# An Operational Investigation
# of the CPS Hierarchy

Olivier Danvy[1] and Zhe Yang[2] [*]

[1] BRICS [***]
Department of Computer Science, University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
`danvy@brics.dk`
[2] Department of Computer Science, New York University
251 Mercer Street, New York, NY 10012, USA
`zheyang@cs.nyu.edu`

**Abstract.** We explore the hierarchy of control induced by successive transformations into continuation-passing style (CPS) in the presence of "control delimiters" and "composable continuations". Specifically, we investigate the structural operational semantics associated with the CPS hierarchy.

To this end, we characterize an operational notion of continuation semantics. We relate it to the traditional CPS transformation and we use it to account for the control operator **shift** and the control delimiter **reset** operationally. We then transcribe the resulting continuation semantics in ML, thus obtaining a native and modular implementation of the entire hierarchy. We illustrate it with several examples, the most significant of which is layered monads.

## 1 Introduction

### 1.1 Background

Continuation-passing style (CPS) programs are usually obtained by CPS transformation. The CPS hierarchy is obtained by iterating the CPS transformation, which yields programs whose types obey the following pattern:

$$\text{Fun} = \text{Val}_0 \rightarrow \text{Cont}_1 \rightarrow \text{Cont}_2 \rightarrow ... \rightarrow \text{Cont}_n \rightarrow \text{Ans}_n$$
$$\text{Cont}_1 = \text{Val}_1 \rightarrow \text{Cont}_2 \rightarrow ... \rightarrow \text{Cont}_n \rightarrow \text{Ans}_n$$
$$...$$
$$\text{Cont}_n = \text{Val}_n \rightarrow \text{Ans}_n$$

In the CPS hierarchy, programs exhibit the familiar pattern of success/failure continuations which is pervasive in functional specifications of backtracking.

---

Enriching CPS with identity and composition makes it possible to simulate Prolog-style backtracking and also to accumulate results. To simulate Prolog-style backtracking, we successively apply the current continuation to all possible choices—failing if there are none. To accumulate results, we use the current continuation to compute the result of the "remaining computation" and stash it away in an accumulator. To combine these two control mechanisms, e.g., to accumulate all the possible results of a non-deterministic computation in a list, we exploit the natural hierarchy between these two processes: the generation should take place in a context where its successive results are accumulated. We therefore CPS-transform the generation process (thus making its continuation continuation-passing) and we supply the accumulation process as its initial continuation [5].

The CPS hierarchy thus offers a fitting platform to express hierarchical backtracking—at the price indicated by the types above: a quadratic inflation of continuations.

The last level of CPS can be avoided by using the identity continuation and the ability to compose continuations. This is often deemed enough when $n = 1$ or $n = 2$ in the type equations above. For higher values of $n$, this quadratic cost can be alleviated with a linguistic device: two new syntactic forms in direct style, whose CPS transformation yields the desired effect of initializing a continuation with identity and of composing continuations. Initializing a continuation with identity is achieved with the control delimiter **reset**. Composing continuations is enabled by the control operator **shift** which captures the current (delimited) continuation and makes it ready to be composed with a subsequent continuation [5,6]. For comparison, the control operator **callcc** captures the current (unlimited) continuation and makes it ready to replace a subsequent continuation [17,23].

The challenge now is how to implement the CPS hierarchy more directly than by repeated CPS transformations and more efficiently than with a definitional interpreter [5]. Filinski showed how to implement the first level natively, using **callcc** and one reference cell [12]. In this article, we show how to implement the entire hierarchy natively, using **callcc** and one reference cell per level.

## 1.2   Related work

The CPS hierarchy was identified and advocated by Danvy and Filinski [5], who also introduced the corresponding hierarchy of control operators **shift**$_n$ and **reset**$_n$ (one per surrounding continuation). At the same time, but independently of CPS, Felleisen invented control delimiters [8], initiating a whole area of work on composable continuations and hierarchies of control [9,10,16,18,19,21,22,27], [31,33,34]. Control delimiters, for example, were instrumental to obtain a full-abstraction result [35].

All researchers in this new area followed Felleisen and defined their new control constructs operationally. They reported a variety of control operators, each of these displaying inventiveness in its modus operandi, its description, and its implementation.

In contrast, **shift** and **reset** are defined by translation into CPS. They have also proven particularly fruitful: because (we believe) control delimiters and composable continuations arise naturally in the CPS hierarchy, a number of applications of **shift** and **reset** were reported through the 90's [4,12,14,24,25,37], up to and including the R5RS [23]. There were only two further studies, however, of the CPS hierarchy and its guidelines: Murthy's, formalizing its type system [28], and Filinski's, establishing its equivalence with computational monads [13,15].

### 1.3    This work

The growing number of applications of **shift** and **reset** leads one to want to combine them. For example, suppose that we want to specialize programs that use **shift** and **reset**, using type-directed partial evaluation [4]. The problem is that type-directed partial evaluation also uses **shift** and **reset**, and we would like these two uses not to interfere with each other. This kind of applications require the CPS hierarchy to layer different uses of **shift** and **reset** at different levels.

It is difficult to implement the CPS hierarchy natively, since the semantics of built-in constructs cannot be altered. We thus take a novel approach using operational semantics: we characterize an operational 'continuation semantics' and following Section 1.1, (1) we enrich it with identity and composition of continuation as provided by **shift** and **reset**, and (2) we transform the result in a new continuation semantics. The new semantics extends the old one with a new pair of **shift** and **reset**; moreover, it can be natively implemented in the old semantics, since all the rules in the new semantics except those of the newly added operators are those of the old semantics, with the addition of one unchanged component. Iterating this process yields a family of semantics—the CPS hierarchy—and its native and modular implementation in ML, à la Filinski [12,13,15]. This general approach provides a native implementation of the new language constructs.

En passant, to make sure that we account for **shift** and **reset** as originally defined (i.e., by CPS transformation), we relate our operational notion of continuation semantics with the traditional CPS transformation.

### 1.4    Applications

*A toy example:* The two following computations declare an outer context `1 + [ ]`. They also declare a delimited context `[50 + [ ]]`, which is abstracted as a function denoted by `k`. This function is successively applied to `0`, yielding `50`, and to `10`, yielding `60`. These two results are added, yielding `110` which is then plugged in the outer context. In both cases, the overall result is `111`.

```
1 + reset (fn () => 50 + shift (fn k => (k 0) + (k 10)))

1 + let fun k v = 50 + v in (k 0) + (k 10) end
```

In the first computation, the context is delimited by `reset` and the delimited context is abstracted into a function with `shift`. The second computation is the continuation-passing counterpart of the first one.

