Draft 4.2 – July 22nd, 1989

A Functional Abstraction of Typed Contexts

Olivier Danvy & Andrzej Filinski

DIKU – Computer Science Department, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen Ø, Denmark
uucp: danvy@diku.dk & andrzej@diku.dk

Abstract

This report investigates abstracting control with functions. This is achieved by defining continuations as functions abstracting lexically a delimited context [**C**[ ]] rather than dynamically an unlimited one **C**[ ], as it is usually the case. Because their co-domain is distinguished from the final domain of Answers, such continuations can be composed, and this contrasts with the simple exceptions of ML and Lisp and the unlimited first-class continuations of Scheme. Making these functional control abstractions first-class offers a new area in programming which this paper explores.

The key points obtained here are: a denotational semantics for a simple, call-by-value, strongly typed expression language with higher-order functions and first-class continuations; its congruence with a continuation semantics; a polymorphic type inference system for that language; a systematic translation to higher-order functions (continuation-passing style); some worked examples of programming with functional abstractions of control; and an extension towards gaining control over embedding contexts.

Sections 1 and 2 present the denotational semantics of the simple expression language and its congruence with a traditional continuation semantics. Section 3 illustrates programming with control abstractions. Section 4 describes its translation to an extended continuation-passing style with higher-order functions, and section 5 describes its type system. Section 6 compares this approach with related work.

Keywords
Continuations, contexts, programming languages, congruence, type inference, program transformation.

Introduction

In most people's minds, continuations reflect the imperative aspects of control in programming languages because they have been introduced for handling full jumps [Strachey & Wadsworth 74]. They abstract an evaluation context **C**[ ] up to the final domain of Answers and leave aside all its applicative aspects: when provided as a first-class object[1], a continuation is a function k that cannot be composed freely, as in $f \circ k$.

This report investigates viewing a continuation as a lexical abstraction of a delimited context [**C**[ ]] by means of a function whose codomain is not the final domain of Answers. Such functions can thus be composed. Our approach continues the one from [Felleisen *et al.* 87] and [Felleisen 88], where an extended λ-calculus and a new SECD-like machine were defined, and [Felleisen *et al.* 88] which presents an algebraic framework where continuations are defined as a sequence of frames and their composition as the dynamic concatenation of these sequences. The present paper describes a more lexical vision of composable continuations, that can be given statically a type. We describe a simple expression language using denotational semantics where the local context is represented with a local continuation κ and an extra parameter γ represents the *surroundings*, *i.e.*, the outer contexts. That parameter γ maps values to final answers like traditional (non-composable) continuations, and the local continuation κ is defined as a surroundings transformer. We build a "surroundings" semantics $\mathcal{E}_{sur}$ from traditional continuation semantics $\mathcal{E}_{cnt}$ on the congruence relation [Sethi & Tang 80]:

$$\mathcal{E}_{sur}\,[\![\mathtt{E}]\!]\,\rho\,\kappa\,\gamma = \mathcal{E}_{cnt}\,[\![\mathtt{E}]\!]\,\rho\,(\kappa\,\gamma)$$

The problem of composing continuations can be compared to modeling jumps and non-local exits in a direct semantics. The failures to model such situations have motivated introducing continuation semantics and now context semantics.

This paper concentrates on programming and transforming with functional abstractions of control: composable continuations. We devise a translation of programs with explicit access to contexts into programs without, but with more familiar higher-order functions in a continuation-passing style. Remarkably enough, the implementation of the translation uses composable continuations in a natural way and thus constitutes a non-trivial example of use for practical purpose.

There are a number of reasons why programming with composable continuations is of interest, ranging from conciseness to modeling non-determinism using backtracking. The latter merely consists of sequentially applying a continuation to several values and getting back a sequence of answers. It is developed in section 3. Let us illustrate the former and give an overview of our investigation method.

Our means is a simple expression language, strongly typed, call-by-value and higher-order, extended with two elementary operations over the current context: `shift` and `reset`. We get control over the context using an expression of the form

```
shift I in E
```

where the current context is abstracted as a function and bound to the identifier `I` before the expression `E` is evaluated in a new context. Conversely, with an expression of the form

```
reset E
```

`E` is evaluated in a new, empty context. This allows to define conveniently and lexically the co-domain of a continuation[2].

For example, in the expression `5 + reset(3 + (shift c in (c 0) + (c 1)))`, the variable `c` is bound to a functional abstraction of the current context: $\lambda n \,.\, 3 + n$ and the expression evaluates to

[1] Such as a reified continuation in Scheme [Rees & Clinger 86].

[2] `shift` and `reset` are related to the **control** operator and the prompts # in [Felleisen 88].

$5 + ((3 + 0) + (3 + 1)) = 12$. This can be visualized with the following applicative-order reduction steps, where "c ∼ [ ]" reads "c abstracts the context [ ]":

```
[5 + reset(3 + (shift c in (c 0) + (c 1)))]
[5 + [3 + (shift c in (c 0) + (c 1))]]
[5 + [(c 0) + (c 1)]] where c ∼ [3 + [ ]]
[5 + [[3 + [0]] + [3 + [1]]]]
[5 + [[3 + 0] + [3 + 1]]]
[5 + [[3] + [4]]]
[5 + [3 + 4]]
[5 + [7]]
[5 + 7]
[12]
12
```

As a special case, if the expression in `shift` applies the continuation only at the root of a term, the `shift` degenerates to an identity operation. We prove it in the end of section 1. A very common operation consists of just discarding the continuation, so we define

```
abort E ≡ shift c in E
```

where `c` does not occur free in `E`. We can define the classical `escape` [Reynolds 72] and Scheme's `call-with-current-continuation` [Rees & Clinger 86] as well, modulo any use of `reset` in `E`:

```
escape c in E ≡ shift k in let c x = abort (k x) in k E
call-with-current-continuation f ≡ shift k in k (f (fn x => abort (k x)))
```

Section 1 describes the denotational semantics of the language and section 5 presents its type system as a set of inference rules.

We are accustomed to seeing higher-order functional programming in continuation-passing style [van Wijngaarden 66] [Mazurkiewicz 71] [Fischer 72] [Reynolds 72] [Steele & Sussman 76] with continuation semantics [Reynolds 74] [Stoy 77] [Schmidt 86]. In essence, one extra functional argument will be applied to the result of the current computation. For a simple example, the function duplicating a list[3]:

```
def duplicate l =   -- list(A) -> list(A)
    if l = [] then [] else (hd l) :: duplicate(tl l)
```

could be written in continuation-passing style like this:

```
def duplicate l =   -- list(A) -> list(A)
    letrec duplicate-c l c =   -- list(A) -> (list(A) -> B) -> B
           if l = [] then c [] else duplicate-c (tl l) (fn r => c((hd l) :: r))
    in duplicate-c l (fn x => x)
```

Some familiarity with continuation semantics makes this second definition understandable: the extra functional argument will be applied to the result of the current computation. However, it is very easy to lose track of what is happening, for example by relaxing the rule that a continuation should always be applied at the root of a term, *i.e.*, as a tail call. To illustrate this "continuation-composing style", let us slightly modify the function above, enveloping the call to the continuation with a list construction:

[3] Of course we could define `duplicate` as *map* $\lambda x.x$, but this would require considering the curried functional *map* instead and we prefer sticking for this first example to simplicity – though avoiding the identity function.


```
def foo l =   -- list(A) -> list(A)
    letrec foo-c l c =   -- list(A) -> (list(B) -> list(A)) -> list(A)
           if l = [] then c [] else foo-c (tl l) (fn r => (hd l) :: (c r))
    in foo-c l (fn x => x)
```

Beyond the traditional methods for reasoning about it, this function is not immediately understandable – not that it is counter-intuitive, but rather because there is no standard intuition about its action. Its effect is to reverse a list:

```
foo [1,2,3] ⇒ [3,2,1]
```

We can best explain it by tracing three recursive calls. Classically, we expect something like:

$$\overset{1}{\rightarrow}\quad\overset{2}{\rightarrow}\quad\overset{3}{\rightarrow}$$
$$\overset{6}{\leftarrow}\quad\overset{5}{\leftarrow}\quad\overset{4}{\leftarrow}$$

where an arrow $\overset{i}{\rightarrow}$ represents a call, an arrow $\overset{j}{\leftarrow}$ represents a return, and i and j label the sequence of calls and returns. Presently, what we get with the function `foo` above is:

$$\overset{1}{\rightarrow}\quad\overset{2}{\rightarrow}\quad\overset{3}{\rightarrow}$$
$$\overset{4}{\leftarrow}\quad\overset{5}{\leftarrow}\quad\overset{6}{\leftarrow}$$

where the first return corresponds to the first call (and precisely not to the last one), and accordingly the last return matches the last call. These "recursive" calls and returns seem to bypass quite a few things, for example the adequacy of a stack to implement them [Dijkstra 60]. However, that diagram provides some better intuition on why the list is reversed. It suggests that this mode of programming stands in the middle between recursion and iteration. In an iterative version, we would have a parameter for accumulating the result, and the step 1 and 4, 2 and 5, and 3 and 6 would actually be paired [Wand 80] – but it should be stressed that this vision does not directly apply here since there is no immediate translation to an iterative version with accumulator.

