Abstracting Control *

Olivier Danvy †

Andrzej Filinski

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, USA
andrzej+@cs.cmu.edu

Abstract

The last few years have seen a renewed interest in continuations for expressing advanced control structures in programming languages, and new models such as Abstract Continuations have been proposed to capture these dimensions. This article investigates an alternative formulation, exploiting the latent expressive power of the standard continuation-passing style (CPS) instead of introducing yet other new concepts. We build on a single foundation: abstracting control as a *hierarchy* of continuations, each one modeling a specific language feature as acting on nested *evaluation contexts*.

We show how *iterating* the continuation-passing conversion allows us to specify a wide range of control behavior. For example, two conversions yield an abstraction of Prolog-style backtracking. A number of other constructs can likewise be expressed in this framework; each is defined independently of the others, but all are arranged in a hierarchy making any interactions between them explicit.

This approach preserves all the traditional results about CPS, *e.g.*, its evaluation order independence. Accordingly, our semantics is directly implementable in a call-by-value language such as Scheme or ML. Furthermore, because the control operators denote simple, typable lambda-terms in CPS, they themselves can be statically typed. Contrary to intuition, the iterated CPS transformation does not yield huge results: except where explicitly needed, all continuations beyond the first one disappear due to the extensionality principle (η-reduction).

Besides presenting a new motivation for control operators, this paper also describes an improved conversion into applicative-order CPS. The conversion operates in one pass by performing all administrative reductions at translation time; interestingly, it can be expressed very concisely using the new control operators. The paper also presents some examples of nondeterministic programming in direct style.

*The core of this work was developed at DIKU, the Computer Science department at the University of Copenhagen, Denmark (danvy@diku.dk, andrzej@diku.dk).

†This work has benefited from visits to the Computer Science departments of Stanford University (thanks to Carolyn L. Talcott), Indiana University (thanks to Daniel P. Friedman), and Kansas State University (thanks to David A. Schmidt) during the academic year 1989-1990.

Introduction

Strachey and Wadsworth's continuations were a breakthrough in understanding imperative constructs of programming languages. They gave a clear and unambiguous semantics to a wide class of control operations such as escapes and coroutines. In recent years, however, there has been a growing interest in a class of control operators [Felleisen *et al.* 87] [Felleisen 88] which do not seem to fit into this framework. The point of these new operators is to abstract control with regular procedures that do not escape when they are applied.

This approach encourages seeing not only procedures as the computational counterpart of functions but extending this view to continuations as well. However, the published semantic descriptions, [Felleisen *et al.* 88] do not actually represent continuations as functions but as concatenable sequences of activation frames, losing the inherent simplicity of the original functional formalism. Does this mean that control operators substantially more powerful than jumps are indeed beyond the limit of a traditional continuation semantics?

In the following, we present a denotational "standard semantics" [Milne & Strachey 76], where continuations are represented with functions and control is abstracted with procedures, and where programs have natural, purely functional counterparts. In doing so, we replace the fundamentally dynamic control scoping specified by prior definitions of composable continuations with a properly static approach, akin to the difference between Lisp and Scheme.

The new idea is that a term is evaluated in a collection of embedded contexts, each represented by a continuation. The denotation of a term is expressed in *extended continuation-passing style (ECPS)*. Essentially, this generalizes ordinary continuation-passing style to a hierarchy of continuations, one for each context. Very importantly, however, it inherits the characteristic, syntactically restricted form of a λ-calculus without nested function applications. As such, it still yields semantic specifications where the evaluation order of the defined language is independent of the evaluation order of the defining one [Reynolds 72].

Of course, extended continuation-passing style is in general more verbose than plain continuation-passing style. This suggests introducing new control operators to retain the ability of expressing programs in direct style, mirroring the rationale for including `call-with-current-continuation` in Scheme [Rees & Clinger 86] [Miller 87, appendix A]. We will show how such control operators can in fact be systematically added to an applicative order λ-calculus.

Sections 1 and 2 investigate extended continuation-passing style and how to convert direct terms in the basic case of one delimited context. Section 3 illustrates nondeterministic programming with control abstractions. Section 4 describes control operators over several embedding contexts and their translation to extended continuation-passing style. After a comparison with related work, our approach is put into perspective.

1 Extended Continuation-Passing Style

This section extends continuation-passing style (CPS) to one *delimited* context. We present it using a conceptually simple, but somewhat naive CPS conversion algorithm; the next section addresses more efficient translations. We will use a λ-calculus-like abstract syntax for conciseness; the extensions to a full Scheme-like language should be immediate.

The CPS translation is a source-to-source transformation, which means that new syntactic terms are built as part of the translation. To emphasize this, we will use the quotes $\ulcorner\ \urcorner$ to denote the construction of new syntactic terms. The construction may be parameterized by λ-calculus terms building syntactic subterms; we enclose these subterms between braces $\{\ \}$. The value of this distinction will become apparent in the next section. We can note that $\ulcorner\ \urcorner$ and $\{\ \}$ correspond to the `quasiquote` and `unquote` constructs in Scheme.

As usual, names of new bound variables introduced by the translation are assumed not to collide with existing ones. The conversion $\overline{[\![\,]\!]}$ to ordinary (call-by-value) CPS is given by the following well-known equations:

$$\begin{aligned}
\overline{[\![x]\!]} &= \ulcorner\lambda\kappa.\kappa\,\{[\![x]\!]\}\urcorner \\
\overline{[\![\pi\,E]\!]} &= \ulcorner\lambda\kappa.\{\overline{[\![E]\!]}\}\,(\lambda a.\kappa\,(\{[\![\pi]\!]\}\,a))\urcorner \\
\overline{[\![E_1\ E_2]\!]} &= \ulcorner\lambda\kappa.\{\overline{[\![E_1]\!]}\}\,(\lambda f.\{\overline{[\![E_2]\!]}\}\,(\lambda a.f\,a\,\kappa))\urcorner \\
\overline{[\![E_1 \rightarrow E_2 \parallel E_3]\!]} &= \ulcorner\lambda\kappa.\{\overline{[\![E_1]\!]}\}\,(\lambda b.b \rightarrow \{\overline{[\![E_2]\!]}\}\,\kappa \parallel \{\overline{[\![E_3]\!]}\}\,\kappa)\urcorner \\
\overline{[\![\lambda x.E]\!]} &= \ulcorner\lambda\kappa.\kappa\,(\lambda\{[\![x]\!]\}.\{\overline{[\![E]\!]}\})\urcorner
\end{aligned}$$

To simplify the equations, we treat primitive functions as operators; when a primitive π is passed as an argument, it can be written as $\lambda x.\pi\,x$.