*More substantial examples:* The CPS hierarchy also makes it possible to express computations like min-max processes or quantifier alternation. For example, the existential quantifier $\exists v.p(v)$ for a condition $p$ over non-deterministically generated values $v$ can be implemented as

```
fun exist v p
    = shift (fn k => if (p v) then true else k ())
```

where the return value of **reset** corresponding to the collection is set to `false`. Similarly, we can implement the universal quantifier with a function `forall`. Using these two functions at different levels, we can write formulae of arbitrary quantifier alternation.

## 2   Operational Semantics of the CPS Hierarchy

This section presents a family of operational semantics that can be directly transcribed into an implementation.

   Starting with an operational semantics $S$ for ML (Section 2.1), we characterize an operational notion of continuation semantics (Section 2.2). We then relate continuation semantics and syntactic CPS transformation, which is a result in itself (Sections 2.3 and 2.4). Based on the CPS transformation, we provide a semantic account of **shift** and **reset** (Section 2.5).

   The semantics $L$, which is $S$ extended with **shift** and **reset**, is no longer a continuation semantics. We induce two continuation semantics $H$ and $I$, and prove that they both simulate the semantics $L$ (Sections 2.6 and 2.7). Moreover, $I$ is directly implementable in $S$, in that the $S$-rules embed into the $I$-rules with the addition of one unchanged component. This component can be implemented by a reference cell. As for the remaining $I$-rules, they correspond to new control operators which can be implemented as functions.

   The resulting semantics $I$ is a continuation semantics, and thus, generalizing, we can iterate the whole transformation (Section 2.8). The resulting family of operational semantics formalizes the CPS hierarchy and is directly implementable in the initial operational semantics of ML (Section 3).

### 2.1   Starting semantics $S$

We use Harper, Duba, and MacQueen's "continuation-based operational semantics" for ML [17] as our starting semantics $S$.[1] Its syntactic categories are defined by the following grammar.

$$e \in Exp ::= x \mid \ell \mid \lambda x.e \mid e_0\,e_1 \qquad \text{—expressions}$$
$$v \in Val ::= \ell \mid \lambda x.e \qquad \text{—values}$$
$$k \in Cont ::= \Box \mid k\,e \mid v\,k \qquad \text{—continuations}$$

Its inference rules specify a judgment of the form "$k \vdash e \Rightarrow v$" which reads "under the continuation $k$, evaluating the expression $e$ yields the answer $v$" (Table 1).

---

[1] For brevity, both WRONG and LET rules in the original semantics are omitted here. The WRONG rule specifies the error case and serves in the formulation of type soundness. The LET rule is only there for ML's let polymorphism.

$$\text{(VAL0)}\frac{}{\square \;\vdash\; v \;\Rightarrow\; v} \qquad\qquad \text{(VAL1)}\frac{\square \;\vdash\; k[v_1] \;\Rightarrow\; v}{k \;\vdash\; v_1 \;\Rightarrow\; v} \;\; (k \neq \square)$$

$$\text{(FN)}\frac{k[\square\, e_1] \;\vdash\; e_0 \;\Rightarrow\; v}{k \;\vdash\; e_0\, e_1 \;\Rightarrow\; v} \;\; (e_0 \text{ not a value}) \qquad \text{(ARG)}\frac{k[v_0\, \square] \;\vdash\; e_1 \;\Rightarrow\; v}{k \;\vdash\; v_0\, e_1 \;\Rightarrow\; v} \;\; (e_1 \text{ not a value})$$

$$\text{(BETA)}\frac{k \;\vdash\; [v_1/x]e \;\Rightarrow\; v}{k \;\vdash\; (\lambda x.e)\, v_1 \;\Rightarrow\; v}$$

**Table 1.** An operational continuation semantics

A continuation $k$ can be thought of as an expression with precisely one "hole" $\square$ in it. We write $k[e]$ and $k[k']$ to denote the expression and continuation obtained by filling the hole in $k$ with an expression $e$ and a continuation $k'$, respectively, where $k[k']$ is the "composition" of $k$ with $k'$.

We are mainly interested in the dynamic semantics of **shift** and **reset**, and thus we do not present typing rules and the related soundness proof, which can be adapted from the work of Harper, Duba, and MacQueen [17] and of Gunter, Rémy, and Riecke [16]. We rely on the static type system of our implementation language, ML, for the type soundness (Section 3).

### 2.2 An operational notion of continuation semantics

Operational semantics give rise to derivation trees. We define a *branchless semantics* as an operational semantics whose rules have one premise at most. Such a semantics gives rise to *branchless* derivation trees, i.e., lists. Note that a branchless semantics directly corresponds to a reduction semantics, where reduction proceeds from the conclusion of a rule to the premise, or to the final result if the rule has no premise. Staying in the world of branchless evaluation semantics makes it easy to refer to a complete computation (as a judgment) as well as a single reduction (as a rule instance).

A continuation semantics, like the one in Table 1, is branchless: the continuation component in its judgments keeps track of the remaining branches in a corresponding direct-style derivation tree; it can be regarded as the stack used to traverse this derivation tree.

### 2.3 CPS transformation

Previous studies of the CPS hierarchy build on the CPS transformation. To justify our study of control operators with the continuation semantics of Table 1, we adapt the call-by-value CPS transformation to its expressions, values, and continuations.

$$\begin{aligned} [\![x]\!]_{Exp} &= \lambda\kappa.\kappa\, x \\ [\![v]\!]_{Exp} &= \lambda\kappa.\kappa\, [\![v]\!]_{Val} \\ [\![e_0\, e_1]\!]_{Exp} &= \lambda\kappa.[\![e_0]\!]_{Exp}\, \lambda v_0.[\![e_1]\!]_{Exp}\, \lambda v_1.v_0\, v_1\, \kappa \end{aligned} \qquad\qquad \begin{aligned} [\![\ell]\!]_{Val} &= \ell \\ [\![\lambda x.e]\!]_{Val} &= \lambda x.[\![e]\!]_{Exp} \end{aligned}$$

$$[\![\Box]\!]_{Cont} = \lambda v.\lambda\kappa.\kappa\, v$$
$$[\![k\, e_1]\!]_{Cont} = \lambda v.\lambda\kappa.[\![k]\!]_{Cont}\, v\, \lambda v_0.[\![e_1]\!]_{Exp}\, \lambda v_1.v_0\, v_1\, \kappa$$
$$[\![v_0\, k]\!]_{Cont} = \lambda v.\lambda\kappa.[\![k]\!]_{Cont}\, v\, \lambda v_1.[\![v_0]\!]_{Val}\, v_1\, \kappa$$

The rationale for $[\![\cdot]\!]_{Cont}$ is that the "hole" $\Box$ in the continuation is an abstraction over a value, as captured in the following lemma, where "$=_{\beta\eta}$" denotes $\beta\eta$-convertibility.