Expressing `foo` in direct style and with access to its context rather than with higher-order functions, and renaming it `reverse` gives:

```
def reverse l =   -- list(A) -> list(A)
    letrec reverse-s l =   -- list(A) -> list(A)
           if l = [] then [] else shift c in (hd l) :: c(reverse-s(tl l))
    in reset (reverse-s l)
```

One can notice that this does not make the function more understandable, but it is interesting that we can give it a semantics both in its higher-order form and in its direct form with access to contexts. Its action can be visualized with the following applicative-order reductions. For readability, lists will be noted between angle brackets.

```
reverse <1,2,3>
[reverse-s <1,2,3>]
[shift c in 1 :: c(reverse-s <2,3>)]
[1 :: c(reverse-s <2,3>)] where c ∼ [[ ]]
[1 :: c(shift c in 2 :: c(reverse-s <3>))] where c ∼ [[ ]]
[2 :: c(reverse-s <3>)] where c ∼ [1 :: c([ ])] where c ∼ [[ ]]
[2 :: c(shift c in 3 :: c(reverse-s <>))] where c ∼ [1 :: c([ ])] where c ∼ [[ ]]
[3 :: c(reverse-s <>)] where c ∼ [2 :: c([ ])] where c ∼ [1 :: c([ ])] where c ∼ [[ ]]
[3 :: c(<>)] where c ∼ [2 :: c([ ])] where c ∼ [1 :: c([ ])] where c ∼ [[ ]]
```

[3 :: [2 :: c(<>)]] where c ∼ [1 :: c([ ])] where c ∼ [[ ]]
[3 :: [2 :: [1 :: c(<>)]]] where c ∼ [[ ]]
[3 :: [2 :: [1 :: [<>]]]]
[3 :: [2 :: [1 :: <>]]]
[3 :: [2 :: [<1>]]]
[3 :: [2 :: <1>]]
[3 :: [<2,1>]]
[3 :: <2,1>]
[<3,2,1>]
<3,2,1>

It should be noted, too, that the function `foo` above has been translated automatically from this definition of `reverse`. (They have been presented in that order for introductory purposes.)

To understand the underlying mechanisms of this example, we have devised a translator from direct style with `shift` and `reset` to higher-order functions, in extended continuation-passing style[4]. Our converter uses the present facilities of composing continuations and appears strikingly concise. We include its Scheme code in appendices A and B.

The rest of this report is organized as follows: section 1 presents the denotational semantics of our expression language extended with `shift` and `reset`. Section 3 develops some worked examples of programming with control abstractions. Section 4 describes the translation of such programs to continuation-composing style with higher-order functions. Section 5 describes its type system. Section 6 compares this approach with related work, and puts it into perspective.

1 A Denotational Semantics

This first section presents a denotational semantics of our expression language. We want to express here in a standard framework how continuations are treated when they model a series of properly nested contexts:

$$\ldots[\mathbf{C}_n[\mathbf{C}_{n-1}[\ldots[\mathbf{C}_2[\mathbf{C}_1[\mathbf{C}_0[\ ]]]]\ldots]]]\ldots$$

The central idea is to abstract the current context $[\mathbf{C}_0[\ ]]$ with the current continuation κ and to hold the state of the outer contexts $\ldots[\mathbf{C}_n[\ldots[\mathbf{C}_1[\ ]]\ldots]]\ldots$ with the extra parameter γ. When the context is reset, the current κ is merely stacked on γ and replaced by an identity function that will eventually be applied and will propagate the current result in the embedding context. Symmetrically, shifting captures the current continuation in a function that will apply it to a result, stacking the then current context in γ. At shifting time, the current context is reset on the fly and eventually a result will be propagated in the embedding context, unless of course the continuation is captured again.

We consider a simple, ML-like, expression language, defined by a straightforward continuation semantics. We then build a new semantics for this language on the congruence relation:

$$\mathcal{E}_{sur}\ [\![\mathtt{E}]\!]\ \rho\ \kappa\ \gamma = \mathcal{E}_{cnt}\ [\![\mathtt{E}]\!]\ \rho\ (\kappa\ \gamma)$$

(this is described in section 2) and we extend it with control operators on κ and γ.

Abstract syntax: `E` ∈ `Expression`; `F` ∈ `Procedure`; `C` ∈ `Constant`; `I` ∈ `Identifier`

[4]Extended because in the presence of `shift` and `reset`, the continuation parameter is not always applied at the root of a term.


```
E ::= C | I | F | E0 E1 | shift I in E | reset E
    | if E0 then E1 else E2 | let (I = E)* in E0 | letrec (I = F)* in E
F ::= fn I => E
```

The concrete syntax is sugared so that the declaration `f x y = E` expands to the curried `f = fn x => fn y => E`. Also, top level recursive definitions are introduced with `def`, as in the introduction. For the sake of simplicity, we leave out tupling.

The semantic algebras are the basic domains of numbers *Int*, booleans *Bool*, strings *String*, identifiers *Ide*, finite lists *List*, and so on. Furthermore, the expressible values, which coincide with the denotable values:

$a, v \in Val = Int + Bool + String + List + Fun$

the environments:

$\rho \in Env = Ide \to Val$

the surrounding contexts:

$\gamma \in Sctx = Val \to Ans$

the continuations:

$\kappa \in Ctn = Sctx \to Val \to Ans = Sctx \to Sctx$

the (unary) functions *Fun*:

$f \in Fun = Ctn \to Sctx \to Val \to Ans = Ctn \to Ctn$

and the domain of answers:

$Ans = Val_\perp$

Two valuation functions $\mathcal{C}$ and $\mathcal{I}$ give the denotation of constants and identifiers:

$\mathcal{C}$: `Constant` → *Val*

$\mathcal{I}$: `Identifier` → *Ide*

The definition of $\mathcal{C}$ is omitted. It maps straightforwardly any constant to its denotation. The definition of $\mathcal{I}$ is also omitted. The valuation function $\mathcal{F}$ maps a syntactic λ-abstraction into a semantic one:

$\mathcal{F}$: `Procedure` → *Env* → *Val*

$\mathcal{F}[\![\texttt{fn I => E}]\!]\rho = inFun(\lambda\kappa\gamma v\,.\,\mathcal{E}[\![\mathtt{E}]\!]([\mathcal{I}\,[\![\mathtt{I}]\!] \mapsto v]\rho)\kappa\gamma)$

The valuation function $\mathcal{E}$ for the expressions follows:

$\mathcal{E}$: `Expression` → *Env* → *Ctn* → *Sctx* → *Ans*

$$\begin{aligned}
\mathcal{E}[\![\mathtt{C}]\!]\rho\kappa\gamma &= \kappa\,\gamma\,\mathcal{C}[\![\mathtt{C}]\!] \\
\mathcal{E}[\![\mathtt{I}]\!]\rho\kappa\gamma &= \kappa\,\gamma\,\rho(\mathcal{I}[\![\mathtt{I}]\!]) \\
\mathcal{E}[\![\texttt{if E}_0\texttt{ then E}_1\texttt{ else E}_2]\!]\rho\kappa\gamma &= \mathcal{E}[\![\mathtt{E}_0]\!]\rho(\lambda\gamma a\,.\,let\ Bool(b) = a \\
&\qquad in\ (b \to \mathcal{E}[\![\mathtt{E}_1]\!]\rho\kappa\gamma \parallel \mathcal{E}[\![\mathtt{E}_2]\!]\rho\kappa\gamma))\ \gamma \\
\mathcal{E}[\![\mathtt{F}]\!]\rho\kappa\gamma &= \kappa\,\gamma\,(\mathcal{F}[\![\mathtt{F}]\!]\rho) \\
\mathcal{E}[\![\mathtt{E}_0\ \mathtt{E}_1]\!]\rho\kappa\gamma &= \mathcal{E}[\![\mathtt{E}_0]\!]\rho(\lambda\gamma a\,.\,let\ Fun(f) = a\ in\ \mathcal{E}[\![\mathtt{E}_1]\!]\rho(f\kappa)\gamma)\gamma \\
\mathcal{E}[\![\texttt{let (I = E}_1\texttt{) in E}_0]\!]\rho\kappa\gamma &= \mathcal{E}[\![\mathtt{E}_1]\!]\rho(\lambda\gamma a.\mathcal{E}[\![\mathtt{E}_0]\!]([\mathcal{I}[\![\mathtt{I}]\!] \mapsto a]\rho)\kappa\gamma)\gamma \\
\mathcal{E}[\![\texttt{letrec (I = F)* in E}]\!]\rho\kappa\gamma &= \mathcal{E}[\![\mathtt{E}]\!]fix(\lambda\rho'\,.\,extend(map(\lambda[\![\texttt{(I = F)}]\!]\,.\,\mathcal{I}[\![\mathtt{I}]\!])[\![\texttt{(I = F)*}]\!], \\
&\qquad map(\lambda[\![\texttt{(I = F)}]\!]\,.\,\mathcal{F}[\![\mathtt{F}]\!]\rho')[\![\texttt{(I = F)*}]\!], \\
&\qquad r))\,\kappa\,\gamma
\end{aligned}$$

using a function *extend*: $Ide^* \times Val^* \times Env \to Env$ that extends an environment given two isomorphic sequences of identifiers and of values.

