Derivation of a Virtual Machine For Four Variants of Delimited-Control Operators

Maika Fujii
Ochanomizu University, Tokyo, Japan

Kenichi Asai
Ochanomizu University, Tokyo, Japan

Abstract

This paper derives an abstract machine and a virtual machine for the λ-calculus with four variants of delimited-control operators: `shift/reset`, `control/prompt`, $\texttt{shift}_0/\texttt{reset}_0$, and $\texttt{control}_0/\texttt{prompt}_0$. Starting from Shan's definitional interpreter for the four operators, we successively apply various meaning-preserving transformations. Both trails of invocation contexts (needed for `control` and $\texttt{control}_0$) and metacontinuations (needed for $\texttt{shift}_0$ and $\texttt{control}_0$) are defunctionalized and eventually represented as a list of stack frames. The resulting virtual machine clearly models not only how the control operators and captured continuations behave but also when and which portion of stack frames is copied to the heap.

2012 ACM Subject Classification Theory of computation → Control primitives; Theory of computation → Lambda calculus; Theory of computation → Operational semantics; Theory of computation → Abstract machines; Software and its engineering → Virtual machines

Keywords and phrases delimited-control operators, functional derivation, CPS transformation, defunctionalization, abstract machine, virtual machine

Digital Object Identifier 10.4230/LIPIcs.FSCD.2021.16

Supplementary Material *Software (Source Code)*: `https://github.com/FujiiMaika/fscd21`
archived at `swh:1:dir:e523c86111370f0dce57a8b6c5506fcf7c35c1f1`

Funding *Kenichi Asai*: supported in part by JSPS KAKENHI under Grant No. JP18H03218.

Acknowledgements We are grateful to Youyou Cong and anonymous reviewers for their valuable comments and suggestions.

1 Introduction

Manipulation of control structure of a program is inevitable. In addition to the standard exception handling, more sophisticated manipulation of control using algebraic effects and handlers has been proposed [4, 25] and is becoming widely used [20]. To support such mechanisms in a compiler, one can either (i) transform the source program into continuation-passing style (CPS), or (ii) implement manipulation of control directly via the modification of a portion of a stack without transforming the program into CPS. There is extensive research comparing which approach (among more variants) is better in which circumstances [12].

However, for four variants of delimited-control operators, i.e., `shift` and `reset` [8, 9], `control` and `prompt` [13], $\texttt{shift}_0$ and $\texttt{reset}_0$ [23], and $\texttt{control}_0$ and $\texttt{prompt}_0$ [16], almost no low-level implementation has been considered. The only exceptions we are aware of are all on `shift/reset`: direct implementation of `shift/reset` in Scheme48 [15], in OchaCaml [22], and the derivation of a virtual machine for `shift/reset` [3]. Without proper low-level implementation strategies for all the four delimited-control operators, we cannot even discuss pros and cons of CPS vs. direct-style implementations for those operators. This omission could affect the low-level implementation strategies for algebraic effects and handlers, since they have a close connection with $\texttt{shift}_0$ and $\texttt{control}_0$ [14, 24].

In this paper, we derive an abstract machine and a virtual machine for the λ-calculus with four delimited-control operators. Starting from Shan's definitional interpreter [28], we successively apply various meaning-preserving transformations, following Danvy's recipe [2, 7]. The overall derivation is similar to our previous work [3] on deriving a virtual machine for `shift/reset`. However, handling of invocation contexts (needed for `control` and $\texttt{control}_0$) and metacontinuations (needed for $\texttt{shift}_0$ and $\texttt{control}_0$) is non-trivial: we need to have a trail of invocation contexts to be a tree structure to support concatenation of invocation contexts and have a metacontinuation to maintain a list of code pointers representing the contexts outside delimiters.

In summary, we make the following contributions in this paper:

- We present the first virtual machine that supports four delimited-control operators and that explains how they manipulate stacks.
- We show it is possible to apply Danvy's method of inter-deriving semantic artifacts to four delimited-control operators, giving another non-trivial example and widening its applicability.
- We clarify how trails and metacontinuations can be represented in a stack, suggesting a low-level implementation strategy for four delimited-control operators.

After introducing four delimited-control operators in the next section, we first show the definitional interpreter in Section 3. We then apply various program transformation to obtain a stack-based interpreter in Section 4, showing an abstract machine in passing. In Sections 5 and 6, we derive a compiler and a virtual machine. Related work is discussed in Section 7 and the paper concludes in Section 8. The appendix shows an example how a program is compiled to a list of instructions and executed on the virtual machine. The omitted OCaml code is available as supplementary material.

2 Four Delimited-Control Operators

Delimited-control operators enable us to capture the current continuation up to the enclosing delimiter and use it in the subsequent program. There are four variants of delimited-control operators: `shift` ($\mathcal{S}$) and `reset` [8, 9], `control` ($\mathcal{F}$) and `prompt` [13], $\texttt{shift}_0$ ($\mathcal{S}_0$) and $\texttt{reset}_0$ [23], and $\texttt{control}_0$ ($\mathcal{F}_0$) and $\texttt{prompt}_0$ [16]. Since the behavior of all the four delimiters (`reset`, `prompt`, $\texttt{reset}_0$, and $\texttt{control}_0$) are exactly the same, we use a uniform notation $\langle\rangle$ for them. The basic behavior of the four operators are to capture the current continuation up to the enclosing delimiter and execute their body. We describe their exact behavior below.

A `shift` expression, $\mathcal{S}c.\, e$, clears the current continuation up to the enclosing delimiter, binds it to c, and execute e. Thus, in $1 + \langle(\mathcal{S}c.\, 2 \times c\, 3) + 4\rangle$, the continuation $\langle[] + 4\rangle$ is cleared, bound to c, and $2 \times c\, 3$ is executed in `reset`. The original expression reduces to $1 + \langle 2 \times c\, 3\rangle$, giving the final result 15.

A `control` expression, $\mathcal{F}c.\, e$, differs from `shift` in that it does not insert a delimiter into the captured continuation. In $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + 4\rangle$, c is bound to $[] + 4$ without surrounding `reset`. If the captured continuation contains another `control`, as in $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + \mathcal{F}c'.\, 4\rangle$, c is bound to $[] + \mathcal{F}c'.\, 4$. The original expression reduces to $1 + \langle 2 \times (3 + \mathcal{F}c'.\, 4)\rangle$, where the second $\mathcal{F}$ captures (and discards) not just $3 + []$ but also the *invocation context* of c, namely $2 \times []$, giving the final result 5. Using $\mathcal{F}$, one can access the context in which the captured continuation is invoked. This is in contrast to the `shift` case: $1 + \langle(\mathcal{S}c.\, 2 \times c\, 3) + \mathcal{S}c'.\, 4\rangle$ reduces to $1 + \langle 2 \times \langle 3 + \mathcal{S}c'.\, 4\rangle\rangle$, giving the final result 9. Using more than one $\mathcal{F}$ in the same context, we can capture multiple invocation contexts.[1] To account for the invocation contexts of captured continuations, an interpreter for $\mathcal{F}$ must maintain a *trail* of continuations [5].