**Lemma 1.** *For all $k \in Cont$ and $v \in Val$, $[\![k[v]]\!]_{Exp} =_{\beta\eta} [\![k]\!]_{Cont}\, [\![v]\!]_{Val}$.*

*Proof.* By structural induction on $k$.

## 2.4 Soundness and completeness of the operational semantics

The following theorem connects the continuation semantics of Section 2.1 and the CPS transformation of Section 2.3.

**Theorem 1.** *For all $k \in Cont$, $e \in Exp$, and $v \in Val$, if $k \vdash e \Rightarrow v$ then*
$$[\![e]\!]_{Exp}\, \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a =_{\beta\eta} [\![v]\!]_{Val}.$$

*Proof.* By rule induction on $k \vdash e \Rightarrow v$.

**Corollary 1.** *For all $e \in Exp$ and $v \in Val$, if $\Box \vdash e \Rightarrow v$ then $[\![e]\!]_{Exp}\, \lambda w.w =_{\beta\eta} [\![v]\!]_{Val}$.*

Because the term $[\![e]\!]_{Exp}\, \lambda w.w$ is convertible to a value $[\![v]\!]_{Val}$, it must reduce to a *value* that is equal to $[\![v]\!]_{Val}$ modulo $\beta\eta$-conversion under normal-order evaluation (by the Normalization theorem), and thus also under applicative-order evaluation (by Plotkin's Indifference theorem [29]). The operational semantics is thus sound with respect to the call-by-value semantics defined by the CPS transformation.

Proving completeness requires a close correspondence between the rules and the translated terms, and as often, "administrative redexes" in the CPS transformation get in the way. To prove completeness (i.e., that the evaluation of the CPS form of a term $e$ with an identity continuation leads to a value $v$, then $\Box \vdash e \Rightarrow v$), we successfully adopted Danvy and Filinski's one-pass CPS transformation [6].

These soundness and completeness results are not surprising. One can also obtain them by proving the equivalence of the continuation semantics and a direct semantics, and then by using Plotkin's Simulation theorem [29]. A more immediate connection between continuation semantics and CPS transformation, however, provides the basic framework for adding control operators.

## 2.5 An operational account of shift and reset

A control operator "reifies" a continuation $k$ into a function $f_k$. In terms of the CPS transformation, such a function appears as a $\lambda$-term:

$$\Lambda_k = \lambda w.[\![k]\!]_{Cont}\, w\, \lambda a.a.$$

Correspondingly, in the continuation semantics, we need to find out how to invoke such a function $f_k$ to be able to use a reified continuation.

Let us fix a continuation $k$. For any given value $v$, the corresponding $v'$ such that $k \vdash v \Rightarrow v'$ is unique, if it exists. This suggests us to define a function $f_k$ for every continuation $k$ as follows.

$$f_k\, v = v' \iff k \vdash v \Rightarrow v'$$

And this is justified by the CPS transformation: as a corollary of Theorem 1 when $e = v$, the term $\Lambda_k\,(\llbracket v \rrbracket_{Val})$ is $\beta\eta$-convertible to

$$(\lambda w.\llbracket k \rrbracket_{Cont}\, w\, \lambda a.a)\, \llbracket v \rrbracket_{Val} \; =_{\beta\eta} \; \llbracket v \rrbracket_{Exp}\, \lambda w.\llbracket k \rrbracket_{Cont}\, w\, \lambda a.a \; =_{\beta\eta} \; \llbracket v' \rrbracket_{Val}.$$

Now we are ready to introduce the control operators **shift** and **reset** by adding the following expression forms and the corresponding rules.

$$e ::= ... \mid \textbf{reset}\; e \mid \textbf{shift}\; c.e \mid \textbf{pushcc}(e, k)$$

**Shift** and **reset** are defined by their CPS transformation [5,6]:

$$\llbracket \textbf{shift}\; c.e \rrbracket_{Exp} = \lambda \kappa.(\lambda c.\llbracket e \rrbracket_{Exp}\, \lambda a.a)\, \lambda w.\lambda \kappa'.\kappa'\,(\kappa\, w)$$
$$\text{—composition of continuations}$$

$$\llbracket \textbf{reset}\; e \rrbracket_{Exp} = \lambda \kappa.\kappa\,(\llbracket e \rrbracket_{Exp}\, \lambda a.a)$$
$$\text{—identity continuation}$$

In both **shift** $c.e$ and **reset** $e$, the type of $e$ should be the same—though not necessarily the same as the final result type. Thus any **shift**-expression must be delimited by a corresponding **reset** or a **shift**—a hidden restriction that cannot be easily expressed in this translation. We address it in Section 2.6.

The translation of **shift** and **reset** are not in CPS because they compose continuations. This programming pattern is abstracted by the meta-control operator **pushcc**:

$$\llbracket \textbf{pushcc}(e, k) \rrbracket_{Exp} = \lambda \kappa.\kappa\,(\llbracket e \rrbracket_{Exp}\, \lambda w.\llbracket k \rrbracket_{Cont}\, w\, \lambda a.a)$$

**Pushcc** is a meta-control operator because its expression form contains continuations. It therefore can only be used to define other control operators.

Correspondingly, in the operational semantics, we can express the composition of functions $f_k$ and $f'_k$ related to continuations $k$ and $k'$ by adding the rule pushcc (Table 2). As for **shift** and **reset**, they are defined by the rules shift and reset in term of **pushcc**.