The denotation of `let`-expressions with multiple bindings is obtained straightforwardly by expansion into nested `let`s, possibly with some variable renaming to prevent name clashes. So far, γ

has been passively transmitted. It intervenes when a captured continuation is applied and when the continuation is reset:

$$\begin{aligned}\mathcal{E}[\![\texttt{shift I in E}]\!]\rho\kappa\gamma &= \mathcal{E}[\![\texttt{E}]\!]([\mathcal{I}[\![\texttt{I}]\!] \mapsto \mathit{inFun}(\lambda\kappa'\gamma\,.\,\kappa(\kappa'\gamma))]\rho)\mathit{id}_{\mathit{Sctx}}\gamma\\ &= \mathcal{E}[\![\texttt{E}]\!]([\mathcal{I}[\![\texttt{I}]\!] \mapsto \mathit{inFun}(B\kappa)]\rho)\mathit{id}_{\mathit{Sctx}}\gamma\\ \mathcal{E}[\![\texttt{reset E}]\!]\rho\kappa\gamma &= \mathcal{E}[\![\texttt{E}]\!]\rho\,\mathit{id}_{\mathit{Sctx}}\,(\kappa\gamma)\end{aligned}$$

using the compositor combinator $B = \lambda fgx\,.\,f(gx)$.

An expression `E` is evaluated in an initial environment ρ_{init} with all the predefined bindings. The initial continuation is $\kappa_{init} = \mathit{id}_{\mathit{Sctx}}$, which expresses the fact that there is an implicit top-level reset. The initial context is $\gamma_{init} = \mathit{id}_{\mathit{Val}}$.

There are a couple of major points to notice, the first being that γ is single-threaded [Schmidt 85]. This ensures that it continuously reflects the current state of the surroundings, which can be captured at any time.

In a combination, the function is evaluated prior its argument. By extension, multiple arguments to a curried function are evaluated from left to right.

At reset time, the current continuation is stacked on γ and replaced by the initial continuation that delimits a new context of computation. It defines the co-domain of the new continuation in the case of a capture.

Shifting consists of capturing κ and resetting the continuation. The environment is extended with a function that will apply the captured continuation to its argument, stacking the then current continuation on top of γ. This realizes composing the two continuations. Unlike [Felleisen *et al.* 88], the extent of a captured context is never expanded, just like the environment part of a functional closure in a lexically-scoped language is fixed at the time of its creation. This aspect is treated in section 6.

At this point, we can prove the assertion of the introduction that if the expression in `shift` applies the continuation exactly at the root of a term, the `shift` degenerates to an identity operation. Provided that `E` is free of `c`:

$$\mathcal{E}[\![\texttt{shift c in c E}]\!]\rho\kappa\gamma = \mathcal{E}[\![\texttt{c E}]\!]\rho'\mathit{id}_{\mathit{Sctx}}\gamma = \mathcal{E}[\![\texttt{E}]\!]\rho'(B\,\kappa\,\mathit{id}_{\mathit{Sctx}})\gamma = \mathcal{E}[\![\texttt{E}]\!]\rho'\kappa\gamma$$

Further, we can express better the `abort` construction from the introduction. Provided that `E` is free of `c`, `abort` is defined as a syntactic extension:

$$\mathcal{E}[\![\texttt{abort E}]\!]\rho\kappa\gamma \equiv \mathcal{E}[\![\texttt{shift c in E}]\!]\rho\kappa\gamma = \mathcal{E}[\![\texttt{E}]\!]\rho'\mathit{id}_{\mathit{Sctx}}\gamma$$

This can be visualized better with:

$$\ldots[\mathbf{C}_1[\mathbf{C}_0[\texttt{abort E}]]]\ldots \Rightarrow \ \ldots[\mathbf{C}_1[\mathbf{C}_{\mathit{empty}}[\texttt{E}]]]\ldots$$

As one can note, if $\texttt{E} \equiv \texttt{shift k in E'}$, the identifier $\mathcal{I}[\![\texttt{k}]\!]$ will be bound to a functional abstraction of the empty context, *i.e.*, of $\mathit{id}_{\mathit{Sctx}}$. This is natural, since there is no access beyond the current context $\mathbf{C}_0$ to, say, $\mathbf{C}_1$. But we could provide such an access and have:

$$\ldots[\mathbf{C}_1[\mathbf{C}_0[\texttt{abort E}]]]\ldots \Rightarrow \ \ldots[\mathbf{C}_1[\texttt{E}]]\ldots$$

Then in the case above where $\texttt{E} \equiv \texttt{shift k in E'}$, the identifier $\mathcal{I}[\![\texttt{k}]\!]$ would be bound to a functional abstraction of $\mathbf{C}_1$. Appendix C describes such a variation on context semantics in which the surroundings parameter γ is defined as a stack and repeated shifting gives access to all of the embedding contexts.

2 A Congruence between Continuation and Surroundings Semantics

Surroundings semantics has been built to extend continuation semantics [Sethi & Tang 78] [Schmidt 86] and thus its restriction excluding `shift` and `reset` is congruent to continuation semantics. It is straightforward to deduce the semantic equations from the ones of the continuation semantics of the λ_v-calculus if we define

$$\begin{aligned}\mathit{Ctn} &= \mathit{Val} \to \mathit{Sctx} \to \mathit{Ans}\\ \mathit{Fun} &= \mathit{Val} \to \mathit{Ctn} \to \mathit{Sctx} \to \mathit{Ans}\end{aligned}$$

since the equations do not change – the new parameter γ is η-reduced everywhere.

Defining as in section 1 functions as continuation transformers and continuations as surroundings transformers requires to introduce the parameter γ explicitly – though it is just passed and never updated or consulted, just as the store would be in the semantic equations of a side-effect free language.

3 Programming in Continuation-Composing Style

The goal of this section is to illustrate some typical cases of programming with functional abstractions of control. First we point out that getting access to the current context differs from declaring an escape point [Reynolds 72] or getting access to the current continuation [Rees & Clinger 86]. Then we carry on with palindromes and simulating the non-deterministic generation of Pythagorean triples.

3.1 Shift vs. escape and catch

First let us point out that `shift` is not Reynolds's `escape`, nor MacLisp's `catch`, nor Scheme's `call/cc`: full access is given to the current continuation, and if it is not used, the computation in the current context terminates:

```
reset (1 + (shift c in 0))
[1 + (shift c in 0)]
[0] where c ~ [1 + [ ]]
0
```

This contrasts with getting `1` as a result, as provided by a conventional exception mechanism.

This second example illustrates again that using immediately a continuation amounts to identity:

```
reset (1 + (shift c in c 0))
[1 + (shift c in c 0)]
[c 0] where c ~ [1 + [ ]]
[1 + [0]]
1
```

This property has been proven in the end of section 1.

The following example puts in a nutshell composing continuations. First the current context is abstracted functionally. Then that abstraction is composed with itself:

```
reset (1 + (shift c in c (c 0)))
[1 + (shift c in c (c 0))]
[c (c 0)] where c ~ [1 + [ ]]
[c [1 + [0]]] where c ~ [1 + [ ]]
```

```
[c 1] where c ~ [1 + [ ]]
[[1 + [1]]]
2
```

This cannot realized as directly in Scheme or in ML.

As can be expected, `c` can be composed arbitrarily:

```
reset (1 + (shift c in c (c (c (c (c (c (c 0))))))
...
7
```

Finally, here is a way to have the cake and eat it too, considering a conditional expression:

```
let c = reset (if (shift k in k) then 2 else 3) in (c true) + (c false)
...
5
```

3.2 Getting palindromes using contexts

This section presents two ways to get a palindrome by mirroring a list. The first is a straightforward extension of `reverse`:

```
def make-palindrome s =   -- list(A) -> list(A)
    letrec mirror l =   -- list(A) -> list(A)
           if l = []
           then s
           else shift c in (hd l) :: c(mirror(tl l))
    in reset (mirror s)
```

The point is to start building the reversed list out of the initial list rather than the empty list. The second version conceptually interleaves building the two parts of the list:

```
def make-palindrome l =   -- list(A) -> list(A)
    letrec mirror l =   -- list(A) -> list(A)
           if l = []
           then []
           else shift c in (hd l) :: (c ((hd l) :: mirror(tl l)))
    in reset (mirror l)
```

We get:

```
make-palindrome [1,2,3]  ⇒  [3,2,1,1,2,3]
```

This suggests a way to generate variably periodic lists, by multiply composing continuations:

```
def bar l =   -- list(A) -> list(A)
    letrec baz l =   -- list(A) -> list(A)
           if l = []
           then []
           else shift c in (hd l) :: c ((hd l) :: (c ((hd l) :: baz(tl l))))
    in reset (baz l)
```

such that:

```
bar [1,2,3]  ⇒  [3,2,1,1,1,2,1,1,1,2,3,2,1,1,1,2,1,1,1,2,3]
```


3.3 Non-determinism

Bounded non-determinism is often simulated with backtracking, using continuation-passing style [Mellish & Hardy 84]. The problem with this approach is that programs using it may become quite complicated, because of the complex control structure. Using `shift` and `reset`, we can write very concise programs, in which the implementation of backtracking is completely hidden where not explicitly needed.

