff, for every c_1, the continuation existing when the object o is used, c is a prefix of c_1.*

Amongst the different approaches given above, it is only in the second one that sequences of names have a prefixal extent. In all other cases, the extent of sequences of names is dynamic. We prefer the notion of prefixal extent because it offers more expressiveness as it will be shown in examples of Section 6.

In the two following sections, we propose two formal semantics: the first of them is a reduction system and the second is a cps translation. These two semantics are proved to be congruent. By their equivalence, both semantics precisely define our duumvirate of control operators as well as they satisfy differing tastes. Both semantics formalize the four previously mentioned choices. The first one favours concision by the use of evaluation contexts. The second one exhibits the connection between stack frames and cps, as well as the machinery for new names creation.

4 Reduction System

In this section, we present a context-rewriting system as Felleisen and Friedman [5]. This rewriting system is an extension of the call-by-value λ-calculus [13] with the marker and call/pc operators. We also introduce a new syntactic construct $\#_\alpha(\ldots)$, called *prompt* with name α, that is intended to represent a mark naming a control block. Terms[3] of the language are defined inductively by the following grammar.

$$M ::= x \mid \langle \alpha, \beta, \ldots, \gamma \rangle \mid (\lambda x.M) \mid (MM) \mid (\mathsf{marker}\ M) \mid (\mathsf{call/pc}\ VM) \mid \#_\varphi(M)$$

A *value*, denoted by V, is either a variable x, a sequence of block names $\langle \alpha, \beta, \ldots, \gamma \rangle$, or an abstraction $(\lambda x.M)$. An expression of the form $(\mathsf{marker}\ M)$ is called a marker-*application*. An expression of the form $(\mathsf{call/pc}\ VM)$ is called a call/pc-*application*. An expression of the form $\#_\alpha(M)$ is called a *prompt*.

We also define an evaluation context $E^d[\,]$ indexed by a depth d, where d represents the number of prompts $\#_\alpha()$ in $E^d[\,]$.

$$\begin{aligned} E^0[\,] &::= (V\ E^0[\,]) \mid (E^0[\,]\ M) \mid (\mathsf{call/pc}\ V\ [\,]) \mid [\,] \\ E^d[\,] &::= (V\ E^d[\,]) \mid (E^d[\,]\ M) \mid (\mathsf{call/pc}\ V\ [\,]) \mid \#_\alpha(E^{d-1}[\,]) \end{aligned}$$

An evaluation context of depth d represents a continuation composed of $d+1$ blocks; amongst them, n blocks are named while the last one is the current block.

We can find in Figure 6 the definition of the context-rewriting system. Equation (1) is the call-by-value β-reduction [13]. In (2), a marker-expression is replaced by a prompt with a new name δ, and the receiver of marker, the expression M, is applied to the sequence of prompt names appearing in $E^d[\,]$ extended with δ. According to (3), when the evaluation in a prompt reaches a value, the prompt is removed, and evaluation proceeds with this value. This rule corresponds to the intuitive explanation given previously: when a value is returned to a mark at the top of the stack, the mark is removed. Equation (4) concerns the reification of a partial continuation. In order to apply this rule, the context $E_1^d[\,]$ should contain the names $\alpha \ldots \gamma$. The partial continuation is represented by the context $E_2^{d'}[\,]$. The operator call/pc applies the receiver f to a reification of the partial continuation returned by a call to $\mathcal{F}$, which transforms a context into a function.

$$\begin{aligned} E^d[(\lambda x.M)V] &\to E^d[M\{V/x\}] \quad \text{with } V \text{ a value} && (1) \\ E^d[\mathsf{marker}\ M] &\to E^d[\#_\delta(M\ \langle \delta, \gamma, \ldots, \alpha \rangle)] && (2) \\ & \qquad \text{with } \delta \notin E^d[M] \\ & \qquad \text{with } E^d[\,] \equiv E_0^0[\#_\alpha(\ldots \#_\gamma(E_d^0[\,])\ldots)] \\ E^d[\#_\alpha(V)] &\to E^d[V] && (3) \\ E_1^d[\#_\varphi(E_2^{d'}[\mathsf{call/pc}\ \langle \varphi, \gamma, \ldots, \alpha \rangle\ f])] &\to E_1^d[\#_\varphi(f\ \mathcal{F}(E_2^{d'}[\,]))] && (4) \\ & \qquad \text{with } E_1^d[\,] \equiv E_0^0[\#_\alpha(\ldots \#_\gamma(E_d^0[\,])\ldots)] \end{aligned}$$

Fig. 6. Context-rewriting system

The four possible ways of reifying a partial continuation are formalised in Figure 7. In the two first definitions of $\mathcal{F}$, the control information held in $E^d[\,]$ is preserved, while in the two last definitions, this information is removed by the function flat. With the first and third definitions, when the partial continuation is invoked, a prompt with a new name φ is inserted.

