Functional Derivation of a Virtual Machine for Delimited Continuations

Kenichi Asai   Arisa Kitani
Ochanomizu University
{asai,kitani.arisa}@is.ocha.ac.jp

Abstract

This paper connects the definitional interpreter for the λ-calculus extended with delimited continuation constructs, shift and reset, with a compiler and a low-level virtual machine that copies a part of a data stack to implement delimited continuations. Following the functional derivation approach proposed and popularized by Danvy, we interrelate the two implementations via a series of meaning-preserving program transformations whose validity is independently known. As a result, this work formally establishes the correctness of a compiler and a low-level stack-copying implementation of delimited continuations. In particular, the resulting virtual machine properly models when to store return addresses into a data stack and which part of a data stack to copy. To our knowledge, this work is the first to prove correctness of such low-level features of delimited continuations. It also shows that the functional derivation approach is equally applicable to establish correctness of low-level implementations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures; D.3.4 [*Programming Languages*]: Processors—Compilers, Interpreters; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation

General Terms Languages, Theory

Keywords Functional Derivation, Delimited Continuation, Virtual Machine, Defunctionalization, CPS Transformation

1. Introduction

First-class continuations play an important role in controlling flow of execution in a flexible manner without converting a program into continuation-passing style (CPS). In particular, delimited continuations [9] find many applications due to its clear semantics through CPS transformation, such as non-deterministic programming [9], typed printf [4], dynamic code generation [16], and let-insertion in partial evaluation [3, 19, 26].

Background and related work As delimited continuations become used in many places, various direct implementations of delimited continuations are proposed that copy a part of a data stack. Gasbichler and Sperber [13] implemented delimited continuations via stack copying on Scheme 48 bytecode. Masuko and Asai [20] implemented delimited continuations at the PowerPC assembly level in the MinCaml compiler framework [25]. Kiselyov [17] describes a general implementation method to introduce delimited continuations into a system equipped with stack overflow and exceptions.

However, the correctness of these approaches relies on informal arguments backed up with practical experience. As such, it is difficult to discuss correctness of low-level implementations such as the one in an assembly language. Although Dybvig, Peyton Jones, and Sabry [12] showed operational and CPS semantics for delimited continuations and prove their equivalence, their formalization is still abstract and does not model, for example, return addresses, that are typically stored in a data stack and copied when continuations are captured. Recently, Rompf, Maier, and Odersky [24] integrated delimited continuations into Scala. While remarkable, their implementation is not suitable for discussing low-level aspects such as stack copying, because delimited continuations are emulated as functions via type-directed, selective CPS transformation.

Our approach To establish correctness of a low-level implementation of delimited continuations, we employ the functional derivation approach proposed and popularized by Danvy [2, 8]. Starting from the definitional interpreter for a language with delimited continuation constructs, we apply a series of meaning-preserving program transformations whose validity is already known. Following Danvy, we use defunctionalization [23], refunctionalization [10], and CPS transformation [22]. In addition, we introduce several other program transformations.

The advantage of this approach is that the correctness of derived implementation follows directly from the correctness of program transformations. This is in contrast to the conventional approach where one had to prove, either by hand or using a proof assistant, that the original implementation and the derived implementation behave the same. Furthermore, since there are already various useful transformations, we can explore the design space of implementations simply by trying one transformation after another. Thus, it is not only used by Danvy's group to derive various virtual machines [1], but also used to derive a compiler and a virtual machine for a multi-stage language that supports backquote and unquote [14].

Biernacka, Biernacki, and Danvy [6] use a functional correspondence to derive an abstract machine for λ-calculus with multi-level delimited continuations. Danvy and Millikin [11] start from Landin's SECD machine with the J operator [18] and derive a family of evaluation functions. However, both of the work deal with a high-level abstract machine that is based on a state transition system rather than a low-level virtual machine that operates on instructions.

Contribution The contribution of this paper is to connect the definitional interpreter for the λ-calculus extended with delimited continuation constructs, shift and reset, with a compiler and a low-level virtual machine that copies a part of a data stack to implement delimited continuations. As a result, this work formally establishes the correctness of a compiler and a low-level stack-copying implementation of delimited continuations.

In the previous work [20], Masuko and the first author presented an invariant on a data stack when continuations are captured and invoked to implement delimited continuations. The virtual machine presented in this paper properly models it, in addition to when to store return addresses into a data stack and which part of a data stack to copy. To our knowledge, this work is the first to prove correctness of such low-level features of delimited continuations. It also shows that the functional derivation approach is equally applicable to establish correctness of low-level implementations.

Prerequisites We assume basic familiarity with CPS transformation [22] and defunctionalization [23]. When functions are transformed into CPS, all the calls become tail calls. Such functions can be regarded as an abstract machine where a state of the machine consists of the arguments of the functions. We will use this fact to derive a virtual machine for delimited continuations.

Organization This paper is organized as follows. In Section 2, we introduce delimited continuation constructs, shift and reset, and show the definitional interpreter for them. In Section 3, we introduce a data stack into the definitional interpreter. It is decomposed into a compiler and a virtual machine in Section 4. The virtual machine is further transformed so that it manipulates a segmented stack in Section 5. The property of the obtained virtual machine is discussed in Section 6. The paper concludes in Section 7.

2. Definitional Interpreter

In this section, we introduce delimited continuation constructs, shift and reset, and show the definitional interpreter for them.

2.1 Shift and Reset

Danvy and Filinski [9] introduced two operators, shift and reset, for capturing the current continuation. As a concrete syntax,

```
shift (fun k -> M)
```

captures the current continuation up to the enclosing reset and binds it to `k` to be used in `M`. The captured continuation is removed from the current continuation: if it is not used in `M`, it is discarded. The context is delimited by reset:

```
reset (fun () -> M)
```

The shift operation executed in `M` captures the continuation only up to this reset expression. The exact semantics of shift and reset is defined in Section 2.3.

For example, consider the following program:

```
1 + reset (fun () ->
             2 + (shift (fun k -> 3 * k (k 4))))
```

The continuation captured by `shift` is up to the enclosing `reset`, namely `2 + []`, which is bound to `k`. Since the captured continuation is removed from the current continuation, the above program is reduced to:

```
1 + reset (fun () -> 3 * (2 + (2 + 4)))
```

and the result becomes 25.

