Delimited Control in OCaml, Abstractly and Concretely System Description

Oleg Kiselyov

FNMOC `oleg@okmij.org`

Abstract. We describe the first implementation of multi-prompt delimited control operators in OCaml that is *direct* in that it captures only the needed part of the control stack. The implementation is a library that requires no changes to the OCaml compiler or run-time, so it is perfectly compatible with existing OCaml source code and byte-code. The library has been in fruitful practical use for four years.
We present the library as an implementation of an abstract machine derived by elaborating the definitional machine. The abstract view lets us distill a minimalistic API, scAPI, sufficient for implementing multi-prompt delimited control. We argue that a language system that supports exception and stack-overflow handling supports scAPI. Our library illustrates how to use scAPI to implement multi-prompt delimited control in a typed language. The approach is general and can be used to add multi-prompt delimited control to other existing language systems.

1 Introduction

The library delimcc of delimited control for byte-code OCaml was released at the beginning of 2006 [1] and has been used for implementing (delimited) dynamic binding [2], a very shallow embedding of a probabilistic domain-specific language [3, 4], CGI programming with nested transactions [5], efficient and comprehensible direct-style code generators [6], normalization of MapReduce-loop bodies by evaluation [7], and automatic bundling of RPC requests [8].

The delimcc library was the first *direct* implementation of delimited control in a typed, mainstream, mature language – it captures only the needed prefix of the current continuation, requires no code transformations, and integrates with native-language exceptions. Captured delimited continuations can be serialized, stored, or migrated, then resumed in a different process.

The delimcc library is an OCaml *library* rather than a fork or a patch of the OCaml system. Like the `num` library of arbitrary-precision numbers, delimcc gives OCaml programmers new datatypes and operations, some backed by C code. The delimcc library does *not* modify the OCaml compiler or run-time in any way, so it ensures perfect binary compatibility with existing OCaml code and other libraries. This library shows that delimited control can be implemented efficiently (without copying the whole stack) and non-invasively in a typed language that was not designed with delimited control in mind and that offers no compiler plug-ins or run-time extensions beyond a basic foreign-function interface. Our goal in this paper is to describe the implementation of delimcc with enough detail and generality so that it can be replicated in other language systems.

The delimcc library implements the so-called multi-prompt delimited control operators that were first proposed by Gunter, Rémy, and Riecke [9] and further developed by Dybvig, Peyton Jones, and Sabry [10]. The multi-prompt operators turn out indispensable for normalization-by-evaluation for strong sums [11]. Further applications of specifically multi-prompt operators include the implementation of delimited dynamic binding [2] and the normalization of loop bodies by evaluation [7]. The delimcc library turns out suitably fast, useful, and working in practice. In this paper, we show that it also works in theory.

We describe the implementation and argue for its generality and correctness. The correctness argument cannot be formal: after all, there is no formal specification of OCaml, with or without delimited control. We informally relate the byte-code OCaml interpreter to an abstract machine, which we rigorously relate to abstract machines for delimited control. The main insight is the discovery that OCaml byte-code already has the facilities needed to implement delimited control efficiently. In fact, any language system accommodating exception handling and recovery from control-stack overflow likely offers these facilities. Languages that use recursion extensively typically deal with stack overflow [12].

Our contributions are as follows.

1. We state the semantics of multi-prompt delimited control in a form that guides the implementer, in §3. We derive a minimalistic API, scAPI, sufficient for implementing delimited control. For generality, we describe scAPI in terms of an abstract state machine, which focuses on activation frame manipulation while eliding idiosyncratic details of concrete language systems. Our scAPI includes the creation of 'stable-point' frames, completely describing the machine state including the contents of non-scratch registers. We should be able to identify the recent stable point frame and copy a part of the stack between two stable points. We do *not* require marking of arbitrary frames, adding new types of frames, or even knowing the format of the stack.
2. On the concrete example of the OCaml byte-code and delimcc, we demonstrate in §4 using the scAPI to implement multi-prompt delimited control.[1] OCaml happens to support scAPI, §4.2.
3. The implementation of delimcc poses challenging typing problems, which previously [10, 13] were handled using unsafe coerce. We use reference cells to derive in §4.1 a safe solution, free from any undefined behavior.
4. The experience with the delimcc library called for an extension of the simple interface [10], to avoid a memory leak in multi-prompt shift, appendix B of the full paper.[2] The new primitive `push_delim_subcont` reinstates the captured continuation along with its delimiter.

[1] The Scheme implementation, mentioned on the delimcc web page, is another concrete example of using scAPI, attesting to the generality of the approach.

[2] Available at `http://okmij.org/ftp/Computation/caml-shift.pdf`

5. We describe serialization of captured delimited continuations so to make them persistent. We show why serialized delimited continuations must refer to some reachable data by name rather than incorporate everything by value. Serialized delimited continuations should be, so to speak, twice delimited.[3]

We review the related work in §5 and then conclude. The performance of the library proved adequate, see [4]. In particular, aborting part of the computation with delimcc is just as fast as raising an OCaml exception. We start by introducing the multi-prompt delimited control and the delimcc library in §2.

The delimcc library source along with validation tests and sample code is freely available from `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`.

2 Multi-prompt Delimited Control

Before discussing the implementation of delimcc, we introduce the library on sample code, informally describing multi-prompt delimited control. The basic delimcc interface, taken from [10], defines two abstract types and four functions:

```
type 'a prompt
type ('a,'b) subcont

val new_prompt   : unit -> 'a prompt
val push_prompt  : 'a prompt -> (unit -> 'a) -> 'a
val take_subcont : 'b prompt -> (('a,'b) subcont -> unit -> 'b) -> 'a
val push_subcont : ('a,'b) subcont -> (unit -> 'a) -> 'b
```

whose semantics is formally discussed in §3. Intuitively, a value of the type `'a prompt` is an exception object, with operations to pack and extract a thunk of the type `unit -> 'a`. The expression `new_prompt ()` produces a fresh exception object; `take_subcont p (fun _ () -> e)` packs `fun () -> e` into the exception object denoted by the prompt `p`, and raises the exception. The expression `push_prompt p (fun () -> e)` is akin to OCaml's `try e with ...` form, evaluating `e` and returning its result. Should `e` raise an exception `p`, it is caught, the contained thunk is extracted, and the result of its evaluation is returned. All other exceptions are re-raised. As an example, let us left fold over a file, reading the file line-by-line and reducing using the given function `f`:

```
(* val fold_file: ('a -> string -> 'a) -> 'a -> in_channel -> 'a *)
let fold_file f z file = let ex = new_prompt () in
 let rec loop z =
  let inp =
   try input_line file with End_of_file -> take_subcont ex (fun _ ()-> z)
  in loop (f z inp)
 in push_prompt ex (fun () -> loop z);;
```

[3] Due to the lack of space, we refer the reader to the long title comments in the file `delimcc.ml` for the explanation of the serialization.

For example, `fold_file (fun z s -> z + 1) 0 cin` returns the line count in the input channel `cin`. The code for `fold_file` is exactly equivalent to

```
let fold_file f z file : 'a = let exception Ex of 'a in
 let rec loop z =
  let inp = try input_line file with End_of_file -> raise (Ex z)
  in loop (f z inp)
 in try loop z with Ex z -> z
```

if OCaml had local exception declaration such as those in SML. OCaml however lacks such exception declarations.[4] The delimcc library thus fills this omission.