A `shift`$_0$ expression, $\mathcal{S}_0 c.\, e$, on the other hand, removes the original `reset` surrounding the `shift`$_0$ expression (but retains the `reset` around the captured continuation as in $\mathcal{S}$). By nesting $\mathcal{S}_0$, one can access the context outside the enclosing `reset`. For example, $\langle 1 + \langle (\mathcal{S}_0 c.\, \mathcal{S}_0 c'.\, 2 \times c'3) + 4 \rangle\rangle$ reduces to $\langle 1 + (\mathcal{S}_0 c'.\, 2 \times c'3)\rangle$ where c is bound to $\langle [] + 4\rangle$ but is discarded. Note that there is no `reset` around $\mathcal{S}_0 c'.\, 2 \times c'3$. Thus, c' is bound to the context $\langle 1 + []\rangle$, which was outside the original `reset`, giving the final result 8. This is in contrast to the `shift` case: $\langle 1 + \langle (\mathcal{S} c.\, \mathcal{S} c'.\, 2 \times c'3) + 4 \rangle\rangle$ reduces to $\langle 1 + \langle (\mathcal{S} c'.\, 2 \times c'3)\rangle\rangle$. Now, c' is bound to an empty context $[]$, giving the final result 7. With more nested occurrences of $\mathcal{S}_0$, arbitrarily outer contexts can be captured. To account for hierarchical contexts, the interpreter for $\mathcal{S}_0$ must maintain a *metacontinuation* [23].

A `control`$_0$ expression, $\mathcal{F}_0 c.\, e$, has both the characteristics of $\mathcal{F}$ and $\mathcal{S}_0$: the captured continuation does not come with a surrounding `reset` and the original `reset` is removed. As such, the interpreter for $\mathcal{F}_0$ must maintain both a trail of continuations and a metacontinuation.

Shan [28] provides a detailed explanation on the difference between the four control operators, as well as an example where the choice of the four operators results in four different result values. Dyvbig, Peyton Jones, and Sabry [11] explain the four delimited-control operators in terms of different primitive control operators.

3 The Definitional Interpreter

Listing 1 shows the definitional interpreter for the λ-calculus extended with four delimited-control operators and the delimiter, written in OCaml. The interpreter is written in continuation-, trail-, and metacontinuation-passing style. Although the main interpreter function `f1` receives a trail and a metacontinuation explicitly, they do not play any roles for the pure λ-calculus terms. If we η-reduce them, the definition coincides with the standard continuation-passing style interpreter.

As in our previous work [3], an environment is represented as two lists, a list of variable names `xs` and a list of values `vs`, instead of an association list. This design comes from the goal of this work. Since we will decompose the interpreter into a compiler and a virtual machine, we separate an environment into the part that depends only on the input term and the part that depends on runtime values. The function `Env.offset` returns the offset of a variable within a given list.

In the interpreter, the current continuation and trail in the innermost surrounding delimiter are stored in the arguments `c` and `t` (of types `c` and `t`, respectively), while the continuations and trails outside the delimiter are stored in metacontinuation `m`, which is a list[2] of pairs of a continuation and a trail of each context. Thus, the context is delimited (in the `Reset (e)` case) by storing `c` and `t` to `m` and evaluating the body `e` in the initial continuation `idc` and the empty trail `TNil`.

To capture the current continuation and trail, one of four control operators is used. In all four cases, the current continuation `c` and trail `t` are captured, bound to `x`, and the body of the control operator is evaluated under appropriate settings.

[1] See [6] for the general case as well as other (typed) examples of the use of $\mathcal{F}$.
[2] We use `MNil` and `MCons` to construct metacontinuations. We cannot use `(c * t) list` as the definition of `m`, because the types `c` and `m` would then be circular.

Listing 1 The definitional interpreter.

```
(* syntax *)
type e = Var of string | Fun of string * e | App of e * e
       | Shift of string * e | Control of string * e
       | Shift0 of string * e | Control0 of string * e
       | Reset of e

type v = VFun of (v -> c -> t -> m -> v)      (* value *)
       | VContS of c * t | VContC of c * t
and  c = v -> t -> m -> v                     (* continuation *)
and  t = TNil | Trail of (v -> t -> m -> v)   (* trail *)
and  m = MNil | MCons of (c * t) * m          (* metacontinuation *)

(* initial continuation : v -> t -> m -> v *)
let idc v t m = match t with
    TNil -> (match m with
        MNil -> v
      | MCons((c,t),m) -> c v t m)
  | Trail(h) -> h v TNil m

(* cons : (v -> t -> m -> v) -> t -> t *)
let rec cons h t = match t with
    TNil -> Trail(h)
  | Trail(h') -> Trail(fun v t' m -> h v (cons h' t') m)

(* apnd : t -> t -> t *)
let apnd t0 t1 = match t0 with
    TNil -> t1
  | Trail(h) -> cons h t1

(* f1 : e -> string list -> v list -> c -> t -> m -> v *)
let rec f1 e xs vs c t m = match e with
    Var(x) -> c (List.nth vs (Env.offset x xs)) t m
  | Fun(x,e) ->
      c (VFun(fun v c' t' m' -> f1 e (x::xs) (v::vs) c' t' m')) t m
  | App(e0,e1) ->
      f1 e0 xs vs (fun v0 t0 m0 ->
      f1 e1 xs vs (fun v1 t1 m1 ->
      (match v0 with
         VFun(f) -> f v1 c t1 m1
        | VContS(c',t') -> c' v1 t' (MCons((c,t1),m1))
        | VContC(c',t') -> c' v1 (apnd t' (cons c t1)) m1)) t0 m0) t m
  | Shift(x,e) -> f1 e (x::xs) (VContS(c,t)::vs) idc TNil m
  | Control(x,e) -> f1 e (x::xs) (VContC(c,t)::vs) idc TNil m
  | Shift0(x,e) -> (match m with
      MCons((c0,t0),m0) -> f1 e (x::xs) (VContS(c,t)::vs) c0 t0 m0)
  | Control0(x, e) -> (match m with
      MCons((c0,t0),m0) -> f1 e (x::xs) (VContC(c,t)::vs) c0 t0 m0)
  | Reset(e) -> f1 e xs vs idc TNil (MCons((c,t),m))

(* f : e -> v *)
let f expr = f1 expr [] [] idc TNil MNil
```

- For `Shift (x, e)` and `Control (x, e)`, the body `e` is evaluated under the initial continuation and the empty trail. This reflects the fact that the original `reset` surrounding the control operator remains for these cases. Even if we use control operators within `e`, we cannot access the contexts outside `reset` because they reside in `m`.
- For `Shift0 (x, e)` and `Control0 (x, e)`, on the other hand, the body `e` is evaluated under the topmost continuation and trail stored in the metacontinuation `m`.[3] This reflects the fact that the original `reset` surrounding the control operator is removed for these cases. By using control operators within `e`, we can access the context outside the innermost `reset`.

The captured continuation and trail are packaged into `VContS` for `Shift (x, e)` and `Shift0 (x, e)` and into `VContC` for `Control (x, e)` and `Control0 (x, e)`. When `VContS` or `VContC` is applied (in the `App` case), it behaves differently depending on whether `reset` is present around the invocation.

- For `VContS (c', t')`, the continuation `c` and trail `t1` at the invocation time are pushed into metacontinuation `m1`. This reflects the fact that the invocation of a continuation captured by `Shift (x, e)` or `Shift0 (x, e)` is surrounded by `reset`. Even if we use control operators within `c'`, we cannot access `c` and `t1` because they reside in the metacontinuation.
- For `VContC (c', t')`, on the other hand, the continuation `c` and trail `t1` at the invocation time are concatenated to the current trail `t'`. This reflects the fact that the invocation of a continuation captured by `Control (x, e)` or `Control0 (x, e)` is *not* surrounded by `reset`; since the invocation-time continuation and trail are put into the trail, they can be captured by using control operators within `c'`.

