Delimited Control in OCaml, Abstractly and Concretely

Oleg Kiselyov

Monterey, CA, U.S.A.

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 and binary code. The library has been in fruitful practical use since 2006.

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. With byte- and native-code OCaml systems as two examples, our library illustrates how to use scAPI to implement multi-prompt delimited control in a typed language. The approach is general and has been used to add multi-prompt delimited control to other existing language systems.

Keywords: delimited continuation, exception, semantics, implementation, abstract machine

1. Introduction

The library delimcc of delimited control for OCaml was first released at the beginning of 2006 [1] and has been used for implementing (delimited)

Email address: `oleg@okmij.org` (Oleg Kiselyov)
URL: `http://okmij.org/ftp/` (Oleg Kiselyov)

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]. Other people have used the library for implementing coroutines [8] and 'fibers', and as the base for direct-style functional reactive programming [9].

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 may be reinstated arbitrarily many times in different dynamic contexts. Captured delimited continuations can be serialized, stored, or migrated, then reinstated in a different process, perhaps several times.

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. Except for the common, sole prohibition on capturing continuations across an OCaml callback invoked from a foreign C function, delimcc imposes no restrictions on the user code. Our 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 exposing enough run-time–system details. Our goal in this paper[1] 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 [11] and further developed by Dybvig, Peyton Jones, and Sabry [12]. The multi-prompt operators turn out indispensable for normalization-by-evaluation for strong sums [13]. Further applications of specifically multi-prompt operators

[1]This present paper is an extended version of the conference paper [10]. We completely re-wrote §2 with a new example and detailed explanations. We have added benchmarks §7, proofs (Appendix A and Appendix B), and new §5, §6, §8.

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 account for its correctness and generality. 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 [14].

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 the 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 most recent stable point frame and safely 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 delimcc, we demonstrate in §4 using the scAPI to implement multi-prompt delimited control on two distinct OCaml language systems.[2] OCaml byte-code happens to support scAPI, §4.2, and so does the native-code OCaml system, §6. The implementations of scAPI are the *only* difference between byte- and native-code delimcc.
3. The implementation of delimcc poses challenging typing problems, which

[2]The Scheme and Haskell implementations, mentioned on the delimcc web page, are further concrete examples of using the scAPI, attesting to the generality of the approach.

previously [12, 15] 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 [12], to avoid a memory leak in multi-prompt shift, §5. The new primitive `push_delim_subcont` reinstates the captured continuation along with its delimiter. (The library implements yet another derived function, `abort`, as primitive, §7, to avoid useless continuation capture.)
5. We describe serialization of captured delimited continuations so to make them persistent: §8. 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.

We discuss two small benchmarks in §7; see [4] for a more detailed discussion of a realistic application that uses delimcc library. For that application at least, the performance of delimcc proved adequate. We review the related work in §9 and then conclude. We start by introducing the multi-prompt delimited control and the delimcc library in §2.

The delimcc library source along with validation tests, benchmarks and sample code is freely available from `http://okmij.org/ftp/continuations/`.

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 [12], 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
```

Their semantics is formally discussed in §3. The reader already familiar with delimited control may view delimcc as a generalization of the ordinary

shift/reset [16] to control delimiters of arbitrarily many 'flavors'. The function `new_prompt` creates a control delimiter – or prompt – of a new, unique flavor. The expression `push_prompt p (fun () -> e)`, the generalization of `reset e`, puts the control delimiter `p` on the stack and then evaluates `e`; `take_subcont p f` removes the prefix of the stack up to the closest stack frame marked with the given `p`. The removed portion of the stack, with the terminating delimiter `p` cut off, is packaged as a continuation object of the abstract type `subcont` and passed to `take_subcont`'s argument `f`. The function `push_subcont` puts the removed stack frames back on the stack, possibly in a different context, thus reinstating the captured delimited continuation.

The delimcc library may also be understood as generalizing exceptions, a wide-spread and familiar feature. 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.

We illustrate the generalization of exceptions by elaborating the example of modifying a search tree:

```
type ('k, 'v) tree =
  | Empty
  | Node of ('k, 'v) tree * 'k * 'v * ('k, 'v) tree
```

It is the standard implementation of a finite map associating keys of the type `'k` with values of the type `'v`. A tree node contains the key, the corresponding value, the left branch with the smaller keys and the right branch with the larger keys. The modification example is standard too: update the value associated with the given key, returning a new tree. The new value is determined by applying the given function to the old value. The only interesting part of the code is the case of the input tree not containing the given key. Our first example throws the ordinary OCaml exception then:

```
exception NotFound
let rec update1 : 'k -> ('v->'v) -> ('k,'v) tree -> ('k,'v) tree =
  fun k f ->
```

```
let rec loop = function
| Empty -> raise NotFound
| Node (l,k1,v1,r) ->
    begin
      match compare k k1 with
      | 0 -> Node(l,k1,f v1,r)
      | n when n < 0 -> Node(loop l,k1,v1,r)
      | _            -> Node(l,k1,v1,loop r)
    end
in loop
```

We will describe several versions of this function; they only differ in the type signature and in the code for the empty input tree case. The following sample application increments the value associated with the key `1` in `tree1`, associating the key with the value `100` if it was missing.