Not every λ-calculus term is obtainable as a result of the CPS conversion. Some of the "unused" terms correspond to control operators in the source language. For example, the operator **escape** (equivalent to Scheme's `call/cc`) can be defined by the equation:

$$\overline{[\![\varepsilon k.E]\!]} = \ulcorner\lambda\kappa.\{\overline{[\![E]\!]}\}\kappa\urcorner[[\![k]\!] \leftarrow \ulcorner\lambda a\kappa'.\kappa\,a\urcorner]$$

where $E_1[x \leftarrow E_2]$ denotes textual substitution of E_2 for free occurrences of x in E_1. Yet even with escaping constructs, the result of the translation is in "ordinary" CPS form, *i.e.*, with no nested function applications. This suggests that there is still a considerable amount of untapped expressive power in the CPS formalism, reflecting control structures whose translations are more general λ-terms. In particular, we can define the two operators **shift** and **reset**, conceptually serving as composition and identity for continuation functions:

$$\begin{aligned}
\overline{[\![\xi k.E]\!]} &= \ulcorner\lambda\kappa.\{\overline{[\![E]\!]}\}\,(\lambda x.x)\urcorner[[\![k]\!] \leftarrow \ulcorner\lambda a\kappa'.\kappa'\,(\kappa\,a)\urcorner] \\
\overline{[\![\langle E\rangle]\!]} &= \ulcorner\lambda\kappa.\kappa\,(\{\overline{[\![E]\!]}\}\,(\lambda x.x))\urcorner
\end{aligned}$$

Shift abstracts the current context as an ordinary, composable procedure and **reset** delimits the scope of such a context. **Shift** also differs from **escape** by not duplicating the current continuation. For example, we have

$$1 + \langle 10 + \xi c.\,c\,(c\,100)\rangle \;\Rightarrow\; 1 + (10 + (10 + 100)) \;\Rightarrow\; 121$$

While the effects of these operators are very similar to operators **control** and **prompt** of [Felleisen 88], there is a significant semantical difference between **shift/reset** and **control/prompt**: the context abstracted by **shift** is determined *statically* by the program text, while **control** captures the context up to the nearest *dynamically* enclosing **prompt**. In general, this leads to different behavior.

We say that the translation results above are expressed in *continuation-composing style*. Such definitions lose the important quality of enforcing strict call-by-value evaluation. However, we can restore that property by converting the defining (pure λ-calculus) terms *once more* into CPS. Generalizing from the transformation equations to a full language definition, we can treat a semantics written in continuation-composing style as a *direct semantics* and obtain a strategy-independent "meta-continuation semantics" from it by the standard CPS conversion. Thus, if we include the defining equations for **shift** and **reset**,

$$\begin{aligned}
\mathcal{E}[\![\xi k.E]\!]\rho\kappa &= \mathcal{E}[\![E]\!]\,\rho[[\![k]\!] \mapsto \lambda v\kappa'.\kappa'(\kappa v)]\,(\lambda x.x) \\
\mathcal{E}[\![\langle E\rangle]\!]\rho\kappa &= \kappa\,(\mathcal{E}[\![E]\!]\rho\,(\lambda x.x))
\end{aligned}$$

in a standard call-by-value continuation semantics and transform the entire result into CPS, we obtain the following language definition in ECPS (omitting straightforward notation for including procedures in the domain of values):

$$\begin{aligned}
\mathcal{E} &: \mathtt{Exp} \rightarrow \mathit{Env} \rightarrow \mathit{Cont} \rightarrow \mathit{MCont} \rightarrow \mathit{Ans} \\
\mathit{Proc} &= \mathit{Val} \rightarrow \mathit{Cont} \rightarrow \mathit{MCont} \rightarrow \mathit{Ans} \\
\kappa \in \mathit{Cont} &= \mathit{Val} \rightarrow \mathit{MCont} \rightarrow \mathit{Ans} \\
\gamma \in \mathit{MCont} &= \mathit{Val} \rightarrow \mathit{Ans}
\end{aligned}$$

$$\begin{aligned}
\mathcal{E}[\![x]\!]\rho\kappa\gamma &= \kappa\,(\rho[\![x]\!])\,\gamma \\
\mathcal{E}[\![\pi\,E]\!]\rho\kappa\gamma &= \mathcal{E}[\![E]\!]\rho\,(\lambda v\gamma'.\kappa\,(\pi\,v)\,\gamma')\,\gamma \\
\mathcal{E}[\![E_1\ E_2]\!]\rho\kappa\gamma &= \mathcal{E}[\![E_1]\!]\rho\,(\lambda f\gamma'.\mathcal{E}[\![E_2]\!]\rho(\lambda a\gamma''.fa\kappa\gamma'')\gamma')\,\gamma \\
\mathcal{E}[\![E_1 \rightarrow E_2 \parallel E_3]\!]\rho\kappa\gamma & \\
&= \mathcal{E}[\![E_1]\!]\rho\,(\lambda b\gamma'.b \rightarrow \mathcal{E}[\![E_2]\!]\rho\kappa\gamma' \parallel \mathcal{E}[\![E_3]\!]\rho\kappa\gamma')\,\gamma \\
\mathcal{E}[\![\lambda x.E]\!]\rho\kappa\gamma &= \kappa\,(\lambda v\kappa'\gamma'.\mathcal{E}[\![E]\!]\,\rho[[\![x]\!] \mapsto v]\,\kappa'\gamma')\,\gamma \\
\mathcal{E}[\![\varepsilon k.E]\!]\rho\kappa\gamma &= \mathcal{E}[\![E]\!]\,\rho[[\![k]\!] \mapsto \lambda v\kappa'\gamma'.\kappa v\gamma']\,\kappa\gamma \\
\mathcal{E}[\![\xi k.E]\!]\rho\kappa\gamma &= \\
&\quad \mathcal{E}[\![E]\!]\,\rho[[\![k]\!] \mapsto \lambda v\kappa'\gamma'.\kappa v(\lambda w.\kappa' w\gamma')]\,(\lambda x\gamma''.\gamma'' x)\gamma \\
\mathcal{E}[\![\langle E\rangle]\!]\rho\kappa\gamma &= \mathcal{E}[\![E]\!]\rho\,(\lambda x\gamma'.\gamma' x)\,(\lambda v.\kappa v\gamma)
\end{aligned}$$

It is instructive to note how the composition $\kappa'\,(\kappa\,v)$ in the "direct semantics" for **shift** is sequentialized into the usual CPS style $\lambda\gamma.\kappa v(\lambda w.\kappa'\,w\,\gamma)$. Similarly, in **reset**, $\kappa\,(\ldots(\lambda x.x))$ becomes $\lambda\gamma.\ldots(\lambda x\gamma'.\gamma'\,x)\,(\lambda v.\kappa v\gamma)$.

These equations look somewhat frightening because of all the γ's. We note, however, that in all equations, except for **shift/reset**, the meta-continuations can be elided because of the extensionality principle, *i.e.*, $\lambda\gamma.\varphi\,\gamma$ is equivalent to simply φ. Thus, the entire semantics need not be cluttered up to describe composable continuations. Also, the static nature of the translation into CPS induces a naturally static type system for the language. This aspect is treated in more depth in [Danvy & Filinski 89].

Let us finally point out the congruence relation [Sethi & Tang 80] between the meta-continuation semantics and an ordinary continuation semantics:

$$\mathcal{E}_{mc}[\![E]\!]\rho_{mc}\kappa\gamma = \gamma\ (\mathcal{E}_c[\![E]\!]\rho_c\kappa)$$

completely analogous to the traditional congruence between continuation and direct semantics:

$$\mathcal{E}_c[\![E]\!]\rho_c\kappa = \kappa\ (\mathcal{E}_d[\![E]\!]\rho_d)$$

where ρ_d and ρ_c are related in the usual way [Stoy 81]. In particular, results concerning recursion and the problem of nontermination can be carried over directly.