Let us resume the inductive proof of Theorem 1 for the three new rules. We only reproduce the most interesting one here, i.e., pushcc.

*Proof.* (excerpt)

The induction hypotheses for the rule pushcc (Table 2) read:

$$\llbracket e \rrbracket_{Exp}\, \lambda w.\llbracket k' \rrbracket_{Cont}\, w\, \lambda a.a =_{\beta\eta} \llbracket v' \rrbracket_{Val} \qquad\qquad\qquad (\text{i}1)$$
$$\llbracket v' \rrbracket_{Exp}\, \lambda w.\llbracket k \rrbracket_{Cont}\, w\, \lambda a.a =_{\beta\eta} \llbracket k \rrbracket_{Cont}\, \llbracket v' \rrbracket_{Val}\, \lambda a.a =_{\beta\eta} \llbracket v \rrbracket_{Val} \qquad (\text{i}2)$$

$$(\text{shift})\dfrac{\square \ \vdash \ (\lambda c.e)\ \lambda w.\mathbf{pushcc}(w,k)\ \Rightarrow\ v}{k\ \vdash\ \mathbf{shift}\ c.e\ \Rightarrow\ v} \qquad (\text{reset})\dfrac{k\ \vdash\ \mathbf{pushcc}(e,\square)\ \Rightarrow\ v}{k\ \vdash\ \mathbf{reset}\ e\ \Rightarrow\ v}$$

$$(\text{pushcc})\dfrac{k'\ \vdash\ e\ \Rightarrow\ v' \qquad k\ \vdash\ v'\ \Rightarrow\ v}{k\ \vdash\ \mathbf{pushcc}(e,k')\ \Rightarrow\ v}$$

**Table 2.** Definitional rules for **shift** and **reset**

Now, we have $[\![\mathbf{pushcc}(e,k')]\!]_{Exp}\ \lambda w.[\![k]\!]_{Cont}\ w\ \lambda a.a$

$$\begin{aligned}
&=_{\beta\eta}\ (\lambda\kappa.\kappa\,([\![e]\!]_{Exp}\ \lambda w.[\![k']\!]_{Cont}\ w\ \lambda a.a))\ \lambda w.[\![k]\!]_{Cont}\ w\ \lambda a.a\\
&=_{\beta\eta}\ [\![k]\!]_{Cont}\ [\![v']\!]_{Val}\ \lambda a.a,\ \text{by (i1)}\\
&=_{\beta\eta}\ [\![v]\!]_{Val},\ \text{by (i2)}
\end{aligned}$$

Theorem 1, and consequently Corollary 1, still hold for the operational semantics and the CPS transformation extended with the control operators. Therefore, the operational semantics gives the same definition for the control operators as those given by the CPS transformation.

## 2.6  A continuation semantics for shift and reset

With the addition of pushcc, the semantics is no longer branchless. To obtain an equivalent branchless semantics, we need to use the idea of the CPS transformation, i.e., flattening derivation trees by remembering branching computations (continuations $k$ in pushcc) in a stack.

We thus induce another semantics $H$ from the extended semantics, referred to as $L$. For clarity, we subscript rules and domains by the semantics they belong to. A judgment in $H$ is of the form $k\ \vdash_H\ e\ \Rightarrow\ v$, where $e \in Exp_L$ (which can be one of the form introduced by the control operators), but $k$ forms a new domain of "global" continuations: $k \in Cont_H = Cont_L \times (Cont_L\ \text{list}) = Cont_L^+$. The global continuations (of type $Cont_H$) are always non-empty lists of continuations, whose head is the current active continuation, and whose tail is a stack of saved continuations.

The $H$-rules are given in Table 3. Most of them are simply the corresponding $L$-rules with the stack $ks$ carried around unchanged. The interesting rules, pushcc$_H$ and VAL0$_H^{\text{CONS}}$, function as the branching rule pushcc$_L$.

The semantics $H$ is branchless. We would like to show that it correctly accounts for $L$, i.e., $\forall\,k, e, v.\ (k\ \vdash_L\ e\ \Rightarrow\ v) \Longleftrightarrow (k :: \text{nil} \vdash_H\ e\ \Rightarrow\ v)$.

**Theorem 2.** *For all $k \in Cont_L$, $e \in Exp_L$ and $v \in Val_L$, if $k \vdash_L\ e\ \Rightarrow\ v$, then*
$\forall\,ks \in Cont_L^*,\ v' \in Val_L.\ (\square :: ks \vdash_H\ v\ \Rightarrow\ v') \Longrightarrow (k :: ks \vdash_H\ e\ \Rightarrow\ v').$

*Proof.* By rule induction on $k \vdash_L\ e\ \Rightarrow\ v$.

For $ks = \text{nil}$, using rule VAL0$_H^{\text{NIL}}$, we obtain the following corollary.

**Corollary 2.** *For all $k \in Cont_L$, $e \in Exp_L$ and $v \in Val_L$, if $k \vdash_L\ e\ \Rightarrow\ v$ then* $k :: \text{nil} \vdash_H\ e\ \Rightarrow\ v.$

$$(\mathsf{VAL0}_\mathsf{H}^\mathsf{NIL}) \frac{}{\square :: \mathrm{nil} \vdash_H v \Rightarrow v} \qquad\qquad (\mathsf{VAL0}_\mathsf{H}^\mathsf{CONS}) \frac{k :: ks \vdash_H v' \Rightarrow v}{\square :: k :: ks \vdash_H v' \Rightarrow v}$$

$$(\mathsf{VAL1}_\mathsf{H}) \frac{\square :: ks \vdash_H k[v_1] \Rightarrow v}{k :: ks \vdash_H v_1 \Rightarrow v} \;(k \neq \square) \quad (\mathsf{FN}_\mathsf{H}) \frac{k[\square\, e_1] :: ks \vdash_H e_0 \Rightarrow v}{k :: ks \vdash_H e_0\, e_1 \Rightarrow v} \;(e_0 \text{ not a value})$$

$$(\mathsf{ARG}_\mathsf{H}) \frac{k[v_0\, \square] :: ks \vdash_H e_1 \Rightarrow v}{k :: ks \vdash_H v_0\, e_1 \Rightarrow v} \;(e_1 \text{ not a value}) \quad (\mathsf{BETA}_\mathsf{H}) \frac{k :: ks \vdash_H [v_1/x]e \Rightarrow v}{k :: ks \vdash_H (\lambda x.e)\, v_1 \Rightarrow v}$$

$$(\mathsf{shift}_\mathsf{H}) \frac{\square :: ks \vdash_H (\lambda c.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_H \mathbf{shift}\, c.e \Rightarrow v} \quad (\mathsf{reset}_\mathsf{H}) \frac{k :: ks \vdash_H \mathbf{pushcc}(e, \square) \Rightarrow v}{k :: ks \vdash_H \mathbf{reset}\, e \Rightarrow v}$$

$$(\mathsf{pushcc}_\mathsf{H}) \frac{k :: k' :: ks \vdash_H e \Rightarrow v}{k' :: ks \vdash_H \mathbf{pushcc}(e, k) \Rightarrow v}$$

