Derivation of an Abstract Machine for λ-calculus with Delimited Continuation Constructs

Arisa Kitani Kenich Asai

Ochanomizu University
2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan

Abstract

The goal of our research is to give a verified virtual machine for the λ-calculus with delimited continuation constructs, shift and reset, and ultimately to give a formal foundation for the direct implementation of shift/reset in the machine language. Toward this goal, we derive an abstract machine and transition rules from the CPS interpreter defining shift/reset, using a series of transformations whose validity is proved. Following the "functional correspondence" approach advocated by Danvy, we first perform the CPS transformation and defunctionalization. Unlike the previous approach, however, we introduce two new transformations, stack introduction and environment storing, to turn the interpreter closer to the actual implementation. After stack introduction, the abstract machine stores the result of execution in a stack rather than holding it in machine instructions. After environment storing, the bindings of necessary variables after function calls are stored explicitly in a stack, mimicking the standard calling convention of the compiled programs. In this article, we show correctness proofs of the two transformations using the bisimulation method. Consequently, we succeeded in deriving an abstract machine that stores bindings in the stack from the original CPS interpreter using validated transformations only.

keywords delimited continuations, shift/reset, abstract machine, interpreter, program transformation, bisimulation

1 Introduction

A continuation represents the rest of an evaluation at a certain point. To capture or to copy a continuation corresponds to editing the control flow of a program. Unlike the global jump, however, control operators can be typed [3, 5] and are safer to use than unrestricted 'goto'. An example use of continuations is for exception handling. Using continuation-capturing operators, we can achieve the same process by discarding the current continuation.

To manipulate continuations in a program, several control operators have been proposed, such as call/cc [16], control/prompt [8], and shift/reset [5]. While call/cc captures whole the rest of the continuation, control/prompt and shift/reset capture the continuation whose scope is delimited by the user.

The implementation of control operators, however, is not easy. The direct implementation typically involves copying part of the stack to the heap and back. Such low-level operations are hard to implement and are error-prone.

To remedy this problem, we use the "functional correspondence" approach in this article. Starting from the definitional CPS interpreter for shit/reset, we obtain the low-level implementation by successively applying transformations whose validity is formally verified. This gives us an abstract machine that is formally verified to behave the same as the definitional CPS interpreter [2].

In this article, we focus on the delimited continuation constructs, shift and reset, and show that we can obtain an abstract machine for λ-calculus with shift and reset. Along the development, we introduce two new transformations, stack introduction and environment storing, and prove their correctness. These two transformations are important in obtaining an abstract machine that is closer to the actual implementation. After stack introduction, the abstract machine stores the result of execution in a stack rather than holding it in machine instructions. After environment storing, the bindings of necessary variables after function calls are stored explicitly in a stack, mimicking the standard calling convention of the compiled programs. As a whole, we can obtain an abstract machine that stores bindings in the stack from the original CPS interpreter using validated transformations only.

The contributions of the paper are as follows:

- We introduce two transformations, stack introduction and environment storing, and prove their correctness.
- We connect using the "functional correspondence" approach the definitional CPS interpreter for λ-calculus with shift/reset and an abstract machine which saves bindings of variables in the stack.
- We present a formally verified abstract machine that properly models two low-level features: the calling convention of compiled programs and copying of the stack.

Related work

Danvy and his colleagues strongly push forward the "functional correspondence" approach to relate various interpreters and abstract/virtual machines. Ager, Biernacki, Danvy, and Midtgaard [2] show that the λ-calculus can be related to various abstract machines, such as the CEK, CLS, and SECD machines, via the CPS transformation and defunctionalization. Based on this idea, Danvy and Millikin [7] applied it to a calculus with Landin's J operator and relate it to the SECD machine. Along the development, they mention bisimulation between the defunctionalized CPS interpreter and the SECD machine. Following this approach, we applied the same technique to the λ-calculus with shift and reset, but proved formally the correctness of stack introduction and introduced a new transformation that stores an environment in the stack.

Biernacka, Biernacki, and Danvy [4] already applied the "functional derivation" approach to the λ-calculus with shift/reset. They applied the CPS transformation and defunctionalization to obtain an abstract machine similar to the SECD machine. This corresponds to the first two steps of our derivation. In the present work, we further transformed the interpreter to obtain an abstract machine that is closer to the actual implementation in the machine language.

Ager, Biernacki, Danvy, and Midtgaard [1] presented how to obtain a compiler and a virtual machine by dividing an abstract machine. Following this approach, Igarashi and Iwaki [9] derived a compiler and a virtual machine for the multi-level language $\lambda^{\bigcirc}$, which supports backquote and unquote. Although we do not derive a compiler in this article, we hope to apply this idea to validate

the direct low-level implementation of shift/reset, which our research group is currently working on [12].

Overview

We first introduce the operators shift and reset (Section 2). They are implemented straightforwardly in the CPS interpreter based on their original definition (Section 3). Then, we transform this interpreter to obtain the lower-level implementation of shift/reset. CPS transformation (Section 4), defunctionalization (Section 5), continuation linearization (Section 6), stack introduction (Section 7), and environment storing (Section 8) are applied to the interpreter. The validity of stack introduction and environment storing is proved in Sections 7.2 and 8.2. Afterwords, we describe the feature of the obtained abstract machine (Section 9). We finally mention the conclusion and issues (Section 10).

2 Shift/reset

The shift operator abstracts the current continuation as a function. It is related to call/cc found in Scheme [16], but is different in that the reset operator delimits the scope of the abstracted continuation and the continuation is destroyed if it is not called later. In this article, the expression `shift(...)` means that the current continuation is abstracted as a function and passed to the function `(...)`. For example, the expression `1 + shift(fun k -> 2 * (k 3))` means that a continuation, which is a function that receives a value and adds 1 to it, is passed to `k`. Thus, we obtain `2 * (1 + 3)`, or `8` as a result.

The reset operator delimits the scope of continuations abstracted by the shift operator. In this article, `reset(...)` means that continuations captured by the shift operator in `(...)` is limited up to this expression. For example, the expression `1 + reset(4 + shift(fun k -> 2 * (k 3)))` means that a continuation, which is a function that receives a value and adds 4 to it, is passed to `k`. We obtain `1 + (2 * (4 + 3))`, or `15` as a result. The first part of the expression, `1 +`, is outside the range of continuations abstracted by the shift.

3 CPS interpreter

The language we consider in this article is the λ-calculus extended with shift and reset. Besides variables, abstraction, and application, we can express shift and reset.

$$t \quad ::= \quad x \mid \lambda x.\, t \mid t_0\ t_1 \mid \texttt{shift}(t) \mid \texttt{reset}(t)$$

Values are either a closure $[x, t, e]$, which is generated by abstraction, or a continuation $[c]$, which is captured by the shift operator.

$$v \quad ::= \quad [x, t, e] \mid [c]$$

Now, we implement this language with a functional programming language, OCaml. Definitions of terms and values are as follows:

```
1 (* term *)
2 type t = Var of string            (* variable *)
3        | Fun of string * t     (* abstraction *)
```

```
(* id : v -> v *)
let id x = x

(* eval : t * e * c -> v *)
let rec eval (t, e, c) = match t with
    Var(x) -> c (get(x, e))
  | Fun(x, t) -> c (VFun(x, t, e))
  | App(t0, t1) -> eval (t1, e, (fun v1
                  -> eval (t0, e, (fun v0
                    -> (match v0 with
                          VFun(x', t', e') -> eval(t', (x', v1) :: e', c)
                        | VCont(c') -> c (c' v1) )))))
  | Shift(t) -> eval (t, e, (fun v
                -> (match v with
                      VFun(x', t', e')
                        -> eval (t', (x', VCont(c)) :: e', id)
                    | VCont(c') -> c' (VCont(c))) ))
  | Reset(t) -> c (eval (t, e, id))

(* eval1 : t -> v *)
let eval1 t = eval (t, [], id)
```

Figure 1: `eval1`: the initial CPS interpreter defining shift and reset

```
        | App of t * t          (* application *)
        | Shift of t                  (* shift *)
        | Reset of t                  (* reset *)
(* value *)
type v = VFun of string * t * e   (* closure *)
        | VCont of c           (* continuation *)
(* environment *)
and e = (string * v) list
(* continuation *)
and c = (v -> v)
```

Environments are implemented as lists of pairs of a variable `x` and a value `v`. The function `get(x, e)` returns the value of a variable `x` in an environment `e`. The evaluator `eval1` that defines shift/reset is implemented as in Figure 1 [6].

This is the standard CPS interpreter for the λ-calculus extended with shift and reset. To execute `Shift(t)`, the evaluator packages the current continuation as `VCont(c)` and passes it to the argument `t` (to be more precise, the closure resulting from executing `t` using `eval`). To execute `Reset(t)`, the evaluator initializes or *resets* the current continuation to `id`, delimiting the scope of continuations abstracted by the shift operator. This is the straightforward implementation of shift/reset based on their original definition. Below, we apply a series of transformations whose validity is verified to this interpreter to obtain the lower-level implementation of shift/reset.

```
1  (* cid : v * d -> v *)
2  (* did : v -> v *)
3  let cid (v, d) = d v
4  let did x = x
5
6  (* eval : t * e * c * d -> v *)
7  let rec eval (t, e, c, d) = match t with
8      Var(x) -> c (get(x, e), d)
9    | Fun(x, t) -> c (VFun(x, t, e), d)
10   | App(t0, t1)
11     -> eval (t1, e, (fun (v1, d1)
12         -> eval (t0, e, (fun (v0, d0)
13             -> (match v0 with
14                   VFun(x', t', e') -> eval(t', (x', v1) :: e', c, d0)
15                 | VCont(c') -> c' (v1, (fun v' -> c (v', d0)))))), d1)), d)
16   | Shift(t)
17     -> eval (t, e, (fun (v, d')
18         -> (match v with
19               VFun(x', t', e') -> eval (t', (x', VCont(c)) :: e', cid, d')
20             | VCont(c') -> c' (VCont(c), d'))), d)
21   | Reset(t) -> eval (t, e, cid, (fun v -> c (v, d)))
22
23 (* eval2 : t -> v *)
24 let eval2 t = eval (t, [], cid, did)
```

Figure 2: `eval2`: CPS transformation of `eval1`

4 CPS transformation

If every recursive call in the evaluator is a tail call, we can consider its arguments as a state and the evaluator as defining the transition rules between states. The evaluator `eval1`, however, has recursive calls which are not tail calls (lines 12 and 17 of `eval1`). Thus, we CPS-transform [14] `eval1` to make it tail recursive. This transformation changes the definition of types as follows:

```
1 type v = (* no change *)            (* value *)
2 and e = (* no change *)       (* environment *)
3 and c = (v * d -> v)          (* continuation *)
4 and d = (v -> v)         (* metacontinuation *)
```

The resulting evaluator is shown in Figure 2. We call this evaluator `eval2`.