Adding a continuation to a trail and appending two trails are realized by `cons` and `apnd`, respectively. A trail is either an empty trail `TNil` or a non-empty trail `Trail` holding a continuation, which represents functional composition of all the invocation contexts (continuations) encountered so far.

The interpreter is identical to Shan's interpreter [28] except for two points. First, Shan uses higher-order functions directly to represent captured continuations, while we use a defunctionalized form. We could have started from the higher-order functions; by applying defunctionalization to it, we obtain Listing 1. Second, Shan concatenates the captured continuation `c'` and trail `t'` with the continuation `c` and trail `t1` at the invocation time as `((cons c' t') v1 (cons c t1))`. By case analysis on `t'`, it is straightforward to verify that Shan's code is equivalent to `(c' v1 (apnd t' (cons c t1)))` which we adopt. The latter is also used by Biernacki, Danvy, and Millikin [5] and Kameyama and Yonezawa [19].

4 Stack Introduction

In this and next sections, we successively apply meaning-preserving program transformations to the definitional interpreter to obtain a compiler and a virtual machine. In this section, we introduce a stack into the interpreter by (1) defunctionalizing continuations (Section 4.1), (2) linearizing them into a list of frames (Section 4.2), and (3) separating static and dynamic data in the frames (Section 4.3). Along the way, we derive a stack-based abstract machine (Section 4.5).

[3] Metacontinuation `m` must be non-empty here. Otherwise, a pattern-match error is raised. (In the supplementary material, a more sensible error message "shift0/control0 is used without enclosing reset" is given.)

Listing 2 Type definition for defunctionalized interpreter.

```
type v = VFun of (v -> c -> t -> m -> v)
       | VContS of c * t | VContC of c * t
and  c = C0 | CApp0 of e * string list * v list * c | CApp1 of v * c
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * t) * m
```

Listing 3 Type definition for linearized interpreter.

```
type v = VFun of (v -> c -> t -> m -> v)
       | VContS of c * t | VContC of c * t
and  f = CApp0 of e * string list * v list | CApp1 of v
and  c = f list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * t) * m
```

4.1 Defunctionalization

We first defunctionalize [26, 27] continuations in the definitional interpreter. In Listing 1, the type `c` is higher order. We turn it into a datatype as shown in Listing 2. The identity continuation is represented as `C0`, while two continuations in the `App` case are represented as `App0` and `App1` where the arguments represent free variables of the respective continuations. The resulting datatype essentially represents evaluation contexts.

We do *not* defunctionalize the argument of `VFun` at this point, because it is not necessary for stack introduction. This choice is arbitrary: we could defunctionalize it and the rest of derivations would go through without any problem. We will defunctionalize it later when we need to do so, to derive an abstract machine and a virtual machine.

We do *not* defunctionalize the argument of `Trail`, either. Even though the type of the argument of `Trail` is the same as `c`, defunctionalizing it together with `c` leads to tree-structured continuations. We can still obtain the same abstract machine and virtual machine, but by defunctionalizing it separately at a later stage, we can keep the definition of `c` to have a list structure (as in our previous work [3]) and postpone the introduction of a tree structure until Section 5.3.

We omit the standard definition of the defunctionalized interpreter due to the lack of space; see the supplementary material. We simply introduce a dispatch function for `c` and use it whenever a continuation is applied. The transformation is the standard defunctionalization and thus the resulting interpreter behaves the same as the definitional interpreter.

4.2 Linearizing Continuations

The type `c` in Listing 2 is isomorphic to a list where `C0` is an empty list and `CApp0` and `CApp1` are conses. Thus, we linearize continuations, i.e., we transform `c` into an OCaml list as shown in Listing 3. The type `c` is now a list of frames, where a frame `f` stores data that were previously held in `CApp0` and `CApp1`.

Obviously, the new interpreter (omitted) behaves the same as the previous one.

4.3 Introducing Stacks

Examining the type `f` in Listing 3, we notice that the constructors `CApp0` and `CApp1` contain both static (compile-time) and dynamic (run-time) data. Static data include the term `e` and the variable list `string list` in `CApp0`, which are fixed once the input program

Listing 4 Type definition for stack-based interpreter.

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  f = CApp0 of e * string list | CApp1
and  c = f list
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

Listing 5 Type definition for delinearized interpreter.

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = C0 | CApp0 of e * string list * c | CApp1 of c
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

is given. Dynamic data include `v list` in `CApp0` and `v` in `CApp1`, which are available only at run-time. Since our goal is to transform the interpreter into a compiler and a virtual machine, we separate these two types of data by introducing a stack.

Listing 4 shows the resulting data definition. The previous continuation `c` is split into a pair of a continuation `c` and a stack `s`. The former is a list of frames, where the frame `f` now keeps only the static data. The runtime data are kept in the stack, which is a list of values. Since the previous `CApp0` included `v list`, the value `v` is extended with `VEnv` to store the `v list` as a single value.[4] Since the new `c` (a list of frames) and `s` (a list of values) are obtained by splitting a single list (a list of `f` in Listing 3), they always have the same length. In the subsequent derivations, we keep this invariant throughout.

Because we only changed the representation of `c` locally, we immediately see that the new interpreter behaves the same as the previous one.

4.4 Delinearizing Continuations

The purpose of defunctionalization (Section 4.1) and linearization of continuations (Section 4.2) was to introduce a data stack. Now that we have introduced a data stack, we transform continuations back to the higher-order form via delinearization. In this section, we convert lists into constructors.

Listing 5 shows the resulting data definition. Here, only the static `f` is incorporated into `c`. The stack `s` remains as a list of values. Note that `c` contains only static data (in contrast to `c` in Listing 2 that contains both static and dynamic data). All the dynamic data are still carried around in `s`. As in Section 4.2, the old and new representations of `c` are isomorphic, and thus the new interpreter behaves the same as the previous one.

[4] The introduction of `VEnv` into `v` is arbitrary. Although we introduced it to emulate caller-save registers often found in the compiled code, a user cannot write a program that evaluates to `VEnv`. Instead, we could introduce a new type for stack items that consists of either a value or a list of values (`VEnv`). In the current paper, we followed our previous work [3] and included `VEnv` directly to `v`.