The exceptions thrown by `take_subcont` are restartable: `take_subcont p (fun sk () -> e)` would bind `sk` to a 'restart object' before raising the exception `p`; `e` may return the object as part of its result. Given the restart object, `push_subcont` restarts the exception, continuing the execution from the point of `take_subcont p` till `push_prompt p`, returning the result of the latter. The following should make it concrete. First we introduce `shift0` that captures a frequently occurring pattern

```
(* val shift0: 'a prompt -> (('b -> 'a) -> 'a) -> 'b *)
let shift0 p f = take_subcont p (fun sk () ->
  f (fun c -> push_prompt p (fun () -> push_subcont sk (fun () -> c))))
```

which is used as follows:

```
type 'a res = Value of 'a | Exc of 'a * (unit -> 'a res)
let accum p z str =
  if str = "" then shift0 p (fun k -> Exc (z,fun () -> k z))
  else z + String.length str
```

We may view `shift0` in this code as raising the exception `p`, with `k` bound to the restart function. When `k` is applied to a value `z`, the execution continues as if the entire `shift0` expression had been replaced by `z`. Since the computation, after restart, may raise the exception again, we have to be able to handle it, hence the call to `push_prompt`. The function `accum` is meant to be a reducer function passed to a fold:

```
let sum_arr arr = let p = new_prompt () in
 push_prompt p (fun () -> Value (Array.fold_left (accum p) 0 arr));;

let t2 = sum_arr [| "FLOPS";"";"2010"|];;
↪ val t2 : int res = Exc (5, <fun>)
let t3 = match t2 with Exc (_,resume) -> resume ();;
↪ val t3 : int res = Value 9
```

The function `sum_arr` sums the lengths of all strings in a string array. Encountering an empty string throws an exception. The function `sum_arr` then returns

[4] Placing exception declarations into an OCaml local module does not fully implement SML local exceptions. In SML, a local exception declaration may refer to a bound type variable. A type variable in OCaml cannot bind into a local structure.

`Exc (z,resume)` reporting the length so far. Evaluating `resume ()` restarts the exception and resumes the accumulation, returning either the final result `Value z` or another exception. The same exception can be restarted more than once, which is particularly useful for probabilistic programming [3]. The functions `accum` and `sum_arr` have demonstrated the application of delimited control to 'invert' an enumerator, that is, to convert the enumerator to a stream [14, 15].

We can use `accum` with `fold_file` defined earlier, to sum the lengths of the strings read from the file, stopping at empty strings. Although `fold_file` itself uses delimited control, the two `take_subcont` use different prompts and so act unaware of each other.

The formal, small-step semantics of these delimited control operators was specified in [9] (`push_prompt` was called `set` and `take_subcont` was called `cupto`) – as a set of re-writing rules. The rules, which operate essentially on the source code, greatly help a programmer to predict the evaluation result of an expression. Alas, the rules offer little guidance for the implementer since typical language systems are stateful machines, whose behavior is difficult to correlate with pure source-code re-writing.

3 Abstract Machine for Multi-prompt Delimited Control

More useful for the implementer is semantics expressed in terms of an abstract machine, whose components and steps can, hopefully, be related to an implementation of a concrete machine at hand. By abstracting away implementation details, abstract state machines let us discern generally applicable lessons. Our first lesson is the identification of a small scAPI for manipulating the control stack. We further learn that any language system supporting exception handling already implements a half of scAPI.

We start with the definitional machine introduced in [10, Figure 1] as a formal specification of multi-prompt delimited control. We reproduce the definition in appendix A for reference. The machine contains features that are recognizable by implementers, such as 'context' – which is a sequence of activation frames, commonly known as '(control) stack.' The machine however contains an extra component, a list of contexts. It is not immediately clear what it may correspond to in concrete machines, raising doubts if delimited control can be added to an existing machine such as the OCaml byte-code without re-designing it.

These worries are unfounded. The machine of [10] can be converted into the equivalent machine described below, which has no extra components such as lists of control stacks. We prove the equivalence in appendix A. Our machine M_{dc}, Figure 1, is bare-bone: it has no environment, arithmetic and many other practically useful features, which are orthogonal and can be easily added. It abstracts away all details except for control stack. The machine can be viewed as a generalization of the environment-less version of the machine of [16].

The program for the machine is call-by-value λ-calculus, augmented with integral-valued prompts and delimited control operators. The operators here are syntactic forms rather than constants: for example, `newP` evaluates each time to

Variables $x, y, \ldots$ Prompts $p, q \in N$

Expressions $e ::= v \mid e\,e \mid \texttt{newP} \mid \texttt{pushP}\,e\,e \mid \texttt{takeSC}\,e\,e \mid \texttt{pushSC}\,e\,e$

Values $v ::= x \mid \lambda x.\, e \mid p \mid D$

Contexts $D ::= \Box \mid De \mid vD \mid \texttt{pushP}\,D\,e \mid \texttt{pushSC}\,D\,e \mid \texttt{takeSC}\,D\,e$
$\mid \texttt{takeSC}\,p\,D \mid \texttt{pushP}\,p\,D$

Transitions between configurations (e, D, q)

$$
\begin{array}{rcll}
(ee', D, q) & \mapsto & (e, D[\Box e'], q) & e \text{ non-value} \\
(ve, D, q) & \mapsto & (e, D[v\Box], q) & e \text{ non-value} \\
(\texttt{pushP}\,ee', D, q) & \mapsto & (e, D[\texttt{pushP}\,\Box e'], q) & e \text{ non-value} \\
(\texttt{takeSC}\,ee', D, q) & \mapsto & (e, D[\texttt{takeSC}\,\Box e'], q) & e \text{ non-value} \\
(\texttt{takeSC}\,pe, D, q) & \mapsto & (e, D[\texttt{takeSC}\,p\Box], q) & e \text{ non-value} \\
(\texttt{pushSC}\,ee', D, q) & \mapsto & (e, D[\texttt{pushSC}\,\Box e'], q) & e \text{ non-value} \\
\\
((\lambda x.\, e)v, D, q) & \mapsto & (e[v/x], D, q) & \\
(\texttt{newP}, D, q) & \mapsto & (q, D, q+1) & \\
(\texttt{pushP}\,pe, D, q) & \mapsto & (e, D[\texttt{pushP}\,p\Box], q) & \\
(\texttt{takeSC}\,pv, D, q) & \mapsto & (vD_1, D_2, q) & D_2[\texttt{pushP}\,pD_1] = D,\ \texttt{pushP}\,pD' \notin D_1 \\
(\texttt{pushSC}\,D'e, D, q) & \mapsto & (e, D[D'], q) & \\
\\
(v, D[D_1], q) & \mapsto & (D_1[v], D, q) & D_1 \neq \Box \\
(\texttt{pushP}\,pv, D, q) & \mapsto & (v, D, q) &
\end{array}
$$

Fig. 1. Abstract machine $\mathsf{M_{dc}}$ for multi-prompt delimited control

a new prompt. In delimcc, we eschew extending the syntax of OCaml. Therefore, we represent `newP` as a function application `new_prompt ()`. Likewise, $\texttt{pushP}\,p\,e$ takes the form `push_prompt p (fun () -> e)` in delimcc. The operation $D[u]$ replaces the hole in context D with u, which may be either an expression or another context; $e[v/x]$ stands for a capture-avoiding substitution of v for variable x in expression e. Prompts p and contexts D may not appear in source programs. The machine operates on configurations (e, D, q) of the current expression e, 'stack' D and the counter for generating fresh prompt names. The initial configuration is $(e, \Box, 0)$; the machine stops when it reaches $(v, \Box, q)$.