Capturing delimited continuations enables us to have control over the type of the context. Consider the following program:

```
reset (fun () ->
         shift (fun k -> fun x -> k x) ^ "world")
"Hello "
```

The type of the body of `reset` appears to be a string, because the result is a concatenation of something with `"world"`. However, the `shift` operator captures the current continuation, `[] ^ "world"`, binds it to `k`, and returns a function. Thus, the above program is reduced to:

```
reset (fun () -> fun x -> x ^ "world")
"Hello "
```

and then to:

```
(fun x -> x ^ "world")
"Hello "
```

to produce finally `"Hello world"`. With this observation, one can implement typed printf using delimited continuations. Interested readers are invited to read the first author's article [4] which comes with introduction to delimited continuations.

```
type t = Var of string                    (* term *)
       | Lam of string * t | App of t * t
       | Shift of string * t | Reset of t

type xs = string list            (* argument list *)

(* offset : string -> xs -> int *)
let offset x xs =
  let rec loop xs n = match xs with
      [] -> raise UnboundVariable
    | a::rest ->
        if x = a then n else loop rest (n + 1)
  in loop xs 0
```

Figure 0. Syntax of λ-calculus with shift and reset and an auxiliary function `offset`

2.2 Syntax

Figure 0 shows the syntax of the input language we use in this paper. It is a standard λ-calculus extended with shift and reset. The figure also shows an `offset` function that returns the position of a variable `x` in a variable list `xs`. The definition of `offset` is the same throughout this paper.

2.3 Definitional Interpreter

Figure 1 shows the definitional interpreter for the λ-calculus with shift and reset [9]. It is a standard CPS interpreter extended with shift and reset. Shift captures the current continuation `c`, binds it to `x`, and resets the current continuation to empty (*i.e.*, an identity function). Reset delimits the continuation captured by shift by resetting the current continuation to empty and applies the continuation `c` of reset in direct style.

An environment is represented as two lists, a list of names `xs` and a list of values `vs`, instead of more usual association list of pairs of names and values. Thus, a value bound to a variable is obtained by two functions, `offset` and `List.nth`. The latter function returns the n'th element of a given list, starting at index zero. The use of two lists makes it easy to decompose an interpreter into a compiler and a virtual machine. It is called a *binding-time improvement* [15, Chapter 12], a standard technique in the partial evaluation community.

Starting from this interpreter, we derive its low-level implementation via a series of program transformations.

```
type v = VFun of (v -> c -> v)          (* value *)
       | VCont of c
and  c = v -> v                  (* continuation *)

(* f1 : t -> xs -> v list -> c -> v *)
let rec f1 t xs vs c = match t with
    Var(x) -> c (List.nth vs (offset x xs))
  | Lam(x,t) ->
      c (VFun(fun v c' ->
               f1 t (x::xs) (v::vs) c'))
  | App(t0,t1) ->
      f1 t0 xs vs (fun v0 ->
      f1 t1 xs vs (fun v1 ->
      match v0 with
          VFun(f) -> f v1 c
        | VCont(c') -> c (c' v1)))
  | Shift(x,t) ->
      f1 t (x::xs) (VCont(c)::vs) (fun v -> v)
  | Reset(t) -> c (f1 t xs vs (fun v -> v))

(* eval1 : t -> v *)
let eval1 t = f1 t [] [] (fun v -> v)
```

Figure 1. Definitional interpreter for λ-calculus with shift and reset (`eval1`)

3. Stack Introduction

In this section, we aim at introducing a data stack into the definitional interpreter. Instead of returning a value, the resulting interpreter passes around a data stack and results of evaluation are pushed onto the data stack. We show that a data stack is systematically introduced by splitting dynamic data hidden in a continuation chain. To be more concrete, we will modify the representation of a continuation `c` in the following way: defunctionalization (Section 3.1), linearization (Section 3.2), stack extraction (Section 3.3), delinearization (Section 3.4), and refunctionalization (Section 3.5). After defunctionalization and linearization, a data stack will be extracted naturally from the type of continuations. Converting the representation of continuations back to the original higher-order functions, we will obtain a stack-based interpreter.

3.1 Defunctionalization

Figure 2 shows the result of defunctionalizing continuations. To defunctionalize continuations, we first assign a constructor for each continuation in a program. In Figure 1, there are three kinds of continuations: the identity function and two continuations in the `App` case. We assign them the constructors `C0`, `CApp0`, and `CApp1`, respectively, with the arguments being free variables of the corresponding continuations. See the definition of type `c` in Figure 2.

We then replace application of a continuation with a call to a dispatch function (`run_c2`), which describes what to do when the continuation object is called.

The equivalence of `eval1` and `eval2` follows from the correctness of defunctionalization [5, 21].

3.2 Linearization

Figure 3 shows the result of linearizing continuations. The definition of type `c` in Figure 2 shows that a continuation consists of a sequence of `CApp0` or `CApp1` with `C0` at the end. Since it is isomorphic to the structure of a list, a continuation can be equivalently represented as a list of frames, where a frame is either `CApp0` or `CApp1` without the `c` component. See the definition of types `f` and `c` in Figure 3. The functions `run_c3`, `f3`, and `eval3` are accordingly modified.

```
type v = VFun of (v -> c -> v)          (* value *)
       | VCont of c
and  c = C0                      (* continuation *)
       | CApp0 of t * xs * v list * c
       | CApp1 of v * c

(* run_c2 : c -> v -> v *)
let rec run_c2 c v = match c with
    C0 -> v
  | CApp0(t1,xs,vs,c) -> f2 t1 xs vs (CApp1(v,c))
  | CApp1(v0,c) ->
      (match v0 with
           VFun(f) -> f v c
         | VCont(c') -> run_c2 c (run_c2 c' v))

(* f2 : t -> xs -> v list -> c -> v *)
and f2 t xs vs c = match t with
    Var(x) -> run_c2 c (List.nth vs (offset x xs))
  | Lam(x,t) ->
      run_c2 c (VFun(fun v c' ->
                        f2 t (x::xs) (v::vs) c'))
  | App(t0,t1) -> f2 t0 xs vs (CApp0(t1,xs,vs,c))
  | Shift(x,t) -> f2 t (x::xs) (VCont(c)::vs) C0
  | Reset(t) -> run_c2 c (f2 t xs vs C0)

(* eval2 : t -> v *)
let eval2 t = f2 t [] [] C0
```

Figure 2. Defunctionalized interpreter (`eval2`)

```
type v = VFun of (v -> c -> v)          (* value *)
       | VCont of c
and  f = CApp0 of t * xs * v list       (* frame *)
       | CApp1 of v
and  c = f list                  (* continuation *)

(* run_c3 : c -> v -> v *)
let rec run_c3 c v = match c with
    [] -> v
  | CApp0(t1,xs,vs)::c ->
      f3 t1 xs vs (CApp1(v)::c)
  | CApp1(v0)::c ->
      (match v0 with
           VFun(f) -> f v c
         | VCont(c') -> run_c3 c (run_c3 c' v))

(* f3 : t -> xs -> v list -> c -> v *)
and f3 t xs vs c = match t with
    Var(x) -> run_c3 c (List.nth vs (offset x xs))
  | Lam(x,t) ->
      run_c3 c (VFun(fun v c' ->
                        f3 t (x::xs) (v::vs) c'))
  | App(t0,t1) -> f3 t0 xs vs (CApp0(t1,xs,vs)::c)
  | Shift(x,t) -> f3 t (x::xs) (VCont(c)::vs) []
  | Reset(t) -> run_c3 c (f3 t xs vs [])

(* eval3 : t -> v *)
let eval3 t = f3 t [] [] []
```