While the evaluator `eval1` is mostly in CPS form having the continuation `c`, `eval2` is CPS-transformed one more time and has two continuations `c` and `d`. As a result of the double CPS transformations, a continuation is divided into two parts.

Validity of this transformation is derived directly from the validity of CPS transformation.

Proposition 1 (Validity of CPS transformation).
For an arbitrary term `t`*,* `eval1` *and* `eval2` *both fail to yield values or both yield values which are structurally equal. (The value which is given by CPS-transforming the result of evaluation in* `eval1` *is equal to the result of evaluation in* `eval2`*.)*

```
(* run_c : c * v * d -> v *)
let rec run_c (c, v, d) = match c with
    CApp1(t', e', c') -> eval(t', e', CApp0(v, c'), d)
  | CApp0(v', c') -> (match v with
                          VFun(x', t', e')
                          -> eval(t', (x', v') :: e', c', d)
                        | VCont(c'') -> run_c (c'', v', DRun(c', d)))
  | CShift(c') ->  (match v with
                       VFun(x', t', e')
                       -> eval(t', (x', VCont(c')) :: e', CReset, d)
                     | VCont(c'') -> run_c (c'', VCont(c'), d))
  | CReset -> run_d (d, v)
(* run_d : d * v -> v *)
and run_d (d, v) = match d with
    DRun(c', d') -> run_c (c', v, d')
  | DReset -> v
```

Figure 3: `run_c and run_d`: the apply functions of `eval3`

```
(* cid : c *)
(* did : d *)
let cid = CReset
let did = DReset

(* eval : t * e * c * d -> v *)
let rec eval (t, e, c, d) = match t with
    Var(x) -> run_c (c, get(x, e), d)
  | Fun(x, t) -> run_c (c, VFun(x, t, e), d)
  | App(t0, t1) -> eval (t1, e, CApp1(t0, e, c), d)
  | Shift(t) -> eval (t, e, CShift(c), d)
  | Reset(t) -> eval (t, e, cid, DRun(c, d))

(* eval3 : t -> v *)
let eval3 t = eval (t, [], cid, did)
```

Figure 4: `eval3`: defunctionalization of `eval2`

5 Defunctionalization

As a result of two CPS transformations, `eval2` uses many higher-order functions. However, in the lower level implementation, which is our destination, such a high-level feature is not supported. We therefore defunctionalize [15] `eval2` to remove higher-order functions. We call the new evaluator `eval3`.

For every higher-order function (a `fun` statement in OCaml) in `eval2`, defunctionalization (1) creates an object which holds necessary data to run the function, and (2) replaces the higher-order function with the created data. Then, we supply apply functions that perform the same operations

as the original higher-order function. In this article, we denote them `run_*`. For example, when a term `Shift(t)` is passed to `eval2`, the evaluator yields a `fun` statement of OCaml (lines 17 to 20 of `eval2`). In the body of this `fun` statement, `c` is a free variable. The variable `c` is needed to execute the function later. In `eval2`, the evaluator refers to the scope right out of the `fun` statement for the binding of `c`. On the other hand, the defunctionalized evaluator has to hold the information when it yields the corresponding object. Thus, the object corresponding to the `fun` statement in executing the term `Shift(t)` is defined to hold `c` as an argument. For other `fun` statements, free variables in the body are similarly held in the objects. Here is the changed definition of types:

```
1  type v = (* no change *)  (* value *)
2  and e = (* no change *)   (* environment *)
3  (* continuation *)
4  and c = CApp1 of t * e * c
5            (* the outer fun statement of App *)
6            (* lines 11-15 in Fig. 2          *)
7        | CApp0 of v * c       (* lines 12-15  *)
8        | CShift of c          (* lines 17-20  *)
9        | CReset      (* cid, lines 19, 21, 24 *)
10 and d = DRun of c * d        (* line 17       *)
11       | DReset               (* did, line 24  *)
```

Next, we equip the evaluator with apply functions (Figure 3). Operations executed in the body of `fun` statements in `eval2` are executed in these apply functions.

Not only replacing each `fun` statement with the corresponding object, but we call the apply functions when the continuation is applied in `eval2`. For example, when the term `Var(x)` is passed to the evaluator, the evaluator obtains the value of `x` from the environment and passes it to the continuation `c` (line 8 of `eval2`). Since `c` becomes a continuation object (rather than an applicable function) after defunctionalization, we call the apply function (`run_c`) to execute the continuation as in `run_c (c, get(x, e), d)`. We similarly change the evaluator for other terms and for `d`, and obtain `eval3` (Figure 4). [1]

By this transformation, continuations represented as `fun` statements are divided into continuation objects which hold data needed to execute the continuations and the apply functions for continuation objects. Since continuation objects indicate steps of evaluation, we can consider them like a list of code.

Validity of this transformation is derived directly from the validity of defunctionalization.

Proposition 2 (Validity of Defunctionalization).
For an arbitrary term `t`*,* `eval2` *and* `eval3` *both fail to yield values or both yield values which are structurally equal. (The value which is given by defunctionalizing the result of evaluation in* `eval2` *is equal to the result of evaluation in* `eval3`*.)*

6 Continuation Linearization

In the last section, we mentioned that continuation objects are like a list of code. However, continuation objects in `eval3` are not represented as a list. Because each continuation object has at

[1] To be more precise, the two apply functions in Figure 3 and `eval` in Figure 4 need to be declared as mutually recursive functions.

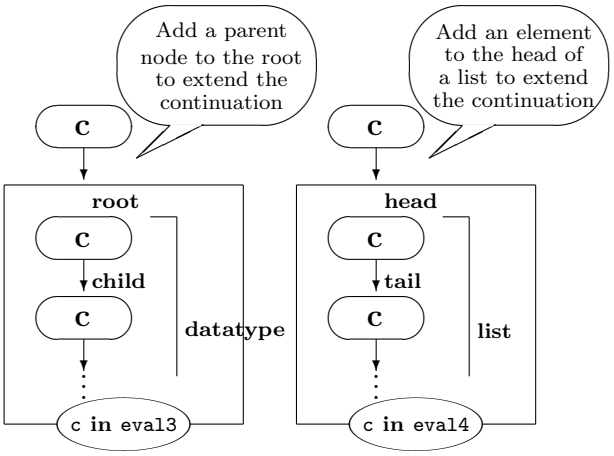

Figure 5: Continuations in `eval3` and `eval4`

most one child, we can transform the evaluator to transfer `c` as a list (Figure 5). We call the new evaluator `eval4`. After this transformation, we can say the argument `c` is a code list.

In `eval3`, continuation objects `CApp1`, `CApp0`, and `CShift` hold data of type `c` as an argument. We therefore change these terms to have no data of type `c`. Instead, the argument `c` is replaced with a list of elements of type `c`. While the data of type `c` is held in a continuation object in `eval3`, a new continuation object is added to the head of the list in `eval4`. Then, data of type `c` to be held is transferred as the tail of the list. In the case of `CReset` which has no data of type `c`, an empty list is passed in place of the continuation object. We make a similar change in terms of `d`. Definition of types is modified as follows:

```
1  (* value *)
2  type v = VFun of string * t * e
3         | VCont of c list (* changed to c list *)
4  and e = (* no change *)         (* environment *)
5  (* continuation *)
6  and c = CApp1 of t * e             (* c removed *)
7        | CApp0 of v
8        | CShift
9               (* CReset removed. use [] instead *)
10 and d = DRun of c list             (* d removed *)
11              (* DReset removed. use [] instead *)
```

The evaluator `eval4` is implemented as in Figure 6. As noted above, the last two arguments of the evaluator are changed to `c list` and `d list`. In line 13, the evaluator passes an empty list (`cid`) in stead of `CReset`, and the current continuation is packaged as `DRun(c)`, which is added to the head of the list. We can observe that the reset operator moves the current continuation from `c` to `d` and `d` is a list of lists of code. Usually, shift/reset is implemented as follows: The reset operator puts a mark on the continuation and the shift operator captures the continuation up to the mark. Here, shift/reset is implemented as follows: The reset operator saves the current continuation into the metacontinuation (`d`) and the shift operator captures the active continuation which is not saved in `d`. These two styles of implementations are convertible with each other.

```
(* identity function *)
(* cid : c list *)
(* did : d list *)
let cid = []
let did = []

(* eval : t * e * c list * d list -> v *)
let rec eval (t, e, c, d) = match t with
    Var(x) -> run_c (c, get(x, e), d)
  | Fun(x, t) -> run_c (c, VFun(x, t, e), d)
  | App(t0, t1) -> eval (t1, e, CApp1(t0, e) :: c, d)
  | Shift(t) -> eval (t, e, CShift :: c, d)
  | Reset(t) -> eval (t, e, cid, DRun(c) :: d)

(* run_c : c list * v * d list -> v *)
and run_c (c, v, d) = match c with
    CApp1(t', e') :: c' -> eval(t', e', CApp0(v) :: c', d)
  | CApp0(v') :: c' -> (match v with
                          VFun(x', t', e')
                            -> eval(t', (x', v') :: e', c', d)
                        | VCont(c'') -> run_c (c'', v', DRun(c') :: d))
  | CShift :: c' -> (match v with
                       VFun(x', t', e')
                         -> eval(t', (x', VCont(c')) :: e', cid, d)
                     | VCont(c'') -> run_c (c'', VCont(c'), d))
  | [] -> run_d (d, v)

(* run_d : d list * v -> v *)
and run_d (d, v) = match d with
    DRun(c') :: d' -> run_c (c', v, d')
  | [] -> v

(* eval4 : t -> v *)
let eval4 t = eval (t, [], cid, did)
```

Figure 6: `eval4`: continuation-linearization of `eval3`

Validity of this transformation is easily seen from the homomorphism between `c` in `eval3` and `c list` in `eval4`.