2 Metacircular Interpreters and Compilers

The traditional CPS translation equations of the last section, while simple, tend to produce unnecessarily large results. The problem is that the constructed terms contain many β- and η-redexes, which must usually be post-reduced in a separate pass [Steele 78]. However, it is possible to avoid building them at all, by performing the reduction directly in conjunction with the translation. The key is to represent continuations in the converted terms as *semantic* functions operating on pieces of abstract syntax, rather than as syntactic terms. This seems to be a new approach to practical CPS conversion, and one which can be expressed very concisely by using exactly the new operators we are defining!

The goal of the efficient translation is to build from a syntactic term E a term with the same meaning as $\overline{[\![E]\!]}$ but without the residual redexes of the latter. We express this using a new conversion function $\overline{[\![\ ]\!]}$ that defers building new syntactic terms until their context of use is known:

$$\begin{aligned}
\overline{[\![x]\!]} &= \lambda\kappa.\kappa\,[\![x]\!] \\
\overline{[\![\pi\ E]\!]} &= \lambda\kappa.\overline{[\![E]\!]}\,(\lambda a.\kappa\,\ulcorner\{[\![\pi]\!]\}\,\{a\}\urcorner) \\
\overline{[\![E_1\ E_2]\!]} &= \lambda\kappa.\overline{[\![E_1]\!]}\,(\lambda f.\overline{[\![E_2]\!]}\,(\lambda a.\ulcorner\{f\}\,\{a\}\,(\lambda t.\{\kappa\,\ulcorner t\urcorner\})\urcorner)) \\
\overline{[\![E_1 \rightarrow E_2 \,[\!]\, E_3]\!]} & \\
&= \lambda\kappa.\overline{[\![E_1]\!]}\,(\lambda b.\ulcorner\{b\} \rightarrow \{\overline{[\![E_2]\!]}\,\kappa\}\,[\!]\,\{\overline{[\![E_3]\!]}\,\kappa\}\urcorner) \\
\overline{[\![\lambda x.E]\!]} &= \lambda\kappa.\kappa\,\ulcorner\lambda\{[\![x]\!]\}k.\{\overline{[\![E]\!]}\,(\lambda a.\ulcorner k\,\{a\}\urcorner)\}\urcorner \\
\overline{[\![\varepsilon k.E]\!]} &= \lambda\kappa.(\overline{[\![E]\!]}\,\kappa)[[\![k]\!] \leftarrow \ulcorner\lambda a k'.\{\kappa\,\ulcorner a\urcorner\}\urcorner] \\
\overline{[\![\xi k.E]\!]} &= \lambda\kappa.(\overline{[\![E]\!]}\,(\lambda x.x))[[\![k]\!] \leftarrow \ulcorner\lambda a k'.k'\,\{\kappa\,\ulcorner a\urcorner\}\urcorner] \\
\overline{[\![\langle E\rangle]\!]} &= \lambda\kappa.\kappa\,(\overline{[\![E]\!]}\,(\lambda x.x))
\end{aligned}$$

The result of converting a term E in an empty context is then given by $\overline{[\![E]\!]}\,(\lambda x.x)$. Note how most of the λ's appearing in these equations are semantical, *i.e.*, not directly within $\ulcorner\urcorner$ and thus will not appear explicitly in the converted term. This is the motivation for having the continuation κ available as a *functional object* that can be applied directly to any syntactic term, rather than a *syntactic λ-abstraction* leaving many administrative redexes to post-reduce.

In fact we can do even better: in the equation for function application, the syntactic λ-abstraction can often be η-reduced away, though this is awkward to express in equational style. Such a reduction is possible when the semantic continuation κ is a function of the form $\lambda a.\ulcorner E\,\{a\}\urcorner$ where a is not used to build E. For example, this is the case for a tail call in a function abstraction, such as the call to `f` in `(lambda (x) (f (g x)))`.

Let us now observe that the equations for $\overline{[\![\ ]\!]}$ are written in a CPS-like style, but with occasional nested function applications. This situation is precisely what **shift/reset** are intended for. We can thus fold the equations back into direct style, using the new operators:

$$\begin{aligned}
\overline{[\![x]\!]} &= [\![x]\!] \\
\overline{[\![\pi\ E]\!]} &= \ulcorner\{[\![\pi]\!]\}\,\{\overline{[\![E]\!]}\}\urcorner \\
\overline{[\![E_1\ E_2]\!]} &= \xi\kappa.\ulcorner\{\overline{[\![E_1]\!]}\}\,\{\overline{[\![E_2]\!]}\}\,(\lambda t.\{\kappa\,\ulcorner t\urcorner\})\urcorner \\
\overline{[\![E_1 \rightarrow E_2 \,[\!]\, E_3]\!]} & \\
&= \xi\kappa.\ulcorner\{\overline{[\![E_1]\!]}\} \rightarrow \{\langle\kappa\,\overline{[\![E_2]\!]}\rangle\}\,[\!]\,\{\langle\kappa\,\overline{[\![E_3]\!]}\rangle\}\urcorner \\
\overline{[\![\lambda x.E]\!]} &= \ulcorner\lambda\{[\![x]\!]\}k.\{\langle\ulcorner k\,\{\overline{[\![E]\!]}\}\urcorner\rangle\}\urcorner \\
\overline{[\![\varepsilon k.E]\!]} &= \xi\kappa.\langle\kappa\,\overline{[\![E]\!]}\rangle[[\![k]\!] \leftarrow \ulcorner\lambda a k'.\{\kappa\,\ulcorner a\urcorner\}\urcorner] \\
\overline{[\![\xi k.E]\!]} &= \xi\kappa.\langle\overline{[\![E]\!]}\rangle[[\![k]\!] \leftarrow \ulcorner\lambda a k'.k'\,\{\kappa\,\ulcorner a\urcorner\}\urcorner] \\
\overline{[\![\langle E\rangle]\!]} &= \langle\overline{[\![E]\!]}\rangle
\end{aligned}$$

This set of equations can be seen as a meta-circular compiler from a language with the new control operators into its purely functional subset. Alternatively, it translates terms of a Scheme-like language (*i.e.*, λ-calculus + **escape**) into standard CPS. Such a conversion has a practical interest for compiling, from [Steele 78] to [Appel & Jim 89], and thus constitutes a significant example of using **shift/reset**: even the pure CPS translation is expressed naturally using the new control operators.

As with all meta-circular definitions, we need to bootstrap it. If we have an interpreter for a language with **shift/reset**, we can use it to execute the translator on itself, obtaining a CPS converter written in pure λ-calculus. On the other hand, we can get an interpretive semantics for the extended language by translating a trivial (*i.e.*, defining **shift** in terms of **shift**, *etc.*) self-interpreter into ECPS. This ensures an automatic consistency between the two methods of language definition.

3 An Application: Nondeterministic Programming

It is well-known that continuation passing-style can be used to simulate backtracking in, *e.g.*, Prolog programs [Mellish & Hardy 84]. In this "downward success" model of nondeterministic execution, alternatives at choice points are considered in sequence. However, such an approach requires the entire program to be expressed with explicit success continuations, complicating its structure considerably. Moreover, in this case simple escaping operators such as `call-with-current-continuation` are not powerful enough to avoid an actual translation.