Figure 3. Linearized interpreter (`eval3`)

```
type v = VFun of (v -> s -> c -> v)     (* value *)
       | VCont of c * s
       | VEnv of v list
and  f = CApp0 of t * xs                (* frame *)
       | CApp1
and  c = f list                  (* continuation *)
and  s = v list                    (* data stack *)

(* run_c4 : c -> v -> s -> v *)
let rec run_c4 c v s = match (c, s) with
    ([], []) -> v
  | (CApp0(t1,xs)::c, VEnv(vs)::s) ->
      f4 t1 xs vs (v::s) (CApp1::c)
  | (CApp1::c, v0::s) ->
      (match v0 with
           VFun(f) -> f v s c
         | VCont(c',s') ->
             run_c4 c (run_c4 c' v s') s)

(* f4 : t -> xs -> v list -> s -> c -> v *)
and f4 t xs vs s c = match t with
    Var(x) ->
      run_c4 c (List.nth vs (offset x xs)) s
  | Lam(x,t) ->
      run_c4 c
             (VFun(fun v s' c' ->
                     f4 t (x::xs) (v::vs) s' c'))
             s
  | App(t0,t1) ->
      f4 t0 xs vs (VEnv(vs)::s) (CApp0(t1,xs)::c)
  | Shift(x,t) ->
      f4 t (x::xs) (VCont(c,s)::vs) [] []
  | Reset(t) -> run_c4 c (f4 t xs vs [] []) s

(* eval4 : t -> v *)
let eval4 t = f4 t [] [] [] []
```

Figure 4. Stack-based interpreter (eval4)

```
type v = VFun of (v -> s -> c -> v)     (* value *)
       | VCont of c * s
       | VEnv of v list
and  c = C0                      (* continuation *)
       | CApp0 of t * xs * c
       | CApp1 of c
and  s = v list                    (* data stack *)

(* run_c5 : c -> v -> s -> v *)
let rec run_c5 c v s = match (c, s) with
    (C0, []) -> v
  | (CApp0(t1,xs,c), VEnv(vs)::s) ->
      f5 t1 xs vs (v::s) (CApp1(c))
  | (CApp1(c), v0::s) ->
      (match v0 with
           VFun(f) -> f v s c
         | VCont(c',s') ->
             run_c5 c (run_c5 c' v s') s)

(* f5 : t -> xs -> v list -> s -> c -> v *)
and f5 t xs vs s c = match t with
    Var(x) ->
      run_c5 c (List.nth vs (offset x xs)) s
  | Lam(x,t) ->
      run_c5 c
             (VFun(fun v s' c' ->
                     f5 t (x::xs) (v::vs) s' c'))
             s
  | App(t0,t1) ->
      f5 t0 xs vs (VEnv(vs)::s) (CApp0(t1,xs,c))
  | Shift(x,t) ->
      f5 t (x::xs) (VCont(c,s)::vs) [] C0
  | Reset(t) -> run_c5 c (f5 t xs vs [] C0) s

(* eval5 : t -> v *)
let eval5 t = f5 t [] [] [] C0
```

Figure 5. Delinearized interpreter (eval5)

This local transformation is obviously correct. Thus, eval2 and eval3 are equivalent.

3.3 Stack Extraction

Examining the types f and c in Figure 3, we notice that the constructors CApp0 and CApp1 contain both static (compile-time) data and dynamic (runtime) data. The term t and the variable list xs are compile-time data: they are fixed once the input program is given. On the other hand, the value list v list and the value v are runtime data: they are available only at runtime. Since our goal is to turn an interpreter into a compiler and a virtual machine, we separate a continuation into two parts. The static part of a continuation will eventually become a list of instructions. (See Section 5.3.) The dynamic part of a continuation becomes a data stack. This separation is also a binding-time improvement.

Typewise, the separation is done as follows. The type c in Figure 3 is defined as f list. We split the definition of f into a pair of new f and v, where the new f contains only the static part of the old f and the dynamic part is compensated by the accompanying v. Because the dynamic part of CApp0 has type v list rather than v, we introduce a new constructor VEnv in v to hold a list of values.

Now, a continuation is represented as a value of type (f * v) list. We further split it into two separate lists: f list * v list. The former is a (new) continuation (or a control stack) and the latter a data stack. See Figure 4. Because of this modification, VCont receives s in addition to c. The same for VFun: it receives s in addition to c.

The changes in types suggest how to rewrite the interpreter: a continuation and a data stack are always passed around together; whenever a frame is pushed onto a continuation, the corresponding value is pushed onto a data stack; and whenever a frame is popped from a continuation, the corresponding value is popped from a data stack. Because of this exact correspondence, the length of the continuation and the data stack are always the same.

Since this transformation simply changes the representation of continuations locally, we immediately see that the transformation is correct, *i.e.*, eval3 and eval4 are equivalent.

This transformation is the essence of stack introduction: a data stack is a dynamic counterpart of a continuation. We can actually observe that the intermediate results and live variables are stored in a data stack. However, we will defer this discussion until Section 3.5, where the interpreter is transformed back to more readable higher-order one.

3.4 Delinearization

The purpose of defunctionalization (Section 3.1) and linearization (Section 3.2) of continuations was to introduce a data stack. Now that we have successfully introduced a data stack, we apply the reverse transformation to the representation of continuations. This will result in a readable stack-based interpreter. Here, we change

```
type v = VFun of (v -> s -> c -> v)    (* value *)
       | VCont of c * s
       | VEnv of v list
and  c = v -> s -> v              (* continuation *)
and  s = v list                     (* data stack *)

(* f6 : t -> xs -> v list -> s -> c -> v *)
let rec f6 t xs vs s c = match t with
    Var(x) -> c (List.nth vs (offset x xs)) s
  | Lam(x,t) ->
      c (VFun(fun v s' c' ->
                f6 t (x::xs) (v::vs) s' c'))
        s
  | App(t0,t1) ->
      f6 t0 xs vs (VEnv(vs)::s)
                           (fun v0 (VEnv(vs)::s) ->
      f6 t1 xs vs (v0::s) (fun v1 (v0::s) ->
      match v0 with
          VFun(f) -> f v1 s c
        | VCont(c',s') -> c (c' v1 s') s))
  | Shift(x,t) ->
      f6 t (x::xs) (VCont(c,s)::vs) []
         (fun v [] -> v)
  | Reset(t) ->
      c (f6 t xs vs [] (fun v [] -> v)) s

(* eval6 : t -> v *)
let eval6 t = f6 t [] [] [] (fun v [] -> v)
```