**Table 3.** An operational continuation semantics for **shift** and **reset**

For the inverse direction, we need to refer to the derivation more explicitly: we use $\preceq$ to denote the sub-derivation relation, and we write $D : J$ if $D$ is a derivation ending with the judgment $J$.

**Theorem 3.** *For all* $ks \in Cont_L^*$, $k \in Cont_L$, $e \in Exp_L$, $v' \in Val_L$, *if* $D : k :: ks \vdash_H e \Rightarrow v'$, *then*

$$\exists v \in Val_L, D'. (k \vdash_L e \Rightarrow v) \wedge (D' \preceq D) \wedge D' : (\square :: ks \vdash_H v \Rightarrow v').$$

*Proof.* By strong induction on the derivation $D$.

For $ks = \mathrm{nil}$ in Theorem 3, we notice that the only possible derivation for $D'$ is one-step, using rule $\mathsf{VAL0}_\mathsf{H}^\mathsf{NIL}$, so the witness $v$ is $v'$, and the following corollary holds.

**Corollary 3.** *For all* $k \in Cont_L$, $e \in Exp_L$ *and* $v \in Val_L$, *if* $k :: \mathrm{nil} \vdash_H e \Rightarrow v$ *then* $k \vdash_L e \Rightarrow v$.

Together, Corollary 2 and Corollary 3 show that the branchless semantics $H$ simulates the semantics $L$. $H$ can be implemented by a definitional interpreter as before [5]. Our goal, however, is to implement the control operators natively as functions (using first-class continuations and cells, like Filinski [12]). The semantics of the built-in constructs cannot be altered in such a setting, thereby preventing us to implement the crucial rule $\mathsf{VAL0}_\mathsf{H}^\mathsf{CONS}$, which is enacted when the active continuation (of type $Cont_L$) is already identity. Such behavior should be put into the continuation: instead of initializing it with an identity continuation $\square$, we should initialize it with an operation to resume the top continuation from the stack. The corresponding transformation of $L$ is described in Section 2.7.

Now we can come back to the typing problem of **shift**. The requirement that all **shift**-expressions must be enclosed by a corresponding **reset** or **shift** can be easily manifested in the semantics $H$ by adding a side condition to the rule

$$(\mathsf{VAL0_I}) \frac{}{\square :: ks \vdash_I v \Rightarrow v} \qquad\qquad (\mathsf{VAL1_I}) \frac{\square :: ks \vdash_I k[v_1] \Rightarrow v}{k :: ks \vdash_I v_1 \Rightarrow v} \ (k \neq \square)$$

$$(\mathsf{FN_I}) \frac{k[\square\, e_1] :: ks \vdash_I e_0 \Rightarrow v}{k :: ks \vdash_I e_0\, e_1 \Rightarrow v} \ (e_0 \text{ not a value})$$

$$(\mathsf{ARG_I}) \frac{k[v_0\, \square] :: ks \vdash_I e_1 \Rightarrow v}{k :: ks \vdash_I v_0\, e_1 \Rightarrow v} \ (e_1 \text{ not a value}) \qquad (\mathsf{BETA_I}) \frac{k :: ks \vdash_I [v_1/x]e \Rightarrow v}{k :: ks \vdash_I (\lambda x.e)\, v_1 \Rightarrow v}$$

$$(\mathsf{shift_I}) \frac{id_L^{pop} :: ks \vdash_I (\lambda c.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_I \mathbf{shift}\, c.e \Rightarrow v} \ (ks \neq \text{nil})$$

$$(\mathsf{reset_I}) \frac{k :: ks \vdash_I \mathbf{pushcc}(e, id_L^{pop}) \Rightarrow v}{k :: ks \vdash_I \mathbf{reset}\, e \Rightarrow v} \qquad (\mathsf{pushcc_I}) \frac{k :: k' :: ks \vdash_I e \Rightarrow v}{k' :: ks \vdash_I \mathbf{pushcc}(e, k) \Rightarrow v}$$

$$(\mathsf{popcc_I}) \frac{ks \vdash_I v \Rightarrow v'}{k' :: ks \vdash_I \mathbf{popcc}(v) \Rightarrow v'} \ (ks \neq \text{nil})$$

**Table 4.** An implementable semantics for **shift** and **reset**

$\mathsf{shift_H}$ that the stack should not be empty. (This makes the definition of **shift** partial; when the stack is empty, we obtain a run-time error.)

$$(\mathsf{shift_H}) \frac{\square :: ks \vdash_H (\lambda k'.e)\, \lambda w.\mathbf{pushcc}(w, k) \Rightarrow v}{k :: ks \vdash_H \mathbf{shift}\, k'.e \Rightarrow v} \ (ks \neq \text{nil})$$

## 2.7   An implementable continuation semantics for shift and reset

The implementable semantics $I$ is also induced from the semantics $L$ with $Cont_I = Cont_H = Cont_L^+$. We introduce a new operator **popcc**:

$$e ::= ... \mid \mathbf{popcc}(e)$$

Intuitively, this operator pops the continuation stack and sends its operand to the popped continuation. Now, the initial continuation can be defined as $id_L^{pop} \stackrel{\text{def}}{=} \mathbf{popcc}(\square)$, which replaces $\square$ in rules $\mathsf{shift_H}$ and $\mathsf{reset_H}$, thus eliminating the need for the rule $\mathsf{VAL0_H^{CONS}}$.