With `shift/reset` in the language, on the other hand, all the control aspects of backtracking can be abstracted into just two primitives, permitting the rest of the program to be written in a natural, direct style. We express the nondeterministic choice as repeated invocations of the success continuation with each of the possible choices; dead ends or failures correspond to discarding the current branch of control:

```
(define flip
  (lambda () (shift c (begin (c #t) (c #f) (fail)))))

(define fail
  (lambda () (shift c "no")))
```

As an example, let us consider the implementation of a nondeterministic finite automaton for recognizing languages denoted by regular expressions [Aho, Hopcroft, & Ullman 74]. While conceptually simple, this application demonstrates the essence of choice and failure in a traditional setting. We will represent the sequence of input symbols as a list. If the automaton recognizes a prefix of this sequence, it returns the remainder; otherwise, it fails. The regular expressions $r_1\,r_2$, $r_1|r_2$ and r^* are encoded as `(& r1 r2)`, `(/ r1 r2)` and `(* r)`, respectively. We see that accepting an input string is indeed reduced to verifying a sequence of guesses generated by the oracle `flip`:

```
(define ndfa
  (lambda (r l)
    (if (atom? r)
        (if (and (not (null? l)) (equal? (car l) r))
            (cdr l)
            (fail))
        (case (car r)
          [(&) (let ([l1 (ndfa (cadr r) l)])
                 (ndfa (caddr r) l1))]
          [(/) (if (flip)
                   (ndfa (cadr r) l)
                   (ndfa (caddr r) l))]
          [(*) (if (flip)
                   l
                   (let ([l1 (ndfa (cadr r) l)])
                     (ndfa r l1)))])))))

(define accept
  (lambda (r l)
    (let ([l1 (ndfa r l)])
      (if (null? l1) "accepted" (fail)))))
```

The function `accept` will return to the point of call whenever the input `l` is in the language denoted by the expression `r`. This algorithm corresponds exactly to the following continuation-composing style procedure (where we have unfolded `flip` and `fail` and uncurried the translation, for clarity):

```
(define ndfa-c
  (lambda (r l k)
    (if (atom? r)
        (if (and (not (null? l)) (equal? (car l) r))
            (k (cdr l))
            "no")
        (case (car r)
          [(&) (ndfa-c (cadr r) l
                 (lambda (l1) (ndfa-c (caddr r) l1 k)))]
          [(/) (begin (ndfa-c (cadr r) l k)
                      (ndfa-c (caddr r) l k)
                      "no")]
          [(*) (begin (k l)
                      (ndfa-c (cadr r) l
                        (lambda (l1) (ndfa-c r l1 k)))
                      "no")])))))

(define accept-c
  (lambda (r l k)
    (ndfa-c r l
      (lambda (l1) (if (null? l1) (k "accepted") "no")))))
```

Here, the argument `k` of `accept-c` will be invoked whenever the string is accepted.

Let us note again that since we have expressed the backtracking primitives using the existing operator `shift`, the ECPS semantic formalism immediately gives a proper denotational description of this facility. In fact, as noted in section 1, η-reduction permits us to write virtually all of the semantics in ordinary CPS style, involving the metacontinuation only for expressing the denotations of `flip` and `fail`.

As a more complex example, we may consider the problem presented in [Abelson & Sussman 85, pp 254–255]: generating all triples of distinct positive integers i, j and k less than or equal to a given integer n that sum to a given integer s. The solution given there amounts to filtering admissible triples from a stream of possible triples, built out of three streams of numbers. Let us see how to solve it using our nondeterministic operators instead.

We first define a procedure `choice`, returning an "appropriate" integer between 1 and n. Then we can generate the triples naively in a very straightforward way:

```
(define choice
  (lambda (n)
    (if (< n 1)
        (fail)
        (if (flip) (choice (- n 1)) n))))

(define triple
  (lambda (n s)
    (let* ([i (choice n)]
           [j (choice (- i 1))]
           [k (choice (- j 1))])
      (if (= (+ i j k) s)
          (list i j k)
          (fail)))))
```

Now `(reset (display (triple 9 15)))` prints the triples as they are generated. Sometimes, however, we want to collect all the results in a list (*cf.* the `bagof` predicate found in many Prolog systems). A possible way to obtain this would be to update a list of solutions imperatively. However, a purely functional alternative exists as well, this time using `shift` to define a procedure `emit`:

```
(define emit
  (lambda (n) (shift c (cons n (c '())))))
```

with the property that every time it is applied to an argument, it adds that argument to a list of "answers". For example, `(reset (begin (emit 1) (emit 2) (emit 3) '()))` yields `(1 2 3)`. The `'()` for ending the list is actually redundant in this case, since the return value of `(emit 3)` could be exploited instead.

This definition can also be converted naturally into ECPS. However, if we try to combine `emit` with `flip/fail`, the collection and generation interfere. We want to specify that once a solution has been generated, subsequent backtracking should not attempt to retract it. A direct solution to this problem would be to first convert the generator (*i.e.*, `triple`) into continuation-composing style, and then invoking it with `emit` as a continuation, so that only the collection process would manipulate contexts directly. However, this staging requires us to deal with CPS-converted programs explicitly, losing the benefit of having a direct style generator. What we need instead is a way to express that the CPS-conversion of `emit` should itself contain a `shift`, *i.e.*, we want a definition of `emit` that would translate into

```
(define emit-c
  (lambda (n k)
    (k (shift c (cons n (c '()))))))
```

Writing `(collect E)` as syntactic sugar for

```
(begin (reset (emit E)) '())
```

the translation of `(collect (triple 9 15))` into CPS becomes:

```
(begin (triple-c 9 15
         (lambda (t) (emit-c t (lambda (x) x))))
       '())
```

which is just what we want. This is the subject of section 4. Using the operator $\texttt{shift}_2$ defined there, instead of `shift` in `emit`, we get exactly the desired behavior, *i.e.*, separating generation and collection into different levels.

Similarly, we can use $\texttt{shift}_2$ to produce a single Boolean result from the nondeterministic automaton, independently of how many ways there are to "parse" the input as a regular expression. Again, conceptually we want to express the following:

```
(define decide
  (lambda (r l)
    (escape k (begin (accept-c r l (lambda (a) (k #t)))
                     #f))))
```

This corresponds to a "joining of nondeterministic paths": only if all execution threads fail to accept do we return `#f`.

4 Generalizing to More Contexts

Many control operators can be defined with respect to extended continuation-passing style. This section investigates the generalization of **shift** to give control over an arbitrary number of embedding contexts, abstracting them as a single procedure. Correspondingly, **reset** is generalized to reset arbitrarily many embedding contexts. This extends the meta-continuation semantics of section 1.

Nested contexts may be used in two settings. In a program they can be used to order meta-level actions such as backtracking, accumulating results, and so on, as in section 3. When designing a language, they can be used to order dynamic features such as an environment for fluid or logical variables, exceptions, a store, i/o streams, and so on.

4.1 Notation

In this section, we are considering the types of terms that have been repeatedly converted into CPS, say m times. We manifest this by subscripting the domain of answers with the number of conversions:

$$Ans_i = Cont_{i+1} \to Ans_{i+1}$$

where $0 \le i < m$ and $Ans_m = Ans$

4.2 A family of semantics

Let us consider a family of continuation semantics indexed with a number of contexts. Each valuation function $\mathcal{E}_m$ is passed an expression, an environment, m continuations, and returns an answer. Each procedure is passed a value, m continuations, and returns an answer. Each continuation is passed a value, the rest of the continuations, and returns an answer. This is captured in the following. (Figure 1 displays the full family of continuation semantics with the answer domain type expanded.)

$$\begin{aligned} \mathcal{E}_m &: \texttt{Exp} \to Env \to Ans_0 \\ Proc &= Val \to Ans_0 \\ Cont_i &= Val \to Ans_i \end{aligned}$$

When $m = 1$, this semantics defines Scheme. In particular, `call-with-current-continuation` abstracts the mth continuation.