Figure 6. Refunctionalized interpreter (eval6)

```
type v = VFun of (s -> c -> v)         (* value *)
       | VCont of c * s
       | VEnv of v list
and  c = s -> v                   (* continuation *)
and  s = v list                     (* data stack *)

(* f7 : t -> xs -> s -> c -> v *)
let rec f7 t xs (VEnv(vs)::s) c = match t with
    Var(x) -> c ((List.nth vs (offset x xs))::s)
  | Lam(x,t) ->
      c (VFun(fun (v::s') c' ->
                f7 t (x::xs) (VEnv(v::vs)::s') c')
         ::s)
  | App(t0,t1) ->
      f7 t0 xs (VEnv(vs)::VEnv(vs)::s)
                         (fun (v0::VEnv(vs)::s) ->
      f7 t1 xs (VEnv(vs)::v0::s)
                                (fun (v1::v0::s) ->
      match v0 with
          VFun(f) -> f (v1::s) c
        | VCont(c',s') -> c ((c' (v1::s'))::s)))
  | Shift(x,t) ->
      f7 t (x::xs) [VEnv(VCont(c,s)::vs)]
         (fun [v] -> v)
  | Reset(t) ->
      c ((f7 t xs [VEnv(vs)] (fun [v] -> v))::s)

(* eval7 : t -> v *)
let eval7 t = f7 t [] [VEnv([])] (fun [v] -> v)
```

Figure 7. Interpreter with combined arguments (eval7)

the representation of continuations from a list back to a defunctionalized form.

Figure 5 shows the result. Notice that we delinearize only continuations, not a data stack. As in Section 3.2, the equivalence of eval4 and eval5 follows from the isomorphism between a list of frames and c in Figure 5.

3.5 Refunctionalization

Finally, Figure 6 shows the result of refunctionalizing continuations. Refunctionalization [10] is the left inverse of defunctionalization. Constructors in a defunctionalized form are transformed into higher-order functions. If refunctionalization succeeds, its correctness follows from the correctness of defunctionalization. Thus, we conclude that eval5 and eval6 are equivalent.

What have we obtained? Compared to the definitional interpreter in Figure 1, the interpreter in Figure 6 receives and returns a data stack as an additional argument. In particular, a continuation c receives a result value v and a data stack s. We can regard it as pushing v on s and passing v::s as a whole to c. (We will actually do so in the next section.)

Thus, the function f6 implements a standard stack-based interpreter: the value of a variable and a function is pushed onto a data stack; the value of an application is computed by evaluating the function part t0, pushing the result v0 on a data stack, evaluating the argument part t1, pushing the result v1 on a data stack, and then calling v0 with the argument v1 pushed on a data stack. The called function extracts the argument from the data stack and continues. In other words, we have successfully interrelated the stack-based interpreter and the standard interpreter via a series of program transformations.

In addition, f6 models saving and restoring of live variables vs. Before the function part t0 is evaluated, vs is saved in a data stack. It is restored back from the data stack after the evaluation of t0, when the evaluation of t1 requires it.

The use of program transformations to derive a stack-based interpreter is particularly effective when an interpreter deals with unfamiliar constructs, such as shift and reset, and it is not obvious how to write a stack-based interpreter for them. To capture a continuation via shift, we see in Figure 6 that a captured continuation needs to hold not only a continuation but also a data stack (as in VCont (c,s)) and clear the current data stack. This behavior is consistent with a commonly-used implementation technique. Using the functional derivation approach, we can certify that such an implementation method is correct.

4. Extracting Virtual Machine

In this section, we decompose the stack-based interpreter into a compiler and a virtual machine. The key transformation is factorization of combinators that control the dynamic behavior of programs as virtual machine instructions. To do so, we first need to move inherited arguments to a data stack.

4.1 Combining Arguments

In the definitional interpreter (Figure 1), the argument vs is distributed (or inherited) to all the recursive calls. In particular, it becomes a free variable of the second continuation of App case. This prevented us from storing vs into a data stack and forced us to pass vs as a separate argument: even if we store vs in a data stack, there was no guarantee that it could be safely extracted when evaluation of t1 needed it.

After defunctionalization and stack extraction, however, we discover that vs can actually be stored in a data stack. Since vs is not a free variable of any recursive calls any more (except for the one in `Lam` case[1]), we can move `vs` to the stack top. The result is shown in Figure 7. In the figure, we also moved the first argument of continuations and functions in `VFun` into a data stack as mentioned in Section 3.5.

This local transformation simply combines two arguments into one, and thus we can immediately conclude that `eval6` and `eval7` are equivalent.

4.2 Factoring Combinators as Instructions

We now want to decompose the interpreter in Figure 7 into a compiler and a virtual machine. A compiler accepts a static program text (*i.e.*, `t` and `xs`) and produces a list of instructions. A virtual machine receives a list of instructions and executes it. Such decomposition can be done by first moving dynamic arguments to all branches and then factoring dynamic parts out as instructions.

To be more concrete, the data stack and continuation arguments of the interpreter `f7` in Figure 7 are moved to each branch of the case dispatch on `t` as follows:

```
let rec f7 t xs = match t with
    Var(x) -> fun (VEnv(vs)::s) c -> ...
  | Lam(x,t) -> fun (VEnv(vs)::s) c -> ...
  | App(t0,t1) -> fun (VEnv(vs)::s) c -> ...
  | Shift(x,t) -> fun (VEnv(vs)::s) c -> ...
  | Reset(x,t) -> fun (VEnv(vs)::s) c -> ...
```

Because we have eliminated an inherited argument, each `fun (VEnv(vs)::s) c -> ...` does not have any free variables, *i.e.*, it is a combinator. If this combinator is close enough to a machine instruction, we can think of `f7 t xs` as compiling a term into an instruction.

This is immediately true for `Var` case. Let us define an instruction `access` as follows:

```
let access n = fun (VEnv(vs)::s) c ->
  c ((List.nth vs n)::s)
```

This combinator is basic enough to treat as an instruction: it takes the n'th element of a vector `vs`, starting at index zero. Thus, the `Var` case of `f7` can be written as follows:

```
    Var(x) -> access (offset x xs)
```

Note that we perform `offset x xs` at compile time. Because we know the position of `x` in `xs` at compile time, we embed the resulting number as an argument to `access` instruction.

For other cases, it is not so straightforward, but once we realize that we can define an infix function composition operator `(>>)` in CPS, we can use it to encode a more complex expression in an instruction-like manner. For the `App` case, for example, we see that `VEnv(vs)` is pushed onto the data stack at the first recursive call (`f7 t0 xs`). Thus, we can introduce a `push_env` instruction to accommodate this change of the data stack. The other cases are similar. The result is shown in Figure 8.

The function `f8` in Figure 8 looks like a compiler: given `t` and `xs`, `f8` recurs over the structure of `t` and returns instructions. In fact, if we defunctionalize instructions (we will do so in Section 5.2), `f8` actually becomes a compiler and the dispatch function becomes a virtual machine. Before doing it, however, let us make some observations.