Using those operations, "non-deterministic functions" can be written in a very natural way. Furthermore, the "non-deterministic values" they return are truly first class: they can be passed as parameters, stored in data structures, *etc.*, with no special considerations.

Let us consider for example the problem of generating Pythagorean triples: we first define a procedure `choice` with one parameter `n`, which will return an integer between 1 and `n`:

```
def choice n =   -- int -> int
    shift c in letrec from s =   -- int -> int
                      if s > n
                      then fail()
                      else let dummy = (c s)
                           in from (s + 1)
               in from 1
```

The (success) continuation at the call point of `choice` will be resumed `n` times. This is achieved by using the sequentiality of call-by-value.

We also need an operator `fail` that is called when no solution can be found, to realize backtracking. Its definition is simple:

```
def fail () = abort "no (more) answers"
```

This will return control to the last choice point.

Using these two constructs we can write the body of the program in direct style, calling `choice` at branching points and `fail` at dead ends. A close analogy to `choice` and `fail` are the `fork()` and `exit()` system calls in Unix-like operating systems, corresponding to the well-known interpretation of a nondeterministic automaton as a collection of (non-communicating) parallel processes.

We can now define `pythagorean-triple`, that will return 3 integers `x`, `y` and `z`, such that $x^2 + y^2 = z^2$:

```
def pythagorean-triple max =   -- int -> int*int*int
    let x = choice max,  y = choice max,  z = choice max
    in if x*x + y*y = z*z
       then (x,y,z)
       else fail()
```

`pythagorean-triple` will return several times with the different results. To view them one by one, we can print them as they are generated. As is expected:

```
print (pythagorean-triple 5) ⇒ (3,4,5) (4,3,5) ⇒ "no (more) answers"
```

which separates the numeric output and the result. It can be noted that using the translator, program execution is quite fast. The above example, but with numbers up to 25 (16 solutions among 15625 triples) took 3 CPU seconds on a Vax 11/785 using Chez Scheme.

The two procedures `choice` and `fail` defined above are not limited to just generation of Pythagorean triples. In fact, they could form the base of a comprehensive package for nondeterministic programming in functional languages, for applications such as simulating nondeterministic automata, branch-and-bound problem solving, etc.

3.4 A new approach to function abstraction and composition

The present approach renews functions as values; either with a control abstraction:

```
reset(f (shift k in k))
```

or with the equivalent function abstraction:

```
fn x => reset (f x)
```

Similarly it leads to a new view of function composition; either by abstracting control:

```
compose f g ≡ reset(f (g (shift k in k)))
```

or equivalently by functional abstraction:

```
compose f g ≡ fn x => reset (f (g x))
```

It can be noted that a control abstraction affects a context, in the same sense that a higher-order function affects the environment. This justifies the two occurrences of `reset` in the functional abstractions above.

As an application we can define the (curried) function `append` that performs all its recursive calls and then abstracts control to return a functional value:

```
def append x =   -- List(A) -> List(A) -> List(A)
    letrec aux x =
           if x = []
           then shift k in k
           else (hd x) :: aux (tl x)
    in reset (aux x)
```

Applying `append` to a list x will return a control abstraction that, when applied to another list y, will install the context prefixing x to y and return the new list.

3.5 An applicative-order fixed point operator

It follows from the definition of Scheme's `call/cc` that evaluation of the following expression does not terminate:

```
(let ((k (call/cc (lambda (c) c))))
    (k k))
```

The reason why it loops is that the captured continuation binds a value to `k` and evaluates the body of the let-expression. In this body, the captured continuation is restored and passed the same continuation – hence the loop.

With `shift` in the language, there is a major difference: we *can* compose continuations. Therefore all we need is to insert around each unfolding a function to unfold recursively, and to add an η-redex for the sake of applicative-orderness, to obtain an applicative-order fixed point operator:

```
def fix-0 f =
    reset (let x = shift c in c c
           in f (fn a => x x a))
```

which is the control counterpart to Curry's fixed point combinator. Indeed, translating `fix-0` into continuation-composing style yields the same as converting Curry's fixed point combinator.

If we apply Böhm's derivation [Böhm 66], which consists of applying a fixed point combinator to the combinator


```
fn y => fn f => f (y f)
```

to get a new combinator, we obtain the control counterpart to Turing's fixed point combinator:

```
def fix-1 =
    reset (let x = shift c in c c
           in fn g => g (fn a => x x g a))
```

Again, the conversion of this into continuation-composing style gives the same as converting the original combinator.

3.6 Flattening a tree into a list

The point of this section is to illustrate how to simulate a two-argument function by abstracting control. The story goes in successive steps.

First, here is the obvious definition that uses `append`:

```
def flatten-0 t =
    if t = empty-tree
    then []
    else if node? t
         then [t]
         else append (flatten-0 (left t)) (flatten-0 (right t))
```

Introducing an accumulator with the invariant `aux-1 t a == append (flatten-0 t) a` yields:

```
def flatten-1 t =
    letrec aux-1 t a =
           if t = empty-tree
           then a
           else if node? t
                then t :: a
                else aux-1 (left t) (aux-1 (right t) a)
    in aux-1 t []
```

Rewriting this definition using control operators instead of function abstractions yields:

```
def flatten-2 t =
    letrec aux-2 t =
           reset (if t = empty-tree
                  then shift a in a
                  else if node? t
                       then t :: (shift a in a)
                       else aux-2 (left t) (aux-2 (right t) (shift a in a)))
    in aux-2 t []
```

3.7 Sharing control vs. sharing data

[Danvy 89] presents many other examples of using control operations.

4 Formal Translation to Continuation-Passing Style

In the following, we develop on the process of mechanically translating shift/reset to higher-order functions. A program converted in this way can then be compiled directly to machine code, using an existing compiler. This gives a large increase in efficiency compared to an interpretative implementation derived directly from the denotational semantics of section 1. Furthermore, the translator

gives a realistic example of an application of shift/reset to a non-trivial problem. To simplify matters, we consider only first-order (uncurried) functions here, but use the same syntax as in the rest of the article.

The delimited context of a subexpression is defined as the surrounding program text up to (but not including) the innermost enclosing square brackets. Using our notation for contexts, the following rewrite rules give a very concise specification of the translation process:

$$
\begin{array}{rcl}
[\mathbf{C}[\overline{\texttt{C}}]] & \Rightarrow & [\mathbf{C}[\texttt{C}]] \\
[\mathbf{C}[\overline{\texttt{I}}]] & \Rightarrow & [\mathbf{C}[\texttt{I}]] \\
[\mathbf{C}[\overline{\texttt{if E}_0\ \texttt{then E}_1\ \texttt{else E}_2}]] & \Rightarrow & [\texttt{if } \overline{\texttt{E}_0} \texttt{ then } [\mathbf{C}[\overline{\texttt{E}_1}]] \texttt{ else } [\mathbf{C}[\overline{\texttt{E}_2}]]] \\
[\mathbf{C}[\overline{\texttt{def f x}_1 \ldots \texttt{x}_n = \texttt{E}}]] & \Rightarrow & [\mathbf{C}[\texttt{def f-c x}_1 \ldots \texttt{x}_n \texttt{ c = [c } \overline{\texttt{E}}]] \\
[\mathbf{C}[\overline{\texttt{reset E}}]] & \Rightarrow & [\mathbf{C}[[\overline{\texttt{E}}]]] \\
[\mathbf{C}[\overline{\texttt{shift c in E}}]] & \Rightarrow & [\texttt{let c = (fn x => } \mathbf{C}[\texttt{x}]\texttt{) in } [\overline{\texttt{E}}]] \\
[\mathbf{C}[\overline{\texttt{t E}_1 \ldots \texttt{E}_n}]] & \Rightarrow & [\mathbf{C}[\texttt{t } \overline{\texttt{E}_1} \ldots \overline{\texttt{E}_n}]], \text{if t is a trivial function} \\
[\mathbf{C}[\overline{\texttt{f E}_1 \ldots \texttt{E}_n}]] & \Rightarrow & [\texttt{f-c } \overline{\texttt{E}_1} \ldots \overline{\texttt{E}_n} \texttt{ (fn x => } \mathbf{C}[\texttt{x}]\texttt{)}]
\end{array}
$$

The functions introduced by function application and `shift` are η-reduced whenever possible. The let-expression in the conversion of `shift` can similarly be β-reduced, but this is not done in the version of the translator presented here.