[3] In order to slightly simplify the definition of the reduction system, we suppose that a call/pc-expression has a value as a first argument.

Four ways of reifying a partial continuation:	Control information removal
$\mathcal{F}_1(E^d[\,]) = \lambda x.\#_\varphi(E^d[x])$ with a fresh φ	$\text{flat}(E^0[\,]) = E^0[\,]$
$\mathcal{F}_2(E^d[\,]) = \lambda x.E^d[x]$	$\text{flat}(V\ E^d[\,]) = (V\ (\text{flat}(E^d[\,])))$
$\mathcal{F}_3(E^d[\,]) = \lambda x.\#_\varphi(\text{flat}(E^d[x]))$ with a fresh φ	$\text{flat}(E^d[\,]\ M) = (\text{flat}(E^d[\,])\ M)$
$\mathcal{F}_4(E^d[\,]) = \lambda x.\text{flat}(E^d[x])$	$\text{flat}(\#_\alpha(E^{d-1}[\,])) = \text{flat}(E^{d-1}[\,])$

Fig. 7. Reification of a partial continuation

5 CPS Translation

A continuation semantics is another natural way to give control operators a semantics. In a traditional continuation semantics, a continuation maps an intermediate value to a final value, i.e. a continuation represents the rest of the computation. This property is not suitable for partial continuations since they are expected to be composable. Felleisen, Wand, Friedman, and Duba [7] proposed a non-traditional continuation semantics where a continuation algebra was derived from the evaluation contexts of the reduction system. This technique, named Abstract Continuation Passing Style, was also used by the second author to define `splitter` [14]. Although such an approach is also possible here, we have adopted Danvy and Filinski's technique [3] to exploit the expressive power of continuation-passing style through the use of multi-level continuations.

As explained in Sections 2 and 3, it is important to uniquely name a new control block. Unlike the operational semantics, we explicitly represent in the cps translation this naming facility by a single-threaded counter passed to each continuation. Whenever a new name is required, the counter is incremented by one, therefore guaranteeing the uniqueness of names.

The domains of values and the cps translation appear in Figure 8. A block is represented by an element of the domain K. A block returns a final answer when given a value, a continuation and a counter. An element of *Cont* is a continuation, which is nothing more than a sequence of named blocks. The semantic function $[\![\ ]\!]$ maps a program, a block, a continuation, and a counter to a final answer. There exists a similarity with the informal definition of Section 2, where the current block appears on top of the stack and the rest of the stack consists of named blocks: the current block is an element of K, and the stack is an element of *Cont*.

The auxiliary functions $\mathcal{P}$ and $\mathcal{D}$ used in the translation of call/pc in Figure 8 are defined in Figure 9. They receive a sequence of names and a continuation. The function $\mathcal{P}$ returns the *prefix* of the continuation having this sequence of names, i.e. it is the part of the continuation that is not reified. The function $\mathcal{D}$ returns the reification of the partial continuation computed by difference as explained in Section 2. Both $\mathcal{P}$ and $\mathcal{D}$ call the function *decompose*, which returns two continuations, the first being the prefix, the other the suffix. The function $\mathcal{F}$ reifies a continuation (i.e. a sequence of named blocks) into a function. Four definitions of $\mathcal{F}$ are given for each case of Section 3. We add to the set of names the special name $\diamond$, which stands for "anonymous". The blocks of a sequence are merged by giving them this anonymous name, which is considered as an invisible name by the function *decompose*.