Proposition 3 (Validity of continuation linearization).
For an arbitrary term `t`, `eval3` *and* `eval4` *both fail to yield values or both yield values which are structurally equal. (The value which is given by linearizing the result of evaluation in* `eval3` *is equal to the result of evaluation in* `eval4`*.)*

```
1  (* cid : c list *)
2  (* did : d list *)
3  let cid = []
4  let did = []
5
6  (* eval : t * s * e * c * d -> v *)
7  let rec eval (t, s, e, c, d) = match t with
8      Var(x) -> run_c (c, get(x, e) :: s, d)
9    | Fun(x, t) -> run_c (c, VFun(x, t, e) :: s, d)
10   | App(t0, t1) -> eval (t1, s, e, CApp1(t0, e) :: c, d)
11   | Shift(t) -> eval (t, s, e, CShift :: c, d)
12   | Reset(t) -> eval (t, [], e, cid, DRun(c, s) :: d)
13
14 (* run_c : c * s * d -> v *)
15 and run_c (c, s, d) = match c with
16     CApp1(t', e') :: c' -> eval(t', s, e', CApp0 :: c', d)
17   | CApp0 :: c' ->
18       (match s with
19            VFun(x', t', e') :: v' :: s'
20            -> eval(t', s', (x', v') :: e', c', d)
21          | VCont(c'', s'') :: v' :: s'
22            -> run_c (c'', v' :: s'', DRun(c', s') :: d))
23   | CShift :: c' ->
24       (match s with
25            VFun(x', t', e') :: s'
26            -> eval(t', [], (x', VCont(c', s')) :: e', cid, d)
27          | VCont(c'', s'') :: s' -> run_c (c'', VCont(c', s') :: s'', d))
28   | [] -> run_d (d, s)
29
30 (* run_d : d * s -> v *)
31 and run_d (d, s) = match (d, s) with
32     (DRun(c', s') :: d', v :: _) -> run_c (c', v :: s', d')
33   | ([], v :: _) -> v
34
35 (* eval5 : t -> v *)
36 let eval5 t = eval (t, [], [], cid, did)
```

Figure 7: `eval5`: Stack introduction of `eval4`

7 Stack introduction

7.1 Transformation to store values in the stack

In `eval4`, values are held in continuation objects. For example, `CApp0` in `eval4` has `v` as its argument. However, in machine languages, intermediate results are stored in a stack. To close this gap, we provide the evaluator with a stack as an argument and store values in the stack. We call the new evaluator `eval5`. In this article, the stack is implemented as a list of values. Since values that were held in continuation objects are now stored in the stack, `c` in `eval4` is divided into the

continuation object and the stack. We need to carry not only `c list` but the stack whenever we use continuations. Accordingly, definition of types are changed as follows:

```
1  (* value *)
2  type v = VFun of string * t * e
3         | VCont of c list * s (* c list and s *)
4  (* stack *)
5  and s = v list                    (* list of v *)
6  and e = (* no change *)         (* environment *)
7  (* continuation *)
8  and c = CApp1 of t * e
9        | CApp0                      (* no value *)
10       | CShift
11 and d = DRun of c list * s    (* c list and s *)
```

While `eval4` transfers values using `c` as in `CApp0(v)`, the new evaluator stores them in the stack. Therefore, a continuation object `CApp0` is modified to have no argument. The type of continuations is modified to `c list * s`. (It was just `c list` in `eval4`.) Arguments of `VCont` and `DRun` are also modified.

Then, the specification of the new evaluator `eval5` is in Figure 7. As noted above, the stack `s` is transferred as an argument of the evaluator and the apply functions. The evaluator stores a value at the top of the stack and passes the entire stack instead of passing only one value (lines 8 and 9 of `eval5`). On applying to continuations, the evaluator restores the stack besides the continuation object list.

7.2 Correctness of the transformation

Intuitive explanation of stack introduction and definition of the evaluator are given in Section 7.1. However, whether stack introduction is a valid transformation is not immediately clear. In fact, it is difficult to show simple equivalence between `eval4` and `eval5` because they use different types for values and continuations. In this section, we show the equivalence between `eval4` and `eval5` by a bisimulation method.

First, we introduce the notion of bisimulation.

Definition 1 (bisimulation [13]).
Two states P and Q are bisimilar*, written $P \sim Q$, if, for all α,*

- *Whenever $P \xrightarrow{\alpha} P'$ then, for some Q', $Q \xrightarrow{\alpha'} Q'$ and $P' \sim Q'$*
- *Whenever $Q \xrightarrow{\alpha'} Q'$ then, for some P', $P \xrightarrow{\alpha} P'$ and $P' \sim Q'$*

A bisimulation is an equivalence relation between state transition systems.

A defunctionalized CPS interpreter implements an abstract machine [2, 4, 7]. We can consider arguments of `eval`, `run_c`, and `run_d` as well as the final result of `run_d` as states. We write them $\langle t, e, c, d\rangle$, $\langle c, v, d\rangle$, $\langle d, v\rangle$, and $\langle v\rangle$, respectively. Then, `eval4` is regarded as a state transition machine having the transition rules denoted in Figure 8. Similarly, we can consider `eval5` as a state transition machine having transition rules shown in Figure 9.

Then, we show the equivalence between `eval4` and `eval5` by proving bisimulation between these state transition systems. Prior to the proof, we determine the relation between states in `eval4` and

$$
\begin{array}{rcl}
t & \Rightarrow & \langle t, [], [], [] \rangle \\
\hline
\langle \texttt{Var}(x), e, c, d \rangle & \Rightarrow & \langle c, get(x, e), d \rangle \\
\langle \texttt{Fun}(x, t), e, c, d \rangle & \Rightarrow & \langle c, \texttt{VFun}(x, t, e), d \rangle \\
\langle \texttt{App}(t0, t1), e, c, d \rangle & \Rightarrow & \langle t1, e, \texttt{CApp1}(t0, e) :: c, d \rangle \\
\langle \texttt{Shift}(t), e, c, d \rangle & \Rightarrow & \langle t, e, \texttt{CShift} :: c, d \rangle \\
\langle \texttt{Reset}(t), e, c, d \rangle & \Rightarrow & \langle t, e, [], \texttt{DRun}(c) :: d \rangle \\
\hline
\langle \texttt{CApp1}(t, e) :: c, v, d \rangle & \Rightarrow & \langle t, e, \texttt{CApp0}(v) :: c, d \rangle \\
\langle \texttt{CApp0}(v) :: c, \texttt{VFun}(x, t, e), d \rangle & \Rightarrow & \langle t, (x, v) :: e, c, d \rangle \\
\langle \texttt{CApp0}(v) :: c, \texttt{VCont}(c'), d \rangle & \Rightarrow & \langle c', v, \texttt{DRun}(c) :: d \rangle \\
\langle \texttt{CShift} :: c, \texttt{VFun}(x, t, e), d \rangle & \Rightarrow & \langle t, (x, \texttt{VCont}(c)) :: e, [], d \rangle \\
\langle \texttt{CShift} :: c, \texttt{VCont}(c'), d \rangle & \Rightarrow & \langle c', \texttt{VCont}(c), d \rangle \\
\langle [], v, d \rangle & \Rightarrow & \langle d, v \rangle \\
\hline
\langle \texttt{DRun}(c') :: d, v \rangle & \Rightarrow & \langle c', v, d \rangle \\
\langle [], v \rangle & \Rightarrow & \langle v \rangle
\end{array}
$$