Conversions must be carried out in the usual evaluation order, *i.e.*, call-by-value, with function arguments evaluated left-to-right. Finally, recall that every top level expression has an implicit reset. When context brackets are no longer needed because the expression they enclose has been completely converted, they can be removed. Following these rules, here are a couple of systematic conversions (again, lists will be enclosed in angle brackets for clarity):

```
[def app a b = if a=<> then b else (hd a)::(app (tl a) b)]
[def app-c a b c = [c (if a=<> then b else (hd a)::(app (tl a) b))]]
[def app-c a b c = [if a=<> then [c b] else [c ((hd a)::(app (tl a) b))]]]
[def app-c a b c = [if a=<> then [c b] else [c ((hd a)::(app (tl a) b))]]]
[def app-c a b c = [if a=<> then c b else [c ((hd a)::(app (tl a) b))]]]
[def app-c a b c = [if a=<> then c b else [c ((hd a)::(app (tl a) b))]]]
[def app-c a b c = [if a=<> then c b else [app-c (tl a) b (fn x=>c ((hd a)::x))]]]
[def app-c a b c = [if a=<> then c b else [app-c (tl a) b (fn x=>c ((hd a)::x))]]]
[def app-c a b c = [if a=<> then c b else [app-c (tl a) b (fn x=>c ((hd a)::x))]]]
def app-c a b c = if a=<> then c b else app-c (tl a) b (fn x=>c ((hd a)::x))

[def f x = reset(1+shift k in (k (k x)))]
[def f-c x c = [c reset(1+shift k in (k (k x)))]]
[def f-c x c = [c [1+shift k in (k (k x)))]]]
[def f-c x c = [c [1+shift k in (k (k x)))]]]
[def f-c x c = [c [1+shift k in (k (k x)))]]]
[def f-c x c = [c [let k = (fn x => 1+x) in [k (k x)]]]]
[def f-c x c = [c [let k = (fn x => 1+x) in [k (k x)]]]]
[def f-c x c = [c [let k = (fn x => 1+x) in [k (k x)]]]]
[def f-c x c = [c [let k = (fn x => 1+x) in [k (k x)]]]]
def f-c x c = c (let k = (fn x => 1+x) in k (k x))
```

β-reduces to `def f-c x c = c (1+1+x)`

Using attribute grammar terminology, each subexpression has an inherited attribute $\mathbf{C}[\ ]$, giving its (delimited) context. A natural way to build it is, by abstract interpretation of the subject program, to keep track of the continuation in an extra parameter, *i.e.*, writing the pseudo-interpreter in continuation-passing style, because at certain points (*e.g.*, `if`) we need explicit access to the continuation as a true function.

Using `shift` and `reset`, however, we can write the translator as a simple recursive-descent compiler. Because the translator treats programs as data, it is expressed in a Scheme-like syntax. "Code generation" consists mostly of rearranging existing Scheme code and adding continuation parameters to functions. As this translator uses the new control primitives, it is not immediately executable in Scheme, but "bootstrapping" by hand conversion to continuation-passing style using the above rules is quite simple.

The various cases in the translator correspond very closely to the rewrite rules above. For example, the function for converting `if` reads (paraphrased):

```
def d-evif ("if" E0 "then" E1 "else" E2) =
    shift c in
        "if" (d-eval E0) "then" (reset (c (d-eval E1)))
                         "else" (reset (c (d-eval E2)))
```

Rewrite rules that do not modify the context correspond to the construct `shift c in c` ..., *i.e.*, an identity operation. The left-to-right translation order of function arguments is inherited directly from the translators list constructor: as for all functions, its arguments are evaluated from left to right, unlike in Scheme where the evaluation order is undefined.

It should be noted that apart from eliminating `shift` and `reset` in object programs, the converter performs a full translation to continuation-passing style, obtaining very reasonable target programs containing no new β- nor η-redexes. It is thus a nontrivial example of using shift/reset for a practical purpose.

The Scheme code for a slightly cut-down and simplified version of the translator can be found in appendix 0. It handles only first-order functions and does not perform some natural optimizations. However, it is sufficiently powerful to translate itself, and the actual output of this conversion can be found in appendix 1. This is a directly executable Scheme program with the same behavior as the original converter. The full converter also handles useful but non-essential constructs such as `case`, `let`, and `letrec`, as well as higher-order functions.

A note on the compiled language: evaluation order for function arguments is strictly left to right, as far as flow of control depends on it. However, to obtain good target programs, calls to trivial functions (including continuation functions!) during argument evaluation may be interleaved in an unspecified way. This means that genuine side-effects (such as `print`, assignments, *etc.*) and runtime-errors (*e.g.*, head of empty list) can still occur in any order. To force explicit sequencing, either a function call or `let` must be used as in Scheme.

5 A Type System

In this section we present a polymorphic type inference system that can handle expressions containing `shift` and `reset`. We focus mainly on the new parts, *i.e.*, shift/reset, and do not treat the classical problems with polymorphic let and letrec, structured types, *etc.* [Milner 78]. To obtain the type of an expression it would be possible to first translate it to higher-order functions, as described in the previous section, and then find the type using traditional methods. However, it is also possible to obtain the type directly, using modified inference rules.

Traditionally [Plotkin 81], type inference rules for expressions are written like this:

$$\frac{\rho \vdash \mathtt{E}_0 : \mathit{bool} \quad \rho \vdash \mathtt{E}_1 : \tau \quad \rho \vdash \mathtt{E}_2 : \tau}{\rho \vdash \mathtt{if\ E}_0\ \mathtt{then\ E}_1\ \mathtt{else\ E}_2 : \tau}$$

This reads: if $\mathtt{E}_0$ is of type *bool*, and $\mathtt{E}_1$ and $\mathtt{E}_2$ of the same type τ in the (type) environment ρ, then the type of the conditional expression in ρ is τ. A rule like this can be directly translated into Prolog or a similar language with unification.

However, it turns out that this simple approach is inadequate for correctly inferring the types of expressions containing `shift` and `reset`. For one point, the premises in a rule are unordered (Prolog's fixed search strategy notwithstanding), and thus the rule does not express the fact that the condition must be evaluated before either of the two alternatives. This is quite important, as evaluation of the condition may also modify the context.

Furthermore, as a `shift` may occur inside a function body, any function call may alter the type of the entire expression. Again, there seems to be no way to accurately specify this kind of global effects in a traditional type system.

To overcome these problems, we must recognize the fact that, just as an expression with free variables can only be given a type relative to an environment, an expression with a "free" (*i.e.*, not syntactically enclosed by a `reset`) `shift` needs a context (represented by a continuation function) to determine its type.

Thus, the basic type relation is "if `E` is evaluated in a context represented by a function from τ to α, the type of the result will be β" As contexts are seldom captured, it is more convenient to express this as "In a context where the (original) result type was α, the type of `E` is τ, and the (new) type of the result will be β." Thus, we write the types like this:

$$\rho, \alpha \vdash \mathtt{E} : \tau, \beta$$

Here, α and β are the old and new result types. If the expression `E` does not modify its context, the two will be identical. In particular, the types of constants are given by axioms like these:

$$\rho, \alpha \vdash \mathtt{true} : \mathit{bool}, \alpha \qquad \rho, \alpha \vdash \mathtt{3} : \mathit{int}, \alpha \qquad \rho, \alpha \vdash \mathtt{"abc"} : \mathit{string}, \alpha$$

As an introduction to the new notation, let us consider the modified form of a simple inference rule: the type of an equality test[5]. This operation takes two operands of the same type (leaving aside the problem of equality for functions), and returns a boolean result. Furthermore, the left argument is always evaluated first. We can write the inference rule like this:

$$\frac{\rho, \delta \vdash \mathtt{E}_1 : \tau, \beta \quad \rho, \alpha \vdash \mathtt{E}_2 : \tau, \delta}{\rho, \alpha \vdash \mathtt{E}_1\ \mathtt{=}\ \mathtt{E}_2 : \mathit{bool}, \beta}$$

This rule can be derived immediately from the semantic equation of an equality test, by decorating it with type information:

$$\mathcal{E}[\![\mathtt{E}_1\ \mathtt{=}\ \mathtt{E}_2]\!]\rho\kappa_{\mathit{bool}\to\alpha} \quad = \quad (\mathcal{E}[\![\mathtt{E}_1]\!]\rho\{\lambda s_\tau.(\mathcal{E}[\![\mathtt{E}_2]\!]\rho\{\lambda t_\tau.(\kappa(s=t))_\alpha\}_{\tau\to\alpha})_\delta\}_{\tau\to\delta})_\beta$$

Rules of this form also translate directly into Prolog, by adding two extra arguments representing the context. The type of an entire (top-level) expression can be inferred using the bridge rule from ordinary to extended types:

[5] In the actual inference system, all built-in (primitive) operators, structure constructors *etc.* are handled in a slightly different way, using only one inference rule and a lookup table, instead of an explicit rule for each operator.

$$\frac{\rho_{\mathit{init}}, \tau \vdash \mathtt{E} : \tau, \alpha}{\vdash \mathtt{E} : \alpha}$$

This expresses the fact that initially, the type of the result will be the ordinary type of `E`, *i.e.*, τ. This corresponds to using the identity function as a continuation. However, `E` may change the context, so the final type α is not yet known.