The machine exhibits familiar to the implementers features: D is a sequence of activation frames, the 'stack'; the first six transitions look like a function call, pushing a new activation frame onto the stack; the last-but-one transition is akin to the function return, popping the frame. (For generality, we only require the sequence of the popped frames D_1 to be non-empty.) The machine also exhibits non-standard stack-manipulation operations: $D[D']$ in the `pushSC` transition pushes several frames D' at once onto the stack; the `takeSC` transition involves locating a particular frame $\texttt{pushP}\,pD_1$ and splitting the stack at that frame. The removed prefix D_1 is passed as a value to the argument of `takeSC`; in a real machine, the stack prefix D_1 would be copied onto heap, the ordinary place of storing composite values. These non-standard stack operations thus constitute an API, which we call scAPI, for implementing multi-prompt delimited control.

To see how scAPI may be supported, we relate scAPI with exception handling, a widely supported feature. As a specification of exception handling we take an abstract machine M_{ex}, Figure 2. The program for M_{ex} is too call-by-value λ-calculus, extended with the operations to raise and catch exceptions. These operations are indexed by exception types. A source programmer has an unlimited supply of exception types to choose from. Exception types, however, are not values and cannot be created at run-time.

Variables $x, y, \ldots$ Exceptions $p, \ldots$

Expressions $e ::= v \mid e\,e \mid \mathtt{raise}_p\, e \mid \mathtt{try}_p\, e\, e$

Values $v ::= x \mid \lambda x.\, e$

Contexts $D ::= \square \mid De \mid vD \mid \mathtt{raise}_p\, D \mid \mathtt{try}_p\, D\, e$

Transitions between configurations (e, D)

$$
\begin{array}{rcll}
(ee', D) & \mapsto & (e, D[\square e']) & e \text{ non-value} \\
(ve, D) & \mapsto & (e, D[v\square]) & e \text{ non-value} \\
(\mathtt{raise}_p\, e, D) & \mapsto & (e, D[\mathtt{raise}_p\, \square]) & e \text{ non-value} \\[1ex]
((\lambda x.\, e)v, D) & \mapsto & (e[v/x], D) & \\
(\mathtt{try}_p\, ee', D) & \mapsto & (e, D[\mathtt{try}_p\, \square e']) & \\
(\mathtt{raise}_p\, v, D) & \mapsto & (e'v, D_2) & D_2[\mathtt{try}_p\, D_1 e'] = D,\ \mathtt{try}_p\, D' e \notin D_1 \\[1ex]
(v, D[D_1]) & \mapsto & (D_1[v], D) & D_1 \neq \square \\
(\mathtt{try}_p\, ve', D) & \mapsto & (v, D) &
\end{array}
$$

Fig. 2. Abstract machine M_{ex} for exception handling

The comparison of Figures 1 and 2 shows many similarities. For example, we observe that the expression $\mathtt{pushP}\, p\, v$ reduces to v in any evaluation context; likewise, $\mathtt{try}_p\, v\ e'$ reduces to v for any D. One may also notice a similarity between raising an exception and `takeSC` that disregards the captured continuation. On the other hand, `takeSC` uses prompts whose new values can be created at run-time; the set of exceptions is fixed during the program execution. To dispel doubts, we state the equivalence result precisely, even more so as we rely on it in the implementation.

First, we have to extend M_{ex} with integers serving as prompts, which can be compared for equality using `==`. Prompts cannot appear in source programs but are generated by an operator `newP`, evaluating each time to a fresh value. We add unit (), pairs (e, e) and pair projections `fst` and `snd`, and the conditional. We call the extended machine $\mathsf{M}'_{\mathsf{ex}}$. Let $\mathsf{M}'_{\mathsf{dc}}$ be M_{dc} with a restriction on source programs: no `pushSC`, all `takeSC` expressions must be of the form $\mathtt{takeSC}\, e\, (\lambda x.\, e')$ where

x is not free in e'. Therefore, contexts D are not values of $\mathsf{M}'_{\mathsf{dc}}$. We define the translation $\lfloor\cdot\rfloor$ of $\mathsf{M}'_{\mathsf{dc}}$ expressions to the expressions of $\mathsf{M}'_{\mathsf{ex}}$ as follows (where p_0 is a dedicated exception type):

$$\begin{aligned}
\lfloor \mathtt{takeSC}\, p\, (\lambda x.\, e)\rfloor &= \mathtt{raise}_{p_0} (\lambda x.\, e, p)\\
\lfloor \mathtt{pushP}\, p\, e\rfloor &= \mathtt{try}_{p_0}\, e\, (\lambda y.\, \mathtt{if}\ p \mathbin{\mathtt{==}} \mathtt{snd}\, y\ \mathtt{then}\ \mathtt{fst}\, y\, ()\ \mathtt{else}\ \mathtt{raise}_{p_0}\, y)
\end{aligned}$$

It is homomorphism in the other cases. The intuition comes from mail-relay systems. The exception is an envelope, the prompt p is an address, the exception handler is a relay station, which matches the address on the envelope with its own. If the address matches, the station opens the envelope; otherwise, it forwards the message to the next relay. More formally, we state: for all $\mathsf{M}'_{\mathsf{dc}}$ source programs e, the machine reaches the terminal configuration iff $\mathsf{M}'_{\mathsf{ex}}$ does so for the source program $\lfloor e\rfloor$. The proof is straightforward bi-simulation.

We conclude that M_{ex} effectively provides the operation to locate a particular stack frame and split the stack at the frame, disregarding the prefix. That particular stack frame, $\mathtt{try}_p\, D\, e'$ is quite like the frame $\mathtt{pushP}\, pD$ that has to be located in M_{dc}. Thus any real machine that supports exception handling implements a part of scAPI.

To see how the stack-copying part of scAPI could be implemented, we turn to stack overflow. Any language system that supports and encourages recursion has to face stack overflow and should be able to recover from it [12]. Recovery typically involves either copying the stack into a larger allocated area, or adjoining a new stack fragment. In the latter case, the implementation needs to handle stack underflow, to switch to the previous stack fragment. In the extreme case, each 'stack' fragment is one-frame long and so all frames are heap-allocated. In every case, the language system has to copy, or adjoin and remove stack fragments. These are exactly the operations of scAPI. The deep analogy between handling stack overflow and underflow on one hand and capturing and reinstating continuations on the other hand has been noted in [12].

We now introduce an equivalent variant of M_{dc} ensuring that a captured continuation is delimited by `pushP` frames on both ends. These frames are *stable points*. Real machines use the control stack as a scratch allocation area and for register spill-over. The state of real machines also contains more components (such as CPU registers), used as a fast cache for various frame data [17]. When capturing continuation, we have to make sure that all these caches are flushed so that the captured activation frames contain the complete state for resuming the computation. As we rely on exception handling for support of a part of scAPI, we identify `pushP` frames with exception handling frames. To our knowledge, the points of exception handling correspond to stable points of concrete machines.

We define the variant $\mathsf{M}^i_{\mathsf{dc}}$ of M_{dc} by changing two transitions to:

$$\begin{aligned}
(\mathtt{takeSC}\, pv, D, q) &\mapsto (vD_1, D_2, q)\\
&\quad D_2[\mathtt{pushP}\, pD_1] = D[\mathtt{pushP}\, p'\square], \quad p' \text{ fresh}, \quad \mathtt{pushP}\, pD' \notin D_1\\
(\mathtt{pushSC}\, D'e, D, q) &\mapsto (e, D[\mathtt{pushP}\, p''D'], q) \quad p'' \text{ fresh}
\end{aligned}$$

Strictly speaking, we ought to have introduced an auxiliary counter q' in the configuration to generate fresh auxiliary prompts p' and p''. We can prove the equivalence of the modified M_{dc} to the original one, using bi-simulation similar to the one in appendix A. The key fact is that the auxiliary prompts are fresh, are not passed as values and so there cannot be any `takeSC` operations referring to these prompts. Any continuation captured by $\mathsf{M}^{i}_{\mathsf{dc}}$ is delimited by `pushP` p' at one end and `pushP` p at the other: the continuation is captured between two stable points, as desired. The re-instated continuation is too sandwiched between two `pushP` frames: `pushP` $p'\square$ is part of the captured continuation, the other frame is inserted by `pushSC`. The presence of `pushP` on both ends also helps in making delimcc well-typed, as we see next.