Figure 8: Transition rules derived from `eval4`

$$
\begin{array}{rcl}
t & \Rightarrow & \langle t, [], [], [], [] \rangle \\
\hline
\langle \texttt{Var}(x), s, e, c, d \rangle & \Rightarrow & \langle c, get(x, e) :: s, d \rangle \\
\langle \texttt{Fun}(x, t), s, e, c, d \rangle & \Rightarrow & \langle c, \texttt{VFun}(x, t, e) :: s, d \rangle \\
\langle \texttt{App}(t0, t1), s, e, c, d \rangle & \Rightarrow & \langle t1, s, e, \texttt{CApp1}(t0, e) :: c, d \rangle \\
\langle \texttt{Shift}(t), s, e, c, d \rangle & \Rightarrow & \langle t, s, e, \texttt{CShift} :: c, d \rangle \\
\langle \texttt{Reset}(t), s, e, c, d \rangle & \Rightarrow & \langle t, [], e, [], \texttt{DRun}(c, s) :: d \rangle \\
\hline
\langle \texttt{CApp1}(t, e) :: c, s, d \rangle & \Rightarrow & \langle t, s, e, \texttt{CApp0} :: c, d \rangle \\
\langle \texttt{CApp0} :: c, \texttt{VFun}(x, t, e) :: v :: s, d \rangle & \Rightarrow & \langle t, s, (x, v) :: e, c, d \rangle \\
\langle \texttt{CApp0} :: c, \texttt{VCont}(c', s') :: v :: s, d \rangle & \Rightarrow & \langle c', v :: s', \texttt{DRun}(c, s) :: d \rangle \\
\langle \texttt{CShift} :: c, \texttt{VFun}(x, t, e) :: s, d \rangle & \Rightarrow & \langle t, [], (x, \texttt{VCont}(c, s)) :: e, [], d \rangle \\
\langle \texttt{CShift} :: c, \texttt{VCont}(c', s') :: s, d \rangle & \Rightarrow & \langle c', \texttt{VCont}(c, s) :: s', d \rangle \\
\langle [], s, d \rangle & \Rightarrow & \langle d, s \rangle \\
\hline
\langle \texttt{DRun}(c', s') :: d, v :: s \rangle & \Rightarrow & \langle c', v :: s', d \rangle \\
\langle [], v :: s \rangle & \Rightarrow & \langle v \rangle
\end{array}
$$

Figure 9: Transition rules derived from `eval5`

in `eval5`. Remember that a continuation object in `eval4` is divided into a stack and a continuation object in `eval5`. With this observation, we define a relation $\_ \stackrel{s}{=}_c \_ \otimes \_$. Intuitively, $c_4 \stackrel{s}{=}_c s \otimes c_5$ means that c_4 in `eval4` is divided into s and c_5 in `eval5` by stack introduction. The notation $\stackrel{s}{=}_c$ intends to be the correspondence between states in `eval4` and in `eval5`. Besides $\stackrel{s}{=}_c$, we need to define three more relations $\stackrel{s}{=}_e$, $\stackrel{s}{=}_d$, $\stackrel{s}{=}_v$. They correspond to the relations of e, c, and d between `eval4` and `eval5`, respectively.

Definition 2 (Relations $\stackrel{s}{=}_c$, $\stackrel{s}{=}_e$, $\stackrel{s}{=}_d$, $\stackrel{s}{=}_v$).
Let e_4, c_4, d_4, and v_4 be values of type `e`, `c`, `d`, *and* `v` *in* `eval4` *and s, e_5, c_5, d_5, and v_5 be values of type* `s`, `e`, `c`, `d`, *and* `v` *in* `eval5`. *Then we define mutually recursive relations $c_4 \stackrel{s}{=}_c s \otimes c_5$, $e_4 \stackrel{s}{=}_e e_5$, $d_4 \stackrel{s}{=}_d d_5$, and $v_4 \stackrel{s}{=}_v v_5$ as the least relations satisfying the following conditions:*

$\boxed{c_4 \overset{s}{=}_c s \otimes c_5}$:

- *If* $c_4 = \texttt{[]}$, $s = \texttt{[]}$ *and* $c_5 = \texttt{[]}$, *then* $c_4 \overset{s}{=}_c s \otimes c_5$
- *If* $c_4 \overset{s}{=}_c s \otimes c_5$ *and* $e_4 \overset{s}{=}_c e_5$, *then* $\texttt{CApp1}_4(t, e_4) :: c_4 \overset{s}{=}_c s \otimes \texttt{CApp1}_5(t, e_5) :: c_5$
- *If* $c_4 \overset{s}{=}_c s \otimes c_5$ *and* $v_4 \overset{s}{=}_c v_5$, *then* $\texttt{CApp0}_4(v_4) :: c_4 \overset{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$
- *If* $c_4 \overset{s}{=}_c s \otimes c_5$, *then* $\texttt{CShift}_4 :: c_4 \overset{s}{=}_c s \otimes \texttt{CShift}_5 :: c_5$

$\boxed{e_4 \overset{s}{=}_e e_5}$:

- *If* $e_4 = \texttt{[]}$ *and* $e_5 = \texttt{[]}$, *then* $e_4 \overset{s}{=}_e e_5$
- *If* $e_4 \overset{s}{=}_e e_5$ *and* $v_4 \overset{s}{=}_v v_5$, *then* $(x, v_4) :: e_4 \overset{s}{=}_e (x, v_5) :: e_5$

$\boxed{d_4 \overset{s}{=}_d d_5}$:

- *If* $d_4 = \texttt{[]}$ *and* $d_5 = \texttt{[]}$, *then* $d_4 \overset{s}{=}_d d_5$
- *If* $d_4 \overset{s}{=}_d d_5$ *and* $c_4 \overset{s}{=}_c s \otimes c_5$, *then* $\texttt{DRun}_4(c_4) :: d_4 \overset{s}{=}_d \texttt{DRun}_5(c_5, s)$

$\boxed{v_4 \overset{s}{=}_v v_5}$:

- *If* $e_4 \overset{s}{=}_e e_5$, *then* $\texttt{VFun}_4(x, t, e_4) \overset{s}{=}_v \texttt{VFun}_5(x, t, e_5)$
- *If* $c_4 \overset{s}{=}_c s \otimes c_5$, *then* $\texttt{VCont}_4(c_4) \overset{s}{=}_v \texttt{VCont}_5(c_5, s)$

Now, we define a relation between the states in `eval4` and `eval5`, and prove that it is bisimilar.

Definition 3 (Relation between evaluators before and after stack introduction). *The relation* $\sim_s$ *is defined as the least relation satisfying one of the following conditions:*

- *If* $c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, $d_4 \overset{s}{=}_d d_5$ *and* $v_4 \overset{s}{=}_v v_5$, *then* $\langle t, e_4, c_4, d_4 \rangle \sim_s \langle t, s, e_5, c_5, d_5 \rangle$ *for any* t.
- *If* $c_4 \overset{s}{=}_c s \otimes c_5$, $d_4 \overset{s}{=}_d d_5$ *and* $v_4 \overset{s}{=}_v v_5$, *then* $\langle c_4, v_4, d_4 \rangle \sim_s \langle c_5, v_5 :: s, d_5 \rangle$
- *If* $d_4 \overset{s}{=}_d d_5$ *and* $v_4 \overset{s}{=}_v v_5$, *then* $\langle d_4, v_4 \rangle \sim_s \langle d_5, v_5 :: s' \rangle$
- *If* $v_4 \overset{s}{=}_v v_5$, *then* $\langle v_4 \rangle \sim_s \langle v_5 \rangle$

Theorem 1 (Bisimulation of $\sim_s$). *The relation* $\sim_s$ *is a bisimulation.*

Proof. We write S_4 and S_5 for states of `eval4` and `eval5`, respectively. Assume that $P \in S_4$, $Q \in S_5$ and $P \sim_s Q$. We prove that $P \xrightarrow{\text{eval4}} P'$ implies $Q \xrightarrow{\text{eval5}} Q'$ and $P' \sim_s Q'$ for some Q', and that $Q \xrightarrow{\text{eval5}} Q'$ implies $P \xrightarrow{\text{eval4}} P'$ and $P' \sim_s Q'$ for some P'. We investigate whether the statements are true in each of the transition rules. By assumption, we have $c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, $d_4 \overset{s}{=}_d d_5$, and $v_4 \overset{s}{=}_v v_5$. Using these conditions, we can derive corresponding relations between arguments after transitions.

Details of the proof for each case is presented in Appendix. □

Because evaluations in `eval4` and `eval5` are both deterministic, validity of the stack introduction is derived from the bisimulation relation between state transition systems.

Theorem 2 (Validity of stack introduction).
For an arbitrary term `t`*,* `eval4` *and* `eval5` *both fail to yield values or both yield values which are structurally equal. (Assume that* v_4 *is the result of evaluation in* `eval4` *and* v_5 *is the result of evaluation in* `eval5`*. Then, we have* $v_4 \stackrel{s}{=}_v v_5$*.)*

This is one of the important results of this research that the validity of stack introduction is shown formally. Although a stack is often used in the low-level description of a language, it is not entirely clear how it relates to high-level description of the language, such as an interpreter, especially in the presence of shift/reset. In fact, while we try to prove the correctness of the transformation, we found and corrected tiny faults in our low-level implementation. To rigorously derive low-level implementations that are equivalent to the high-level description, it is essential to establish the validity of the transformation.

8 Environment storing

8.1 Transformation to save the environment in the stack

On function calls, the evaluator `eval5` saves the current environment in the continuation object (line 10 in `eval5`). Remember that a continuation object list is like a code list. In a machine language, there is no operation that saves environments in the code. Thus, we transform the evaluator to save the environment in the stack, instead of the continuation object. We call the new evaluator `eval6`.

We modify `CApp1` to have no environments. We package the environment as a value to be stored in the stack. Therefore, the definition of types is modified as follows:

```
1  (* value *)
2  type v = VFun of string * t * e
3         | VCont of c list * s
4         | VEnv of e   (* to store environment *)
5  and s = (* no change *)              (* stack *)
6  and e = (* no change *)        (* environment *)
7  (* continuation *)
8  and c = CApp1 of t          (* no environment *)
9        | CApp0
10       | CShift
11 and d = (* no change *)
```

The specification of the evaluator `eval6` is in Figure 10. When functions are called, the new evaluator packages the environment as the value `VEnv(e)`, and stores it in the stack (line 10). This environment is loaded when executing `CApp1` (lines 16-18). This models the behavior of saving and restoring values of variables.