Figure 1 Abstract machine.

e	$\Rightarrow$	$\langle e,\ [],\ [],\ C_0,\ [],\ \mathit{TNil}, []\rangle$
$\langle x,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ \mathit{List.nth}\ vs\ (\mathit{offset}\ x\ xs),\ s,\ t,\ m\rangle$
$\langle \lambda x.e,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ \mathit{VFun}(e,\ x,\ xs,\ vs),\ s,\ t,\ m\rangle$
$\langle e_0\ e_1,\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e_0,\ xs,\ vs,\ \mathit{CApp}_0(e_1,\ xs,\ c),\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle$
$\langle \mathit{Shift}(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContS}(c,\ s,\ t) :: vs,\ C_0,\ [],\ \mathit{TNil},\ m\rangle$
$\langle \mathit{Control}(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContC}(c,\ s,\ t) :: vs,\ C_0,\ [],\ \mathit{TNil},\ m\rangle$
$\langle \mathit{Shift}_0(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ (c_0,\ s_0,\ t_0) :: m_0\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContS}(c,\ s,\ t) :: vs,\ c_0,\ s_0,\ t_0,\ m_0\rangle$
$\langle \mathit{Control}_0(x,\ e),\ xs,\ vs,\ c,\ s,\ t,\ (c_0,\ s_0,\ t_0) :: m_0\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ \mathit{VContC}(c,\ s,\ t) :: vs,\ c_0,\ s_0,\ t_0,\ m_0\rangle$
$\langle \mathit{Reset}(e),\ xs,\ vs,\ c,\ s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ xs,\ vs,\ C_0,\ [],\ \mathit{TNil},\ (c,\ s,\ t) :: m\rangle$
$\langle C_0,\ v,\ [],\ \mathit{TNil},\ []\rangle$	$\Rightarrow$	v
$\langle C_0,\ v,\ [],\ \mathit{TNil},\ (c,\ s,\ t) :: m\rangle$	$\Rightarrow$	$\langle c,\ v,\ s,\ t,\ m\rangle$
$\langle C_0,\ v,\ [],\ \mathit{Trail}(h),\ m\rangle$	$\Rightarrow$	$\langle h,\ v,\ \mathit{TNil},\ m\rangle$
$\langle \mathit{CApp}_0(e,\ xs,\ c),\ v,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ xs,\ vs,\ \mathit{CApp}_1(c),\ v :: s,\ t,\ m\rangle$
$\langle \mathit{CApp1}(c),\ v,\ \mathit{VFun}(e,\ x,\ xs,\ vs) :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle e,\ x :: xs,\ v :: vs,\ c,\ s,\ t,\ m\rangle$
$\langle \mathit{CApp}_1(c),\ v,\ \mathit{VContS}(c',\ s',\ t') :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ v,\ s',\ t',\ (c,\ s,\ t) :: m\rangle$
$\langle \mathit{CApp}_1(c),\ v,\ \mathit{VContC}(c',\ s',\ t') :: s,\ t,\ m\rangle$	$\Rightarrow$	$\langle c',\ v,\ s',\ \mathit{apnd}\ t'\ (\mathit{cons}\ (\mathit{Hold}\ (c,\ s))\ t),\ m\rangle$
$\langle \mathit{Hold}\ (c,\ s),\ v,\ t,\ m\rangle$	$\Rightarrow$	$\langle c,\ v,\ s,\ t,\ m\rangle$
$\langle \mathit{Append}\ (h,\ h'),\ v,\ t,\ m\rangle$	$\Rightarrow$	$\langle h,\ v,\ \mathit{cons}\ h'\ t,\ m\rangle$

4.5 Abstract Machine

In this section, we briefly describe the abstract machine that can be derived from the interpreter in Section 4.4. Since all the interpreters in this paper receive a continuation and a metacontinuation, all the calls to interpreter functions (such as `f1` and the dispatch function for continuations) are tail calls. As such, we can easily derive an abstract machine by simply regarding the arguments to interpreter functions as a state of the abstract machine. The derived abstract machine is shown in Figure 1. Although we omit the code for the interpreter, one can imagine how it looks like from the abstract machine. To extract the abstract machine, we further performed the following transformations:

- We defunctionalized the argument of `VFun`. A function is now represented as a closure. We will perform the same transformation later; see Section 5.3.
- We defunctionalized the argument of `Trail`. The `Trail` data are constructed in the two branches of `cons` (see Listing 1). The first one is represented as `Hold` that holds an invocation context; the second one as `Append` that appends two trails. We will perform the same transformation later; see Section 5.3 for details.
- Instead of `MNil` and `MCons`, we use standard lists for metacontinuations.

Because we have introduced a stack into the interpreter, we obtain a stack-based abstract machine. This is in contrast to the previous abstract machines [5, 11, 28] which do not carry a stack explicitly. The obtained abstract machine clearly describes the behavior of control operators. When one of the control operators is used, the current continuation c, stack s, and trail t are captured, put into a stack, and bound to x. Then, the body of the control operator is executed. For *Shift* and *Control*, the current continuation and trail are cleared, whereas for Shift_0 and $\mathit{Control}_0$, the ones in the metacontinuation are used. The `reset` operator pushes the current c, s, and t on the metacontinuation m, and initializes them.

When a continuation captured by *Shift* or Shift_0 is invoked, the current c, s, and t are pushed onto m and the captured state is reinstated. When a continuation captured by *Control* or $\mathit{Control}_0$ is invoked, on the other hand, t is extended by c and s (via *cons*), and the result is in turn extended by t' (via *apnd*).

4.6 Refunctionalizing Continuations

Finally, Listing 6 shows the refunctionalized interpreter where defunctionalized continuations are transformed back to higher-order functions. It is similar to the definitional interpreter in Listing 1, but passes around a stack. Typewise, all the occurrences of a continuation `c` in Listing 1 are replaced by pairs `c * s` of a continuation and a stack. Furthermore, the type `c` and the type of the argument of `VFun` are modified to receive a stack.

Compared to the definitional interpreter `f1` in Listing 1, the refunctionalized interpreter `f6` receives an additional stack argument `s`, and whenever it returns a value, a continuation `c` is applied to the value together with a stack `s`. We can also observe that the references to free variables in the definitional interpreter (`vs` and `v0` in the `App` case) are now realized by passing those values via the stack. We push those values at the recursive calls and pop them when the corresponding continuations are called. Since stacks are extracted from continuations and stacks have the same structure as (now refunctionalized) continuations, popping a value would never fail: popped values correspond to the dynamic arguments of `CApp0` and `CApp1`. This is the consequence of the invariant we keep between continuations and stacks. Similarly, `idc` corresponds to `C0`, which has no dynamic counterpart. Thus, the stack argument of `idc` (the second argument of `idc` in Listing 6) must be an empty stack.

The argument of `Trail` needs explanation. Since we have not defunctionalized the argument of `Trail` yet, we need type conversion to store a continuation `c` in a trail. See the first argument to `cons` in the `App` case. The continuation `c` is turned into `fun v t m -> c v s1 t m` with `s1` being a free variable. Later when we defunctionalize it, the stack `s1` will be extracted; see Section 5.3.

It is not straightforward to obtain the refunctionalized interpreter from the previous one. One has to verify that the previous interpreter is in defunctionalized form [10]. However, once it is obtained, it is simple to verify its correctness: by defunctionalizing the refunctionalized interpreter, we can obtain the previous one.

5 Deriving a Virtual Machine

In this section, we derive a virtual machine from the refunctionalized interpreter obtained in Section 4.6. We first combine arguments so that values are passed via a stack (Section 5.1). We then stage the interpreter into a compiler that operates on instructions represented as functions (Section 5.2). By defunctionalizing the instructions (Section 5.3) and linearizing instructions (Section 5.4) and stacks (Section 5.5), we obtain the virtual machine (Section 6).

5.1 Combining Arguments

In Listing 6, functions in `VFun` as well as continuations `c` receive both a value `v` and a stack `s`. In a low-level implementation, such as a virtual machine, we want to pass all the values via a stack rather than passing a value and a stack separately. Listing 7 shows the type definition of the result of such a transformation.