4 Implementation in OCaml

In the previous section, we have introduced the deliberately general and minimalistic scAPI that is sufficient to implement delimited control, and shown that a concrete language system supporting handling of exceptions and of stack overflow is likely to implement scAPI. We now demonstrate both points on the concrete example of OCaml: that is, we describe the implementation of delimcc. In §4.2 we show how exactly OCaml, which supports exceptions and handles stack overflow, implements scAPI. In fact, the OCaml byte-code interpreter is an instance of $\mathsf{M}'_{\mathsf{ex}}$ extended with the operations for copying parts of stack. §4.3 then explains the implementation of delimcc in terms of scAPI, closely following the 'abstract implementation' in §3. The OCaml byte-code interpreter is written in C; our delimcc code is in OCaml (using thin C wrappers for scAPI), giving us more confidence in the correctness due to the expressive language and the use of types. OCaml is a typed language; the delimcc interface is also typed. Having avoided types so far we confront them now.

4.1 Implementing Typed Prompts

We describe the challenges of implementing delimited control in a typed language on a simpler example, of realizing the $\mathsf{M}'_{\mathsf{dc}}$ machine, with the restricted form of `takeSC`, in terms of exception handling. Earlier, in §3, we explained the implementation on abstract machines. The version of that code in OCaml:

```
let take_subcont p thunk = raise (P0 (thunk,p))
let push_prompt p thunk  = try thunk () with
   (P0 (v,p')) as y -> if p = p' then v () else raise y
```

is ill-typed for two reasons. First, the type of a prompt in delimcc, §2 (whose interface is based on [9, 10]) is parametrized by the so-called answer-type, the type of values yielded by the `push_prompt` that pushed it. The prompts `p` and `p'` in the above code are generally pushed by different `push_prompt`s and hence may have different types. In OCaml, we can only compare values of the same type. To solve the problem, we implement prompts as records with an `int` component, called 'mark', making `new_prompt` produce a unique value for that field. We can then compare prompts by comparing their marks. (The overhead of marks proved negligible.) A deeper problem is that the typing of `try e1 with ex ->`

e2 in OCaml requires e1 and e2 be of the same type. Hence thunk and v in our code must have the same type. However, thunk produces the value to return by push_prompt p and v is 'thrown to' push_prompt p'. Generally, p and p', and so thunk and v, have different types. It is only when the marks of p and p' have the same value that v and thunk have the same type. Dependent types, or at least recursive and existential types [18] seem necessary.

The post-office intuition helps us again: we usually do not communicate with a mailman directly; rather, we use a shared mailbox. The correspondence between take_subcont and push_prompt is established through a common prompt, a shared value. This prompt is well-suited for the role of the mailbox. A reference cell of the type 'a option ref may act as a mailbox to exchange values of the type 'a; the empty mailbox contains None. Since in our code take_subcont sends to push_subcont a thunk, it is fitting to rather use (unit -> 'a) ref as the mailbox type.

```
type 'a prompt = {mbox: (unit -> 'a) ref; mark: unit ref}
let mbox_empty () = failwith "Empty mbox"

let mbox_receive p =     (* val mbox_receive : 'a prompt -> 'a *)
  let k = !(p.mbox) in p.mbox := mbox_empty; k ()
let new_prompt () = {mbox = ref mbox_empty; mark = ref ()};;
```

The mark field of the prompt should uniquely identify the prompt. Since we already use reference cells, and since OCaml has the physical equality ==, it behooves us to take a unit ref as prompt's mark. We rely on the fact that each evaluation of ref () gives a unique value, which is == only to itself. If physical equality is not provided, we can always emulate it via equi-mutability.

To send a thunk to a push_prompt, the operation take_subcont deposits the thunk into the shared mailbox and 'alerts' the receiver, by sending the exception containing the mark of the mailbox. Since the type of the mark is always unit ref regardless of the type of the thunk, we no longer have any typing problems.

```
exception P0 of unit ref
let take_subcont p thunk = p.mbox := thunk; raise (P0 p.mark)
let push_prompt p thunk = try thunk ()
   with (P0 mark') as y ->
    if p.mark == mark' then mbox_receive p else raise y;;
```

Anticipating the continuation capture in §4.3, we make the code more uniform:

```
let push_prompt p thunk =
  try let res = thunk () in p.mbox := (fun () -> res); raise (P0 p.mark)
  with (P0 mark') as y ->
    if p.mark == mark' then mbox_receive p else raise y;;
```

The inferred type is 'a prompt -> (unit -> 'a) -> 'a, befitting delimcc. The value produced by push_prompt is in every case the value received from the mailbox. Our earlier typing problems are clearly eliminated.

4.2 scAPI in OCaml

We now precisely specify scAPI and describe how the OCaml byte-code implements it. We formulate scAPI as the interface

```
module EK : sig   type ek   type ekfragment
 val get_ek : unit -> ek
 val add_ek : ek -> ek -> ek
 val sub_ek : ek -> ek -> ek
 val pop_stack_fragment  : ek -> ek -> ekfragment
 val push_stack_fragment : ekfragment -> unit
end
```

with two abstract types, `ek` and `ekfragment`. The former identifies an exception frame; `get_ek ()` returns the identity of the latest exception frame. There are *no* operations to scan the stack looking for a particular frame. A stack fragment between two exception frames is represented by `ekfragment`. Given the stack of the form $D_2[\mathtt{try}_{\mathrm{ek1}}[D_1[\mathtt{try}_{\mathrm{ek2}}\, D']]]$, `pop_stack_fragment ek1 ek2` transforms the stack to $D_2[\mathtt{try}_{\mathrm{ek1}}\, D']$ returning the removed part $D_1[\mathtt{try}_{\mathrm{ek2}}\, \square]$ as `ekfragment`. One of the exception frames is captured as part of `ekfragment`. The operation `push_stack_fragment ekfragment` splices such an `ekfragment` in at the point of the latest exception frame, turning the stack from $D_2[\mathtt{try}_{\mathrm{ek}}\, D']$ to $D_2[\mathtt{try}_{\mathrm{ek}}[D_1[\mathtt{try}_{\mathrm{ek2}}\, D']]]$. These stack operations clearly correspond to the transitions of $\mathsf{M}^{i}_{\mathsf{dc}}$ in §3. We never capture the top stack fragment D' and never copy onto the top of the stack D' because D' contains ephemeral local data [17]. When the captured `ekfragment` is pushed back onto the stack, the identities of the exception frames captured in the fragment may change. If we obtained the identities of the captured frames before, we should adjust our `ek` values; hence the operations `add_ek` and `sub_ek`.

The OCaml byte-code interpreter [19], an elaboration of the abstract machine ZAM [17], supports exceptions, pairs, conditionals, comparison, state to generate unique identifiers – and is thus an instance of $\mathsf{M}'_{\mathsf{ex}}$. Exception frames are linked together; the dedicated register `trapsp` of the interpreter keeps the pointer to the latest exception frame. Therefore, we can identify exception frames by their pointers; `ek` is such a pointer, relative to the beginning of the stack `caml_stack_high`, in units of `value`. Evaluating `try e with ...` creates a new exception frame before evaluating `e`. Reading `trapsp` in `e` by executing `get_ek ()` gives us the identity of the created exception frame. Since the relative pointer is just an integer, `add_ek` and `sub_ek` are integer addition and subtraction. OCaml handles stack overflow by copying the stack into a larger allocated memory block. That implies that either there are no absolute pointers to stack values stored in data structures, or there is a way to adjust them. In fact, the only absolute pointers into stack are the link pointers in exception frames. The OCaml byte-code has a procedure to adjust such pointers after copying the stack. The operations `pop_stack_fragment` and `push_stack_fragment` are the variants of interpreter's stack-copying procedure. These operations along with `get_ek` can be invoked from OCaml code via the foreign-function interface.