After this transformation, continuation objects have no data except for the term `t`. It means that it is possible to yield code lists from terms only.

```
(* cid : c list *)
(* did : d list *)
let cid = []
let did = []

(* eval : t * s * e * c list * d list -> v *)
let rec eval (t, s, e, c, d) = match t with
    Var(x) -> run_c (c, get(x, e) :: s, d)
  | Fun(x, t) -> run_c (c, VFun(x, t, e) :: s, d)
  | App(t0, t1) -> eval (t1, VEnv(e) :: s, e, CApp1(t0) :: c, d)
  | Shift(t) -> eval (t, s, e, CShift :: c, d)
  | Reset(t) -> eval (t, [], e, CReset :: [], DRun(c, s) :: d)

(* run_c : c list * s * d -> v *)
and run_c (c, s, d) = match c with
    CApp1(t') :: c' ->
      (match s with
           v :: VEnv(e') :: s -> eval(t', v :: s, e', CApp0 :: c', d))
  | CApp0 :: c' ->
      (match s with
           VFun(x', t', e') :: v' :: s'
           -> eval(t', s', (x', v') :: e', c', d)
         | VCont(c'', s'') :: v' :: s'
           -> run_c (c'', v' :: s'', DRun(c', s') :: d))
  | CShift :: c' ->
      (match s with
           VFun(x', t', e') :: s'
             -> eval(t', [], (x', VCont(c', s')) :: e', CReset :: [], d)
         | VCont(c'', s'') :: s' -> run_c (c'', VCont(c', s') :: s'', d))
  | [] -> run_d (d, s)

(* run_d : d  list * s -> v *)
and run_d (d, s) = match (d, s) with
    (DRun(c', s') :: d', v :: _) -> run_c (c', v :: s', d')
  | ([], v :: _) -> v

(* eval6 : t -> v *)
let eval6 t = eval (t, [], [], cid, did)
```

Figure 10: `eval6`: an environment storing version of `eval5`

t	$\Rightarrow$	$\langle t, [], [], [], [] \rangle$
$\langle \texttt{Var}(x), s, e, c, d \rangle$	$\Rightarrow$	$\langle c, get(x, e) :: s, d \rangle$
$\langle \texttt{Fun}(x, t), s, e, c, d \rangle$	$\Rightarrow$	$\langle c, \texttt{VFun}(x, t, e) :: s, d \rangle$
$\langle \texttt{App}(t0, t1), s, e, c, d \rangle$	$\Rightarrow$	$\langle t1, \texttt{VEnv}(e) :: s, e, \texttt{CApp1}(t0) :: c, d \rangle$
$\langle \texttt{Shift}(t), s, e, c, d \rangle$	$\Rightarrow$	$\langle t, s, e, \texttt{CShift} :: c, d \rangle$
$\langle \texttt{Reset}(t), s, e, c, d \rangle$	$\Rightarrow$	$\langle t, [], e, [], \texttt{DRun}(c, s) :: d \rangle$
$\langle \texttt{CApp1}(t) :: c, v :: \texttt{VEnv}(e) :: s, d \rangle$	$\Rightarrow$	$\langle t, v :: s, e, \texttt{CApp0} :: c, d \rangle$
$\langle \texttt{CApp0} :: c, \texttt{VFun}(x, t, e) :: v :: s, d \rangle$	$\Rightarrow$	$\langle t, s, (x, v) :: e, c, d \rangle$
$\langle \texttt{CApp0} :: c, \texttt{VCont}(c', s') :: v :: s, d \rangle$	$\Rightarrow$	$\langle c', v :: s', \texttt{DRun}(c, s) :: d \rangle$
$\langle \texttt{CShift} :: c, \texttt{VFun}(x, t, e) :: s, d \rangle$	$\Rightarrow$	$\langle t, [], (x, \texttt{VCont}(c, s)) :: e, [], d \rangle$
$\langle \texttt{CShift} :: c, \texttt{VCont}(c', s') :: s, d \rangle$	$\Rightarrow$	$\langle c', \texttt{VCont}(c, s) :: s', d \rangle$
$\langle [], s, d \rangle$	$\Rightarrow$	$\langle d, s \rangle$
$\langle \texttt{DRun}(c', s') :: d, v :: s \rangle$	$\Rightarrow$	$\langle c', v :: s', d \rangle$
$\langle [], v :: s \rangle$	$\Rightarrow$	$\langle v \rangle$

Figure 11: Transition rules derived from `eval6`

8.2 Correctness of the transformation

As in Section 7.2, we consider `eval6` as a state transition machine, whose transition rules are in Figure 11.

As in Section 7.2, we define the relation between s_5, c_5 in `eval5` and s_6, c_6 in `eval6`. Intuitively, $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ means that the environments stored in c_5 in `eval5` are stored in s_6 in `eval6`. We also define the relations $\stackrel{e}{=}_e$, $\stackrel{e}{=}_d$, and $\stackrel{e}{=}_v$.

Definition 4 (Relations $\stackrel{e}{=}_c$, $\stackrel{e}{=}_e$, $\stackrel{e}{=}_d$, $\stackrel{e}{=}_v$).
Let s_5, e_5, c_5, d_5, and v_5 be values of type `s`, `e`, `c`, `d`, *and* `v` *of* `eval5` *and s_6, e_6, c_6, d_6, and v_6 be values of type* `s`, `e`, `c`, `d`, *and* `v` *of* `eval6`. *Then we define mutually recursive relations $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, $e_5 \stackrel{e}{=}_e e_6$, $d_5 \stackrel{e}{=}_d d_6$, and $v_5 \stackrel{e}{=}_v v_6$ as the least relations satisfying the following conditions:*

$\boxed{s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6}$:

- *If $c_5 = []$, $s_5 = []$, $c_6 = []$, and $s_6 = []$, then $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$*
- *If $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ and $e_5 \stackrel{e}{=}_e e_6$, then $s_5 \otimes \texttt{CApp1}_5(t, e_5) :: c_5 \stackrel{e}{=}_c \texttt{VEnv}(e_6) :: s_6 \odot \texttt{CApp1}_6(t) :: c_6$*
- *If $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ and $v_5 \stackrel{e}{=}_v v_6$, then $v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6 :: c_6$*
- *If $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, then $s_5 \otimes \texttt{CShift}_5 :: c_5 \stackrel{e}{=}_c s_6 \odot \texttt{CShift}_6 :: c_6$*

$\boxed{e_5 \stackrel{e}{=}_e e_6}$:

- *If $e_5 = []$ and $e_6 = []$, then $e_5 \stackrel{e}{=}_e e_6$*
- *If $e_5 \stackrel{e}{=}_e e_6$ and $v_5 \stackrel{e}{=}_v v_6$, then $(x, v_5) :: e_5 \stackrel{e}{=}_e (x, v_6) :: e_6$*

$\boxed{d_5 \stackrel{e}{=}_d d_6}$:

- *If $d_5 = []$ and $d_6 = []$, then $d_5 \stackrel{e}{=}_d d_6$*

- *If* $d_5 \stackrel{e}{=}_d d_6$ *and* $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$*, then* $\mathtt{DRun}_5(c_5, s_5) :: d_5 \stackrel{e}{=}_d \mathtt{DRun}_6(c_6, s_6) :: d_6$

$\boxed{v_5 \stackrel{e}{=}_v v_6}$:

- *If* $e_5 \stackrel{e}{=}_e e_6$*, then* $\mathtt{VFun}_5(x, t, e_5) \stackrel{e}{=}_v \mathtt{VFun}_6(x, t, e_6)$
- *If* $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$*, then* $\mathtt{VCont}_5(c_5, s_5) \stackrel{e}{=}_v \mathtt{VCont}_6(c_6, s_6)$

Now, we define a relation between the states in `eval5` and `eval6`, and prove that it is bisimilar.

Definition 5 (Relation between evaluators before and after environment storing). *The relation* $\sim_e$ *is defined as the least relation satisfying one of the following conditions:*

- *If* $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$*,* $e_5 \stackrel{e}{=}_e e_6$*,* $d_5 \stackrel{e}{=}_d d_6$ *and* $v_5 \stackrel{e}{=}_v v_6$*, then* $\langle t, s_5, e_5, c_5, d_5\rangle \sim_e \langle t, s_6, e_6, c_6, d_6\rangle$ *for any* t*.*
- *If* $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$*,* $d_5 \stackrel{e}{=}_d d_6$ *and* $v_5 \stackrel{e}{=}_v v_6$*, then* $\langle c_5, v_5 :: s_5, d_5\rangle \sim_e \langle c_6, v_6 :: s_6, d_6\rangle$
- *If* $d_5 \stackrel{e}{=}_d d_6$ *and* $v_5 \stackrel{e}{=}_v v_6$*, then* $\langle d_5, v_5 :: s_5\rangle \sim_e \langle d_6, v_6 :: s_6\rangle$
- *If* $v_5 \stackrel{e}{=}_v v_6$*, then* $\langle v_5\rangle \sim_e \langle v_6\rangle$

Then, $\sim_e$ *is a bisimulation.*

Theorem 3 (Bisimulation of $\sim_e$). *The relation* $\sim_e$ *is a bisimulation.*

Proof. The proof is similar to the one for Theorem 1. Details are presented in [10]. □

Because evaluations in `eval5` and `eval6` are both deterministic, validity of the stack introduction is derived from the bisimulation relation between state transition systems.

Theorem 4 (Validity of environment storing). *For an arbitrary term* `t`*,* `eval5` *and* `eval6` *both fail to yield values or both yield values which are structurally equal. (Assume that* v_5 *is the result of evaluation in* `eval5` *and* v_6 *is the result of evaluation in* `eval6`*. Then, we have* $v_5 \stackrel{e}{=}_v v_6$*.)*

9 Abstract Machine

As a result of transformations above, we obtained the abstract machine whose transition rules are in Figure 11. This abstract machine is similar to Landin's SECD machine [11]. However, our abstract machine differs from the SECD machine in two points. That is:

- environments are saved in the stack, and
- shift/reset is supported (instead of J operator)

The former models the calling convention typically found in an implementation in a machine language. This shows that the "functional correspondence" approach scales to derive lower-level implementations than previously known.