In Table 4, the $I$-rules are the same as the $H$-rules except for $\mathsf{VAL0_I}$ (replacing $\mathsf{VAL0_H^{NIL}}$ and $\mathsf{VAL0_H^{CONS}}$), $\mathsf{shift_I}$, $\mathsf{reset_I}$, and $\mathsf{popcc_I}$.

Semantics $I$ simulates semantics $L$, as shown by the following theorem.

**Theorem 4.** *For all $e \in Exp_L$ and $v \in Val_L$, $(\square :: \text{nil} \ \vdash_I \ e \ \Rightarrow \ v) \Longleftrightarrow (\square :: \text{nil} \vdash_H e \Rightarrow v)$.*

*Proof.* By two straightforward inductions.

This new semantics $I$ has two properties:

(1) It is branchless. More specifically, for any intermediate judgment $k \ \vdash_I e \Rightarrow v$ in a proof tree, the rest of the computation after $e$ is evaluated is totally captured in the global continuation $k$. We can thus iterate the above process to add control operators at subsequent levels.

(2) It can be directly implemented in the starting semantics $S$ using references and first-class continuations in the following way:[2] the head of the global continuation $k$ is the current continuation, while the tail is stored in a reference cell. All the $S$-rules automatically 'extend' to the corresponding $I$-rules, without touching the reference cell; the new rules defining the control operators can then be directly implemented by encoding the four constants **shift**, **reset**, **pushcc** and **popcc** as functions, using **callcc** to capture the current continuation and **throw** to restore it.

## 2.8     An inductive construction of the CPS hierarchy

The semantic transformation (from Section 2.5 to Section 2.7) can be generalized and iterated: at each step, we transform an input semantics $S_i$ into an output semantics $S_{i+1}$ that preserves certain *inductive conditions* (such as "branchlessness"). The operational continuation semantics displayed in Table 1 satisfies these inductive conditions, and we used it as the starting semantics $S_1$.
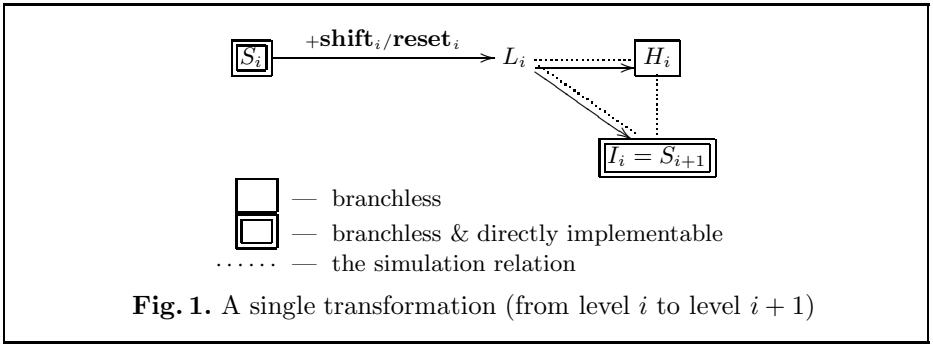


**Fig. 1.** A single transformation (from level $i$ to level $i+1$)

Figure 1 summarizes the development of a single transformation: we start with the branchless semantics $S_i$ of level $i$. Adding **shift**$_i$ and **reset**$_i$ with related inference rules yields the semantics $L_i$ which is no longer branchless (see Section 2.5). Then we replace the global continuation of this level by a stack of such continuations, which forms the global continuation of the next level, and we obtain a semantics $H_i$ where we restore the branchlessness property (see Section 2.6). Since $H_i$ is not directly implementable in $S_i$, we apply another transformation to $L_i$ to obtain a semantics $I_i$, which is both branchless and directly implementable (see Section 2.7). Semantics $I_i$ simulates $H_i$, which in turn simulates $L_i$. With its newly introduced control operators **shift**$_i$ and **reset**$_i$, $I_i$ satisfies the inductive conditions. Therefore, we can use it as the semantics $S_{i+1}$ for the next level of the hierarchy.

---

[2] We did not put references and **callcc** in the starting semantics and we only use them for the implementation. In fact, making references available to the user causes no problem, whereas **callcc** interferes with **shift** and **reset**.

*For space reasons, the rest of this section is omitted. But it is available in the extended version of this article [7].*

## 3   An Implementation of the CPS Hierarchy

We implement the CPS hierarchy by transcribing the transformation of Section 2.8 in Standard ML of New Jersey [1], using the structure `SMLofNJ.Cont` which provides `callcc` and `throw`. The implementation uses a signature `SHIFT_RESET` to specify the operations provided by a semantics $S$ for the user and for the construction of the next control level, a structure `innermost_level` to model the first level in the hierarchy (which thus provides no control operators), and a functor `sr_outer` to construct next-level control operators, parameterized by the answer type `ans` and by the control level `inner` immediately preceding it. Essentially, the signature `SHIFT_RESET` corresponds to the inductive conditions for the semantics $S$, and the functor `sr_outer` corresponds to the transformation from a semantics $S = S_i$ to the next-level semantics $I = S_{i+1}$.

The implementations of control operators are thus hidden inside the module system, and they are accessed via the name of the structure that corresponds to their level. Having devised an ordering of the control effects, a user then implements it through the order of functor applications. Hierarchical occurrences of **shift** and **reset** are thus no longer referred to by their relative index, which had been criticized in the literature [16,27].

We implemented the functor `sr_outer` by transcribing line-by-line the added semantic rules in semantics $I$ (four new functions, one per operator) and the definition of the constant $id_L^{pop}$.[3] We also use two auxiliary functions: a function `replace_gcont`, used implicitly in the semantics, captures and replaces the current global continuation, and a function `cont2gcont`, required by the inductive conditions for semantics $S$, converts a first-class continuation to a global continuation. The code is thus very concise: the pretty-printed program defining `innermost_level` and `sr_outer` takes about 40 lines of ML code (Figure 3).

We also provide a functor for the usual first level of control operators (**shift**$_1$ and **reset**$_1$):

```
functor initial_control_level (type ans) : SHIFT_RESET
= sr_outer (type ans = ans   structure inner = innermost_level)
```

Specializing this functor for the first level of the CPS hierarchy yields a result similar to Filinski's implementation of **shift** and **reset** [12]. The main difference is that here we use an explicit stack of continuations whereas Filinski uses an implicit one through functional abstraction. (An analogy: one can represent environments in an interpreter as a list or as a function.)