4.3 Implementing delimcc in Terms of scAPI

In this section we show how to use scAPI to implement the delimcc interface, presented in §2. One may view this section as an example of transcribing the abstract implementation, $\mathsf{M}^i_{\mathsf{dc}}$ in §3, into OCaml, keeping the code well-typed. The transcription is mostly straightforward, after we remove the final obstacle that we now explain.

Recall that $\mathsf{M}^i_{\mathsf{dc}}$ requires locating on the stack a `pushP` p frame with a particular prompt value p and copying parts of stack between two `pushP` frames. OCaml, via scAPI, supports copying parts of stack between exception frames. We can also obtain the identity of the latest exception frame. However, scAPI gives us no way to scan the stack looking for a frame with a particular identity. §4.1 showed how to relate a `push_prompt` frame to an exception frame and how to locate on stack a `push_prompt p` frame with a particular prompt value p – alas, flushing the stack up to that point. We have to find a way to identify a `pushP` frame without disturbing the stack.

The solution is easy: `push_prompt` should maintain its own stack of its invocations, called 'parallel stack' or `pstack`. The `pstack` is a mutable list of `pframe`s, which we can easily scan. A `pframe` on `pstack` corresponds to a `push_prompt` on the real stack and contains the identity of `push_prompt`'s exception frame and the mark of the prompt (see §4.1) 'pushed' at that point:

```
exception DelimCCE
type pframe = {pfr_mark : unit ref; pfr_ek : ek}
type pstack = pframe list ref
let ptop : pstack = ref []
```

`DelimCCE` is the dedicated exception type, called p_0 in M_{ex} and `P0` in §4.1. Unlike the latter, the exception no longer carries the prompt's identity since we obtain this identity from `pstack`, accessed via the global variable `ptop`. Essentially, `pstack` maintains the association between the 'pushed' prompts and the corresponding `push_prompt`'s frames on the real stack – precisely what we need for implementing $\mathsf{M}^i_{\mathsf{dc}}$.

From now on, the transcription from $\mathsf{M}^i_{\mathsf{dc}}$ to OCaml is straightforward. First we implement the `pushP` pe and `pushP` pv transitions of M_{dc} (inherited by $\mathsf{M}^i_{\mathsf{dc}}$):

```
let push_prompt_aux (p : 'a prompt) (body : unit -> 'a) : 'a =
  let pframe = {pfr_mark = p.mark; pfr_ek = get_ek ()} in
  let () = ptop := pframe :: (!ptop) in
  let res = body () in p.mbox := fun () -> res; raise DelimCCE

let push_prompt (p : 'a prompt) (body : unit -> 'a) : 'a =
  try push_prompt_aux p body with
  | DelimCCE -> (match !ptop with h::t ->
     assert (h.pfr_mark == p.mark); ptop := t; mbox_receive p)
  | e -> match !ptop with
          h::t -> assert(h.pfr_mark==p.mark); ptop:=t; raise e
```

The `try`-block establishes an exception frame, on the top of which we build the call frame for the evaluation of the `body` – or, of the wrapper `push_prompt_aux`.

That call frame will be at the very bottom of `ekfragment` when the continuation is captured. The wrapper pushes a new `pframe` onto `pstack`, which `push_prompt` removes upon normal or exceptional exit. The `assert` expresses the invariant: every exception frame created by `push_prompt` corresponds to a `pframe`. That `pframe` is on the top of `pstack` iff `push_prompt`'s exception frame is the latest exception frame. The `body` may finish normally, returning a value. It may also invoke `take_subcont` capturing and removing the part of the stack up to `push_prompt`, thus sending the value to `push_prompt` 'directly'. We use a mailbox for such communication, see §4.1. In fact, the above code is an elaboration of the code in §4.1, using `prompt`, `mbox_receive` defined in that section.

The code for `take_subcont` is too an elaboration of the code in §4.1; now it has to capture the continuation rather than simply disregarding it. In $\mathsf{M}^i_{\mathsf{dc}}$, we capture the continuation between two `pushP` frames, that is, between two exception frames. The captured continuation:

```
type ('a,'b) subcont =
  {subcont_ek : ekfragment; subcont_ps : pframe list; subcont_bs : ek;
   subcont_pa : 'a prompt;  subcont_pb : 'b prompt}
```

includes two mailboxes (to receive a value when the continuation is reinstated and to send the result), the copy of the OCaml stack `ekfragment`, and the corresponding copy of the parallel stack. The latter is a list of `pframe`s in reverse order. We note in `subcont_bs` the base of the `ekfragment`, the identity of the exception frame left on the stack after the `ekfragment` is removed. We need the base to adjust `pfr_ek` fields of `pframe`s when the continuation is reinstated.

The transition `takeSC` of $\mathsf{M}^i_{\mathsf{dc}}$ requires locating the latest frame `pushP` p with the given prompt p and splitting the stack at that point. This job is now done by `unwind`, which scans the `pstack` returning `h`, the `pframe` corresponding to a given prompt (identified by its `mark`).

```
let rec unwind acc mark = function
 | []   -> failwith "No prompt was set"
 | h::t as s ->
    if h.pfr_mark == mark then (h,s,acc) else unwind (h::acc) mark t
```

The function also splits `pstack` at `h`, returning the part up to but not including `h` as `acc`, in reverse frame order.

The function `take_subcont` straightforwardly implements the `takeSC` transition of $\mathsf{M}^i_{\mathsf{dc}}$, removing the fragments from the real and parallel stack, packaging them into a `subcont` structure. First, however, `take_subcont` must push the frame `pushP` p' with a fresh prompt p'. That prompt will never be referred to in any `take_subcont` function, see §3; therefore, we should not register the `pushP` p' frame in `pstack`. We use `push_prompt_simple` to push such an 'ephemeral' prompt, used only as a mailbox.

```
let push_prompt_simple (p: 'a prompt) (body: unit -> unit) : 'a =
  try body (); raise DelimCCE with DelimCCE -> mbox_receive p

let take_subcont (p: 'b prompt) (f: ('a,'b) subcont -> unit->'b) : 'a =
```

```
  let pa = new_prompt () in push_prompt_simple pa (fun () ->
    let (h,s,subcontchain) = unwind [] p.mark !ptop in
    let () = ptop := s in let ek = h.pfr_ek in let sk = get_ek () in
    let ekfrag = pop_stack_fragment ek sk in
    p.mbox := f {subcont_ek = ekfrag; subcont_pa = pa;
                 subcont_pb = p; subcont_ps = subcontchain;
                 subcont_bs = ek})
```

The function push_subcont is the transcription of $\mathsf{M}^i_{\mathsf{dc}}$'s transition pushSC.

```
let push_subcont (sk : ('a,'b) subcont) (m : unit -> 'a) : 'b =
  let pb = sk.subcont_pb in push_prompt_simple pb (fun () ->
    let base = sk.subcont_bs in let ek = get_ek () in
    List.iter (fun pf ->
     ptop := {pf with pfr_ek = add_ek ek (sub_ek pf.pfr_ek base)} :: !ptop)
         sk.subcont_ps;
    sk.subcont_pa.mbox := m; push_stack_fragment sk.subcont_ek)
```

When we push the ekfragment onto the stack, the identities of the exception frames therein may change. We have to 're-base' pfk_ek fields of pframes in the parallel stack fragment to restore the correspondence.