6 Application

The operators marker and call/pc can be used to express Danvy and Filinski's [3] example about the non-deterministic finite state automaton and Danvy's example of computation of the prefixes of a list [2]. In Sections 6.2 and 6.3, we illustrate how

$$
\begin{aligned}
[\![\lambda x.M]\!]\kappa\kappa^*d &= \kappa(\lambda\kappa'.\lambda x.\lambda\kappa'^*d'.[\![M]\!]\kappa'\kappa'^*d')\kappa^*d \\
[\![MN]\!]\kappa\kappa^*d &= [\![M]\!]\ (\lambda m\kappa'^*d'.[\![N]\!](\lambda n\kappa''^*d''.m\kappa n\kappa''^*d'')\kappa'^*d')\kappa^*d \\
[\![x]\!]\kappa\kappa^*d &= \kappa\ x\kappa^*d \\
[\![\textsf{marker}\ M]\!]\kappa\kappa^*d &= [\![M]\!]\ (\lambda m\kappa^*_m d_m.\ m\ \theta_0\ \ (\langle d_m\rangle\S(\mathrm{map}\ (\lambda x.x\downarrow 1)\ \kappa^*_m)) \\
&\qquad (\langle\langle d_m,\kappa\rangle\rangle\S\kappa^*_m)\ (d_m+1))\kappa^*d \\
[\![\textsf{call/pc}\ c\ M]\!]\kappa\kappa^*d &= [\![M]\!]\ (\lambda m\kappa^*_m d_m.\ m\ \theta_0\ \ \mathcal{D}(c,(\langle\langle d_m,\kappa\rangle\rangle\S\kappa^*_m)) \\
&\qquad \mathcal{P}(c,(\langle\langle d_m,\kappa\rangle\rangle\S\kappa^*_m))\ (d_m+1))\quad \kappa^*\ d \\
\theta_0 &= \lambda v\kappa^*d.(((\kappa^*\downarrow 1)\downarrow 2)v)(\kappa^*\dagger 1)d \qquad \text{the initial block} \\
\kappa_0 &= \lambda v.\lambda\kappa^*.\lambda d.v \qquad \text{the initial continuation} \\
\kappa \in K &: Val \to Cont \to N \to Ans \\
f \in Fun &: K \to Val \to Cont \to N \to Ans \\
\kappa^* \in Cont &: (N\times K)^* \\
[\![\ ]\!] &: Prog \to K \to Cont \to N \to Ans
\end{aligned}
$$

Fig. 8. CPS translation

$decompose : Cont \times N^* \times Cont \to \langle Cont, Cont\rangle$

$decompose(cont, names, acc) =$

if $null?(cont)$ then $wrong($`"not in extent"`$)$

elif $null?(names)$ then $\langle acc, reverse(cont)\rangle$

elif $names\downarrow 1 = \diamond$ then $decompose(cont, names\dagger 1, acc)$

elif $(cont\downarrow 1)\downarrow 1 = \diamond$ then $decompose(cont\dagger 1, names, \langle cont\downarrow 1\rangle\S acc)$

elif $names\downarrow 1 = (cont\downarrow 1)\downarrow 1$ then $decompose(cont\dagger 1, names\dagger 1, \langle cont\downarrow 1\rangle\S acc)$

else $wrong($`"not in extent"`$)$

$\mathcal{P} : N^* \times Cont \to Cont$ **prefix**

$\mathcal{P}(c,\kappa^*) = decompose(reverse(\kappa^*), reverse(c), \langle\rangle)\downarrow 1$

$\mathcal{D} : N^* \times Cont \to Fun$ **difference**

$\mathcal{D}(c,\kappa^*) = \mathcal{F}(decompose(reverse(\kappa^*), reverse(c), \langle\rangle)\dagger 1)$

$\mathcal{F} : Cont \to Fun$

$$
\begin{aligned}
\mathcal{F}_1(\kappa^*) &= \lambda\kappa_1 v_1 d_1\kappa^*_1.((\theta_0\ v_1)\ (d_1+1))\ (\kappa^*\S\langle\langle d_1,\kappa_1\rangle\rangle\S\kappa^*_1) \\
\mathcal{F}_2(\kappa^*) &= \lambda\kappa_1 v_1 d_1\kappa^*_1.((\theta_0\ v_1)\ d_1)\ (\kappa^*\S\langle\langle\diamond,\kappa_1\rangle\rangle\S\kappa^*_1) \\
\mathcal{F}_3(\kappa^*) &= \lambda\kappa_1 v_1 d_1\kappa^*_1.((\theta_0\ v_1)\ (d_1+1))\ ((\mathrm{map}\ (\lambda x.\langle\diamond, x\dagger 1\rangle)\ \kappa^*)\S\langle\langle d_1,\kappa_1\rangle\rangle\S\kappa^*_1) \\
\mathcal{F}_4(\kappa^*) &= \lambda\kappa_1 v_1 d_1\kappa^*_1.((\theta_0\ v_1)\ d_1)\ ((\mathrm{map}\ (\lambda x.\langle\diamond, x\dagger 1\rangle)\ \kappa^*)\S\langle\langle\diamond,\kappa_1\rangle\rangle\S\kappa^*_1)
\end{aligned}
$$