The argument `v` is removed from the argument of `VFun` and `c`. When we call such a function, we push `v` to the stack before the call. When the function is called, we pop `v` from the stack before the function body is executed. We do the same for the interpreter function: we remove the `vs` argument and push it on the stack. As a result, the type of the interpreter function `f7` after the transformation becomes as follows:

```
(* f7 : e -> string list -> c -> s -> t -> m -> v *)
```

Listing 6 Refunctionalized interpreter (cons and apnd are the same as in Listing 1).

```
type v = VFun of (v -> c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = v -> s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m

(* initial continuation : v -> s -> t -> m -> v *)
let idc v [] t m = match t with
    TNil -> (match m with
        MNil -> v
      | MCons((c,s,t),m) -> c v s t m)
  | Trail(h) -> h v TNil m

(* f6 : e -> string list -> v list -> c -> s -> t -> m -> v *)
let rec f6 e xs vs c s t m = match e with
  | Var(x) -> c (List.nth vs (Env.offset x xs)) s t m
  | Fun(x,e) ->
      c (VFun(fun v c' s' t' m' -> f6 e (x::xs) (v::vs) c' s' t' m'))
        s t m
  | App(e0,e1) ->
      f6 e0 xs vs (fun v0 (VEnv(vs)::s0) t0 m0 ->
      f6 e1 xs vs (fun v1 (v0::s1) t1 m1 ->
      (match v0 with
         VFun(f) -> f v1 c s1 t1 m1
       | VContS(c',s',t') -> c' v1 s' t' (MCons((c,s1,t1),m1))
       | VContC(c',s',t') ->
         c' v1 s' (apnd t' (cons (fun v t m -> c v s1 t m) t1)) m1))
      (v0::s0) t0 m0) (VEnv(vs)::s) t m
  | Shift(x,e) -> f6 e (x::xs) (VContS(c,s,t)::vs) idc [] TNil m
  | Control(x,e) -> f6 e (x::xs) (VContC(c,s,t)::vs) idc [] TNil m
  | Shift0(x,e) -> (match m with
      MCons((c0,s0,t0),m0) ->
        f6 e (x::xs) (VContS(c,s,t)::vs) c0 s0 t0 m0)
  | Control0(x,e) -> (match m with
      MCons((c0,s0,t0),m0) ->
        f6 e (x::xs) (VContC(c,s,t)::vs) c0 s0 t0 m0)
  | Reset(e) -> f6 e xs vs idc [] TNil (MCons((c,s,t),m))

(* f : e -> v *)
let f expr = f6 expr [] [] idc [] TNil MNil
```

Listing 7 Type definition for interpreter with combined arguments.

```
type v = VFun of (c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list
and  c = s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
```

Since we simply changed the way two arguments are passed locally, we immediately see that the new interpreter behaves the same as the previous one.

5.2 Introducing Combinators as Instructions

In this section, we extract a compiler from the interpreter. Looking at the type of `f7` in the previous section, we notice that the first two arguments are static and the rest of the arguments are dynamic. We first define the type `i` of instructions (in Listing 8) as the dynamic part of the interpreter, which represents the work to be done when dynamic data are received. We then regard the interpreter as a compiler that accepts two static data and returns an instruction. Listing 8 shows the result.

The interpreter function `f8`, or a compiler, processes only the static data: the input term `e` and a list of variable names `xs`. It then produces an instruction of type `i`, which performs the rest of the work when dynamic data are given.

For example, in the case of `Var (x)`, the compiler emits an instruction `access`, which, given dynamic data, returns the corresponding value in the environment. In the case of `App (e0, e1)`, we define `push_env`, `pop_env`, and `call`, and concatenate these instructions using `(>>)`. We employ the same technique as the previous work [3]: we store the return address `VK` (added to the definition of `v`) to the stack in `return` and retrieve it in `call`.

This interpreter behaves the same as the previous one, because if we inline all the instructions, we obtain the interpreter in the previous section.

5.3 Defunctionalizing Instructions

In this section, we defunctionalize the functional instructions introduced in the previous section into the ones that are closer to machine instructions. Specifically, we defunctionalize the argument of `VFun`, `i`, `c`, and the argument of `Trail`, separately, and change the representation of `m`. See Listing 9.

First, the argument of `VFun` (see `push_closure` and `call` in Listing 8) is defunctionalized to a closure. Second, the instruction `i` is defunctionalized. All the functional instructions are turned into constructors as shown in `i` in Listing 9. The corresponding dispatch function (omitted) is a virtual machine: given an instruction and the current dynamic state, it performs necessary operations. Observe how a virtual machine is naturally derived by defunctionalizing functional instructions. Note also that the instruction is *not* linear: it includes `ISeq` corresponding to `(>>)` and thus has a tree structure.

Third, `c` is defunctionalized. There are two cases that constitute the value of `c` in Listing 8: the identity continuation `idc`, which is closed, and the second argument to `i0` in `(>>)`, `fun s' t' m' -> i1 c s' t' m'`. Since the free variables of the latter are `i1` and `c`, we can represent `c` as a list of `i`, regarding the former as an empty list and the latter as cons list.

Fourth, the argument of `Trail` is defunctionalized and given a new type `h`. The `Trail` data are constructed in the two branches of `cons` (see Listing 1): its argument is either a continuation `h` or `fun v t' m -> h v (cons h' t') m` which has `h` and `h'` as free variables. They are represented as `Hold` and `Append` in Listing 9, respectively. Note that `h` has a tree structure. Finally, the metacontinuation `m` is turned into an OCaml list, as no circular dependency arises any more.[5]

Since all these changes are instances of defunctionalization and a simple local change of data representation, the behavior of the new interpreter is the same as the previous one.

[5] Unlike the definitional interpreter. See footnote 2.

Listing 8 Interpreter using combinators factored as instructions.

```
type v = VFun of (c -> s -> t -> m -> v)
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list | VK of c
and  c = s -> t -> m -> v
and  s = v list
and  t = TNil | Trail of (v -> t -> m -> v)
and  m = MNil | MCons of (c * s * t) * m
type i = c -> s -> t -> m -> v

(* (>>) : i -> i -> i *)
let (>>) i0 i1 =
  fun c s t m -> i0 (fun s' t' m' -> i1 c s' t' m') s t m

(* instructions *)
let access n = fun c (VEnv(vs)::s) t m -> c ((List.nth vs n)::s) t m
let push_closure i = fun c (VEnv(vs)::s) t m ->
  c (VFun(fun c' (v::s') t' m' -> i c' (VEnv(v::vs)::s') t' m')::s)
    t m
let return = fun _ (v::VK(c)::s) t m -> c (v::s) t m
let push_env = fun c (VEnv(vs)::s) t m ->
  c (VEnv(vs)::VEnv(vs)::s) t m
let pop_env = fun c (v::VEnv(vs)::s) t m -> c (VEnv(vs)::v::s) t m
let call = fun c (v1::v0::s) t m -> match v0 with
    VFun(f) -> f idc (v1::VK(c)::s) t m
  | VContS(c',s',t') -> c' (v1::s') t' (MCons((c,s,t),m))
  | VContC(c',s',t') ->
    c' (v1::s') (apnd t' (cons (fun v t m -> c (v::s) t m) t)) m
let shift i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(VContS(c,s,t)::vs)::[]) TNil m
let control i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(VContC(c,s,t)::vs)::[]) TNil m
let shift0 i = fun c (VEnv(vs)::s) t (MCons((c0,s0,t0),m0)) ->
  i c0 (VEnv(VContS(c,s,t)::vs)::s0) t0 m0
let control0 i = fun c (VEnv(vs)::s) t (MCons((c0,s0,t0),m0)) ->
  i c0 (VEnv(VContC(c,s,t)::vs)::s0) t0 m0
let reset i = fun c (VEnv(vs)::s) t m ->
  i idc (VEnv(vs)::[]) TNil (MCons((c,s,t),m))

(* f8 : e -> string list -> i *)
let rec f8 e xs = match e with
    Var(x) -> access (Env.offset x xs)
  | Fun(x,e) -> push_closure ((f8 e (x::xs)) >> return)
  | App(e0,e1) ->
      push_env >> (f8 e0 xs) >> pop_env >> (f8 e1 xs) >> call
  | Shift(x,e) -> shift (f8 e (x::xs))
  | Control(x,e) -> control (f8 e (x::xs))
  | Shift0(x,e) -> shift0 (f8 e (x::xs))
  | Control0(x,e) -> control0 (f8 e (x::xs))
  | Reset(e) -> reset (f8 e xs)

(* f : e -> v *)
let f expr = f8 expr [] idc (VEnv([])::[]) TNil MNil
```