When $m = 2$, this semantics coincides with the meta-continuation semantics of section 1.

Each $m+1$-semantics is a traditional continuation semantics of the m-semantics (*cf.* section 4.3 for their congruence), and thus the relation between $Cont_i$ and $Cont_{i+1}$ also holds between $Proc$ and $Cont_1$ — actually, θ_0 as defined in figure 1 is the denotation of the identity procedure.

Alternatively the domain equations could be set up to define procedures as continuation transformers, and correspondingly each $Cont_i$ would be a $Cont_{i+1}$ transformer, instead of the rather long types of figure 1. However these more compact types would impede the η-reductions that keep ECPS terms concise.

4.3 A family of congruences

The new congruence of continuation semantics mentioned in section 1 can be generalized to the family of congruence relations displayed in figure 2, starting from the traditional direct semantics $\mathcal{E}_0$. Each semantics $\mathcal{E}_{m+1}$ is the result of the CPS conversion of $\mathcal{E}_m$ considered as a direct semantics.

4.4 A family of control operators

Resetting n contexts (where $0 \le n < m$) is a natural generalization of section 1. We abstract n continuations on the $n+1$st one and install n initial θ_i continuations (*cf.* figures 3 and 4).

Extending **shift** is also a natural generalisation of section 1. We want to shift n contexts (where $0 \le n < m$) into a procedural object. This is captured by evaluating the body of the $\textbf{shift}_n$-expression in an environment binding the $\textbf{shift}_n$-identifier to a procedural abstraction of these contexts, and with n initial θ_i continuations. When invoked, the control abstraction will restore the n continuations and abstract the n current continuations in the $n+1$st one (*cf.* figures 3 and 4).

4.5 Translation to extended continuation-passing style

From the general definitions of **shift** and **reset** above, we immediately obtain the equations for $n = 0$:

$$\begin{aligned} \xi_0 k.E &= E[k \leftarrow \lambda x.x] \\ \langle E \rangle_0 &= E \end{aligned}$$

With these at the base, we can express how every operator is converted into a lower-numbered one by the CPS translation.

$$\begin{aligned} \overline{[\![\xi_{n+1}\, k.E]\!]} &= \ulcorner \lambda\kappa.\xi_n k'.\{\overline{[\![E]\!]}\}\,(\lambda x.x)\urcorner[k \leftarrow \lambda a\kappa'.\kappa'\langle k'(\kappa\, a)\rangle_n] \\ \overline{[\![\langle E \rangle_{n+1}]\!]} &= \ulcorner \lambda\kappa.\kappa\,(\{\overline{[\![E]\!]}\}\,(\lambda x.x)\rangle_n \urcorner \end{aligned}$$

$$
\begin{array}{rcl}
\mathcal{E}_m & : & \mathtt{Exp} \to \mathit{Env} \to \mathit{Cont}_1 \to \ldots \to \mathit{Cont}_{i+1} \to \ldots \to \mathit{Cont}_m \to \mathit{Ans} \\
\theta_0 \in \mathit{Proc} & = & \mathit{Val} \to \mathit{Cont}_1 \to \ldots \to \mathit{Cont}_{i+1} \to \ldots \to \mathit{Cont}_m \to \mathit{Ans} \;=\; \mathit{Cont}_0 \\
\theta_i \in \mathit{Cont}_i & = & \mathit{Val} \to \mathit{Cont}_{i+1} \to \ldots \to \mathit{Cont}_m \to \mathit{Ans}
\end{array}
$$

where $\theta_i = \lambda v \kappa_{i+1} \kappa_{i+2} \ldots \kappa_m . \kappa_{i+1} v \kappa_{i+2} \ldots \kappa_m$, for $0 \leq n < m$, and $\theta_m = \lambda v.v$

Intuitively, each θ_i transmits a result to an outer context, when $1 \leq n < m$

κ_{i+2}, ..., and κ_m can be η-reduced in the definitions of θ_i

Figure 1: A family of m continuation semantics

$$
\begin{array}{c}
\kappa_1^1(\mathcal{E}_0[\![E]\!]\rho^0) = \mathcal{E}_1[\![E]\!]\rho^1\kappa_1^1 \\
\kappa_2^2(\kappa_1^2(\mathcal{E}_0[\![E]\!]\rho^0)) = \kappa_2^2(\mathcal{E}_1[\![E]\!]\rho^1\kappa_1^2) = \mathcal{E}_2[\![E]\!]\rho^2\kappa_1^2\kappa_2^2 \\
\vdots \\
\kappa_{n+1}^{n+1}(\kappa_n^{n+1}(\ldots(\kappa_1^{n+1}(\mathcal{E}_0[\![E]\!]\rho^0))\ldots)) = \ldots = \kappa_{n+1}^{n+1}(\mathcal{E}_n[\![E]\!]\rho^n\kappa_1^{n+1}\ldots\kappa_n^{n+1}) = \mathcal{E}_{n+1}[\![E]\!]\rho^{n+1}\kappa_1^{n+1}\ldots\kappa_n^{n+1}\kappa_{n+1}^{n+1}
\end{array}
$$

Figure 2: A family of congruences

Each environment and continuation is superscripted with the number of contexts. By introducing two new control operators for each context, we obtain a family of congruent semantics of a language $\mathcal{L}_n$ for each n. Each $\mathcal{L}_i$-program has the same meaning in any $\mathcal{L}_j$, for $j > i$. The only changing summand in the value domain are procedures, as accounted for in figure 1.

$$
\begin{array}{rcl}
\mathcal{E}_m[\![\langle E \rangle_n]\!]\rho\kappa_1\ldots\kappa_n\kappa_{n+1}\kappa_{n+2}\ldots\kappa_m & = & \mathcal{E}_m[\![E]\!]\rho\theta_1\ldots\theta_n(\lambda v\kappa'_{n+2}\ldots\kappa'_m.\theta_0 v\kappa_1\ldots\kappa_n\kappa_{n+1}\kappa'_{n+2}\ldots\kappa'_m)\kappa_{n+2}\ldots\kappa_m \\
\mathcal{E}_m[\![\xi_n k.E]\!]\rho\kappa_1\ldots\kappa_n\kappa_{n+1}\kappa_{n+2}\ldots\kappa_m & = & \mathcal{E}_m[\![E]\!]\rho[[\![k]\!] \mapsto p]\theta_1\ldots\theta_n\kappa_{n+1}\kappa_{n+2}\ldots\kappa_m
\end{array}
$$

where $p = \lambda v\kappa'_1\ldots\kappa'_n\kappa'_{n+1}\kappa'_{n+2}\ldots\kappa'_m.\theta_0 v\kappa_1\ldots\kappa_n(\lambda w\kappa''_{n+2}\ldots\kappa''_m.\theta_0 w\kappa'_1\ldots\kappa'_n\kappa'_{n+1}\kappa''_{n+2}\ldots\kappa''_m)\kappa'_{n+2}\ldots\kappa'_m$
and $\theta_i = \lambda v\kappa_{i+1}\ldots\kappa_m.\kappa_{i+1}v\kappa_{i+2}\ldots\kappa_m$ for $1 \leq i < m$