In terms of the latter, our abstract machine can be regarded as modeling the implementation of shift/reset in a machine language. Observe that s@d corresponds to a stack in the implementation of a machine language. Then, @ becomes the position of the reset mark and the shift operator captures the continuation up to the reset mark. When `CShift` is executed in our abstract machine, the current stack and code are captured in `VCont`. This corresponds exactly to the creation of a continuation closure which contains a pointer to the copied stack and a code pointer representing the continuation in the low-level implementation [12]. Since the abstract machine is derived using validated transformations only, it effectively shows the correctness of the low-level implementation of shift/reset.

10 Conclusion and Issues

In this paper, we have derived through a series of transformations an abstract machine for the λ-calculus with shift/reset that saves the bindings of variables in the stack. The used transformations are CPS transformation, defunctionalization, continuation linearization, stack introduction, and environment storing. Among them, we have formally shown the correctness of the last two transformations. Although the obtained abstract machine is similar to Landin's SECD machine, there exists an important difference: our abstract machine saves environments in the stack, properly modeling the standard calling convention of function calls. Moreover, the abstract machine models the direct implementation of shift/reset that copies a part of the stack, opening a door for formally validating the low-level implementation of shift/reset.

In the future, we plan to divide the abstract machine into a compiler and a virtual machine, following the method shown by Ager, Biernacki, Danvy, and Midtgaard [1]. Preliminary investigation shows that we can obtain the virtual machine without problems by currying `eval6` so that it receives a term first (corresponding to a compiler) and then the rest of the arguments (corresponding to a virtual machine). Ultimately, we hope to extend the functional correspondence approach to cover the low-level implementations and establish the correctness of the direct implementation of shift/reset written in the assembly language [12].

References

[1] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard: "From Interpreter to Compiler and Virtual Machine: A Functional Derivation," Technical Report RS-03-14, BRICS, Aarhus, Denmark (March 2003).

[2] M. S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard: "A Functional Correspondence between Evaluators and Abstract Machines," Technical Report RS-03-13, BRICS, Aarhus, Denmark (March 2003).

[3] K. Asai, and Y. Kameyama: "Polymorphic Delimited Continuations," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 239–254 (November 2007).

[4] M. Biernacka, D. Biernacki, and O. Danvy "An Operational Foundation for Delimited Continuations in the CPS Hierarchy," *Logical Methods in Computer Science*, Vol. 1 (2:5), pp. 1–39 (November 2005).

[5] O. Danvy, and A. Filinski: "A Functional Abstraction of Typed Contexts," Technical Report 89/12, DIKU, University of Copenhagen (July 1989).

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

[7] O. Danvy, and K. Millikin: "A Rational Deconstruction of Landin's J Operator," *Logical Methods in Computer Science*, Vol. 4 (4:12), pp. 1–67, (November 2008).

[8] M. Felleisen: "The Theory and Practice of First-Class Prompts," *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 180–190 (January 1988).

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

[10] A. Kitani and K. Asai: "Derivation of a Abstract Machine for Delimited continuation Constructs," Technical Report OCHA-IS 08-3, Ochanomizu University, (February 2009). In Japanese. English version is in preparation.

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

[12] M. Masuko and K. Asai "Direct Implementation of Shift and Reset in the MinCaml Compiler," Submitted for publication (March 2009).

[13] R. Milner: *Communication and Concurrency*, Prentice Hall International Series in Computer Science, (1995).

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

[15] J. C. Reynolds: "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).

[16] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten: "Revised^6 report on the algorithmic language Scheme", http://www.r6rs.org/ (2007).

A Proof of Theorem 1

Here we present details of the proof omitted in the body of this article.

Theorem 4 (Bisimulation of $\sim_s$)
The relation $\sim_s$ is a bisimulation.

Proof. We write S_4 and S_5 as states of `eval4` and `eval5`, respectively. Assume that $P \in S_4$, $Q \in S_5$ and $P \sim_s Q$. We prove that $P \xrightarrow{\text{eval4}} P'$ implies $Q \xrightarrow{\text{eval5}} Q'$ and $P' \sim_s Q'$ for some Q', and that $Q \xrightarrow{\text{eval5}} Q'$ implies $P \xrightarrow{\text{eval4}} P'$ and $P' \sim_s Q'$ for some P'.

First, we prove the former. By the assumption, we have $c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, $d_4 \overset{s}{=}_d d_5$, and $v_4 \overset{s}{=}_v v_5$.

Case $P : \langle \texttt{Var}(x), e_4, c_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle c_4, get(x, e_4), d_4 \rangle$:
In this case
$Q : \langle \texttt{Var}(x), s, e_5, c_5, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle c_5, get(x, e_5) :: s, d_5 \rangle$,
and we have
$c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, and $d_4 \overset{s}{=}_d d_5$
Because $e_4 \overset{s}{=}_e e_5$, we have
$get(x, e_4) \overset{s}{=}_v get(x, e_5)$
So, P' and Q' satisfies the second clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{Fun}(x, t), e_4, c_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle c_4, \texttt{VFun}_4(x, t, e_4), d_4 \rangle$:
We can prove that $P \xrightarrow{\text{eval4}} P'$ implies $Q \xrightarrow{\text{eval5}} Q'$ and $P' \sim_s Q'$ for some Q', similarly to the case above.

Case $P : \langle \texttt{App}(t0, t1), e_4, c_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t1, e_4, \texttt{CApp}_4(t0, e_4) :: c_4, d_4 \rangle$:
In this case,
$Q : \langle \texttt{App}(t0, t1), s, e_5, c_5, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle t1, s, e_5, \texttt{CApp}_5(t0, e_5) :: c_5, d_5 \rangle$
and we have
$c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, and $d_4 \overset{s}{=}_d d_5$
Because $c_4 \overset{s}{=}_c s \otimes c_5$ and $e_4 \overset{s}{=}_e e_5$, we have
$\texttt{CApp}_4(t1, e_4) :: c_4 \overset{s}{=}_c s \otimes \texttt{CApp}_5(t1, e_5) :: c_5$
So, P' and Q' satisfies the first clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{Shift}(t), e_4, c_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t, e_4, \texttt{CShift}_4 :: c_4, d_4 \rangle$:
We can prove that $P \xrightarrow{\text{eval4}} P'$ implies $Q \xrightarrow{\text{eval5}} Q'$ and $P' \sim_s Q'$ for some Q', similarly to the case above.

Case $P : \langle \texttt{Reset}(t), e_4, c_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t, e_4, [], \texttt{DRun}_4(c_4) :: d_4 \rangle$:
In this case,
$Q : \langle \texttt{Reset}(t), s, e_5, c_5, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle t, [], e_5, [], \texttt{DRun}_5(c_5, s) :: d_5 \rangle$,
and we have
$c_4 \overset{s}{=}_c s \otimes c_5$, $e_4 \overset{s}{=}_e e_5$, and $d_4 \overset{s}{=}_d d_5$
We have $[] \overset{s}{=}_c [] \otimes []$ due to Definition 2.
Because $c_4 \overset{s}{=}_c s \otimes c_5$, we have
$\texttt{DRun}_4(c_4) :: d_4 \overset{s}{=}_d \texttt{DRun}_5(c_5, s) :: d_5$
So, P' and Q' satisfies the first clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{CApp1}_4(t, e_4) :: c_4, v_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t, e_4, \texttt{CApp0}_4(v_4) :: c_4, d_4 \rangle$:
In this case,
$Q : \langle \texttt{CApp1}_5(t, e_5) :: c_5, v_5 :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle t, v_5 :: s, e_5, \texttt{CApp0}_5 :: c_5, d_5 \rangle$,
and we have
$\texttt{CApp1}_4(t, e_4) :: c_4 \overset{s}{=}_c s \otimes \texttt{CApp1}_5(t, e_5) :: c_5$, $d_4 \overset{s}{=}_d d_5$, and $v_4 \overset{s}{=}_v v_5$
Because $\texttt{CApp1}_4(t, e_4) :: c_4 \overset{s}{=}_c s \otimes \texttt{CApp1}_5(t, e_5) :: c_5$, we have

$c_4 \stackrel{s}{=}_c s \otimes c_5$

Then,

$\texttt{CApp0}_4(v_4) :: c_4 \stackrel{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$.

So, P' and Q' satisfies the first clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{CApp0}_4(v_4) :: c_4, \texttt{VFun}_4(x, t, e_4), d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t, (x, v_4) :: e_4, c_4, d_4 \rangle$:

In this case,

$Q : \langle \texttt{CApp0}_5 :: c_5, \texttt{VFun}_5(x, t, e_5) :: v_5 :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle t, s, (x, v_5) :: e_5, c_5, d_5 \rangle$,

and we have

$\texttt{CApp0}_4(v_4) :: c_4 \stackrel{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$, $e_4 \stackrel{s}{=}_e e_5$, and $d_4 \stackrel{s}{=}_d d_5$

Because $\texttt{CApp0}_4(v_4) :: c_4 \stackrel{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$, we have

$c_4 \stackrel{s}{=}_c s \otimes c_5$ and $v_4 \stackrel{s}{=}_v v_5$

We also have $(x, v_4) :: e_4 \stackrel{s}{=}_e (x, v_5) :: e_5$ due to $v_4 \stackrel{s}{=}_v v_5$ and $e_4 \stackrel{s}{=}_e e_5$.