Listing 9 Type definition for interpreter with defunctionalized instructions and continuations.

```
type v = VFun of i * v list
       | VContS of c * s * t | VContC of c * s * t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of i | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of i | IControl of i | IShift0 of i | IControl0 of i
       | IReset of i | ISeq of i * i
and  c = i list
and  s = v list
and  h = Hold of c * s | Append of h * h
and  t = TNil | Trail of h
type m = (c * s * t) list
```

Listing 10 The function `flat` to remove `ISeq`.

```
(* flat: i -> i list *)
let rec flat i = match i with
    IAccess (n) -> [IAccess (n)]
  | ...
  | ISeq (i0, i1) -> flat i0 @ flat i1
```

5.4 Linearizing Instructions

In the previous section, we used `ISeq` to combine two instructions. As such, an instruction had a tree structure. We can turn it into a linear list by flattening the tree into an OCaml list. With this transformation, `i` in `VFun` becomes `i list` (or equivalently, `c`) and `ISeq` is removed from `i`.

Although the transformation is intuitively clear, to show its correctness, we need to prove that the instructions form a monoid. Namely, the grouping of instructions does not matter as long as the order of instructions is preserved. We briefly sketch the proof. We first define a flattening function (Listing 10) that turns `i` into a list of `i`'s without `ISeq`. We can define similar functions (`flatV`, `flatC`, etc.) that flatten all the instructions appearing in given data (a value, a continuation, etc., respectively). We then prove the following equivalences:

- `flat (f9 e xs)` = `f10 e xs`, stating that the list of instructions generated by the new compiler is the same as flattening the instruction generated by the old compiler, and
- `flatV (run_i9 i c s t m)` = `run_c10 (flat i @ flatC c) (flatS s) (flatT t) (flatM m)`, stating that running `i` under `c` in the old virtual machine yields the same result as running the flattened instructions of `i` and `c` in the new virtual machine (or both do not terminate).

The former is proved by induction on the structure of `e` and the latter on the number of steps the old virtual machine takes. One has to be careful in the case when `i` is `ISeq`. Although the old virtual machine takes a step to execute it, there is no corresponding execution step in the new virtual machine, since `ISeq` is already flattened. Therefore, the termination behavior of the two virtual machines is different when the instruction list contains infinitely many `ISeq`'s: the former continues indefinitely executing `ISeq`'s while the latter terminates since all the `ISeq`'s are already flattened and removed. This does not happen, since all the instructions are finite.

Listing 11 Interpreter with linearized trails.

```
type v = VFun of c * v list | VContS of t | VContC of t
       | VEnv of v list | VK of c
and  i = IAccess of int | IPush_closure of c | IReturn
       | IPush_env | IPop_env | ICall
       | IShift of c | IControl of c | IShift0 of c | IControl0 of c
       | IReset of c
and  c = i list
and  s = v list
and  t = (c * s) list
type m = t list
```

Figure 2 Virtual machine.

$$
\begin{array}{rcl}
c & \Rightarrow & \langle c,\ [\mathit{VEnv}([])],\ [],\ []\rangle \\
\hline
\langle [],\ v :: [],\ [],\ []\rangle & \Rightarrow & v \\
\langle [],\ v :: [],\ [],\ ((c,\ s) :: t) :: m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle [],\ v :: [],\ (c,\ s) :: t,\ m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle \mathit{IAccess}(n) :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ (\mathit{List.nth}\ vs\ n) :: s,\ t,\ m\rangle \\
\langle \mathit{IPushClosure}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VFun}(c',\ vs) :: s,\ t,\ m\rangle \\
\langle \mathit{IReturn} :: \_,\ v :: \mathit{VK}(c) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ v :: s,\ t,\ m\rangle \\
\langle \mathit{IPushEnv} :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VEnv}(vs) :: \mathit{VEnv}(vs) :: s,\ t,\ m\rangle \\
\langle \mathit{IPopEnv} :: c,\ v :: \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VEnv}(vs) :: v :: s,\ t,\ m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VFun}(c',\ vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(v :: vs) :: \mathit{VK}(c) :: s,\ t,\ m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VContS}((c',\ s') :: t') :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ v :: s',\ t',\ ((c,\ s) :: t) :: m\rangle \\
\langle \mathit{ICall} :: c,\ v :: \mathit{VContC}((c',\ s') :: t') :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ v :: s',\ t' \mathbin{@} (c,\ s) :: t,\ m\rangle \\
\langle \mathit{IShift}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(\mathit{VContS}((c,\ s) :: t) :: vs) :: [],\ [],\ m\rangle \\
\langle \mathit{IControl}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(\mathit{VContC}((c,\ s) :: t) :: vs) :: [],\ [],\ m\rangle \\
\langle \mathit{IShift}_0(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0\rangle & \Rightarrow & \langle c' \mathbin{@} c_0,\ \mathit{VEnv}(\mathit{VContS}((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0\rangle \\
\langle \mathit{IControl}_0(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ ((c_0,\ s_0) :: t_0) :: m_0\rangle & \Rightarrow & \langle c' \mathbin{@} c_0,\ \mathit{VEnv}(\mathit{VContC}((c,\ s) :: t) :: vs) :: s_0,\ t_0,\ m_0\rangle \\
\langle \mathit{IReset}(c') :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c',\ \mathit{VEnv}(vs) :: [],\ [],\ ((c,\ s) :: t) :: m\rangle
\end{array}
$$

5.5 Linearizing Trails

Finally, we transform the type `t` of trails, which had a tree structure (Listing 9), into a linear list. By regarding `TNil` as an empty list, `Hold` as a singleton list consisting of `c * s`, and `Append` as a list append, we can represent `t` as a list of `c * s`. The resulting type definitions are shown in Listing 11. Now that `t` becomes `(c * s) list`, we change the type of `VContS` and `VContC` from `c * s * t` to `t` by piling up the `c * s` pair onto `t`. Similarly, `m` can be represented as `t list`.

To establish the correctness of this transformation, we need to show that the new virtual machine behaves the same as before:

```
flatV (run_c10 c s t m) = run_c11 c (flatS s) (flatT t) (flatM m)
```

where `flat` functions are defined similarly to the ones in the previous section to flatten the type of trails. The above equivalence is shown by induction on the number of execution steps the old virtual machine takes.