Figure 3: Two families of control operators: $\mathbf{shift}_n$ and $\mathbf{reset}_n$

$$
\begin{array}{rcl}
\mathcal{E}_m[\![\langle E \rangle_n]\!]\rho\kappa_1\ldots\kappa_n\kappa_{n+1} & = & \mathcal{E}_m[\![E]\!]\rho\theta_1\ldots\theta_n(\lambda v.\theta_0 v\kappa_1\ldots\kappa_n\kappa_{n+1}) \\
\mathcal{E}_m[\![\xi_n k.E]\!]\rho\kappa_1\ldots\kappa_n & = & \mathcal{E}_m[\![E]\!]\rho[[\![k]\!] \mapsto p]\theta_1\ldots\theta_n
\end{array}
$$

where $p = \lambda v\kappa'_1\ldots\kappa'_n\kappa'_{n+1}.\theta_0 v\kappa_1\ldots\kappa_n(\lambda w.\theta_0 w\kappa'_1\ldots\kappa'_n\kappa'_{n+1})$
and $\theta_i = \lambda v\kappa_{i+1}.\kappa_{i+1}v$ for $1 \leq i < m$

Figure 4: Definition of $\mathbf{shift}_n$ and $\mathbf{reset}_n$ where all the outer continuations have been η-reduced

Intuitively, in $\mathbf{shift}_{n+1}$ the innermost context is captured as the continuation κ, and the n outer ones as k'; they are then composed into one primitive function bound to k.

4.6 A family of control combinators

There are two common patterns in the definitions of $\mathbf{shift}_n$ and $\mathbf{reset}_n$: a series of contexts are abstracted in a continuation; and a series of θ_i are installed. We can capture this pattern in a family of regular combinators A_n for abstracting and R_n for resetting contexts:

$$A_n : Ans_{n-1} \to Ans_0 \;=\; [Cont_n \to Ans_n] \to Ans_0$$
$$A_n f \kappa_1 \ldots \kappa_m = f(\lambda v.\theta_0 v \kappa_1 \ldots \kappa_n)\kappa_{n+1} \ldots \kappa_m$$

$$R_n : Ans_0 \to Ans_n \to Ans$$
$$R_n f \kappa_{n+1} \ldots \kappa_m = f\theta_1 \ldots \theta_n \kappa_{n+1} \ldots \kappa_m$$
where $\theta_i = \lambda v \kappa_{i+1} \kappa_{i+2} \ldots \kappa_m.\kappa_{i+1} v \kappa_{i+2} \ldots \kappa_m$
and $0 \le i < m$.

Given A_n and R_n we can reexpress the semantics of the families of control operators as follows, η-reducing all the outer continuations:

$$\begin{aligned}\mathcal{E}_m[\![\langle E\rangle_n]\!]\rho &= A_{n+1}\,(R_n\,(\mathcal{E}_m[\![E]\!]\rho))\\ \mathcal{E}_m[\![\xi_n k.E]\!]\rho &= A_n\,(\lambda c.R_n\,(\mathcal{E}_m[\![E]\!]\,\rho[[\![k]\!] \mapsto \lambda v.A_{n+1}\,(c\,v)]))\end{aligned}$$

Because A_n and R_n are regular, this description respects CPS.[1]

In the rest of this section, we describe how to implement A_n and R_n using Church numerals and untyped combinators. This description relies on observing that all the θ_i are extensionally equal to the polymorphic constant $\theta = \lambda v\kappa.\kappa v$. Confusing all the θ_i is type incorrect since it discards type information. However, this description can be transliterated into Scheme as a definitional interpreter.

Inductive combinators can be expressed without ellipses using combinatory logic and Church numerals [Barendregt 85]:

$$\begin{cases} \underline{0} &= KI \\ \underline{n+1} &= SB\underline{n} \end{cases} \quad \text{s.t. } \underline{n}\,M\,N = \underbrace{M(M(\ldots(M}_{n}\ N)\ldots))$$

For example, we can define the family of regular duplicating combinators:

$$W_n = \underline{n}(BW)K \quad \text{such that} \quad W_n M N = M\underbrace{N \ldots N}_{n}$$

R_n applies its one argument to n instances of θ. It may be defined as:

$$R_n E = W_n E \theta$$

where again $\theta = \lambda v\kappa.\kappa v = CI$

[1] A combinator is regular when its first argument does not disappear, is not duplicated, and occurs first in the resulting combination. The following usual combinators are regular: compositor $B = \lambda fgx.f(gx)$, permutator $C = \lambda fxy.fyx$, identity $I = \lambda x.x$, cancellator $K = \lambda xy.x$, distributor $S = \lambda fgx.fx(gx)$, and duplicator $W = \lambda fx.fxx$.

A_n combines a function and n continuations. It may be defined using an auxiliary family:

$$\begin{aligned} A_n f &= Z_n^f \theta_0 = Z_n^f \theta \\ Z_0^f &= f \\ Z_{i+1}^f &= \lambda a.\lambda\kappa_{n-i}.Z_i^f(\lambda v.av\kappa_{n-i}) \\ &= B(BZ_i^f)C \\ &= C(BBB)CZ_i^f = B(CBC)BZ_i^f \end{aligned}$$

This yields the two possible definitions:

$$A_n = \lambda g.\underline{n}(C(BBB)C)g\theta = \lambda g.\underline{n}(B(CBC)B)g\theta$$

As usual, defining combinators makes it possible to derive an instruction set for implementing a semantic specification.

5 Comparison with Related Work

The idea of representing "the rest of the computation" as a function or a procedure has occurred more or less independently in [van Wijngaarden 66] for transforming Algol 60 programs to eliminate all labels, in [Mazurkiewicz 71] for proving algorithms, and in [Fischer 72] to prove the generality of a deletion implementation strategy, based on adding a functional argument and transforming returns into calls [Morris 72]. Continuations were presented as a device for formalizing control flow in the denotational specification of programming languages with jumps [Strachey & Wadsworth 74]. This device was characterized as yielding language specifications independent of the evaluation order of the defining language [Reynolds 72] [Plotkin 75]. Relations between direct and continuation semantics were investigated in [Reynolds 74], together with their congruence [Sethi & Tang 80]. Continuation-passing style was made popular in [Steele & Sussman 76], and conversion to CPS has become a common device for compiling Scheme [Steele 78] and more recently ML programs [Appel & Jim 89].

Programming with continuations has appeared as an intriguing possibility offered by control operators such as Landin's J, Reynolds's `escape`, and `call-with-current-continuation` in Scheme. Such first-class continuations are more general than MacLisp's `catch/throw` mechanism and ML's exceptions since they allow a previous scope to be restored, just like applying a functional value reestablishes an earlier environment. First-class continuations have been investigated mainly as powerful, but unstructured devices requiring a deep intuition and operational skill [Friedman, Haynes, & Kohlbecker 84] [Haynes & Friedman 87]. However, some progress has been made towards a more declarative view of them, based on a category-theoretical duality between values and continuations [Filinski 89].