Let us now consider, for example, the problem of inferring the type of an expression like "`37 = abort(42)`". In this case, the type of the entire computation is *int* even though the topmost operator returns a boolean result. We can write the specification for `abort` like this:

$$\frac{\rho, \sigma \vdash \mathtt{E} : \sigma, \beta}{\rho, \alpha \vdash \mathtt{abort(E)} : \tau, \beta}$$

One may note that τ and α occur only once in the rule, *i.e.*, an `abort` may occur in any type context.

Describing `reset` goes similarly. Here we type-inference the subexpression in a new, empty context, and use the result in the original one, protecting it from any modifications:

$$\frac{\rho, \sigma \vdash \mathtt{E} : \sigma, \tau}{\rho, \alpha \vdash \mathtt{reset(E)} : \tau, \alpha}$$

To process the conditional form, in contrast to ML, it is no longer sufficient to treat `if` as just a ternary operator with respect to typing. The semantics of `if` requires the test to be evaluated first, and further that the two branches have identical (or rather, compatible) types in all aspects, so that no matter which one is taken, the result will be of the same type:

$$\frac{\rho, \delta \vdash \mathtt{E}_0 : \mathit{bool}, \beta \quad \rho, \alpha \vdash \mathtt{E}_1 : \tau, \delta \quad \rho, \alpha \vdash \mathtt{E}_2 : \tau, \delta}{\rho, \alpha \vdash \mathtt{if\ E}_0\ \mathtt{then\ E}_1\ \mathtt{else\ E}_2 : \tau, \beta}$$

Here are some type-correct conditional expressions to illustrate these various points. The given types are only valid if the expressions occur at the top level (or as arguments to `reset`). The third example uses `shift`, although it has not been described yet.

`1 = (if 2 = 3 then 4 else abort true)`: *bool*
`if (abort 7) then "a" else "b"`: *int*
`if (shift f in [(f true)]) then 4 else 5`: *list*(*int*)

Finally, let us consider functions. In a ML-like system, a function only has a domain and co-domain. When functions can use `shift` and `reset`, however, the types get slightly more complicated. Recall that the translator turns each function "`f`" into a new function "`f-c`" with an extra parameter, the continuation, which represents the context at the point of call. Thus, we can associate four types with a converted function: its own domain and co-domain and those of its continuation parameter. In true continuation-passing style, the co-domains of both the function and the continuation are the domain of Answers, while the domain of the continuation is the co-domain of the original function. With `shift` and `reset` in the language, however, they may all be different.

We write these functional types in the following way:

Domain-F/CoDomain-C → Domain-C/CoDomain-F

The reason for this notation is that, in the normal case, a function takes an argument of type Domain-F and calls the continuation with (that is, "returns") a value of type Domain-C. However, a function also has a control aspect, *i.e.*, when called in a context where the final answer is of type CoDomain-C, the new final answer will be of type CoDomain-F.

A simple function (one that does not use `shift`) is polymorphic in CoDomain-C = CoDomain-F (since it does not alter the context). Here are some examples of various cases. They are given with their translation to continuation-passing style.

$$\texttt{fn x => (x = 1)}\colon\ int/A \to bool/A$$
$$\texttt{fn x => fn c => c(x = 1)}\colon\ int \to (bool \to A) \to A$$

Here, the two occurrences of A in the first line respectively match the two in the second line. This is the usual pattern for functions which do not modify the context: The final result type is preserved.

The following illustrates a function which aborts:

$$\texttt{fn x => abort(x = 1)}\colon\ int/A \to B/bool$$
$$\texttt{fn x => fn c => (x = 1)}\colon\ int \to (B \to A) \to bool$$

Finally, here is a function which shifts:

$$\texttt{fn x => shift c in (c x) = 1}\colon\ A/int \to A/bool$$
$$\texttt{fn x => fn c => (c x) = 1}\colon\ A \to (A \to int) \to bool$$

We can now write the inference rule of an abstraction. Here, the type of a function body is inferred in both a new environment and a new context, but the functional expression itself does not alter its context:

$$\frac{[\mathtt{x} \mapsto \sigma]\rho, \alpha \vdash \mathtt{E} : \tau, \beta}{\rho, \delta \vdash \texttt{fn x => E} : (\sigma/\alpha \to \tau/\beta), \delta}$$

With this notation, the inference rule for `shift` becomes relatively simple. Recalling that a continuation function never modifies the context:

$$\frac{[\mathtt{f} \mapsto (\tau/\delta \to \alpha/\delta)]\rho, \sigma \vdash \mathtt{E} : \sigma, \beta}{\rho, \alpha \vdash \texttt{shift f in E} : \tau, \beta}$$

As one would expect, this rule is quite similar to the one for `abort`, except that the types τ and α are not discarded, but become parts of the continuation function type.

The only missing part now is function application. As with a conditional, we first evaluate the function and its argument (in that order), and then perform the actual application:

$$\frac{\rho, \delta \vdash \mathtt{F} : (\sigma/\alpha \to \tau/\varepsilon), \beta \qquad \rho, \varepsilon \vdash \mathtt{E} : \sigma, \delta}{\rho, \alpha \vdash \texttt{F E} : \tau, \beta}$$

We may note that the type system just described is derived from a semantics which specifies left-to-right evaluation of operator and function arguments. This means that the types of the following two functions will be different:

```
def foo f g = ((f 1) = (g 1))
def bar f g = ((g 1) = (f 1))
```


In a semantics where argument evaluation order is explicitly unspecified, we would prefer `foo` and `bar` to have the same type. This can be done by strengthening the rules slightly to disallow expressions whose *type*[6] is determined by a specific evaluation strategy. Thus, expressions like `abort(1) = true` and `5 = abort("x")` would still be type-correct (with types *int* and *string* respectively), but *e.g.* `abort(1) = abort(false)` would not be.

We write the modified type inference rules by checking that every argument evaluation order induces the same type. For example, the rule for an equality test becomes:

$$\frac{\rho, \delta \vdash \mathtt{E_1} : \tau, \beta \qquad \rho, \alpha \vdash \mathtt{E_2} : \tau, \delta \qquad\qquad \rho, \alpha \vdash \mathtt{E_1} : \tau, \delta' \qquad \rho, \delta' \vdash \mathtt{E_2} : \tau, \beta}{\rho, \alpha \vdash \mathtt{E_1\ =\ E_2} : bool, \beta}$$

We note that this modification only applies to language constructs where a specific evaluation order is not defined by the semantics. For example, the condition of an `if`-expression is still evaluated before any of the two branches, so that the rule for `if` does not change.

6 Comparison with Related Work

6.1 The λ_c-calculus

The idea of getting control over the whole continuation rather than noting a part of it, and to compose these continuations, originates in Matthias Felleisen's Ph.D. thesis [Felleisen 87] and in a number of related articles. A common pattern to composing continuations is the concatenation of their constituents – *e.g.*, the control string of a CEK-machine [Felleisen 88], and this approach culminates in [Felleisen *et al.* 88] with an algebraic framework where continuations are defined as sequences of frames.

The present work contributes to that endeavor by devising an expression language and its denotational semantics, and adding a type system. More generally, it identifies continuations as a functional abstraction of delimited contexts, and defines a semantics where the embedding contexts are represented with one new argument. This approach induces a lexical scope, which contrasts with the dynamic one previously obtained: when two continuations are composed, the computation is continued up to the last lexical reset (either direct or inherited from the last continuation that has been applied – which is the core of lexical scope) and returns at the point of application as any ordinary function, and the computation continues. Under a dynamic scope, that reset is erased since continuations are appended, and the resulting continuation would be captured as one object. To put it precisely, we abstract a continuation κ as:

$$inFun(\lambda\, \kappa'\, \gamma\, v\, .\, \kappa\ (\kappa'\ \gamma)\ v)$$

using the extra parameter γ, and this contrasts with abstracting a continuation κ as:

$$inFun(\lambda\, \kappa'\, v\, .\, (\kappa' \oplus \kappa)\ v)$$

as in [Felleisen *et al.* 88], where continuations are defined as sequences of frames and their composition as the concatenation of these sequences, and a prompt marks the end of a sequence.