5 Related Work

The paper [9] that introduced multi-prompt delimited control presented its implementation in SML/NJ, relying on local exceptions and call/cc. Later the same authors offered an OCaml implementation [13], using "a very naive experimental brute-force version of callcc that copies the stack", along with Obj.magic, or unsafe coerce. Not only copying of the entire control stack to and from the heap on each use of control operators that is problematic. Since now delimited continuations capture (much more) of the stack than needed, the values referred from the unneeded part cannot be garbage-collected: The implementation has a memory leak. Furthermore, the correctness of the OCaml call/cc implementation [20] is not obvious as it copies the stack regardless of whether the byte-code interpreter is at a stable point or not. Perhaps for that reason the users of call/cc are warned that its "Use in production code is not advised"[20].

Multi-prompt delimited control was further developed and formalized in [10], who also presented indirect implementations in Scheme and Haskell. The Scheme implementation used call/cc, and the Haskell used the continuation monad along with unsafeCoerce.

A direct and efficient implementation of single-prompt delimited control (shift/reset) was first described in [21], specifically for Scheme48. The implementation relied on the hybrid stack/heap strategy for activation frames, particular to Scheme48 and a few other Scheme systems. The implementation required several modifications of the Scheme48 run-time. On many benchmarks, the paper [21] showed the impressive performance of the direct implementation of shift/reset compared to the call/cc emulation. The implementation, alas, has not been available as part of Scheme48. The paper specifically left to future work relating the implementation to the specification of shift/reset.

Recently there has been interest in direct implementations (as compared to the call/cc-based one [22] in SML/NJ) of the single prompt shift/reset in the typed setting [23, 24]. Supporting delimited control required modifying the compiler or the run-time, or both.

Many efficient implementations of undelimited continuations have been described in Scheme literature, e.g. [12]. Clinger et al. [25] is a comprehensive survey. Their lessons hold for delimited control as well.

Sekiguchi et al. [26] use exceptions to implement multi-prompt delimited control in Java and C++. Their method relies on source- or byte-code translation, changing method signatures and preventing mixing the translated code with untranslated libraries. The run-time overhead is especially notable for the control-operator–free portions of the code. A similar, more explicit transformation technique for source Scheme programs is described in [27], with proofs of correctness. The approach, alas, targets undelimited continuations, which brings unnecessary complications. The translation is untyped, deals only with a subset of Scheme and too has difficulties interfacing third-party libraries.

6 Conclusions

We have presented abstract and concrete implementations of multi-prompt delimited control. The concrete implementation is the delimcc OCaml library, which has been fruitfully used for over four years. The abstract implementation has related delimited control to exception handling and distilled scAPI, a minimalistic API, sufficient for the implementation of delimited control. A language system accommodating exception handling and stack-overflow recovery is likely to support scAPI. The OCaml byte-code does support scAPI, and thus permits, *as it is*, the implementation of delimited control. We described the implementation of delimcc as an example of using scAPI in a typed language.

OCaml exceptions and delimited control integrate and benefit each other. OCaml exception frames naturally implement stable points of scAPI. Exception handlers may be captured in delimited continuations, and re-instated along with the captured continuation; exceptions remove the prompts. Conversely, delimcc effectively provides local exception declarations, hitherto missing in OCaml.

In the future, we would like to incorporate the lessons learned in efficient implementations of undelimited continuations, in particular, stack segmentation of [12]. Determining if the native-code OCaml compiler can support scAPI efficiently requires further investigation.[5] We also want to apply the scAPI-based approach to implementing delimited control in other language systems. The formal part of the paper can be extended further by adding state and stack-copying primitives to $\mathsf{M}'_{\mathsf{ex}}$ and relating the result to $\mathsf{M}^{i}_{\mathsf{dc}}$.

[5] The main difficulty is the natively compiled code's using the C stack, which may contain unboxed values. The naive copying of such stack fragments to and from the heap requires many movements and GC root registrations.

Acknowledgements I thank Paul Snively for inspiration and encouragement. I am immensely grateful to Chung-chieh Shan for numerous helpful discussions and advice that improved the content and the presentation. Many helpful suggestions by anonymous reviewers and Kenichi Asai are greatly appreciated.

Bibliography

[1] Kiselyov, O.: Native delimited continuations in (byte-code) OCaml. `http://okmij.org/ftp/Computation/Continuations.html#caml-shift` (2006)

[2] Kiselyov, O., Shan, C.c., Sabry, A.: Delimited dynamic binding. In: ICFP. (2006) 26–37

[3] Kiselyov, O., Shan, C.c.: Embedded probabilistic programming. In: Proc. IFIP Working Conf. on DSL. Volume 5658 of LNCS., Springer (2009) 360–384

[4] Kiselyov, O., Shan, C.c.: Monolingual probabilistic programming using generalized coroutines. In: Uncertainty in Artificial Intelligence. (2009)

[5] Kiselyov, O.: Persistent delimited continuations for CGI programming with nested transactions. Continuation Fest 2008. `http://okmij.org/ftp/Computation/Continuations.html#shift-cgi` (2008)

[6] Kameyama, Y., Kiselyov, O., Shan, C.c.: Shifting the stage: Staging with delimited control. In: PEPM. (2009) 111–120

[7] Kiselyov, O., Shan, C.c.: Lifted inference: Normalizing loops by evaluation. In: Proc. 2009 Workshop on Normalization by Evaluation, BRICS (2009)

[8] Kiselyov, O.: Ask-by-need: On-demand evaluation with effects. `http://okmij.org/ftp/Computation/Continuations.html#ask-by-need` (2007)

[9] Gunter, C.A., Rémy, D., Riecke, J.G.: A generalization of exceptions and control in ML-like languages. In: Functional Programming Languages and Computer Architecture. (1995) 12–23

[10] Dybvig, R.K., Peyton Jones, S.L., Sabry, A.: A monadic framework for delimited continuations. J. Functional Progr. **17** (2007) 687–730

[11] Balat, V., Di Cosmo, R., Fiore, M.P.: Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In: POPL'04. (2004) 64–76

[12] Hieb, R., Dybvig, R.K., Bruggeman, C.: Representing control in the presence of first-class continuations. In: PLDI'90. (1990) 66–77

[13] Gunter, C.A., Rémy, D., Riecke, J.G.: Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/` (1998)

[14] Kiselyov, O.: Zipper in Scheme. Usenet posting to the `comp.lang.scheme` newsgroup; `http://okmij.org/ftp/Scheme/zipper-in-scheme.txt` (2004)

[15] Kiselyov, O.: Zipper as a delimited continuation. Message to the Haskell mailing list; `http://okmij.org/ftp/Haskell/Zipper1.lhs` (2005)

[16] Felleisen, M.: The theory and practice of first-class prompts. In: POPL. (1988) 180–190

[17] Leroy, X.: The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA (1990)

[18] Glew, N.: Type dispatch for named hierarchical types. In: ICFP. (1999) 172–182

[19] Leroy, X.: The bytecode interpreter. version 1.96. `byterun/interp.c` in OCaml distribution (2006)

[20] Leroy, X.: Ocaml-callcc: call/cc for ocaml. `http://pauillac.inria.fr/~xleroy/software.html#callcc` (2005)

[21] Gasbichler, M., Sperber, M.: Final shift for call/cc: Direct implementation of shift and reset. In: ICFP. (2002) 271–282