Recent trends in investigating first-class continuations require the ability of *composing* them [Felleisen *et al.* 87]. This has been explored in GL [Johnson 87], where continuations were composed in the semantic equations. Unfortunately *this makes the semantics order-dependent*, comparably to converting a meta-circular interpreter containing `shift` and `reset` only once. The approach is changed in [Johnson & Duggan 88], where continuations are composed by appending their representation, precisely as with the `control` operator in [Felleisen *et al.* 88]. Further, since

a continuation represents a context, context delimiters have been introduced in [Felleisen 88] as prompts. In our framework, a prompt naturally is the direct style counterpart of initializing the continuation of a CPS program with the identity function.

Formal descriptions of the `control/prompt` approach rely on representing continuations as prompt-delimited sequences of activation frames, and their composition as the concatenation of these sequences. In contrast, the `shift/reset` approach is based on viewing a program as computing a function expressed in extended continuation-passing style. For example, and as in section 1,

```
(let ([f (lambda (x) (shift k (k (k x))))])
  (+ 1 (reset (+ 10 (f 100)))))
```

is really just another way of expressing

```
(let ([f-c (lambda (x k) (k (k x)))])
  (+ 1 (f-c 100 (lambda (v) (+ 10 v)))))
```

while `prompt/control` admits no such simple static interpretation.

The operator `control` is often referred to as Felleisen & Friedman's $\mathcal{F}$ operator. Its operational definition leaves some room for variation, and one restricted form, also known as $\mathcal{F}$-, is characterized by wrapping a `prompt` around all continuation applications. The effect is to close the extent of a context at the point of abstraction, instead of joining it to the context at the point of application. This is very much like building an explicit closure with `FUNCTION` around a `LAMBDA` in Lisp 1.5, which experience has shown to be preferable in general for both theoretical and practical reasons, leading to lexically scoped procedures in Scheme. In fact, $\mathcal{F}$- coincides operationally with $\mathbf{shift}_1$ for the simple case of one delimited context, just like static and dynamic scoping agree when the latter is used in a controlled way.

[Sitaram & Felleisen 90] proposes a hierarchy of control operators. In that framework, specialized operators are defined within the language using `control` and `prompt`. Using them instead of the general primitives `control` and `prompt` realizes the hierarchy. In contrast, our approach is fundamentally hierarchic rather than being based on any particular discipline of programming. In fact, our development is similar to what Strachey and Wadsworth proposed in their original report [footnote 11]: to associate a continuation with each embedding context.

[Danvy & Filinski 89] investigates abstracting control in one delimited context, as in section 1 and 2. It also presents a number of examples and a static type system.

[Danvy 89] contrasts imperative and functional abstractions of control by modeling them with the cancellator and with the Curry combinators, respectively. It also stresses the duality between sharing data structures and sharing control when constructing these data structures.[2]

Abstracting control and evaluating a term in a series of embedded contexts can be related to computational reflection and the reflective tower [Smith 82]. Leaving aside the fact that a reflective tower addresses all the elements of the computational field and not only control, our contexts correspond to the levels in the reflective tower. In particular, the valuation function is comparable to the continuation-passing counterpart of the valuation function in a traditional continuation semantics for Scheme though there are infinitely many contexts:

$$\texttt{Exp} \rightarrow Env \rightarrow Cont \rightarrow MCont \rightarrow Ans$$

where $MCont = Cont^*$ in [Wand & Friedman 88]
and $MCont = (Cont \times Env)^*$ in [Danvy & Malmkjær 88].
Finally reifying continuations and composing them [Malmkjær 89] pushes and pops continuations off the meta-continuation, dynamically, which necessitates reflexive types. This is not the case for our description, and thus computation never goes up and down in the tower.

Conclusions and Issues

This article presents continuations as functions representing embedding contexts when evaluating λ-terms with control operators. This view is structured by making λ-terms with control operators denote purely functional λ-terms in extended continuation-passing style, combining the conciseness of direct style and the expressive power of extended continuation-passing style. It offers a naturally hierarchical framework to specify programming languages with advanced control structures independently from any evaluation order and in a statically typable way, which captures the original motivations for continuations.

This development has lead us to generalize the algorithm of translation from direct style to CPS and ECPS, and to formulate it meta-circularly. More notably, this investigation made it possible to improve the efficiency of the translation, by distinguishing between syntactic (run time) and semantic (compile time, or rather translation time) λ-abstractions and thus avoiding both building and post-reducing β- and η-redexes.

The translation can still be improved by generalizing Reynolds's concept of *serious* and *trivial* terms [Reynolds 72]: a n-trivial term is a term that does not need to be passed the nth, $n+1$st, *etc.* continuations because it does not use them but merely transmits them unchanged. Such a term need not be translated further.

We have implemented both semantics and translations by transliterating the semantic equations into Scheme, defining *de facto* yet another dialect of Scheme. Due to η-reduction, computations only pay for the contexts they actually use.

There are many possible developments to this work. For example, the set of control operators presented here is somewhat arbitrary, and not completely satisfactory, in the sense that the desirable relation

$$\xi_n k.kE \equiv E$$

does not hold for $n \geq 2$. Also it might be necessary to select a continuation representing a specific outer context; or each continuation up to another context, *e.g.* with

$$\xi(k_1, ..., k_n).E$$

Work is going on to devise other families of operators characterized by simpler relations or more closely corresponding to some particular pattern of use.

One might also consider having infinitely many continuations so that it would be possible to abstract arbitrarily

[2] For example, listing the suffixes of a list encourages sharing the tails of the original list. Symmetrically, listing the prefixes of a list encourages sharing the construction of these prefixes, as treated in Pavel Curtis's **(Algorithms)** of Lisp Pointers II-3/4.

many contexts. However, this raises some deep foundational problems. An idea is to construct the inverse limit of the CPS conversion. On the other hand, having a limited number of contexts may correspond to a computational reality. Again, this is left for further research.

Finally, there appear to be some deep connections between our development and the monad-based computational λ-calculus described in [Moggi 89]. One would expect the generalized notion of continuation presented here to be expressible as a suitable monad structure over the λ-calculus. Furthermore, the ability to compose and reset continuations in turn seems sufficient to encode any such structure as an instance of the ECPS formalism. This potential equivalence is currently being investigated.

The study of advanced control structures in programming languages is still a comparatively young area of computer science, and it is yet far from clear what directions it will take. We believe, however, that the basic idea of a generalized continuation-passing style, expressed in a functional, statically typed framework stands an excellent chance of eventually leading towards a better understanding of abstracted control.

Acknowledgements

To Karoline Malmkjær, Neil Jones, and all the other members of the Mix group at DIKU. Thanks are also due to Julia Lawall, Dan Friedman, Bob Hieb, and Kent Dybvig for insightful interactions while the first author was visiting Indiana University. This work has benefited from Carolyn Talcott's interest and comments, and from thoughtful observations about congruences by Pierre Jouvelot and David Schmidt. Special thanks to Bruce Duba and Matthias Felleisen for lively discussions during the spring of 1990, following thorough comments on the extended abstract of this paper. Finally, thanks are due to the referees.