---

[3] We use the function `SMLofNJ.Cont.isolate` to coerce a non-returning function to a continuation. This function can be defined as follows.

```
fun isolate f = callcc (fn x => f (callcc (fn y => throw x y)))
```

```
signature SHIFT_RESET                                    (* control level i *)
= sig
    type answer                                     (* answer type of level i *)
    val reset : (unit -> answer) -> answer
    val shift : (('a -> answer) -> answer) -> 'a
    type 'a gcont                                     (*  Cont_S^i (=  Cont_L^i) *)
    val replace_gcont : 'a gcont -> ('b gcont -> 'a) -> 'b
                        (* captures current global continuation (of type 'b gcont), *)
                        (* and replaces it with the first argument (of type 'a gcont) *)
    val cont2gcont : 'a cont -> 'a gcont                 (*  Cont_S^0  ->  Cont_S^k *)
  end

structure innermost_level :> SHIFT_RESET                        (* level 0 *)
= struct                            (* here, global continuation = ML continuation *)
    exception InnermostLevelNoControl
    type answer = unit
    type 'a gcont = 'a cont                          (* uses ML continuation for Cont_S^0 *)

    fun replace_gcont new_c e_thunk
        = callcc (fn old_c => throw new_c (e_thunk old_c))
    fun cont2gcont c = c
    fun reset _ = raise InnermostLevelNoControl
    fun shift _ = raise InnermostLevelNoControl
  end

functor sr_outer (type ans structure inner: SHIFT_RESET) :> SHIFT_RESET
where type answer = ans                              (* from S = S_i to I = S_{i+1} *)
= struct
    exception MissingReset
    exception Fatal
    type answer = ans
    type 'a gcont                                     (* Cont_I = Cont_S^+ *)
        = (answer inner.gcont) list * 'a inner.gcont
    val stack = ref [] : (answer inner.gcont) list ref            (*  ks *)

    fun replace_gcont (new_ks, new_k) e_thunk
                        (* captures and replaces the global continuation, recursively *)
        = inner.replace_gcont new_k
            (fn cur_k => let val cur_gcont = (!stack, cur_k)
                         in stack := new_ks; (e_thunk cur_gcont) end)
    fun cont2gcont action
        = ([], inner.cont2gcont action)
    fun popcc v                                          (* rule popcc_I *)
        = case !stack of                          (* side condition  (ks  ≠ nil) *)
            [] => raise Fatal
          | k'::ks => (stack := ks; inner.replace_gcont k' (fn _ => v))
    val id_popcc = inner.cont2gcont (isolate popcc)              (* id_L^{pop} *)
    fun pushcc k e_thunk                                 (* rule pushcc_I *)
        = inner.replace_gcont k
            (fn k' => (stack := k' :: !stack; e_thunk ()))
    fun reset e_thunk                                    (* rule reset_I *)
        = pushcc id_popcc e_thunk
    fun shift k_abstraction                              (* rule shift_I *)
        = case !stack of                          (* side condition  (ks  ≠ nil) *)
            [] => raise MissingReset
          | _ => inner.replace_gcont id_popcc
                    (fn (k : 'a inner.gcont)
                     => k_abstraction (fn w => pushcc k (fn () => w)))
end
```

Fig. 2.    A native implementation of the CPS hierarchy in Standard ML of New Jersey

# 4    Application: layering monadic effects

As a significant application of composable continuations, Filinski's work on adding user-defined monadic effects to ML-like languages by *monadic reflection* shows that composing continuations is a universal effect, which can be used to simulate all effects expressible using a monad [12,13]. The original work only allowed one monadic effect, but recently, Filinski has extended the technique to allow layering effects by relating a heterogeneous tower of monads to a tower of continuation monads, and then implementing them using a collection of cells to hold the meta-continuations [15].

Independently, we directly adapted Filinski's original one-level implementation with minimal changes to parameterize the functor that generates a *monad representation* by the monad representation layered beneath it, which also gives an inductive implementation of a monadic hierarchy. We essentially put in each structure of a monad representation the corresponding level in the control hierarchy; the functor that generates an outer monad representation is passed the control level of the monad representation at the inner layer, and applies functor `sr_outer` to construct its own control level.

The benefits of this representation of layered monads is the same as in Filinski's work [15]: it is a direct implementation, i.e., no level of interpretation and no level of translation hinder it [26,36].

More detail and several illustrative examples are available in the extended version of this article [7].

# 5    Related Work

## 5.1    Felleisen's seminal work

As already mentioned in Section 1.2, the notion of control delimiters in direct style is due to Felleisen [8]. As already pointed out by Danvy and Filinski [5], control delimiters are significant because they fit in each level of the CPS hierarchy very naturally: they correspond to resetting the current continuation to the identity function; and indeed the control delimiter **reset** is equivalent to Felleisen's. As for abstracting control, programming practice suggested the control operator **shift** which is equivalent to one of the variants of Felleisen's $\mathcal{F}$-operator.

Felleisen's work relies on a notion of control stack, and has inspired a number of similar control operators. Danvy and Filinski's work relies on CPS, and has inspired a number of applications, for two compound reasons we believe:

*Expressiveness:* Programming intuitions run strong in the world of control stacks. But lacking guidelines, how does one know, e.g., whether one has landed on [the continuation equivalent of] Algol 60's control stack or on Lisp's control stack— i.e., on the control equivalent of lexical scope or on dynamic scope (whichever may be best)? And how does one use the result?

Conversely, the world of CPS is a structured one, which offers guidelines and holds much untapped expressive power. For example [12,13], Filinski has

shown that the expressive power of the CPS hierarchy is equivalent to the one of computational monads. In fact, our new examples could equally well be expressed using a tower of monads.

More specifically, operational descriptions of control hierarchies offer the possibilities to shadow control delimiters, to capture them or not when abstracting control, to restore them or not when reinstating abstracted control, and to dynamically search through them at run time. CPS shields us against the most extravagant of these mind-boggling possibilities, since by definition, programs with **shift** and **reset** denote CPS programs. These CPS programs may have many layers of continuations, but they are (1) purely functional and (2) statically typed.

*Efficient implementation:* A stack-based implementation of control tends to exert a cost which is linear in the use of each captured continuation. Besides, and this is a well-known thesis in the continuation community [3], it faces a real problem of duplicated continuations.