```
try update1 1 succ tree1
with NotFound -> insert 1 100 tree1
```

We re-write the example using `delimcc` to raise the 'exception' (we shall elide the code that is common with `update1`):

```
let rec update2 : ('k,'v) tree option prompt ->
  'k -> ('v->'v) -> ('k,'v) tree -> ('k,'v) tree =
  fun pnf k f ->
    let rec loop = function
    | Empty -> take_subcont pnf (fun _ () -> None)
   ...
```

The sample application takes the following form.

```
let pnf = new_prompt () in
match push_prompt pnf (fun () -> Some (update2 pnf 1 succ tree1)) with
  | Some tree -> tree
  | None      -> insert 1 100 tree1
```

`push_prompt` acts as `try`, catching the exception raised by `take_subcont`, extracting the thunk `fun () -> None` and evaluating it. Apart from the syntactic sugar, the two examples differ in the manner of creating the exception object: whereas `NotFound` is created at compile-time, `pnf` is produced dynamically, and then passed as the first argument to `update2`. The difference

is superficial since ordinary exception objects can also be created dynamically (as so-called 'local exceptions', provided in SML and fully supported by OCaml since version 3.12).

The two `update` examples are inefficient: first, `update` has to navigate down the tree to the point where it expects to find the key `1`, throwing the exception if the key is not found. Then the function `insert` (not shown: it is standard and quite like `update`) again has to navigate to exactly the same spot in the tree, this time creating a new node. Restartable exceptions like those in Common Lisp offer an elegant solution, letting the exception handler take a corrective action and resume the execution from the point where it was interrupted by the exception. Raising of an exception may now return, acting as a regular function application. Restartable exceptions are therefore easy to implement, in principle:

```
let rec update3 : 'k -> ('v->'v) -> ('k,'v) tree -> ('k,'v) tree =
  fun k f ->
    let rec loop = function
    | Empty -> Node(Empty,k,upd_handle k,Empty)
    ...
```

We raise a restartable exception by invoking a global function `upd_handle`, passing it the missing key. The function may throw a real exception or yield the value to put into the updated tree; `update3` will then return normally.

This simplistic, Common-Lisp–like solution is quite problematic. First of all, each caller of `update3` should be able to decide on the value to associate with the missing key. Therefore, the restartable exception handler, as regular exception handlers, should be bound dynamically rather than globally. However, implementing dynamic binding in the presence of exceptions is notably tricky, see [2] for the survey of problems. The main drawback is the exception restart's happening implicitly, upon the return from `upd_handle`. Therefore, `upd_handle` cannot, for example, restart the same exception several times, to try several alternatives of exception recovery. Multiple restarts are useful for implementing non-determinism and probabilistic programming [3]. Shortly we will see another advantage of explicit exception restarts.

The library delimcc implements restartable exceptions with multiple, explicit restarts. The value of the type `subcont` is the restart object, created by `take_subcont` as it raises an exception. Passing the restart object to the function `push_subcont` resumes the interrupted computation. The rewritten `update2` below not only throws the exception when the key is not

found; `update4` also collects the data needed for recovery – the exception object `c` and the missing key – and packs them into the envelope `ReqNF`:

```
type ('k,'v) res = Done of ('k,'v) tree
  | ReqNF of 'k * ('v,('k,'v) res) subcont

let rec update4 : ('k,'v) res prompt ->
  'k -> ('v->'v) -> ('k,'v) tree -> ('k,'v) tree =
  fun pnf k f ->
    let rec loop = function
    | Empty -> Node(Empty,k,
                take_subcont pnf (fun c () -> ReqNF (k,c)),Empty)
    ...
```

The caller of `update4` will receive the envelope from the exception and decide if and how to proceed. The sample application

```
let pnf = new_prompt () in
match push_prompt pnf (fun () -> Done (update4 pnf 1 succ tree1)) with
  | Done tree    -> tree
  | ReqNF (k,c)  ->
      match push_subcont c (fun () -> 100) with Done x -> x
```

extracts the restart object from the envelope and uses it to resume the exception. The function call `push_subcont c (fun () -> 100)` resumes the evaluation of `update4` as if the expression `take_subcont pnf (...)` returned `100`. We have started with the expression `Done (update4 pnf 1 succ tree1)`, whose evaluation was interrupted by the exception; `push_prompt` has caught the exception, yielding `ReqNF (k,c)` rather than the value `Done tree` expected as the result of our expression. The restarted expression does not raise any further exceptions, finishing normally, with the result `Done tree`. The result becomes the value yielded by `push_subcont`. (The last `Done x` pattern-match in the sample application is therefore total.)

Our sample applications that relied on restartable exceptions had a subtle flaw. Upon the exception restart a new node is added to the tree, changing the height of its branch and potentially requiring rebalancing. We should have written