[22] Filinski, A.: Representing monads. In: POPL. (1994) 446–457

[23] Masuko, M., Asai, K.: Direct implementation of shift and reset in the MinCaml compiler. In: ACM SIGPLAN Workshop on ML. (2009)

[24] Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In: ICFP. (2009) 317–328

[25] Clinger, W.D., Hartheimer, A.H., Ost, E.M.: Implementation strategies for first-class continuations. Higher-Order and Symbolic Computation **12** (1999) 7–45

[26] Sekiguchi, T., Sakamoto, T., Yonezawa, A.: Portable implementation of continuation operators in imperative languages by exception handling. In: Advances in Exception Handling Techniques. Volume 2022 of LNCS., Springer (2001) 217–233

[27] Pettyjohn, G., Clements, J., Marshall, J., Krishnamurthi, S., Felleisen, M.: Continuations from generalized stack inspection. In: ICFP. (2005) 216–227

[28] Biernacki, D., Danvy, O., Millikin, K.: A dynamic continuation-passing style for dynamic delimited continuations. Report RS-05-16, BRICS, Denmark (2005)

A Deriving M_{dc} from the Definitional Machine

In this section we recall the definitional machine for multi-prompt delimited control and prove its equivalence to the machine in Figure 1. The proof is standard and patterned after [28].

$$
\begin{array}{ll}
\text{Variables} & x, y, \ldots \qquad \text{Prompts} \quad p, q \in N \\
\text{Expressions} & e \ ::= \ v \mid e\,e \mid \texttt{newP} \mid \texttt{pushP}\,e\,e \mid \texttt{takeSC}\,e\,e \mid \texttt{pushSC}\,e\,e \\
\text{Values} & v \ ::= \ x \mid \lambda x.\,e \mid p \mid E \\
\text{Contexts} & D \ ::= \ \Box \mid D e \mid v D \mid \texttt{pushP}\,D e \mid \texttt{pushSC}\,D e \mid \texttt{takeSC}\,D e \mid \texttt{takeSC}\,p D \\
\text{Sequences} & E \ ::= \ [] \mid p : E \mid D : E
\end{array}
$$

Transitions between configurations (e, D, E, q)

$$
\begin{array}{rcll}
(ee', D, E, q) & \mapsto & (e, D[\Box e'], E, q) & e \text{ non-value} \\
(ve, D, E, q) & \mapsto & (e, D[v\Box], E, q) & e \text{ non-value} \\
(\texttt{pushP}\,ee', D, E, q) & \mapsto & (e, D[\texttt{pushP}\,\Box e'], E, q) & e \text{ non-value} \\
(\texttt{takeSC}\,ee', D, E, q) & \mapsto & (e, D[\texttt{takeSC}\,\Box e'], E, q) & e \text{ non-value} \\
(\texttt{takeSC}\,pe, D, E, q) & \mapsto & (e, D[\texttt{takeSC}\,p\Box], E, q) & e \text{ non-value} \\
(\texttt{pushSC}\,ee', D, E, q) & \mapsto & (e, D[\texttt{pushSC}\,\Box e'], E, q) & e \text{ non-value} \\
\\
((\lambda x.\,e)v, D, E, q) & \mapsto & (e[v/x], D, E, q) & \\
(\texttt{newP}, D, E, q) & \mapsto & (q, D, E, q+1) & \\
(\texttt{pushP}\,pe, D, E, q) & \mapsto & (e, \Box, p : D : E, q) & \\
(\texttt{takeSC}\,pv, D, E, q) & \mapsto & (v(D : E_1), \Box, E_2, q) & E_1 +\!\!+ (p : E_2) = E,\ p \notin E_1 \\
(\texttt{pushSC}\,E'e, D, E, q) & \mapsto & (e, \Box, E' +\!\!+ (D : E), q) & \\
\\
(v, D, E, q) & \mapsto & (D[v], \Box, E, q) & D \neq \Box \\
(v, \Box, p : E, q) & \mapsto & (v, \Box, E, q) & \\
(v, \Box, D : E, q) & \mapsto & (v, D, E, q) &
\end{array}
$$

Fig. 3. Definitional machine $\mathsf{M}_{\mathsf{defn}}$ for multi-prompt delimited control from [10, Figure 1] (adjusted for style). Prompts p and sequences E may not appear in source programs.

Compared to M_{dc} in Figure 1, the definitional machine has an extra component, the sequence E, whose elements are contexts and prompts. We write $u : E$ for a sequence whose first element is u and the rest is E; we write $E_1 +\!\!+ E_2$ for the concatenation of two sequences. The rest of the notation is explained in §3. The machine starts in the configuration $(e, \Box, [], 0)$ and stops when it reaches $(v, \Box, [], q)$.

To prove the equivalence of the definitional machine with M_{dc}, we first relate configurations of the two machines. To distinguish the definitional machine, we place the diacritic mark $\widehat{\cdot}$ over all components of its configuration. We define the family of relations $\sim$ as the least relational family satisfying the following:

Relating configurations $\sim_c$

$(\widehat{e}, \widehat{D}, \widehat{E}, \widehat{q}) \sim_c (e, D, q)$ iff $\widehat{e} \sim_e e$, $(\widehat{D}, \widehat{E}) \sim_d D$, $\widehat{q} = q$

Relating expressions: $\widehat{e} \sim_e e$ iff $\widehat{e} = e$ except for

$\widehat{E} \sim_e D$ iff $(\widehat{\Box}, \widehat{E}) \sim_d D$

Relating contexts:

$(\widehat{\Box}, \widehat{[]}) \sim_d \Box$

$(\widehat{D[\Box e]}, \widehat{E}) \sim_d D[\Box e]$ iff $\widehat{e} \sim_e e$, $(\widehat{D}, \widehat{E}) \sim_d D$

$(\widehat{D[v\Box]}, \widehat{E}) \sim_d D[v\Box]$ iff $\widehat{v} \sim_e v$, $(\widehat{D}, \widehat{E}) \sim_d D$

$(D[\widehat{\texttt{pushP}\,\Box e}], \widehat{E}) \sim_d D[\texttt{pushP}\,\Box e]$ iff $\widehat{e} \sim_e e$, $(\widehat{D}, \widehat{E}) \sim_d D$

and similarly for `pushSC`, `takeSC`

$(\widehat{\Box}, \widehat{p : E}) \sim_d D[\texttt{pushP}\, p\Box]$ iff $(\widehat{\Box}, \widehat{E}) \sim_d D$

$(\widehat{\Box}, \widehat{D : E}) \sim_d D$ iff $(\widehat{D}, \widehat{E}) \sim_d D$

Lemma 1. *If $(\widehat{D}, \widehat{E}) \sim_d D$ then there exist D_1 and D_2 such that $D = D_2[D_1]$ and $(\widehat{D}, \widehat{[]}) \sim_d D_1$ and $(\widehat{\Box}, \widehat{E}) \sim_d D_2$. Conversely, if $(\widehat{D}, \widehat{[]}) \sim_d D_1$ and $(\widehat{\Box}, \widehat{E}) \sim_d D_2$ then $(\widehat{D}, \widehat{E}) \sim_d D_2[D_1]$.*

The proof is by induction on the structure of $\widehat{D}$.