Fig. 9. Reification of a partial continuation

our duumvirate of control operators can be used in two new applications of partial continuations. But beforehand, we show that marker and call/pc can simulate `call/cc`.

6.1 Simulating `call/cc`

The control operators marker and call/pc provide the user with a facility to specify the part of the rest of the computation he precisely wishes to reify. On the other hand, `call/cc` is able to reify the whole rest of the computation. Therefore, it is not surprising that `call/cc` can be simulated by marker and call/pc. With a toplevel-based implementation of Scheme, `call/cc` also has some weird interactions with the toplevel. Consider for instance a continuation reified at interaction n and used at interaction $n+1+m$: either - (i) it is a full continuation that comprises the toplevel mechanism which must take care of being multiply returned to, or, - (*ii*) it excludes the toplevel mechanism so it is a bounded continuation more akin to be described as a partial continuation associated with an abortive effect.

Using the primitives `marker` and `call/pc`, we can define `call/cc`. Any program ⟨*prog*⟩ using `call/cc` can be replaced by the following program.

```
(marker
 (lambda (lowest)
   (let ((call/cc (lambda (f)
                    ((call/pc lowest
                            (lambda (pc)
                              (pc (lambda ()
                                   (f (lambda (v)
                                        (call/pc lowest
                                               (lambda (any)
                                                (pc (lambda () v))))))))))))))
     ⟨prog⟩)))
```

6.2 Run-Time Partial Evaluation: Path Reification

Let us consider a binary tree `tree` and a predicate `pred`; let us search `tree` for a leaf that satisfies `pred`. We are not interested in the leaf itself but in the direct path that leads to this leaf. In Figure 10, we illustrate the search and direct paths that lead to leaf 2. Moreover, we would like the direct path to be represented by a unary function: given a leaf v, this function would build a new tree that is the same as `tree`, except for the leaf v that replaces the leaf satisfying `pred`.

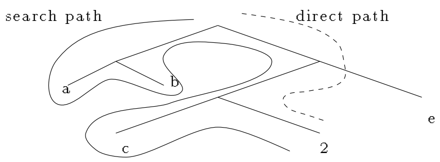

Fig. 10. Search and Direct Paths

Here is the function `search`.

```
(define (search tree lowest pred)
  (letrec ((loop (lambda (tree fail)
                   (if (pair? tree)
                       (or (marker (lambda (fail)
                                     (cons (loop (car tree) fail) (cdr tree))))
                           (cons (car tree) (loop (cdr tree) fail)))
                       (if (pred tree)
                           (call/pc lowest (lambda (pc) pc))
                           (abort fail (lambbda () #f)))))))
    (marker (lambda (fail) (loop tree fail)))))
```

Partial continuations are particularly useful to solve this problem.

1. The operator `marker` marks each point of choice where the search should backtrack to, in case of failure.

2. Backtracking is implemented by aborting to a mark. The **abort** operator simply reifies a partial continuation, discards it, and forces the evaluation of the thunk.

```
(define (abort mark thunk)
 (call/pc mark (lambda (pc) (thunk))))
```

3. When the search succeeds, the partial continuation up to the call of `search` is reified. This yields the expected function that represents the path.
4. We could have used first-class continuations, but it would have required a complex protocol to allow the function `search` to return a value to its caller.
5. It is possible to write such a function in "continuation-passing style"; two continuations would be required: a *success* continuation and a *failure* continuation. On the contrary, the solution that we propose is in "direct style" and only passes the failure continuation. The continuation-passing style and the direct version differ by their performance. The former allocates two closures (representing the success and failure continuations) per node visited. In the worst case, $O(2^n)$ closures must be allocated in the heap, where n is the depth of the found leaf. On the contrary, the direct style version does not allocate closures, but only copies the reified partial continuation into the heap; the cost is proportional to n.