So, P' and Q' satisfies the first clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{CApp0}_4(v_4) :: c_4, \texttt{VCont}_4(c_4'), d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle c_4', v_4, \texttt{DRun}_4(c_4) :: d_4 \rangle$:

In this case,

$Q : \langle \texttt{CApp0}_5 :: c_5, \texttt{VCont}_5(c_5', s') :: v_5 :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle c_5', v_5 :: s', \texttt{DRun}_5(c_5, s) :: d_5 \rangle$,

and we have

$\texttt{CApp0}_4(v_4) :: c_4 \stackrel{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$, $d_4 \stackrel{s}{=}_d d_5$, and $v_4 \stackrel{s}{=}_v v_5$

Because $\texttt{CApp0}_4(v_4) :: c_4 \stackrel{s}{=}_c v_5 :: s \otimes \texttt{CApp0}_5 :: c_5$, we have

$c_4 \stackrel{s}{=}_c s \otimes c_5$ and $v_4 \stackrel{s}{=}_v v_5$

Then,

$\texttt{DRun}_4(c_4) :: d_4 \stackrel{s}{=}_d \texttt{DRun}_5(c_5, s) :: d_5$

So, P' and Q' satisfies the second clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{CShift}_4 :: c_4, \texttt{VFun}_4(x, t, e_4), d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle t, (x, \texttt{VCont}_4(c_4)) :: e_4, [], d_4 \rangle$:

In this case,

$Q : \langle \texttt{CShift}_5 :: c_5, \texttt{VFun}_5(x, t, e_5) :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle t, [], (x, \texttt{VCont}_5(c_5, s)) :: e_5, [], d_5 \rangle$,

and we have

$\texttt{CShift}_4 :: c_4 \stackrel{s}{=}_c s \otimes \texttt{CShift}_5 :: c_5$, $d_4 \stackrel{s}{=}_d d_5$, and $\texttt{VFun}_4(x, t, e_4) \stackrel{s}{=}_v \texttt{VFun}_5(x, t, e_5)$

We have $[] \stackrel{s}{=}_c [] \otimes []$ due to Definition 2.

Because $\texttt{CShift}_4 :: c_4 \stackrel{s}{=}_c s \otimes \texttt{CShift}_5 :: c_5$, we have

$c_4 \stackrel{s}{=}_c s \otimes c_5$

We also have $e_4 \stackrel{s}{=}_e e_5$ because $\texttt{VFun}_4(x, t, e_4) \stackrel{s}{=}_v \texttt{VFun}_5(x, t, e_5)$.

Then,

$(x, \texttt{VCont}_4(c_4)) :: e_4 \stackrel{s}{=}_e (x, \texttt{VCont}_5(c_5, s)) :: e_5$

So, P' and Q' satisfies the first clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{CShift}_4 :: c_4, \texttt{VCont}_4(c_4'), d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle c_4', \texttt{VCont}_4(c_4), d_4 \rangle$:

In this case,

$Q : \langle \texttt{CShift}_5 :: c_5, \texttt{VCont}_5(c_5', s') :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle c_5', \texttt{VCont}_5(c_5, s) :: s', d_5 \rangle$,

and we have

$\texttt{CShift}_4 :: c_4 \stackrel{s}{=}_c s \otimes \texttt{CShift}_5 :: c_5$, $d_4 \stackrel{s}{=}_d d_5$, and $\texttt{VCont}_4(c_4') \stackrel{s}{=}_v \texttt{VCont}_5(c_5', s')$

Because $\texttt{CShift}_4 :: c_4 \stackrel{s}{=}_c s \otimes \texttt{CShift}_5 :: c_5$, we have

$c_4 \stackrel{s}{=}_c s \otimes c_5$

Then,

$\texttt{VCont}_4(c_4) \stackrel{s}{=}_v \texttt{VCont}_5(c_5, s)$
We also have $c'_4 \stackrel{s}{=}_c s' \otimes c'_5$ because $\texttt{VCont}_4(c'_4) \stackrel{s}{=}_v \texttt{VCont}_5(c'_5, s')$.
So, P' and Q' satisfies the second clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle [], v_4, d_4 \rangle \xrightarrow{\text{eval4}} P' : \langle d_4, v_4 \rangle$:
In this case,
$Q : \langle [], v_5 :: s, d_5 \rangle \xrightarrow{\text{eval5}} Q' : \langle d_5, v_5 :: s \rangle$,
and we have
$d_4 \stackrel{s}{=}_d d_5$, and $v_4 \stackrel{s}{=}_v v_5$
P' and Q' satisfies the third clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle \texttt{DRun}_4(c'_4) :: d_4, v_4 \rangle \xrightarrow{\text{eval4}} P' : \langle c'_4, v_4, d_4 \rangle$:
In this case,
$Q : \langle \texttt{DRun}_5(c'_5, s') :: d_5, v_5 :: s \rangle \xrightarrow{\text{eval5}} Q' : \langle c'_5, v_5 :: s', d_5 \rangle$,
and we have
$\texttt{DRun}_4(c'_4) :: d_4 \stackrel{s}{=}_d \texttt{DRun}_5(c'_5, s') :: d_5$, and $v_4 \stackrel{s}{=}_v v_5$
Because $\texttt{DRun}_4(c'_4) :: d_4 \stackrel{s}{=}_d \texttt{DRun}_5(c'_5, s') :: d_5$, we have
$c'_4 \stackrel{s}{=}_c s' \otimes c'_5$ and $d_4 \stackrel{s}{=}_d d_5$
So, P' and Q' satisfies the second clause of Definition 3, that is, $P' \sim_s Q'$.

Case $P : \langle [], v_4 \rangle \xrightarrow{\text{eval4}} P' : \langle v_4 \rangle$:
In this case,
$Q : \langle [], v_5 :: s \rangle \xrightarrow{\text{eval5}} Q' : \langle v_5 \rangle$
and we have
$v_4 \stackrel{s}{=}_v v_5$
P' and Q' satisfies the fourth of Definition 3, that is, $P' \sim_s Q'$.

Therefore, the former, $P \xrightarrow{\text{eval4}} P'$ implies $Q \xrightarrow{\text{eval5}} Q'$ and $P' \sim_s Q'$ for some Q', is proved.
In terms of the latter, we can prove it in a similar way. Thus, $\sim_s$ is a bisimulation. □

B Proof of Theorem 3

We can prove Theorem 3 in a similar way to Theorem 1.

Theorem 5 (Bisimulation of $\sim_e$)
The relation $\sim_e$ is a bisimulation.

Proof. We write S_5 and S_6 as states of `eval5` and `eval6`, respectively. Assume that $P \in S_5$, $Q \in S_6$ and $P \sim_e Q$. We prove that $P \xrightarrow{\text{eval5}} P'$ implies $Q \xrightarrow{\text{eval6}} Q'$ and $P' \sim_e Q'$ for some Q', and that $Q \xrightarrow{\text{eval6}} Q'$ implies $P \xrightarrow{\text{eval5}} P'$ and $P' \sim_e Q'$ for some P'.

First, we prove the former. By the assumption, we have $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, $e_5 \stackrel{e}{=}_e e_6$, $d_5 \stackrel{e}{=}_d d_6$, and $v_5 \stackrel{e}{=}_v v_6$.

Case $P : \langle \texttt{Var}(x), s_5, e_5, c_5, d_5 \rangle \xrightarrow{\text{eval5}} P' : \langle c_5, get(x, e_5), d_5 \rangle$:
In this case
$Q : \langle \texttt{Var}(x), s_6, e_6, c_6, d_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle c_6, get(x, e_6) :: s_6, d_6 \rangle$,
and we have

$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, $e_5 \stackrel{e}{=}_e e_6$, and $d_5 \stackrel{e}{=}_d d_6$

Because $e_5 \stackrel{e}{=}_e e_6$, we have

$get(x, e_5) \stackrel{e}{=}_v get(x, e_6)$

So, P' and Q' satisfies the second clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{Fun}(x,t), s_5, e_5, c_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle c_5, \texttt{VFun}_5(x,t,e_5), d_5\rangle$:

We can prove that $P \stackrel{\text{eval5}}{\longrightarrow} P'$ implies $Q \stackrel{\text{eval6}}{\longrightarrow} Q'$ and $P' \sim_e Q'$ for some Q', similarly to the case above.

Case $P : \langle \texttt{App}(t0,t1), s_5, e_5, c_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle t1, s_5, e_5, \texttt{CApp}_5(t0, e_5) :: c_5, d_5\rangle$:

In this case,

$Q : \langle \texttt{App}(t0,t1), s_6, e_6, c_6, d_6\rangle \stackrel{\text{eval6}}{\longrightarrow} Q' : \langle t1, \texttt{VEnv}(e_6) :: s_6, e_6, \texttt{CApp}_6(t0) :: c_6, d_6\rangle$

and we have

$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, $e_5 \stackrel{e}{=}_e e_6$, and $d_5 \stackrel{e}{=}_d d_6$

Because $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ and $e_5 \stackrel{e}{=}_e e_6$, we have

$s_5 \otimes \texttt{CApp}_5(t1, e_5) :: c_5 \stackrel{e}{=}_c \texttt{VEnv}(e_6) :: s_6 \odot \texttt{CApp}_6(t1) :: c_6$

So, P' and Q' satisfies the first clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{Shift}(t), s_5, e_5, c_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle t, s_5, e_5, \texttt{CShift}_5 :: c_5, d_5\rangle$:

We can prove that $P \stackrel{\text{eval5}}{\longrightarrow} P'$ implies $Q \stackrel{\text{eval6}}{\longrightarrow} Q'$ and $P' \sim_e Q'$ for some Q', similarly to the case above.