6 Virtual Machine

Figure 2 shows the state transition rules for the virtual machine obtained from the interpreter in the previous section. The main state consists of a tuple (c, s, t, m) of four elements: a continuation, a stack, a trail, and a metacontinuation. We show an example how a program is compiled to a list of instructions and executed on the virtual machine in the appendix.

The virtual machine succinctly models the low-level behavior of control operators. Just as in the abstract machine, when one of the control operators is used, the current continuation (or a pointer to an instruction) c, stack s, and trail t are captured and put into a stack. Then, the body of the control operator is executed. For *IShift* and *IControl*, the current continuation and trail are cleared, whereas for IShift_0 and $\mathit{IControl}_0$, the ones in the metacontinuation are used. The `reset` operator pushes the current c, s, and t on the metacontinuation m, and initializes them.

When a continuation captured by *IShift* or IShift_0 is invoked, the current c, s, and t are pushed onto m and the captured state is reinstated. When a continuation captured by *IControl* or $\mathit{IControl}_0$ is invoked, on the other hand, the current c and s are added to t to which the captured trail t' is appended.

Although we maintain s, t, and m separately in the virtual machine, we can represent them as a single stack. Remember that s is a list of values. Since t is a list of pairs of c and s, it has the form:

$$[(c, [v; \ldots; v]); \ldots; (c, [v; \ldots; v])]$$

Thus, if we represent c as a single value (e.g., using `VK`) pointing to the first instruction designated by c, and if we maintain the positions of c in t using pointers, we can represent t as a list of values. Furthermore, since m is a list of trails (a list of lists of pairs of c and s), it can be represented as a list of values, too, if we maintain pointers to each element of m.

If we represent s, t, and m as a single stack, we notice that we can sometimes avoid copying s and t. When c, s, and t are pushed to m in the rules for *IReset* and the *VContS* and *VContC* cases of *ICall*, the ordering of s, t, and m does not change. Thus, we can simply rearrange the pointers to the head of a stack, trail, and metacontinuation appropriately, without copying s and t. Similarly for s_0, t_0, and m_0 in the rules for IShift_0 and $\mathit{IControl}_0$. When do we have to copy s and t? It is when we use control operators or apply captured continuations. The s and t must be copied, in the former case to be stored in *VContS* or *VContC*, and in the latter case to use what was stored.

Finally, in the rules for IShift_0 and $\mathit{IControl}_0$, the body instructions c' of the control operators and the instructions c_0 in the metacontinuation are concatenated. This concatenation reflects the fact that the body of IShift_0 and $\mathit{IControl}_0$ has access to the context outside the current enclosing `reset`. (In the abstract machine, the concatenation was realized by executing the body under the continuation stored in the metacontinuation.) Implementation-wise, this suggests that we need to keep track of a list of pointers to these continuations, which is an interesting observation that has not been observed before.

7 Related Work

We are not aware of any work that derives a virtual machine for the four delimited-control operators other than `shift/reset`. Deriving a virtual machine for other language constructs includes Ager, Biernacki, Danvy, and Midtgaard's work [1] for λ-calculus (of various flavors) and Igarashi and Iwaki's work [18] for a staged language.

As for an abstract machine, Biernacki, Danvy, and Millikin [5] present abstract machines for the four delimited-control operators as definitional and derive a CPS interpreter for `control/prompt`. Shan [28] derives an abstract machine for `control/prompt` from the CPS interpreter for `control/prompt`. In both work, the derivation is done for `control/prompt` only. Their abstract machines are similar to ours but do not maintain a stack explicitly.

Dyvbig, Peyton Jones, and Sabry [11] show an abstract machine for primitive control operators that can implement four delimited-control operators with named prompts. Since they use their own primitive control operators, their CPS interpreter is quite different from ours. They do not use trails and represent concatenation of contexts using a metacontinuation, which is a list of continuations. Based on this abstract machine, Kiselyov [21] implements the control operators in OCaml by emulating the behavior of the abstract machine using OCaml's exception handling mechanism.

Hillerström, Lindley, and Atkey [17] show CPS translations and abstract machine semantics for algebraic effects and handlers. It would be interesting to see if the program transformation approach can be used in this setting, too.

8 Conclusion

In this paper, we have derived a compiler and a virtual machine for the four delimited-control operators from the definitional interpreter. The resulting virtual machine suggests a low-level implementation method for delimited continuations.

Although we focused on the behavior of the delimited-control operators, we also want to consider their type systems. We are currently trying to build a type system for the four delimited-control operators (the one for `control/prompt` is in [6]). Once we obtain a type system, we plan to implement the four delimited-control operators in OchaCaml [22] based on the virtual machine developed in this paper. That would form a solid foundation on which a different implementation of algebraic effects and handlers can be considered.

References

1 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. *BRICS Report Series*, 03(14), 2003.

2 Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 8–19. ACM, 2003. `doi:10.1145/888251.888254`.

3 Kenichi Asai and Arisa Kitani. Functional derivation of a virtual machine for delimited continuations. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria*, pages 87–98. ACM, 2010. `doi:10.1145/1836089.1836101`.

4 Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebraic Methods Program.*, 84(1):108–123, 2015. `doi:10.1016/j.jlamp.2014.02.001`.

5 Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. *ACM Trans. Program. Lang. Syst.*, 38(1):2:1–2:25, 2015. `doi:10.1145/2794078`.

6 Youyou Cong, Chiaki Ishio, Kaho Honda, and Kenichi Asai. A functional abstraction of typed invocation contexts. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 12:1–12:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.FSCD.2021.12`.

7 Olivier Danvy. Defunctionalized interpreters for programming languages. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 131–142. ACM, 2008. `doi:10.1145/1411204.1411206`.

8 Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP 1990, Nice, France, 27-29 June 1990*, pages 151–160. ACM, 1990. `doi:10.1145/91556.91622`.

9 Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Math. Struct. Comput. Sci.*, 2(4):361–391, 1992. doi:10.1017/S0960129500001535.

10 Olivier Danvy and Kevin Millikin. Refunctionalization at work. *Sci. Comput. Program.*, 74(8):534–549, 2009. doi:10.1016/j.scico.2007.10.007.

11 R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007. doi:10.1017/S0956796807006259.

12 Kavon Farvardin and John H. Reppy. From folklore to fact: comparing implementations of stacks and continuations. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 75–90. ACM, 2020. doi:10.1145/3385412.3385994.

13 Matthias Felleisen. The theory and practice of first-class prompts. In Jeanne Ferrante and P. Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 180–190. ACM Press, 1988. doi:10.1145/73560.73576.

14 Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, 2017. doi:10.1145/3110257.

15 Martin Gasbichler and Michael Sperber. Final shift for call/cc: direct implementation of shift and reset. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, pages 271–282. ACM, 2002. doi:10.1145/581478.581504.

16 Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In John Williams, editor, *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995*, pages 12–23. ACM, 1995. doi:10.1145/224164.224173.

17 Daniel Hillerström, Sam Lindley, and Robert Atkey. Effect handlers via generalised continuations. *J. Funct. Program.*, 30:e5, 2020. doi:10.1017/S0956796820000040.

18 Atsushi Igarashi and Masashi Iwaki. Deriving compilers and virtual machines for a multi-level language. In Zhong Shao, editor, *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*, volume 4807 of *Lecture Notes in Computer Science*, pages 206–221. Springer, 2007. doi:10.1007/978-3-540-76637-7_14.

19 Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In Jacques Garrigue and Manuel V. Hermenegildo, editors, *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings*, volume 4989 of *Lecture Notes in Computer Science*, pages 239–254. Springer, 2008. doi:10.1007/978-3-540-78969-7_18.