The following example shows that the same function can be used several times to construct different trees. Moreover, the last result illustrates that the trees built by the function `path` are shared with the initial tree `the-tree`, as much as possible.

```
(define the-tree (cons (cons 'a 'b)    (cons (cons 'c 2) 'e)))
(define path     (marker (lambda (init) (search the-tree init number?))))

(path 'd)                          → ((a . b) . ((c . d) .e))
(path '3)                          → ((a . b) . ((c . 3) .e))
(eq? (car (path 'd)) the-tree)     → #t
```

Unfortunately, the partial continuation returned by `search` is only able to construct a new tree. Instead, we could parameterise the function by two constructors `cons1` and `cons2`.

```
(define (search tree lowest pred cons1 cons2)
 (letrec ((loop (lambda (tree fail)
                 (if (pair? tree)
                     (or (marker (lambda (fail)
                                  (cons1 (loop (car tree) fail) (cdr tree))))
                         (cons2 (car tree) (loop (cdr tree) fail)))
                     (if (pred tree)
                         (call/pc lowest (lambda (pc) pc))
                         (abort fail (lambbda () #f)))))))
  (marker (lambda (fail) (loop tree fail)))))
```

In the following example, the function `path` computes the depth of the leaf found by `search`, i.e. the length of the direct path.

```
(define path (marker (lambda (init)
                      (search the-tree init number?
                              (lambda (x y) (+ 1 x))
                              (lambda (x y) (+ 1 y))))))
(path 0)       → 3
```

But this solution requires the function `search` to be passed the constructors before reifying the partial continuation. We propose below a solution that needs the values of the constructors only when the partial continuation is invoked. (It is possible to design a version without side-effects at the price of extra-allocations of closures.)

```
(define (search tree lowest pred)
 (let ((cons1 'any)
       (cons2 'any))
  (letrec ((loop
            (lambda (tree fail)
             (if (pair? tree)
                 (or (marker
                      (lambda (fail)
                       ((lambda (x y) (cons1 x y)) (loop (car tree) fail)
                                                   (cdr tree))))
                     ((lambda (x y) (cons2 x y)) (car tree)
                                                 (loop (cdr tree) fail)))
                 (if (pred tree)
                     (call/pc lowest (lambda (pc)
                                      (lambda (v cons11 cons22)
                                        (set! cons1 cons11)
                                        (set! cons2 cons22)
                                        (pc v))))
                     (abort fail (lambbda () #f)))))))
    (marker (lambda (fail) (loop tree fail))))))
```

Now, the same partial continuation can be used for several purposes. Below, in `[1]`, a tree is built; in `[2]`, the depth is computed; in `[3]`, the result is a tree that mirrors `the-tree` with respect to the direct path.

```
(define path  (marker (lambda (init) (search the-tree init number?))))
(define (rcons x y) (cons y x))

(path cons cons 'here)                → ((a . b) . ((c . here) .e)) [1]
(path (lambda (x y) (+ 1 x))
      (lambda (x y) (+ 1 y)) 0)       → 3                           [2]
(path rcons rcons 'here)              → ((e .(here . c)) . (a . b)) [3]
```

This technique can be referred to as *run-time partial evaluation*. Consel and Danvy [1] define partial evaluation as “a source-to-source program transformation technique for specialising programs with respect to parts of their input”. For example, let f be a function requiring two arguments, if f is applied to x and y with x known at compile-time, the process of partial evaluation generates the definition of a new function f_x which requires one argument, with f_x such that $\forall y, f_x(y) = f(x,y)$.

On the contrary, let us consider a program where a two-argument function f is applied to x, y_1 and to x, y_2 but x, y_1, y_2 will be known at run-time only. The technique of partial evaluation cannot be used here since this process requires the value x to be known at partial-evaluation-time. One could imagine to generate at run-time a specialised version of f, but as opposed to partial evaluation, we would not generate the text of the definition of the function f_x, but a closure f_x specialised for x.

The function `search` generates a specialised function for the tree passed in argument; `search` just unfolds the recursive calls until a leaf satisfies the predicate.