Therefore alternative implementations have been sought. For example, Filinski already showed that **shift**$_1$ and **reset**$_1$ can be implemented concisely in terms of **callcc**, which itself can be implemented efficiently [3,12,20]. Through an alternative (but equivalent) formalism, our work essentially generalizes this concise implementation to the whole CPS hierarchy, with no new cost and an equivalent use.

## 5.2   Filinski's work

As a significant application of the CPS hierarchy, Filinski's work on adding user-defined monadic effects to ML-like languages by *monadic reflection* shows that composing continuations is a universal effect, which can be used to simulate all effects expressible using a monad [12,13,15].

## 5.3   Gunter, Rémy, and Riecke's work

Gunter, Rémy, and Riecke present a new set of control operators generalizing exceptions and continuations, and its associated operational semantics and type system [16]. The strength of these operators lies in their static type system—in comparison, and even though we do not doubt that there is one for the CPS hierarchy (cf. Murthy's work [28]), we do not present one here explicitly; instead, we rely on ML's type system in our implementation.

Independently of their type system, Gunter, Rémy, and Riecke's operators are not cast in stone. In their own words, "We do not feel, though, that there is a clear answer to the question of which operational rule is right; suffice it to say that we have picked one, and that the other rules lead to strong type soundness as well." Similarly, we do not contend that **shift** and **reset** are the ultimate control operators—Filinski's operators **kreflect** and **kreify**, for example, could well be preferred [12,13]. But we do believe that the key to their simplicity and expressiveness is the CPS hierarchy.

Gunter, Rémy, and Riecke's operators are also implemented with **callcc**.

## 5.4   Operational semantics

Operational semantics, especially small-step reduction semantics, is often used to specify control operators formally. Several researchers have investigated the type soundness of languages with control operators via syntactic approaches based on operational semantics, such as Wright and Felleisen, and Harper, Duba, and MacQueen for first-class continuations [17,39], Gunter, Rémy, and Riecke for generalizing exceptions and continuations [16], and Murthy for the CPS hierarchy [28]. Here, we use operational semantics to derive our implementation and to prove its correctness. Also, matching the CPS hierarchy, we present a family of continuation semantics instead of one monolithic semantics. This family can be natively programmed in ML without resorting to an informal notion of control stack.

## 5.5   Continuations

After 25 years of existence [32], continuations still remain a challenging topic, to the point that ad-hoc frameworks are routinely preferred. For example, we find it significant that alternative and independent solutions were sought to compile goal-directed evaluation [30] and to abstract delimited control [8,16], even though two levels of continuations provide a simple, natural, and directly implementable solution to both problems. This indicates that continuations require more basic research. We have tried to contribute to this research by characterizing a specific notion of operational continuation semantics and by formalizing its connection to the traditional CPS transformation.

## 6   Conclusion

The CPS transformation is ubiquitous in many areas of computer science, including logic, constructive mathematics, programming languages, and programming. Iterating it yields a concise and expressive framework for delimiting and abstracting control—the CPS hierarchy—which appears substantial and fruitful but has been explored very little so far. In this article, we have contributed to exploring it by (1) characterizing an operational analogue of continuation semantics; (2) developing an analogue of the CPS transformation for such an operational continuation semantics; (3) making it account for the family of control operators **shift** and **reset**; (4) providing a native implementation of the CPS hierarchy in the statically typed language Standard ML; and (5) illustrating the implementation both with classical and with new applications, and in particular with a direct implementation of layered monads.

## Acknowledgments

# References

1. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

2. Robert (Corky) Cartwright, editor. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, Snowbird, Utah, July 1988. ACM Press.

3. William Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1), 1999.

4. Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

5. Olivier Danvy and Andrzej Filinski. Abstracting control. In Wand [38], pages 151–160.

6. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

7. Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy (extended version). Technical Report BRICS RS-98-35, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.

8. Matthias Felleisen. The theory and practice of first-class prompts. In Ferrante and Mager [11], pages 180–190.

9. Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, February 1987.

10. Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, and Bruce F. Duba. Abstract continuations: A mathematical semantics for handling full functional jumps. In Cartwright [2], pages 52–62.

11. Jeanne Ferrante and Peter Mager, editors. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988. ACM Press.

12. Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994. ACM Press.

13. Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Technical Report CMU-CS-96-119.

14. Andrzej Filinski. From normalization-by-evaluation to type-directed partial evaluation. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.

15. Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999. ACM Press. To appear.

16. Carl Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proceedings of the*

*Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, California, June 1995. ACM Press.

17. Robert Harper, Bruce F. Duba, and David MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.

18. Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, SIGPLAN Notices, Vol. 25, No. 3, pages 128–136, Seattle, Washington, March 1990. ACM Press.

19. Robert Hieb, R. Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.

20. Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

21. Gregory F. Johnson. GL – a denotational testbed with continuations and partial continuations as first-class objects. In Mark Scott Johnson, editor, *Proceedings of the ACM SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, SIGPLAN Notices, Vol. 22, No 7, pages 154–176, Saint-Paul, Minnesota, June 1987. ACM Press.

22. Gregory F. Johnson and Dominic Duggan. Stores and partial continuations as first-class objects in a language and its environment. In Ferrante and Mager [11], pages 158–168.

23. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised[5] report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

24. Julia L. Lawall. *Continuation Introduction and Elimination in Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, July 1994.

25. Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

26. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, California, January 1995. ACM Press.

27. Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations, a duumvirate of control operators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag.

28. Chethan R. Murthy. Control operators, hierarchies, and pseudo-classical type systems: A-translation at work. In Olivier Danvy and Carolyn L. Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations*, Technical report STAN-CS-92-1426, Stanford University, pages 49–72, San Francisco, California, June 1992.

29. Gordon D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

30. Todd A. Proebsting. Simple translation of goal-directed evaluation. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 1–6, Las Vegas, Nevada, June 1997. ACM Press.
31. Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184, Orlando, Florida, January 1991. ACM Press.
32. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, December 1993.
33. Dorai Sitaram. Handling control. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 147–155, Albuquerque, New Mexico, June 1993. ACM Press.
34. Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.
35. Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In Wand [38], pages 161–175.
36. Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.
37. Philip Wadler. Monads and composable continuations. *LISP and Symbolic Computation*, 7(1):39–55, January 1994.
38. Mitchell Wand, editor. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990. ACM Press.
39. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.