First, some instructions receive arbitrarily long instructions as an argument. For example, `push_closure` receives the result of evaluating `f8 t (x::xs)` (which can be very long) followed by `return`. Although it is not practical to provide long instructions as an argument to another instruction, this is not a problem, because

[1] For a function value, we need to copy `vs` to create a closure. See Section 5.4.

```
type v = VFun of (s -> c -> v)          (* value *)
       | VCont of c * s
       | VEnv of v list
       | VK of c
and  c = s -> v                  (* continuation *)
and  s = v list                    (* data stack *)

type i = s -> c -> v              (* instruction *)

(* (>>) : i -> i -> i *)
let (>>) i1 i2 = fun s c ->
  i1 s (fun s' -> i2 s' c)

(* access : int -> i *)
let access n = fun (VEnv(vs)::s) c ->
  c ((List.nth vs n)::s)

(* push_closure : i -> i *)
let push_closure code = fun (VEnv(vs)::s) c ->
  c (VFun(fun (v::s') c' ->
            code (VEnv(v::vs)::s') c')
     ::s)

(* return : i *)
let return = fun (v::VK(c')::s) _ ->
  c' (v::s)

(* push_env : i *)
let push_env = fun (VEnv(vs)::s) c ->
  c (VEnv(vs)::VEnv(vs)::s)

(* pop_env : i *)
let pop_env = fun (v0::VEnv(vs)::s) c ->
  c (VEnv(vs)::v0::s)

(* call : i *)
let call = fun (v1::v0::s) c ->
  match v0 with                  (* dummy *)
      VFun(f) -> f (v1::VK(c)::s) (fun [v] -> v)
    | VCont(c',s') -> c ((c' (v1::s'))::s)

(* shift : i -> i *)
let shift code = fun (VEnv(vs)::s) c ->
  code [VEnv(VCont(c,s)::vs)] (fun [v] -> v)

(* reset : i -> i *)
let reset code = fun (VEnv(vs)::s) c ->
  c ((code [VEnv(vs)] (fun [v] -> v))::s)

(* f8 : t -> xs -> i *)
let rec f8 t xs = match t with
    Var(x) -> access (offset x xs)
  | Lam(x,t) ->
      push_closure (f8 t (x::xs) >> return)
  | App(t0,t1) ->
      push_env >> f8 t0 xs >> pop_env >> f8 t1 xs
      >> call
  | Shift(x,t) -> shift (f8 t (x::xs))
  | Reset(t) -> reset (f8 t xs)

(* eval8 : t -> v *)
let eval8 t = f8 t [] [VEnv([])] (fun [v] -> v)
```

Figure 8. Interpreter using combinators factored as instructions (`eval8`)

```
type v = VFun of (s -> c -> d -> v)    (* value *)
       | VCont of c * s
       | VEnv of v list
       | VK of c
and  c = s -> d -> v              (* continuation *)
and  d = v -> v      (* dump (metacontinuation) *)
and  s = v list                     (* data stack *)

type i = s -> c -> d -> v        (* instruction *)
```

Figure 9. 2CPS interpreter (eval9), type part

we can store the result of compilation `f8 t (x::xs) >> return` somewhere in the memory, and give a pointer to the first instruction as an argument to `push_closure`.

Secondly, the output of the compilation is not a list of instructions but a tree of instructions. This is again not a problem, because the operator `(>>)` is associative. We can always safely expand a tree of instructions into a list of instructions.

Thirdly, Figure 8 is written in such a way that a return address (`c`) is stored in a data stack when a closure is called, rather than passing it as a continuation argument (see `call` instruction). For this purpose, a new constructor `VK` is introduced in type `v`. Because `c` is stored in a data stack, the continuation argument `fun [v] -> v` of `f` in `call` is a dummy continuation which will never be used. In `return` instruction, we see that the continuation argument is discarded and the continuation stored in a data stack is used.

As this modification shows, there is no predefined instruction set that we have to use. We can define and introduce a new instruction as we need it, so that the outcome best fits our intention. The functional derivation approach does not imply a mechanical process. Rather, it helps us to explore the design space of various implementation methods.

Readers may doubt whether such modification is correct. The equivalence between `eval7` and `eval8` is established by first inlining `(>>)` and all the instructions. The resulting interpreter is identical to `eval7` except for two places. The `Lam` case:

```
c (VFun(fun (v::VK(c')::s') _ ->
         f7 t (x::xs) (VEnv(v::vs)::s') c')
   ::s)
```

and the last part of `App` case:

```
match v0 with                   (* dummy *)
    VFun(f) -> f (v1::VK(c)::s) (fun [v] -> v)
  | VCont(c',s') -> c ((c' (v1::s'))::s)
```

Now, it is easy to see they are equivalent to `eval7`: the continuation argument is simply moved to the data stack. With this simple modification, a standard calling convention of storing return addresses in a data stack is derivable.

Finally, the `reset` instruction is rather unusual. It installs a new data stack `[VEnv(vs)]` and executes `code` with it. However, we usually use only one data stack. Rather than using multiple data stacks, we want to reuse a single data stack in a consistent way. This is the topic of the next section.

5. Representing Dumps

The obtained combinators in Figure 8 can be almost regarded as virtual machine instructions, because most of them simply pass on a data stack, modifying a top few frames if necessary. If we use the real machine stack instead of passing a data stack as an argument, the instructions do correspond to instructions in a stack machine.

However, there are still a few exceptions. When a captured continuation `VCont(c',s')` is applied in `call`:

```
(* (>>) : i -> i -> i *)
let (>>) i1 i2 = fun s c ->
  i1 s (fun s' -> i2 s' c)

(* access : int -> i *)
let access n = fun (VEnv(vs)::s) c ->
  c ((List.nth vs n)::s)

(* push_closure : i -> i *)
let push_closure code = fun (VEnv(vs)::s) c ->
  c (VFun(fun (v::s') c' ->
            code (VEnv(v::vs)::s') c')
     ::s)

(* return : i *)
let return = fun (v::VK(c')::s) _ ->
  c' (v::s)

(* push_env : i *)
let push_env = fun (VEnv(vs)::s) c ->
  c (VEnv(vs)::VEnv(vs)::s)

(* pop_env : i *)
let pop_env = fun (v0::VEnv(vs)::s) c ->
  c (VEnv(vs)::v0::s)

(* instructions same as Figure 8 up to here *)

(* call : i *)
let call = fun (v1::v0::s) c d -> match v0 with
    VFun(f) ->          (* dummy *)
      f (v1::VK(c)::s) (fun [v] d -> d v) d
  | VCont(c',s') ->
      c' (v1::s') (fun v -> c (v::s) d)

(* shift : i -> i *)
let shift code = fun (VEnv(vs)::s) c ->
  code [VEnv(VCont(c,s)::vs)] (fun [v] d -> d v)

(* reset : i -> i *)
let reset code = fun (VEnv(vs)::s) c d ->
  code [VEnv(vs)] (fun [v] d -> d v)
       (fun v -> c (v::s) d)

(* f9 : t -> xs -> i *)
let rec f9 t xs = match t with
    Var(x) -> access (offset x xs)
  | Lam(x,t) ->
      push_closure (f9 t (x::xs) >> return)
  | App(t0,t1) ->
      push_env >> f9 t0 xs >> pop_env >> f9 t1 xs
      >> call
  | Shift(x,t) -> shift (f9 t (x::xs))
  | Reset(t) -> reset (f9 t xs)

(* eval9 : t -> v *)
let eval9 t =
  f9 t [] [VEnv([])] (fun [v] d -> d v)
     (fun v -> v)
```