6.3 Insertion of Values

Let `l1` be a list of length m, and let `l2` be an A-list associating n values with numbers. We want to construct a new list where each value of `l2` is inserted in `l1` at the position which it is associated with in `l2`. We suppose that `l2` is not sorted in ascendent position-order. One solution consists in sorting the list `l2` and then constructing the final result by merging the list `l1` and the values at the positions indicated in the sorted list `l2`. This solution requires $O(n \log(n))$ comparisons for sorting `l2` and $O(m+n)$ comparisons for insertion of values.

We propose another solution that requires neither to sort the list `l2` nor to compare positions for insertions. The function `walk-list` builds a vector with $m+1$ marks. Each mark delimits a control block that conses an element of the list `l1`. The insertion of values is performed in `insert-vals`: for a position and a value, `insert-vals` reifies a partial continuation with respect to the mark in the corresponding position in the vector and re-installs this partial continuation after consing the value.

```
(define (walk-list l acc next-action)
 (if (null? l)
     (marker (lambda (k) (next-action (cons k acc))))
     (cons (car l)
           (marker (lambda (k)
                     (walk-list (cdr l) (cons k acc) next-action))))))

(define (insert-vals l mark-vect)
  (if (null? l)
      '()
      (let ((position (car (car l)))
            (value (cdr (car l))))
       ((call/pc (vector-ref mark-vect position)
                 (lambda (pc)
                   (cons value
                         (pc (lambda ()
                               (insert-vals (cdr l) mark-vect))))))))))

(define (insert l1 l2)
 (marker (lambda (k)
           (walk-list l1
                      (list k)
                      (lambda (mark*)
                        (insert-vals l2 (list->vector (reverse mark*))))))))

(insert '(3 7 33 23 53)
        '((2 3000) (5 6000) (3 4000) (1 2000) (0 1000) (4 5000)))
→ (1000 3 2000 7 3000 33 4000 23 5000 53 6000)
```

This solution reifies n partial continuations and invokes each of them only once. The exact cost is highly dependent of the implementation. We can nevertheless approximate the cost of the n invocations to roughly $O(mn)$ since there are at most m blocks to concatenate per invocation. This approach offers a speed-up when `l2` is much longer than `l1` ($n \log(n) > nm$).

In this problem, the positions of the elements of `l1` remain constant during the insertion of values. It is clear that the complexity of the solution that we propose is enhanced by the use of a random access data structure: a vector here. It is easy to imagine another solution with side-effects. (Let us consider a vector of buckets, and let us add (by a side-effect) each value of `l2` to a bucket at the desired position; the concatenation of the buckets yields the result.) However, the remarkable feature of the solution with partial continuations is that it *does not use side-effects*.

Moreover, this example shows that a new programming style can be adopted with partial continuations. We know that we have to cons the m values of list `l1` but we do not know where to insert values of `l2`: partial continuations allow us to split the traversals of `l1` and `l2` in two separate functions. This example also illustrates that our duumvirate of control operators needs not be extended to a hierarchy of control operators. Indeed, we are able to capture any part of the computation whatever the marks appearing inside. And finally, this example exhibits a typical use of marks in their prefixal extent.

7 Related Work

Queinnec and Serpette's [15] `splitter-call/pc` is certainly the closest approach to marker-call/pc. Indeed, `splitter` leaves a special mark on the evaluation stack and passes this mark to its receiver; `call/pc` reifies the portion of the current stack appearing above this mark[4]. However, `splitter-call/pc` is less expressive than marker-call/pc since `call/pc` is restricted to the dynamic extent of the mark. Consequently, examples like `insert` in the previous section cannot be programmed with `splitter` since marks must be usable outside their dynamic extent.

Moreover, we have tried to simplify the semantics as much as possible. We have included in the semantics the minimal number of features necessary for marker-call/pc. We neither require the full power of a store nor an extent parameter as in [15]. Another semantics of `splitter` appears in [14] in Abstract Continuation Passing Style [7], however this semantics subsumes the existence of both a store and a physical comparison without making this store explicit.

Danvy and Filinski's `shift/reset` and Hieb and Dybvig's `spawn` have in common that partial continuations incorporate the mark up to which they were reified. This approach is totally opposite to ours; indeed, we consider that a mark is pushed on a stack to give a name to the block below it and a mark is pushed on the stack on user's request. Including a mark in the partial continuation is therefore meaningless and can be seen as a "dangling" name which, whenever the partial continuation is invoked, will name the current block.