Case $P : \langle \texttt{Reset}(t), s_5, e_5, c_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle t, [], e_5, [], \texttt{DRun}_5(c_5, s_5) :: d_5\rangle$:

In this case,

$Q : \langle \texttt{Reset}(t), s_6, e_6, c_6, d_6\rangle \stackrel{\text{eval6}}{\longrightarrow} Q' : \langle t, [], e_6, [], \texttt{DRun}_6(c_6, s_6) :: d_6\rangle$,

and we have

$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, $e_5 \stackrel{e}{=}_e e_6$, and $d_5 \stackrel{e}{=}_d d_6$

We have $[] \otimes [] \stackrel{e}{=}_c [] \odot []$ due to Definition 2.

Because $s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$, we have

$\texttt{DRun}_5(c_5, s_5) :: d_5 \stackrel{e}{=}_d \texttt{DRun}_6(c_6, s_6) :: d_6$

So, P' and Q' satisfies the first clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{CApp1}_5(t, e_5) :: c_5, v_5 :: s_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle t, v_5 :: s_5, e_5, \texttt{CApp0}_5 :: c_5, d_5\rangle$:

In this case,

$Q : \langle \texttt{CApp1}_6(t) :: c_6, v_6 :: \texttt{VEnv}(e_6) :: s_6, d_6\rangle \stackrel{\text{eval6}}{\longrightarrow} Q' : \langle t, v_6 :: s_6, e_6, \texttt{CApp0}_6 :: c_6, d_6\rangle$,

and we have

$s_5 \otimes \texttt{CApp1}_5(t, e_5) :: c_5 \stackrel{e}{=}_c \texttt{VEnv}(e_6) :: s_6 \odot \texttt{CApp1}_6(t) :: c_6$, and $d_5 \stackrel{e}{=}_d d_6$

Because $s_5 \otimes \texttt{CApp1}_5(t, e_5) :: c_5 \stackrel{e}{=}_c \texttt{VEnv}(e_6) :: s_6 \odot \texttt{CApp1}_6(t, e_6) :: c_6$, we have

$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$

Then,

$v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6(t) :: c_6$

So, P' and Q' satisfies the first clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{CApp0}_5 :: c_5, \texttt{VFun}_5(x,t,e_5) :: v_5 :: s_5, d_5\rangle \stackrel{\text{eval5}}{\longrightarrow} P' : \langle t, s_5, (x, v_5) :: e_5, c_5, d_5\rangle$:

In this case,

$Q : \langle \texttt{CApp0}_6 :: c_6, \texttt{VFun}_6(x,t,e_6) :: v_6 :: s_6, d_6\rangle \stackrel{\text{eval6}}{\longrightarrow} Q' : \langle t, s_6, (x, v_6) :: e_6, c_6, d_6\rangle$,

and we have

$v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6 :: c_6$, $e_5 \stackrel{e}{=}_e e_6$, and $d_5 \stackrel{e}{=}_d d_6$

Because $v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6 :: c_6$, we have
$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ and $v_5 \stackrel{e}{=}_v v_6$
We also have $(x, v_5) :: e_5 \stackrel{e}{=}_e (x, v_6) :: e_6$ due to $v_5 \stackrel{e}{=}_v v_6$ and $e_5 \stackrel{e}{=}_e e_6$.
So, P' and Q' satisfies the first clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{CApp0}_5(v_5) :: c_5, \texttt{VCont}_5(c_5', s_5') :: v_5 :: s_5, d_5 \rangle \xrightarrow{\text{eval5}} P' : \langle c_5', v_5 :: s_5', \texttt{DRun}_5(c_5, s_5) :: d_5 \rangle$:
In this case,
$Q : \langle \texttt{CApp0}_6 :: c_6, \texttt{VCont}_6(c_6', s_6') :: v_6 :: s_6, d_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle c_6', v_6 :: s_6', \texttt{DRun}_6(c_6, s_6) :: d_6 \rangle$,
and we have
$v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6 :: c_6$, $d_5 \stackrel{e}{=}_d d_6$, and $v_5 \stackrel{e}{=}_v v_6$
Because $v_5 :: s_5 \otimes \texttt{CApp0}_5 :: c_5 \stackrel{e}{=}_c v_6 :: s_6 \odot \texttt{CApp0}_6 :: c_6$, we have
$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$ and $v_5 \stackrel{e}{=}_v v_6$
Then,
$\texttt{DRun}_5(c_5, s_5) :: d_5 \stackrel{e}{=}_d \texttt{DRun}_6(c_6, s_6) :: d_6$
So, P' and Q' satisfies the second clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{CShift}_5 :: c_5, \texttt{VFun}_5(x, t, e_5) :: s_5, d_5 \rangle \xrightarrow{\text{eval5}} P' : \langle t, [], (x, \texttt{VCont}_5(c_5, s_5)) :: e_5, [], d_5 \rangle$:
In this case,
$Q : \langle \texttt{CShift}_6 :: c_6, \texttt{VFun}_6(x, t, e_6) :: s_6, d_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle t, [], (x, \texttt{VCont}_6(c_6, s_6)) :: e_6, [], d_6 \rangle$,
and we have
$s_5 \otimes \texttt{CShift}_5 :: c_5 \stackrel{e}{=}_c s_6 \odot \texttt{CShift}_6 :: c_6$, $d_5 \stackrel{e}{=}_d d_6$, and $\texttt{VFun}_5(x, t, e_5) \stackrel{e}{=}_v \texttt{VFun}_6(x, t, e_6)$
We have $[] \otimes [] \stackrel{e}{=}_c [] \otimes []$ due to Definition 2.
Because $s_5 \otimes \texttt{CShift}_5 :: c_5 \stackrel{e}{=}_c s_6 \odot \texttt{CShift}_6 :: c_6$, we have
$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$
We also have $e_5 \stackrel{e}{=}_e e_6$ because $\texttt{VFun}_5(x, t, e_5) \stackrel{e}{=}_v \texttt{VFun}_6(x, t, e_6)$.
Then,
$(x, \texttt{VCont}_5(c_5, s_5)) :: e_5 \stackrel{e}{=}_e (x, \texttt{VCont}_6(c_6, s_6)) :: e_6$
So, P' and Q' satisfies the first clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \texttt{CShift}_5 :: c_5, \texttt{VCont}_5(c_5', s_5') :: s_5, d_5 \rangle \xrightarrow{\text{eval5}} P' : \langle c_5', \texttt{VCont}_5(c_5, s_5) :: s_5', d_5 \rangle$:
In this case,
$Q : \langle \texttt{CShift}_6 :: c_6, \texttt{VCont}_6(c_6', s_6') :: s_6, d_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle c_6', \texttt{VCont}_6(c_6, s_6) :: s_6', d_6 \rangle$,
and we have
$s_5 \otimes \texttt{CShift}_5 :: c_5 \stackrel{e}{=}_c s_6 \odot \texttt{CShift}_6 :: c_6$, $d_5 \stackrel{e}{=}_d d_6$, and $\texttt{VCont}_5(c_5', s_5') \stackrel{e}{=}_v \texttt{VCont}_6(c_6', s_6')$
Because $s_5 \otimes \texttt{CShift}_5 :: c_5 \stackrel{e}{=}_c s_6 \odot \texttt{CShift}_6 :: c_6$, we have
$s_5 \otimes c_5 \stackrel{e}{=}_c s_6 \odot c_6$
Then,
$\texttt{VCont}_5(c_5, s_5) \stackrel{e}{=}_v \texttt{VCont}_6(c_6, s_6)$
We also have $s_5' \otimes c_5' \stackrel{e}{=}_c s_6' \odot c_6'$ because $\texttt{VCont}_5(c_5', s_5') \stackrel{e}{=}_v \texttt{VCont}_6(c_6', s_6')$.
So, P' and Q' satisfies the second clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle [], v_5 :: s_5, d_5 \rangle \xrightarrow{\text{eval5}} P' : \langle d_5, v_5 :: s_5 \rangle$:
In this case,
$Q : \langle [], v_6 :: s_6, d_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle d_6, v_6 :: s_6 \rangle$,
and we have
$d_5 \stackrel{e}{=}_d d_6$, and $v_5 \stackrel{e}{=}_v v_6$
P' and Q' satisfies the third clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle \mathtt{DRun}_5(c'_5, s'_5) :: d_5, v_5 :: s_6 \rangle \xrightarrow{\text{eval5}} P' : \langle c'_5, v_5 :: s'_5, d_5 \rangle$:

In this case,

$Q : \langle \mathtt{DRun}_6(c'_6, s'_6) :: d_6, v_6 :: s_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle c'_6, v_6 :: s'_6, d_6 \rangle$,

and we have

$\mathtt{DRun}_5(c'_5, s'_5) :: d_5 \overset{e}{=}_d \mathtt{DRun}_6(c'_6, s'_6) :: d_6$, and $v_5 \overset{e}{=}_v v_6$

Because $\mathtt{DRun}_5(c'_5, s'_5) :: d_5 \overset{e}{=}_d \mathtt{DRun}_6(c'_6, s'_6) :: d_6$, we have

$s'_5 \otimes c'_5 \overset{e}{=}_c s'_6 \otimes c'_6$ and $d_5 \overset{e}{=}_d d_6$

So, P' and Q' satisfies the second clause of Definition 5, that is, $P' \sim_e Q'$.

Case $P : \langle [], v_5 :: s_5 \rangle \xrightarrow{\text{eval5}} P' : \langle v_5 \rangle$:

In this case,

$Q : \langle [], v_6 :: s_6 \rangle \xrightarrow{\text{eval6}} Q' : \langle v_6 \rangle$

and we have

$v_5 \overset{e}{=}_v v_6$

P' and Q' satisfies the fourth of Definition 5, that is, $P' \sim_e Q'$.

Therefore, the former, $P \xrightarrow{\text{eval5}} P'$ implies $Q \xrightarrow{\text{eval6}} Q'$ and $P' \sim_e Q'$ for some Q', is proved.

In terms of the latter, we can prove it in a similar way. Thus, $\sim_e$ is a bisimulation. □