Figure 9. 2CPS interpreter (eval9), continued

```
type i = IAccess of int          (* instruction *)
       | IPush_closure of i | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of i | IReset of i
       | ISeq of i * i

type v = VFun of (s -> c -> d -> v)    (* value *)
       | VCont of c * s
       | VEnv of v list
       | VK of c
and  c = i list                   (* continuation *)
and  d = (c * s) list                     (* dump *)
and  s = v list                     (* data stack *)
```

Figure 10. Interpreter with defunctionalized instructions (eval10), type part

```
  | VCont(c',s') -> c ((c' (v1::s'))::s)
```

c' is applied to a saved data stack s', which is different from the current data stack s. In reset,

```
  c ((code [VEnv(vs)] (fun [v] -> v))::s)
```

code is executed on a newly created data stack [VEnv(vs)] which is again different from the current data stack s. Put differently, the interpreter in Figure 8 is not wholly written in CPS, because it contains non-tail calls to continuations. The goal of this section is to represent this direct-style information as a data structure. It turns out that the resulting data structure corresponds to a list of delimited data stacks.

5.1 CPS Transformation

To represent a direct-style application of continuations as a data structure, we transform this almost CPS interpreter into CPS one more time. The newly introduced continuations, or metacontinuations, correspond to *dumps* in the SECD machine [7, 18]. Figure 9 shows the result of CPS-transforming Figure 8.

Because eval8 was already written mostly in CPS, the second CPS transformation does not change functions very much except for their types. Although new functions receive a dump, it is usually η-reduced and does not appear in the function definition. A dump appears only in those instructions that explicitly modify it, namely, call, shift, and reset.

The equivalence between eval8 and eval9 follows from the correctness of CPS transformation.

5.2 Defunctionalization and Linearization

To represent a dump as a data structure, we defunctionalize and linearize a dump and a continuation. At this moment, we defunctionalize instructions, too. Both the processes are mechanical. The result is shown in Figure 10.

Through defunctionalization and linearization of a dump, the identity dump fun v -> v becomes an empty list. The dump fun v -> c (v::s) d in call and reset has three free variables c, s, and a dump d itself. After linearization, it becomes (c * s) list.

Likewise, through defunctionalization and linearization of a continuation, the identity continuation fun [v] d -> d v becomes an empty list. The continuation fun s' -> i2 s' c in (>>) has two free variables i2 and a continuation c itself. After linearization, it becomes a list of instructions.

The dispatch function (run_i10) for instructions is a virtual machine: given an instruction, it operates on a data stack and a dump. The defunctionalization of (>>) yields a sequence instruction ISeq that connects two instructions into one.

```
(* run_d10 : d -> v -> v *)
let rec run_d10 d v = match d with
    [] -> v
  | (c,s)::d' -> run_c10 c (v::s) d'

(* run_c10 : c -> s -> d -> v *)
and run_c10 c s d = match c with
    [] -> (match s with [v] -> run_d10 d v)
  | g::c -> run_i10 g s c d

(* run_i10 : i -> s -> c -> d -> v *)
and run_i10 i s c d = match i with
    IAccess(n) -> (match s with (VEnv(vs)::s) ->
      run_c10 c ((List.nth vs n)::s) d)
  | IPush_closure(code) ->
     (match s with (VEnv(vs)::s) ->
        run_c10 c
                (VFun(fun (v::s') c' d' ->
                   run_i10 code (VEnv(v::vs)::s')
                           c' d')
                 ::s) d)
  | IReturn -> (match s with (v::VK(c')::s) ->
      run_c10 c' (v::s) d)
  | IPush_env -> (match s with (VEnv(vs)::s) ->
      run_c10 c (VEnv(vs)::VEnv(vs)::s) d)
  | IPop_env -> (match s with (v0::VEnv(vs)::s) ->
      run_c10 c (VEnv(vs)::v0::s) d)
  | ICall -> (match s with (v1::v0::s) ->
      match v0 with                    (* dummy *)
          VFun(f) -> f (v1::VK(c)::s) [] d
        | VCont(c',s') ->
            run_c10 c' (v1::s') ((c,s)::d))
  | IShift(code) -> (match s with (VEnv(vs)::s) ->
      run_i10 code [VEnv(VCont(c,s)::vs)] [] d)
  | IReset(code) -> (match s with (VEnv(vs)::s) ->
      run_i10 code [VEnv(vs)] [] ((c,s)::d))
  | ISeq(f,g) -> run_i10 f s (g::c) d

(* (>>) : i -> i -> i *)
let (>>) f g = ISeq(f,g)

(* f10 : t -> xs -> i *)
let rec f10 t xs = match t with
    Var(x) -> IAccess(offset x xs)
  | Lam(x,t) ->
      IPush_closure(f10 t (x::xs) >> IReturn)
  | App(t0,t1) ->
      IPush_env >> f10 t0 xs >> IPop_env
      >> f10 t1 xs >> ICall
  | Shift(x,t) -> IShift(f10 t (x::xs))
  | Reset(t) -> IReset(f10 t xs)

(* eval10 : t -> v *)
let eval10 t =
  run_i10 (f10 t []) [VEnv([])] [] []
```

Figure 10. Interpreter with defunctionalized instructions (eval10), continued

The equivalence between `eval9` and `eval10` follows from the correctness of defunctionalization and the correctness argument for linearization we made in Section 3.2.

In the obtained interpreter, all the instructions operate only on a data stack and a dump. They also receive a continuation as an argument, but looking at the dispatch function (`run_c10`), we notice that the only role of the continuation argument is to designate the next instruction. If we inline `run_i10` into `run_c10` (as we will do in the next section), the continuation argument disappears.

We can now regard `s` placed on top of `d` as a single system stack. This interpretation actually models the copy of a part of a data stack when a continuation is captured. We will discuss it in detail in Section 6.

5.3 Code Flattening

The instructions obtained so far are structured as a tree: the instruction `ISeq` appends two trees of instructions. In this section, we transform it into a list of instructions.

First, we observe (by unfolding `run_c10`) that the following two expressions are equivalent for any `i`, `c`, `s`, and `d`:

```
run_c10 (i::c) s d
run_i10 i s c d
```

Thus, we can replace all the recursive calls in `run_i10` (as well as the initial call to `run_i10` in `eval10`) with calls to `run_c10`. With this replacement, we can inline `run_i10` into `run_c10` so that `run_c10` dispatches over the first instruction of `c`.