On the other hand Queinnec and Serpette's `splitter` and Felleisen, Wand, Friedman, and Duba's [7], [4] $\#(), \mathcal{F}$ avoid to copy the mark (or prompt) when reifying a partial continuation. Those two trends are the origin of the debate about the "dynamicness" of control operators. Although we have decided not to copy a mark when reifying a continuation, a similar problem also arises when a partial continuation is applied. We have to know whether a new mark should be inserted between the current block and the partial continuation: should we name the current block by a new name not given by the user or should this block be merged with the first block of the partial continuation?

In order to be able to reify a partial continuation between the current point and any mark (not the last one) Felleisen and Sitaram [17] and Danvy and Filinski [3] introduce a hierarchy of control operators. Therefore, their languages have $2n$ times control operators instead of 2. But, what is the intuitive difference between control operators of level n or level $n+1$?

Moreau and Ribbens [11] also introduce a mechanism of prompt to mark the extent of the `call/cc` operator. Such a prompt was used to optimise the invocation of a continuation in the extent of the `call/cc` by which it was reified. In their reduction system, the prompt could be used to represent explicitly two extents: the dynamic extent or another notion of extent, similar to prefixal extent except that the prefix notion uses equality of blocks instead of equality of block names.

8 Conclusion

We presented an operator marker that reifies the block names of the current continuation. This is the minimum requirement for our semantics. A nice integration with Scheme would be to merge marker and `call/cc`, thus reifying a sequence of block names into a regular continuation. On the other hand, call/pc would extract the names that it needs from this regular continuation. Hence, only one primitive should be added to the language Scheme in order to reify partial continuations.

[4] In fact, the lowest position where this mark appears.

We defined a partial continuation as the difference of two continuations when one is the prefix of the other. Although other notions of prefixes might be imagined, we think that they must, at least, be able to properly evaluate our examples.

9 Acknowledgement

Luc Moreau wishes to thank INRIA for a three months visit of Projet ICSLA where this work was initiated. Both authors have been partially funded by Projet Tournesol 94016, and Christian Queinnec has been partially funded by GDR-PRC de Programmation du CNRS. The authors also wish to thank Olivier Danvy, Matthias Felleisen, and the anonymous referees for their helpful comments.

References

1. Charles Consel and Olivier Danvy. Tutorial Notes on Partial Evaluation. In *Proceedings of the Twentieth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Charleston, January 1993.
2. Olivier Danvy. On Listing List Prefixes. *Lisp Pointers*, 2(3-4):42–46, 1989.
3. Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, June 1990.
4. Matthias Felleisen. The Theory and Practice of First-Class Prompts. In *Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 180–190, January 1988.
5. Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).
6. Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond Continuations. Technical Report 216, Indiana University, February 1987.
7. Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract Continuations : A Mathematical Semantics for Handling Full Functional Jumps. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 52–62, July 1988.
8. Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. Obtaining Coroutines with Continuations. *Comput. Lang.*, 11(3/4):143–153, 1986.
9. Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.
10. Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic and Computation, Special Issue on Continuations*, 1(7):83–110, January 1994.
11. Luc Moreau and Daniel Ribbens. Sound Rules for Parallel Evaluation of a Functional Language with callcc. In *ACM conference on Functional Programming and Computer Architecture (FPCA'93)*, pages 125–135, Copenhagen, June 1993. ACM.
12. Chetan R. Murthy. Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work. In *Workshop on Continuations*, pages 49–71. ACM Sigplan, June 1992.
13. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.
14. Christian Queinnec. Value Transforming Style. In M. Billaud, P. Castéran, MM. Corsini, K. Musumbu, and A. Rauzy, editors, *WSA '92—Workshop on Static Analysis*, number 81-82 in Bigre, pages 20–28, Bordeaux (France), September 1992.
15. Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1991.
16. John C. Reynolds. The Discoveries of Continuations. *Lisp and Symbolic and Computation, Special Issue on Continuations*, 6(3/4):233–248, November 1993.
17. Dorai Sitaram and Matthias Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.
18. Christopher Strachey and Christopher P. Wadsworth. A Mathematical Semantics for Handling Full Jumps. Technical Monography PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, 1974.