Here is a concrete example where these two approaches give two distinct results:

```
let f n = shift k in n, g x = shift c in add1(c x) in reset(f (g 2))
[f (g 2)]
[f (shift c in add1(c 2))]
[add1(c 2)] where c ~ [f [ ]]
```

[6] Naturally, we cannot in general determine statically whether the *value* of the expression depends on evaluation order.

```
[add1 [f 2]]
[add1 [shift k in 2]]
[add1 [2]] where k ~ [[ ]]
[add1 2]
[3]
3
```

The expression `f (g 2)` evaluates to 3 in our model but according to [Felleisen *et al.* 88] it rewrites to 2. The reason is that with two continuations $\kappa' \simeq add1$ and $\kappa = \kappa_{init}$ we get:

$$\mathcal{E}\,[\![\texttt{shift k in n}]\!]\,([\mathcal{I}[\![\texttt{n}]\!] \mapsto \mathcal{C}[\![2]\!]]\rho_{init})\,\kappa\,(\kappa'\,\gamma_{init}) \Rightarrow \kappa'\,\gamma_{init}\,2 \Rightarrow 3$$

which contrasts with:

$$\mathcal{E}\,[\![\texttt{control k in n}]\!]\,([\mathcal{I}[\![\texttt{n}]\!] \mapsto \mathcal{C}[\![2]\!]]\rho_{init})\,(\kappa' \circ \kappa) \Rightarrow \kappa_{init}\,2 \Rightarrow 2$$

since κ' and κ have been composed and appended as one object, delimited by the prompt of κ'. To say it in other words, when a continuation is applied, it creates a new context in our formalism while it does not in [Felleisen *et al.* 88].

This compares accurately with an environment extension, that can be made lexically from a closure environment or dynamically from the current environment. This dynamic scope could be better seen if continuations were tagged with exception labels as in ML.

Our more lexical approach to contexts seems to induce a limitation: one cannot shift the composition of two continuations. However, since continuations are abstracted as true functions, their composition can as well be abstracted explicitly with: `fn a => (c (k a))`, assuming `c` and `k` to be bound to two continuations.

Finally this lexical approach to contexts has made it possible to infer statically a type for any expression, as described in section 5.

A point in [Felleisen *et al.* 88] is the consideration of stack-implementability. This seems a bit premature. Experience has shown that a first-order language can be conveniently described and implemented with a stack. Introducing higher-order functions in a lexical scope has proven not to be stack-implementable, in general. These aspects concern the environment. It is not surprising that describing and implementing higher-order forms of control with a lexical scope is not stack-implementable either, in general. However, first-order programs are still stack-implementable in the presence of shift/reset, as demonstrated by the translation to extended continuation-passing style, where functions are never returned but only passed as parameters (*i.e.*, there are only downward funargs [van Wijngaarden 66] [Fischer 72]). For example, we have successfully transliterated into Pascal the example `foo` of the introduction.

6.2 GL

[Johnson & Duggan 88] reports on the programming language GL and its "partial" continuations. They can be composed with a *fork* operator. GL is described using denotational semantics too. However it is untyped and imperative with a store, whereas the present approach proposes a typed expression language and a formal definition of partial continuations. GL was first introduced in [Johnson 87], where applying a partial continuation did create a new context. Notably, applying a partial continuation does not create any new context in [Johnson & Duggan 88].

6.3 Reflective towers

The idea of having a dedicated parameter for embedding contexts originates in a previous work on procedural reflection [Danvy & Malmkjær 88], where the context parameter stood for the meta-continuation [Wand & Friedman 88]. The present investigation owes to reflection, by giving unlimited access to continuations (but neither expressions nor environments). This connects shifting with reification since it gives access to the current continuation, and resetting with reflection, since computation starts in a new context. Further, defining the initial context as the fixed point of an elementary context makes it possible to reify indefinitely and thus realizes the illusion of having infinitely many levels as offered in a reflective tower.

Actually, the work presented here bears a close relationship with reflective towers, since it investigates the reification of continuations. In the Blond dialect where reifiers are expressed as μ-abstractions [Danvy & Malmkjær 89], the two main constructions of the present paper can be expressed by the two following reifiers:

```
(common-define reset
    (mu (r k e)
        (k (meaning e r (lambda (x) x)))))

(common-define shift
    (mu (r k c e)
        (meaning e
                 (extend-reified-environment (list c) (list k) r)
                 (lambda (x) x))))
```

However, there is no implicit `reset` at the top level in the reflective tower.

6.4 Converting to continuation-passing style for compiling

Converting to continuation-passing style as part of the compilation process was introduced in [Steele 78]. Since then converting to continuation-passing style has been used for compiling functional programs with first-class continuations [Kranz *et al.* 86] [Appel & Jim 89]. Steele's work includes a converter from Scheme programs in direct style to continuation-passing style. This converter is in continuation-passing style but does not compose the continuation part, although it does reset it here and there. As a result, converted programs contain numerous β-redexes that have to be processed at a later stage. In contrast, our converter does compose continuations. Translated programs have no redexes left and built-in functions are left intact. It does not use side-effects and thus is functional and even self-applicable. This program makes us conjecture that composing continuations could optimize in some sense its corresponding class of tree processing.

6.5 Non-determinism

Finally, composing continuations provides a surprisingly direct insight into non-determinism, by backtracking as in section 3. We are currently exploring its connections with resumption semantics [Schmidt 86].

Conclusion and Issues

This report investigates the functional abstraction of delimited contexts $\mathbf{C}[\ ]$ rather than unlimited ones $\mathbf{C}[\ ]$, as continuations. Because these continuations are abstracted as true functions, they can be composed. We have devised a simple expression language, its congruence with traditional continuation semantics and its type system as an illustration. The type inferencer is expressed in Prolog. Programs are desugared by a YACC-generated parser and interpreted by a Scheme program. We have also set up a translator from direct style to extended continuation-passing style to offer two different angles for viewing a program, as well as an efficiency improvement.

Writing this article, we aimed to contribute to a better understanding of composable continuations. Presently, we are getting closer to seeing whether they allow to reduce the algorithm complexity of some kinds of tree processing for abstract interpretation. The general idea concerns iterating over a tree to find a fixed point: if the treatment for each node only depends on its parent node, *i.e.*, on its context, then composing the continuation at that point would avoid re-traversing all the way down to that node at the next iteration. As one may note, it is the same idea that led to procedures: factorizing code. Here, we factorize processing. Another promising aspect is to define formally the duality of shift/reset and call/return in a procedural language.

A last word: traditionally, the way to access the continuation in a functional program is to rewrite it in continuation-passing style. In many cases, the functionality of ML's exceptions or Scheme's `call-with-current-continuation` is sufficient to achieve the desired effect without such a rewrite. However, continuations as provided in Scheme do not use the full power of a continuation-passing style program, namely that it becomes possible to install any function where a continuation is expected and to compose it at some other place in the program. With `shift` and `reset`, exactly this becomes possible, so that all the benefits of continuation-passing style may be used, while keeping the program in direct style: continuations can be explicited when wanted (and not everywhere as with continuation-passing style) and up to a delimited context (and not to the global one as in Scheme). They provide a functional abstraction of control.

References

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

[Böhm 66] Carl Böhm: *The CUCH as a Formal and Description Language.* In Formal Language: Description Languages for Computer Programming, T. B. Steel Jr. (ed.), pp. 179–197, North-Holland (1966).

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

[Danvy & Malmkjær 89] Olivier Danvy and Karoline Malmkjær: *Aspects of Computational Reflection in a Programming Language* (1989). (Working paper).

[Danvy 89] Olivier Danvy: *Programming with Tighter Control.* BIGRE special issue: Putting the Scheme Language to Work (May 1989). (To appear).

[Dijkstra 60] Edger W. Dijkstra: *Recursive Programming.* In Programming Systems and Languages, Saul Rosen (ed.), chapter 3C, pp. 221–227, McGraw-Hill, New York (1960).

[Felleisen 87] Matthias Felleisen: *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana (August 1987).

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

[Felleisen *et al.* 87] Matthias Felleisen, Daniel P. Friedman, Bruce Duba, and John Merrill: *Beyond Continuations.* Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana (February 1987).

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

[Fischer 72] Michael J. Fischer: *Lambda Calculus Schemata.* In 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).

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

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

[Kranz *et al.* 86] David Kranz *et al.*: *ORBIT: An Optimizing Compiler for Scheme.* In proceedings of the SIGPLAN '86 Symposium on Compiler Construction, pp. 219–233, Palo Alto, California (June 1986).

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

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

[Milner 78] Robin Milner: *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences, 17:348–375 (1978).

[Plotkin 81] Gordon Plotkin: *A Structural Approach to Operational Semantics.* Technical Report FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (September 1981).

[Rees & Clinger 86] Jonathan Rees and William Clinger (eds.): *Revised*^{3} *Report on the Algorithmic Language Scheme.* SIGPLAN Notices, 21(12):37–79 (December 1986).