We then remove `ISeq` and replace it with an append operation `@` on instructions, using a singleton list as a unit of instructions. The result is shown in Figure 11. The function `f11` now returns a list of instructions rather than a single instruction. In `IPush_closure` case of `run_c11`, `code` and `c'` is appended, but because `c'` is always a dummy continuation `[]`, we do not actually have to manipulate code at runtime. In the next section, the dummy continuation disappears as the result of defunctionalization.

The equivalence between `eval10` and `eval11` follows from the correctness of inlining and associativity of `(>>)`. Wand [27, 28] used similar program transformation based on associativity of combinators.

5.4 Defunctionalizing Function

Finally, defunctionalization of functions in `VFun` yields a closure representation of functions. The type of `VFun` is changed to

```
type v = VFun of c * v list             (* value *)
       | ...
```

Namely, a closure holds a code pointer and a list of values for its free variables. The only call site of this closure is in the `ICall` case. Corresponding modification to the virtual machine (`run_c11`) is simple:

```
 | IPush_closure(code)::c ->
    (match s with (VEnv(vs)::s) ->
       run_c11 c (VFun(code,vs)::s) d)
 | ICall::c -> (match s with (v1::v0::s) ->
     match v0 with
         VFun(code,vs) ->
           run_c11 code (VEnv(v1::vs)::VK(c)::s) d
       | VCont(c',s') ->
           run_c11 c' (v1::s') ((c,s)::d))
```

The defunctionalized `VFun` constructed in the `IPush_closure` case is destructed in the `ICall` case. Rather than defining an apply function for `VFun`, it is inlined into the body of the `ICall` case. At that time, the dummy continuation `[]` is also inlined and reduced away.

```
type i = IAccess of int          (* instruction *)
       | IPush_closure of i list | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of i list | IReset of i list

type v = VFun of (s -> c -> d -> v)    (* value *)
       | VCont of c * s
       | VEnv of v list
       | VK of c
and  c = i list                  (* continuation *)
and  d = (c * s) list                    (* dump *)
and  s = v list                    (* data stack *)

(* run_d11 : d -> v -> v *)
let rec run_d11 d v = match d with
    [] -> v
  | (c,s)::d' -> run_c11 c (v::s) d'

(* run_c11 : i list -> s -> d -> v *)
and run_c11 c s d = match c with
    [] -> (match s with [v] -> run_d11 d v)
  | IAccess(n)::c ->
     (match s with (VEnv(vs)::s) ->
        run_c11 c ((List.nth vs n)::s) d)
  | IPush_closure(code)::c ->
     (match s with (VEnv(vs)::s) ->
       run_c11 c
         (VFun(fun (v::s') c' d' ->
           run_c11 (code@c') (VEnv(v::vs)::s') d')
          ::s) d)
  | IReturn::c -> (match s with (v::VK(c')::s) ->
      run_c11 c' (v::s) d)
  | IPush_env::c -> (match s with (VEnv(vs)::s) ->
      run_c11 c (VEnv(vs)::VEnv(vs)::s) d)
  | IPop_env::c ->
     (match s with (v0::VEnv(vs)::s) ->
        run_c11 c (VEnv(vs)::v0::s) d)
  | ICall::c -> (match s with (v1::v0::s) ->
      match v0 with                   (* dummy *)
          VFun(f) -> f (v1::VK(c)::s) [] d
        | VCont(c',s') ->
            run_c11 c' (v1::s') ((c,s)::d))
  | IShift(code)::c ->
     (match s with (VEnv(vs)::s) ->
        run_c11 code [VEnv(VCont(c,s)::vs)] d)
  | IReset(code)::c ->
     (match s with (VEnv(vs)::s) ->
        run_c11 code [VEnv(vs)] ((c,s)::d))

(* f11 : t -> xs -> i list *)
let rec f11 t xs = match t with
    Var(x) -> [IAccess (offset x xs)]
  | Lam(x,t) ->
      [IPush_closure((f11 t (x::xs))@[IReturn])]
  | App(t0,t1) ->
      [IPush_env]@(f11 t0 xs)@[IPop_env]
      @(f11 t1 xs)@[ICall]
  | Shift(x,t) -> [IShift(f11 t (x::xs))]
  | Reset(t) -> [IReset(f11 t xs)]

(* eval11 : t -> v *)
let eval11 t = run_c11 (f11 t []) [VEnv([])] []
```

Figure 11. Interpreter with linear instructions (`eval11`)

c	$\Rightarrow$	$\langle c, [\texttt{VEnv}([])], []\rangle$
$\langle \texttt{IAccess}(n) :: c, \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c, (\texttt{List.nth}\ vs\ n) :: s, d\rangle$
$\langle \texttt{IPush\_closure}(c') :: c, \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c, \texttt{VFun}(c', vs) :: s, d\rangle$
$\langle \texttt{IReturn} :: \_, v :: \texttt{VK}(c) :: s, d\rangle$	$\Rightarrow$	$\langle c, v :: s, d\rangle$
$\langle \texttt{IPush\_env} :: c, \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c, \texttt{VEnv}(vs) :: \texttt{VEnv}(vs) :: s, d\rangle$
$\langle \texttt{IPop\_env} :: c, v :: \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c, \texttt{VEnv}(vs) :: v :: s, d\rangle$
$\langle \texttt{ICall} :: c, v_1 :: \texttt{VFun}(c', vs) :: s, d\rangle$	$\Rightarrow$	$\langle c', \texttt{VEnv}(v_1 :: vs) :: \texttt{VK}(c) :: s, d\rangle$
$\langle \texttt{ICall} :: c, v_1 :: \texttt{VCont}(c', s') :: s, d\rangle$	$\Rightarrow$	$\langle c', v_1 :: s', (c, s) :: d\rangle$
$\langle \texttt{IShift}(c') :: c, \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c', [\texttt{VEnv}(\texttt{VCont}(c, s) :: vs)], d\rangle$
$\langle \texttt{IReset}(c') :: c, \texttt{VEnv}(vs) :: s, d\rangle$	$\Rightarrow$	$\langle c', [\texttt{VEnv}(vs)], (c, s) :: d\rangle$
$\langle [], [v], []\rangle$	$\Rightarrow$	v
$\langle [], [v], (c, s) :: d\rangle$	$\Rightarrow$	$\langle c, v :: s, d\rangle$

Figure 12. The obtained virtual machine

This transformation is an instance of defunctionalization and is correct.

6. Discussion

Because all the calls in `run_d11` and `run_c11` are tail calls, they can be directly converted to a virtual machine. Figure 12 shows the final virtual machine we obtained in this paper, after inlining `run_d11` into `run_c11`. In addition to the standard calling convention (*i.e.*, saving and restoring return addresses and values of live variables), it precisely specifies the behavior of stack copying at continuation capture and invocation. When a context is delimited (see the behavior of `IReset` in Figure 12), the next instruction c (= the current continuation) and the current data stack s are pushed on a dump and the current data stack is cleared. When a continuation is captured (via `IShift`), the next instruction c and the current data stack s are copied to the heap, the current data stack is cleared, and the body c' of shift is executed. When the captured continuation is invoked, the next instruction c and the current data stack s are pushed on a dump and the saved continuation and the data stack are restored.