Lemma 2. *If $(\widehat{\Box}, \widehat{E_1}) \sim_d D_1$ and $(\widehat{\Box}, \widehat{E_2}) \sim_d D_2$ then $(\widehat{\Box}, \widehat{E_1 +\!\!+ E_2}) \sim_d D_2[D_1]$.*

Lemma 3. *If $(\widehat{\Box}, \widehat{E}) \sim_d D$ and $E = E_1 +\!\!+ E_2$ then there exist D_1 and D_2 such that $D = D_2[D_1]$ and $(\widehat{\Box}, \widehat{E_1}) \sim_d D_1$ and $(\widehat{\Box}, \widehat{E_2}) \sim_d D_2$. Conversely, if $(\widehat{\Box}, \widehat{E}) \sim_d D$ and $D = D_2[D_1]$ then there exist E_1 and E_2 such that $E = E_1 +\!\!+ E_2$ and $(\widehat{\Box}, \widehat{E_1}) \sim_d D_1$ and $(\widehat{\Box}, \widehat{E_2}) \sim_d D_2$.*

The proof is by induction on the length of E_1, using Lemma 1.

Lemma 4. *If $(\widehat{D}, \widehat{[]}) \sim_d D$ and $\widehat{e} \sim_e e$ then $\widehat{D[e]} \sim_e D[e]$.*

The proof is by structural induction on $\widehat{D}$.

As usual, we write $\mapsto^+$ for the transitive closure of the transition relation, and $\mapsto^*$ for the transitive reflexive closure.

Proposition 1 (Equivalence). *For all $\widehat{e}$ and e such that $\widehat{e} \sim_e e$, $(\widehat{e}, \widehat{\Box}, \widehat{[]}, \widehat{0}) \mapsto^* (\widehat{v}, \widehat{\Box}, \widehat{[]}, \widehat{q})$ for some $\widehat{v}$ iff $(e, \Box, 0) \mapsto^* (v, \Box, q)$ for some v such that $\widehat{v} \sim_e v$.*

The proof depends on the following lemma:

Lemma 5. *Let $\widehat{C}$ be the configuration of $\mathsf{M}_{\mathit{defn}}$ and let C be the related configuration of M_{dc}. Then:*

1. *If $\widehat{C} \mapsto \widehat{C'}$ for some $\widehat{C'}$, then $C \mapsto^* C'$ for some C' and $\widehat{C'} \sim_c C'$.*

2. *If* $C \mapsto C'$ *for some* C' *then there exists* C'' *and* $\widehat{C'}$ *such that* $C' \mapsto^* C''$, $\widehat{C} \mapsto^+ \widehat{C'}$, *and* $\widehat{C'} \sim_c C''$
3. *If* C *is a terminal configuration, then there exists terminal* $\widehat{C'}$ *such that* $\widehat{C} \mapsto^* \widehat{C'}$ *and* $\widehat{C'} \sim_c C$

Only the cases where $\widehat{C}$ includes $\widehat{\mathtt{pushP}\,pe}$, $\widehat{\mathtt{takeSC}\,pv}$, $\widehat{\mathtt{pushSC}\,E'e}$, and $\widehat{v}$ are interesting. In the other configurations, the machines clearly 'move in lockstep'. The machines turn out to move in lockstep for $\widehat{C}$ including $\widehat{\mathtt{pushP}\,pe}$, $\widehat{\mathtt{takeSC}\,pv}$ (seen from Lemma 3) and $\widehat{\mathtt{pushSC}\,E'e}$ (proved using Lemma 2). If C is a terminal configuration $(v, \square, q)$, the related $\widehat{C}$ may be a non-terminal configuration $(\widehat{v}, \widehat{\square}, \widehat{E}, \widehat{q})$ where $\widehat{E}$ is the list made entirely of $\square$. In the number of steps equal to the length of the list, the machine reaches the terminal configuration that is related to C. A non-terminal configuration $C = (v, D, q)$ may be related to one of $(\widehat{v}, \widehat{D}, \widehat{E}, \widehat{q})$ with $\widehat{D} \neq \widehat{\square}$, $(\widehat{v}, \widehat{\square}, \widehat{p : E}, \widehat{q})$, or $(\widehat{v}, \widehat{\square}, \widehat{D : E}, \widehat{q})$. In the first case, we use Lemmas 1,4. In the second case, $C' = (\mathtt{pushP}\,pv, D', q)$ and $C'' = (v, D', q)$ where $D = D'[\mathtt{pushP}\,p\square]$. In the third case, we apply the last rule in Figure 3, may be more than once if $\widehat{D}$ is empty.

B Plugging a Memory Leak

Experience with delimcc called for the addition of push_delim_subcont to its interface. The new function can in principle be written in terms of the existing ones:

```
let push_delim_subcont (sk : ('a,'b) subcont) (m : unit -> 'a) : 'b =
      push_prompt sk.subcont_pb (fun () -> push_subcont sk v)
```

However, that implementation has a memory leak, which we demonstrate. The function push_delim_subcont expresses a common pattern, already seen in shift0 of §2, of pushing a *delimited* continuation. The same pattern occurs in implementations of user-level threads or co-routines, where the memory leak becomes the problem, as was kindly pointed out by Christophe Deleuze; the following is a simplified version of his code.

```
type state = Done | Pause of (unit, state) subcont
let p = new_prompt ()

let pause () = take_subcont p (fun sk () -> Pause sk)
let proc () = while true do pause () done; Done
let rec sched_loop = function | Done -> ()
   | Pause sk ->
      sched_loop (
        push_prompt p (fun () -> push_subcont sk (fun () -> ())))
```

Our example has only one, continually running thread proc, which pauses on each iteration. The scheduler keeps resuming the thread. Since take_subcont removes the scheduler's prompt p, the scheduler has to push it again – hence the pattern expressed in `push_delim_subcont`. Informally, the scheduler has to re-establish the thread-kernel boundary. Evaluating `sched_loop (push_prompt p proc)` several thousand times leads to abnormal termination after the program exhausts all available memory.

To see the problem clearly we use the abstract machine $\mathsf{M}^{i}_{\mathsf{dc}}$, to which we add a new expression $\mathtt{loop}\, e_1 e_2$, a new frame type $\mathtt{loop}\, e_1 \square$ and the corresponding transitions:

$$(\mathtt{loop}\, e' e, D, q) \mapsto (e, D[\mathtt{loop}\, e' \square], q) \qquad e \text{ non-value}$$
$$(\mathtt{loop}\, e' v, D, q) \mapsto (\mathtt{loop}\, e' e', D, q)$$

Let e_b be $\mathtt{takeSC}\, p\, (\lambda x.\, x)$. Tracing transitions in $\mathsf{M}^{i}_{\mathsf{dc}}$ shows $\mathtt{pushP}\, p\, (\mathtt{loop}\, e_b e_b)$ evaluating to $\mathtt{loop}\, e_b\, (\mathtt{pushP}\, p' \square)$, to be called D_1. The prompt p' is fresh. The value D_1 corresponds to the result of `pause ()`. To resume the thread, we evaluate $\mathtt{pushP}\, p\, (\mathtt{pushSC}\, D_1\, ())$, which reduces to $\mathtt{pushP}\, p\, (\mathtt{pushP}\, p''\, (\mathtt{loop}\, e_b e_b))$. Here, p'' is the fresh prompt introduced by the $\mathtt{pushSC}$ transition of $\mathsf{M}^{i}_{\mathsf{dc}}$. The result is the value $\mathtt{pushP}\, p''\, (\mathtt{loop}\, e_b\, (\mathtt{pushP}\, p' \square))$, called D_2, which is longer than D_1 by an extra frame $\mathtt{pushP}\, p''$. Resuming D_2, gives D_3 that is longer still. The memory leak becomes apparent.

The solution is to implement `push_delim_subcont` as a new library primitive, taking the code at the beginning of the section as the specification. We transform the code by inlining `push_subcont` and collapsing the two adjacent `pushP` frames: when there is already $\mathtt{pushP}\, p$ at the top of the stack, the $\mathtt{pushSC}$ transition of $\mathsf{M}^{i}_{\mathsf{dc}}$ no longer needs to push the $\mathtt{pushP}\, p''$ frame.