References

[Abelson & Sussman 85] Harold Abelson and Gerald Jay Sussman with Julie Sussman: *Structure and Interpretation of Computer Programs*, The MIT Press, Cambridge, Massachusetts (1985)

[Aho, Hopcroft, & Ullman 74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974)

[Appel & Jim 89] Andrew W. Appel, Trevor Jim: *Continuation-Passing, Closure-Passing Style*, proceedings of the Sixteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp 293-302, Austin, Texas (January 1989)

[Barendregt 85] Hendrick P. Barendregt: *The Lambda Calculus, Its Syntax and Semantics*, revised edition, Studies in Logic and the Foundations of Mathematics, Vol. 103, North-Holland (1985)

[Danvy & Malmkjær 88] Olivier Danvy, Karoline Malmkjær: *Intensions and Extensions in a Reflective Tower*, proceedings of the 1988 ACM Conference on Lisp and Functional Programming pp 327-341, Snowbird, Utah (July 1988)

[Danvy & Filinski 89] Olivier Danvy, Andrzej Filinski: *A Functional Abstraction of Typed Contexts*, DIKU Rapport 89/12, DIKU, University of Copenhagen, Copenhagen, Denmark (August 1989)

[Danvy 89] Olivier Danvy: *Programming with Tighter Control*, pp 10-29 of the BIGRE journal, No 65 on *Putting Scheme to Work*, André Pic, Michel Briand, and Jean Bézivin (eds.), Brest, France (July 1989)

[Felleisen *et al.* 87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, John Merrill: *Beyond Continuations*, Technical Report No 216, Computer Science Department, Indiana University, Bloomington, Indiana (February 1987)

[Felleisen 88] Matthias Felleisen: *The Theory and Practice of First-Class Prompts*, proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp 180-190, San Diego, California (January 1988)

[Felleisen *et al.* 88] Matthias Felleisen, Mitchell Wand, Daniel P. Friedman, Bruce F. Duba: *Abstract Continuations: A Mathematical Semantics for Handling Full Functional Jumps*, proceedings of the 1988 ACM Conference on Lisp and Functional Programming pp 52-62, Snowbird, Utah (July 1988)

[Filinski 89] Andrzej Filinski: *Declarative Continuations: An Investigation of Duality in Programming Language Semantics*, proceedings of the Summer Conference on Category Theory and Computer Science, Lecture Notes in Computer Science No 389 pp 224-249, D. H. Pitt, D. E. Rydeheard, P. Dybjer, A. M. Pitts, & A. Poigné (eds.), Springer-Verlag, Manchester, UK (September 1989)

[Fischer 72] Michael J. Fischer: *Lambda Calculus Schemata*, proceedings of the ACM conference *Proving assertions about programs* pp 104-109, SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14 (January 1972)

[Friedman, Haynes, & Kohlbecker 84] Daniel P. Friedman, Christopher T. Haynes, Eugene Kohlbecker: *Programming with Continuations*, NATO ASI Series, Vol. F8, *Program Transformation and Programming Environments* pp 263-274, P. Pepper (ed.), Springer-Verlag Berlin Heidelberg (1984)

[Haynes & Friedman 87] Christopher T. Haynes, Daniel P. Friedman: *Embedding Continuations in Procedural Objects*, TOPLAS, Vol. 9, No 4 pp 582-598 (October 1987) Preliminary version: *Constraining Control*, proceedings of the 12th ACM Symposium on Principles of Programming Languages pp 245-254 (January 1985)

[Johnson 87] Gregory F. Johnson: *GL – A Denotational Testbed with Continuations and Partial Continuations as First-Class Objects*, proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques pp 154-176, Saint-Paul, Minnesota (June 1987)

[Johnson & Duggan 88] Gregory F. Johnson, Dominic Duggan: *Stores and Partial Continuations as First-Class Objects in a Language and its Environment*, proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages pp 158-168, San Diego, California (January 1988)

[Malmkjær 89] Karoline Malmkjær: *On some Semantic Aspects of the Reflective Tower*, proceedings of the 4th Conference on Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science No ??, Michael Main, Austin Melton, Michael Mislove, and David Schmidt (eds.), Springer-Verlag, New Orleans, Louisiana (April 1989)

[Mazurkiewicz 71] Antoni W. Mazurkiewicz: *Proving Algorithms by Tail Functions*, Information and Control Vol. 18 pp 220-226 (1971)

[Mellish & Hardy 84] Chris Mellish, Steve Hardy: *Integrating Prolog in the POPLOG Environment*, in *Implementations of PROLOG*, John A. Campbell (ed.) pp 147-162, Ellis Horwood (1984)

[Miller 87] James S. Miller: *MultiScheme: A Parallel Processing System Based on MIT Scheme*, PhD thesis, MIT/LCS/TR-402 (September 1987)

[Milne & Strachey 76] Robert E. Milne, Christopher Strachey: *A Theory of Programming Language Semantics*, Chapman and Hall, London, and John Wiley, New York (1976)

[Moggi 89] Eugenio Moggi: *Computational Lambda-calculus and Monads*, proceedings of 4th Conference on Logic in Computer Science pp 14-23, IEEE (1989).

[Morris 72] James H. Morris Jr.: *A Bonus from van Wijngaarden's Device*, Communications of the ACM Vol. 15, No 8, p 773 (August 1972)

[Plotkin 75] Gordon Plotkin: *Call-by-Name, Call-by-Value, and the λ-calculus*, Theoretical Computer Science, Vol. 1 pp 125-159 (1975)

[Rees & Clinger 86] Jonathan Rees, William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme*, SIGPLAN Notices, Vol. 21, No 12 pp 37-79 (December 1986)

[Reynolds 72] John Reynolds: *Definitional Interpreters for Higher-Order Programming Languages*, proceedings 25th ACM National Conference pp 717-740, New York (1972)

[Reynolds 74] John Reynolds: *On the Relation between Direct and Continuation Semantics*, 2nd Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science No 14 pp 141-156, Springer-Verlag, Jacques Loeckx (ed.), Saarbrücken, West Germany (July 1974)

[Sethi & Tang 80] Ravi Sethi, Adrian Tang: *Constructing Call-by-Value Continuation Semantics*, Journal of the ACM, Vol. 27, No 3 pp 580-597 (July 1980)

[Sitaram & Felleisen 90] Dorai Sitaram, Matthias Felleisen: *Control Delimiters and their Hierarchies*, to appear in *Lisp and Symbolic Computation*, Rice University, Houston, Texas (June 1989)

[Smith 82] Brian C. Smith: *Reflection and Semantics in a Procedural Language*, PhD thesis, MIT/LCS/TR-272, MIT, Cambridge, Massachusetts (January 1982)

[Steele & Sussman 76] Guy L. Steele Jr., Gerald J. Sussman: *Lambda, the Ultimate Imperative*, MIT-AIL, AI Memo No 353, Cambridge, Massachusetts (March 1976)

[Steele 78] Guy L. Steele Jr.: *RABBIT: A Compiler for SCHEME*, MIT-AIL, AI-TR-474, Cambridge, Massachusetts (May 1978)

[Stoy 81] Joseph E. Stoy: *The Congruence of Two Programming Language Definitions*, Theoretical Computer Science, Vol. 13 pp 151-174 (1981)

[Strachey & Wadsworth 74] Christopher Strachey, Christopher P. Wadsworth: *Continuations: A Mathematical Semantics for Handling Full Jumps*, Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1974)

[van Wijngaarden 66] Adriaan van Wijngaarden: *Recursive Definition of Syntax and Semantics*, in *Formal Language Description Languages for Computer Programming* pp 13-24, T. B. Steel Jr. (ed.), North-Holland (1966)

[Wand & Friedman 88] Mitchell Wand, Daniel P. Friedman: *The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower*, Volume 1, Issue 1 of *Lisp and Symbolic Computation* (May 1988)