Figure 12 shows more than that. Remember that a dump d is a list of (c, s), and c is a code pointer (the return address for the evaluation of the current context). Thus, the top of a dump holds a return address. This validates the invariant we presented in the previous work [20] to implement delimited continuations: a return address and a reset pointer always reside immediately under the current data stack.

Readers may think that copying a data stack s for every reset and continuation invocation is not realistic. This can be easily avoided, if we regard a data stack s and a dump d as a single stack. Assume that d has the form $[(c_1, s_1); \ldots ; (c_n, s_n)]$. If we know the length of each data stack s_i, we can simply expand a data stack s and a dump d as a single stack:

$$s @ \texttt{VK}(c_1) @ s_1 @ \ldots @ \texttt{VK}(c_n) @ s_n$$

Then, resetting the current context amounts to simply pushing the current c (without copying s), and the continuation invocation amounts to copying the saved data stack s' (without copying s). The length of each data stack can be remembered by pushing a pointer to the end of the current data stack. Thus, the virtual machine even explains the need for a reset pointer!

7. Conclusion

In this paper, we have connected the definitional interpreter for the λ-calculus with shift and reset with a compiler and a low-level stack-copying virtual machine. The obtained virtual machine explains and certifies a number of implementation techniques for delimited continuations. The derivation process is not actually straightforward: we sometimes have to imagine the result of program transformation and what it means. However, the functional derivation approach surely provides us with the guidance as to which ways we can go and explore. Furthermore, connection via simple program transformation appears to suggest the meaning of various implementation techniques in a succinct way.

Acknowledgments

We received valuable comments and suggestions from Olivier Danvy as well as anonymous reviewers; they improved the paper in various ways.

References

[1] Ager, M. S., D. Biernacki, O. Danvy, and J. Midtgaard “From Interpreter to Compiler and Virtual Machine: A Functional Derivation,” BRICS Research Report RS-03-14, Department of Computer Science, Aarhus University, 36 pages (March 2003).

[2] Ager, M. S., D. Biernacki, O. Danvy, and J. Midtgaard “A functional correspondence between evaluators and abstract machines,” *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP’03)*, pp. 8–19 (September 2003).

[3] Asai, K. “Logical Relations for Call-by-value Delimited Continuations,” *Trends in Functional Programming (TFP 2005)*, Vol. 6, pp. 63–78, Intellect (2007).

[4] Asai, K. “On Typing Delimited Continuations: Three New Solutions to the Printf Problem,” *Higher-Order and Symbolic Computation*, Vol. 22, No. 3, pp. 275–291, Kluwer Academic Publishers (September 2009).

[5] Banerjee, A., N. Heintze, and J. G. Riecke “Design and Correctness of Program Transformations Based on Control-Flow Analysis,” In N. Kobayashi and B. C. Pierce, editors, *Theoretical Aspects of Computer Software (LNCS 2215)*, pp. 420–447 (October 2001).

[6] Biernacka, M., D. Biernacki, and O. Danvy “An Operational Foundation for Delimited Continuations in the CPS Hierarchy,” *Logical Methods in Computer Science*, Vol. 1, No. 2:5, pp. 1–39 (November 2005).

[7] Danvy, O. “A Rational Deconstruction of Landin’s SECD Machine,” In C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, editors, *Implementation and Application of Functional Languages (LNCS 3474)*, pp. 52–71 (September 2004).

[8] Danvy, O. “Defunctionalized interpreters for programming languages,” *Proceedings of the 13th ACM SIGPLAN Inter-*

national Conference on Functional Programming (ICFP'08), pp. 131–142 (September 2008).

[9] Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[10] Danvy, O., K. Millikin "Refunctionalization at Work," *Science of Computer Programming*, Vol. 74, No. 8, pp. 534–549, Elsevier (June 2008).

[11] Danvy, O., K. Millikin "A Rational Deconstruction of Landin's SECD Machine with the J Operator," *Logical Methods in Computer Science*, Vol. 4, No. 4:12, pp. 1–67 (November 2008).

[12] Dybvig, R. K., S. L. Peyton Jones, and A. Sabry "A Monadic Framework for Delimited Continuations," *Journal of Functional Programming*, Vol. 17, No. 6, pp. 687–730, Cambridge University Press (November 2007).

[13] Gasbichler, M., and M. Sperber "Final Shift for Call/cc: Direct Implementation of Shift and Reset," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp. 271–282 (October 2002).

[14] Igarashi, A., and M. Iwaki "Deriving Compilers and Virtual Machines for a Multi-Level Language," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 206–221 (November 2007).

[15] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[16] Kameyama, Y., O. Kiselyov, and C. C. Shan "Shifting the Stage: Staging with Delimited Control," *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'09)*, pp. 111–120 (January 2009).

[17] Kiselyov, O. "Delimited Control in OCaml, Abstractly and Concretely, System Description," *Proceedings of the Tenth International Symposium on Functional and Logic Programming (FLOPS 2010)*, to appear (April 2010).

[18] Landin, P. J. "The mechanical evaluation of expressions," *The Computer Journal*, Vol. 6, No. 4, pp. 308-320 (January 1964).

[19] Lawall, J. L., and O. Danvy "Continuation-Based Partial Evaluation," *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pp. 227–238 (June 1994).

[20] Masuko, M., and K. Asai "Direct Implementation of Shift and Reset in the MinCaml Compiler," *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, pp. 49–60 (September 2009).

[21] Nielsen, L. R. "A Denotational Investigation of Defunctionalization," BRICS Research Report RS-00-47, Department of Computer Science, Aarhus University, 50 pages (December 2000).

[22] Plotkin, G. D. "Call-by-name, call-by-value, and the λ-calculus," *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125–159 (December 1975).

[23] Reynolds, J. C. "Definitional Interpreters for Higher-Order Programming Languages," *Proceedings of the ACM National Conference*, Vol. 2, pp. 717–740, (August 1972), reprinted in *Higher-Order and Symbolic Computation*, Vol. 11, No. 4, pp. 363–397, Kluwer Academic Publishers (December 1998).

[24] Rompf, T., I. Maier, M. Odersky "Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform," *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, pp. 317–328 (August 2009).

[25] Sumii, E. "MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language," *ACM SIGPLAN Workshop on Functional and Declarative Programming in Education (FDPE '05)*, pp. 27–38 (September 2005).

[26] Sumii, E., and N. Kobayashi "A Hybrid Approach to Online and Offline Partial Evaluation," *Higher-Order and Symbolic Computation*, Vol. 14, Nos. 2/3, pp. 101–142, Kluwer Academic Publishers (2001).

[27] Wand, M. "Semantics-Directed Machine Architecture," *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pp. 234–241 (January 1982).

[28] Wand, M. "Loops in Combinator-Based Compilers," *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, pp. 190–196 (January 1983).