20 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25–27, 2013*, pages 145–158. ACM, 2013. doi:10.1145/2500365.2500590.

21 Oleg Kiselyov. Delimited control in ocaml, abstractly and concretely. *Theor. Comput. Sci.*, 435:56–76, 2012. doi:10.1016/j.tcs.2012.02.025.

22 Moe Masuko and Kenichi Asai. Caml light+ shift/reset= caml shift. *Theory and Practice of Delimited Continuations (TPDC 2011)*, pages 33–46, 2011.

23 Marek Materzok and Dariusz Biernacki. Subtyping delimited continuations. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 81–93. ACM, 2011. doi:10.1145/2034773.2034786.

24 Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Typed equivalence of effect handlers and delimited control. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 30:1–30:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.30.

25 Gordon D. Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009. doi:10.1007/978-3-642-00590-9_7.

26 John C. Reynolds. Definitional interpreters for higher-order programming languages. In John J. Donovan and Rosemary Shields, editors, *Proceedings of the ACM annual conference, ACM 1972, 1972, Volume 2*, pages 717–740. ACM, 1972. doi:10.1145/800194.805852.

27 John C. Reynolds. Definitional interpreters for higher-order programming languages. *High. Order Symb. Comput.*, 11(4):363–397, 1998. doi:10.1023/A:1010027404223.

28 Chung-chieh Shan. A static simulation of dynamic delimited control. *High. Order Symb. Comput.*, 20(4):371–401, 2007. doi:10.1007/s10990-007-9010-4.

A Example Execution

In this appendix, we show an example how the compiler and the virtual machine work. We use the control term in Section 2: $1 + \langle(\mathcal{F}c.\,2 \times c\,3) + \mathcal{F}c'.\,4\rangle$. It is straightforward to support numbers and binary operators; see the supplementary material. State transition rules for the new instructions are summarized in Figure 3.

The list of instructions output by the compiler is:

$$[\mathit{IPushEnv}_1;\ \mathit{INum}(1);\ \mathit{IPopEnv}_1;\ \mathit{IReset}(c_1);\ \mathit{IOp}_1(+)]$$

where

$$\begin{array}{lcl} c_1 & = & [\mathit{IPushEnv}_2;\ \mathit{IControl}_1(c_2);\ \mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \\ c_2 & = & [\mathit{IPushEnv}_3;\ \mathit{INum}(2);\ \mathit{IPopEnv}_3;\ \mathit{IPushEnv}_4;\ \mathit{IAccess}(0);\ \mathit{IPopEnv}_4; \\ & & \ \mathit{INum}(3);\ \mathit{ICall};\ \mathit{IOp}_3(*)] \\ c_3 & = & [\mathit{INum}(4)] \end{array}$$

We use subscripts to disambiguate instructions that appear more than once.

The list of instruction is executed as in Figure 4. We can observe that the trails $3 + []$ (i.e., $([\mathit{IOp}_2(+)], [\mathit{VNum}(3)])$) and $2 \times []$ (i.e., $([\mathit{IOp}_3(*)], [\mathit{VNum}(2)])$) are concatenated at the second invocation of *IControl* and are captured in vc_2.

$$\boxed{\begin{array}{rcl} \langle \mathit{INum}(n) :: c,\ \mathit{VEnv}(vs) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n) :: s,\ t,\ m\rangle \\ \langle \mathit{IOp}(+) :: c,\ \mathit{VNum}(n_0) :: \mathit{VNum}(n_1) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n_0 + n_1) :: s,\ t,\ m\rangle \\ \langle \mathit{IOp}(*) :: c,\ \mathit{VNum}(n_0) :: \mathit{VNum}(n_1) :: s,\ t,\ m\rangle & \Rightarrow & \langle c,\ \mathit{VNum}(n_0 * n_1) :: s,\ t,\ m\rangle \end{array}}$$

Figure 3 State transition rules for *INum* and *IOp*.

Instruction: $[\mathit{IPushEnv}_1;\ \mathit{INum}(1);\ \mathit{IPopEnv}_1;\ \mathit{IReset}(c_1);\ \mathit{IOp}_1(+)]$

$$\begin{array}{lcl}
c_1 & = & [\mathit{IPushEnv}_2;\ \mathit{IControl}_1(c_2);\ \mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \\
c_2 & = & [\mathit{IPushEnv}_3;\ \mathit{INum}(2);\ \mathit{IPopEnv}_3;\ \mathit{IPushEnv}_4;\ \mathit{IAccess}(0);\ \mathit{IPopEnv}_4;\ \mathit{INum}(3);\ \mathit{ICall};\ \mathit{IOp}_3(*)] \qquad c_3 \ = \ [\mathit{INum}(4)] \\
c_4 & = & [\mathit{IOp}_1(+)] \qquad c_5 \ = \ [\mathit{IPopEnv}_2;\ \mathit{IControl}_2(c_3);\ \mathit{IOp}_2(+)] \qquad c_6 \ = \ [\mathit{IOp}_3(*)] \qquad c_7 \ = \ [\mathit{IOp}_2(+)] \\
vc_1 & = & \mathit{VContC}((c_5, [\mathit{VEnv}([])]) :: []) \qquad vc_2 \ = \ \mathit{VContC}((c_7, [\mathit{VNum}(3)]) :: (c_6, [\mathit{VNum}(2)]) :: [])
\end{array}$$

$$\begin{array}{llrrrr}
 & \langle & \mathit{IPushEnv}_1 :: \ldots, & \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{INum}(1) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_1 :: \ldots, & \mathit{VNum}(1) :: \mathit{VEnv}([]) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IReset}(c_1) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VNum}(1) :: [], & [], & []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_2 :: \ldots, & \mathit{VEnv}([]) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IControl}_1(c_2) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VEnv}([]) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_3 :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(2) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_3 :: \ldots, & \mathit{VNum}(2) :: \mathit{VEnv}(vc_1 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPushEnv}_4 :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IAccess}(0) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_4 :: \ldots, & vc_1 :: \mathit{VEnv}(vc_1 :: []) :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(3) :: \ldots, & \mathit{VEnv}(vc_1 :: []) :: vc_1 :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{ICall} :: \ldots, & \mathit{VNum}(3) :: vc_1 :: \mathit{VNum}(2) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IPopEnv}_2 :: \ldots, & \mathit{VNum}(3) :: \mathit{VEnv}([]) :: [], & (c_6, [\mathit{VNum}(2)]) :: [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IControl}_2(c_3) :: \ldots, & \mathit{VEnv}([]) :: \mathit{VNum}(3) :: [], & (c_6, [\mathit{VNum}(2)]) :: [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{INum}(4) :: [], & \mathit{VEnv}(vc_2 :: []) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & [], & \mathit{VNum}(4) :: [], & [], & ((c_4,\ [\mathit{VNum}(1)]) :: []) :: []\rangle \\
\Rightarrow & \langle & \mathit{IOp}_1(+) :: [], & \mathit{VNum}(4) :: \mathit{VNum}(1) :: [], & [], & []\rangle \\
\Rightarrow & \langle & [], & \mathit{VNum}(5) :: [], & [], & []\rangle
\end{array}$$

Figure 4 An example execution of $1 + \langle(\mathcal{F}c.\, 2 \times c\, 3) + \mathcal{F}c'.\, 4\rangle$ on the virtual machine.