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

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

[Schmidt 85] David A. Schmidt: *Detecting Global Variables in Denotational Definitions.* ACM Transactions on Programming Languages and Systems, 7(2):299–310 (April 1985).

[Schmidt 86] David A. Schmidt: *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc. (1986).

[Sethi & Tang 78] Ravi Sethi and Adrian Tang: *Transforming Direct into Continuation Semantics for a Simple Imperative Language* (April 1978). (Unpublished manuscript).

[Sethi & Tang 80] Ravi Sethi and Adrian Tang: *Constructing Call-by-Value Continuation Semantics.* Journal of the ACM, 27(3):580–597 (July 1980).

[Steele & Sussman 76] Guy L. Steele, Jr. and Gerald J. Sussman: *Lambda, the Ultimate Imperative.* AI Memo 353, MIT-AIL, Cambridge, Massachusetts (March 1976).

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

[Stoy 77] Joseph E. Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press (1977).

[Strachey & Wadsworth 74] Christopher Strachey and 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] A. van Wijngaarden: *Recursive Definition of Syntax and Semantics.* In Formal Languages: Description Languages for Computer Programming, T. B. Steel Jr. (ed.), pp. 13–24, North-Holland (1966).

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

[Wand 80] Mitchell Wand: *Continuation-Based Program Transformation Strategies.* Journal of the ACM, 27(1):164–180 (January 1980).

A The Converter

This appendix contains a listing of a minimal selfapplicable version of the CPS converter mentioned in section 4. For conciseness it is expressed in and operates on programs in a Scheme-based syntax, rather than the ML-like notation used in the rest of this report.

```
(define (d-eval e r)
  (if (or (atom? e) (equal? (car e) 'quote))
      e
      (if (equal? (car e) 'if)
          (d-evcon (cadr e) (caddr e) (cadddr e) r)
          (if (equal? (car e) 'define)
              (d-evdef (cadr e) (caddr e) (gensym "k") r)
              (if (equal? (car e) 'shift)
                  (d-evsft (cadr e) (caddr e) (gensym "x") r)
                  (if (equal? (car e) 'reset)
                      (reset (d-eval (cadr e) r))
                      (d-apply (car e)
                               (d-evlis (cdr e) r)
                               (gensym "x")
                               r)))))))

(define (d-evlis l r)
    (if (null? l)
        '()
        (cons (d-eval (car l) r) (d-evlis (cdr l) r))))

(define (d-evcon e e1 e2 r)
    (shift c
        (list 'if
              (d-eval e r)
              (reset (c (d-eval e1 r)))
              (reset (c (d-eval e2 r))))))

(define (d-evdef f e k r)
    (list 'define
         (cons (concat (car f) "-c") (append (cdr f) (list k)))
         (reset (list k (d-eval e r)))))

(define (d-evsft f e x r)
  (shift c
     (list 'let
           (list (list f (eta (list x) (c x))))
           (reset (d-eval e r)))))

(define (d-apply f l x r)
    (if (member f r)
        (shift c
           (cons (concat f "-c")
                 (append l (list (eta (list x) (c x))))))
        (cons f l)))

(define (eta vl e)
    (if (and (pair? e) (equal? (cdr e) vl))
        (car e)
        (list 'lambda vl e)))

(define (list-def p)
   (if (null? p)
       '()
       (if (equal? (caar p) 'define)
           (cons (car (cadar p)) (list-def (cdr p)))
           (list-def (cdr p)))))

(define (conv-list l r)
   (if (null? l)
       '()
       (cons (reset (d-eval (car l) r)) (conv-list (cdr l) r))))

(define (conv-prog p) (conv-list p (list-def p)))
```

B The Converter Converted

This appendix contains the actual output of the converter applied to itself. While somewhat less readable, it is a pure Scheme program equivalent to the direct-style converter.

```
(define (d-eval-c e r k0)
```

```
   (if (or (atom? e) (equal? (car e) 'quote))
       (k0 e)
       (if (equal? (car e) 'if)
           (d-evcon-c (cadr e) (caddr e) (cadddr e) r k0)
           (if (equal? (car e) 'define)
               (d-evdef-c (cadr e) (caddr e) (gensym "k") r k0)
               (if (equal? (car e) 'shift)
                   (d-evsft-c (cadr e) (caddr e) (gensym "x") r k0)
                   (if (equal? (car e) 'reset)
                       (k0 (d-eval-c (cadr e) r (lambda (x26) x26)))
                       (d-evlis-c
                          (cdr e)
                          r
                          (lambda (x29)
                             (d-apply-c
                                (car e)
                                x29
                                (gensym "x")
                                r
                                k0)))))))))
(define (d-evlis-c l r k32)
   (if (null? l)
       (k32 '())
       (d-eval-c
          (car l)
          r
          (lambda (x35)
             (d-evlis-c
                (cdr l)
                r
                (lambda (x37) (k32 (cons x35 x37))))))))
(define (d-evcon-c e e1 e2 r k39)
   (let ((c k39))
      (d-eval-c
         e
         r
         (lambda (x41)
            (list 'if x41 (d-eval-c e1 r c) (d-eval-c e2 r c))))))
(define (d-evdef-c f e k r k47)
   (k47 (list 'define
              (cons (concat (car f) "-c") (append (cdr f) (list k)))
              (d-eval-c e r (lambda (x54) (list k x54))))))
(define (d-evsft-c f e x r k57)
   (let ((c k57))
      (eta-c
         (list x)
         (c x)
         (lambda (x61)
            (list 'let
                  (list (list f x61))
                  (d-eval-c e r (lambda (x64) x64)))))))
(define (d-apply-c f l x r k66)
   (if (member f r)
       (let ((c k66))
          (eta-c
             (list x)
             (c x)
             (lambda (x72)
                (cons (concat f "-c") (append l (list x72))))))
       (k66 (cons f l))))
(define (eta-c vl e k77)
   (if (and (pair? e) (equal? (cdr e) vl))
       (k77 (car e))
       (k77 (list 'lambda vl e))))
(define (list-def-c p k84)
   (if (null? p)
       (k84 '())
       (if (equal? (caar p) 'define)
           (list-def-c
              (cdr p)
              (lambda (x91) (k84 (cons (car (cadar p)) x91))))
           (list-def-c (cdr p) k84))))
(define (conv-list-c l r k95)
   (if (null? l)
       (k95 '())
       (conv-list-c
          (cdr l)
          r
          (lambda (x100)
             (k95 (cons (d-eval-c (car l) r (lambda (x98) x98))
                        x100))))))
(define (conv-prog-c p k102)
   (list-def-c p (lambda (x103) (conv-list-c p x103 k102))))
```


C Modifying the denotational semantics

The semantics of section 1 can be slightly modified to access the embedding context at shifting time rather than merely resetting it. This makes it possible to get control over embedding contexts by shifting repeatedly. This section illustrates the idea and points out the corresponding modifications to the semantics, the congruence and the type system of the language.

For example, in `reset(3 + reset(4 * shift k in shift c in E))`, `k` is bound to $\lambda n\,.\,4 \times n$ and `c` is bound to $\lambda n\,.\,3 + n$.
If `E` = `c(k 1)` then the result is $3 + (4 \times 1) = 7$ *i.e.*, $[\mathbf{C_c}[\mathbf{C_k}[1]]]$
If `E` = `k(c 1)` then the result is $4 \times (3 + 1) = 16$ *i.e.*, $[\mathbf{C_k}[\mathbf{C_c}[1]]]$

The modification requires redefining the semantic algebra of contexts, the initial continuation, shifting and resetting. It makes it clear that we push and pop continuations on and off γ, at reset time and at shift time, respectively.

$\gamma \in Sctx = Ctn^*$
$\gamma_{init} = ()$
$\kappa_{init} = propagate = \lambda\,\gamma\,a\,.\;\; cases\ \gamma\ of\ () \to a \parallel (\kappa, \gamma') \to \kappa\,\gamma'\,a\ end$
$\mathcal{E}\,[\![\texttt{reset E}]\!]\,\rho\,\kappa\,\gamma = \mathcal{E}\,[\![\texttt{E}]\!]\,\rho\,propagate\,(\kappa, \gamma)$
$\mathcal{E}\,[\![\texttt{shift I in E}]\!]\,\rho\,\kappa\,(\kappa'', \gamma) = \mathcal{E}\,[\![\texttt{E}]\!]\,([\mathcal{I}\,[\![\texttt{I}]\!] \mapsto inFun(\lambda\,v\,\kappa'\,\gamma\,.\,\kappa\,(\kappa', \gamma)\,v)]\rho)\,\kappa''\,\gamma$

As one can note, the operator `reset` is no longer idempotent, *i.e.*, `reset (reset E)` is not equivalent to `reset E`.

The type system needs a representation of the context, as a stack of embedding continuations. The initial stack is empty. The conversion to higher-order functions changes accordingly.