```
let pnf = new_prompt () in
rebalance (match push_prompt pnf ...)
```

which is not optimal however: if the key was found no rebalancing is needed since the resulting tree has the same structure as the input tree. We may need to rebalance the tree only after the key lookup failure and the addition of a new node. The optimal solution is to proceed upon the assumption of no rebalancing; if we eventually discover that the key was missing and a new node has to be adjoined, we go 'back in time' and add the call to `rebalance` at the beginning. This scenario, however far-fetched it may seem, is implementable:

```
let pnf = new_prompt () in
match push_prompt pnf (fun () -> Done (update4 pnf 1 succ tree1)) with
  | Done tree    -> tree
  | ReqNF (k,c)  ->
     rebalance (match push_subcont c (fun () -> 100) with Done x -> x)
```

The benefit of explicit restarts is the ability to restart the interrupted computation in a different context, in our case, in the context of the extra function call, to `rebalance`. One can easily imagine examples where the restarted computation may throw other exceptions, and we would use `try` or `push_prompt` in place of `rebalance` to handle them.

The function that computes the modified value may also throw (restartable) exceptions. For example, instead of `succ`, we could pass to `update` the following function:

```
exception TooBig
let upd_fun n = if n > 5 then raise TooBig else succ n
```

adjusting our sample application to catch `TooBig`

```
try
  let pnf = new_prompt () in
  match push_prompt pnf (fun () ->
     Done (update4 pnf 7 upd_fun tree1)) with ...
with TooBig -> Empty
```

The `TooBig` exception will be raised in the dynamic context of the restartable exception handling established by `push_prompt`. However, `TooBig` is of a different 'flavor' from `pnf` and so the two exceptions (as well as two restartable exceptions that use different prompts) act unaware of each other.

The formal, small-step semantics of these delimited control operators was specified in [11] (`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 [12, 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.' On the other hand, the operation of popping a single activation frame off the stack (which corresponds to a function return in typical concrete machines) has no equivalent in the definitional machine. Mainly, the machine contains an extra component, a list of contexts. It is not immediately clear what it may correspond to in concrete machines, making it harder for the implementer to see how to map a concrete machine such as OCaml byte-code to the definitional machine. Perhaps such a mapping is not possible without extending the OCaml interpreter.

These worries are unfounded. The machine of [12] can be converted into the equivalent machine described below, which has no extra components such as lists of control stacks and is hence more familiar. 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 the control stack. The machine can be viewed as a generalization of the environment-less version of the machine of [17].

The program for the machine is call-by-value λ-calculus, augmented with integral-valued prompts and delimited control operators. The operators here

$$
\begin{array}{llcl}
\text{Variables} & x, y, \ldots & \text{Prompts} & p, q \in N \\
\text{Expressions} & e & ::= & v \mid e\,e \mid \texttt{newP} \mid \texttt{pushP}\,e\,e \mid \texttt{takeSC}\,e\,e \mid \texttt{pushSC}\,e\,e \\
\text{Values} & v & ::= & x \mid \lambda x.\,e \mid p \mid D \\
\text{Contexts} & D & ::= & \Box \mid De \mid vD \mid \texttt{pushP}\,D\,e \mid \texttt{pushSC}\,D\,e \mid \texttt{takeSC}\,D\,e \\
 & & & \mid \texttt{takeSC}\,p\,D \mid \texttt{pushP}\,p\,D \\
\text{Single Frame} & & ::= & \Box e \mid v\Box \mid \texttt{pushP}\,\Box\,e \mid \texttt{pushSC}\,\Box\,e \mid \texttt{takeSC}\,\Box\,e \\
 & & & \mid \texttt{takeSC}\,p\,\Box \mid \texttt{pushP}\,p\,\Box
\end{array}
$$

Transitions between configurations (e, D, q)

$$
\begin{array}{rcll}
(ee', D, q) & \mapsto & (e, D[\Box e'], q) & e \text{ non-value} \\
(ve, D, q) & \mapsto & (e, D[v\Box], q) & e \text{ non-value} \\
(\texttt{pushP}\,ee', D, q) & \mapsto & (e, D[\texttt{pushP}\,\Box e'], q) & e \text{ non-value} \\
(\texttt{takeSC}\,ee', D, q) & \mapsto & (e, D[\texttt{takeSC}\,\Box e'], q) & e \text{ non-value} \\
(\texttt{takeSC}\,pe, D, q) & \mapsto & (e, D[\texttt{takeSC}\,p\Box], q) & e \text{ non-value} \\
(\texttt{pushSC}\,ee', D, q) & \mapsto & (e, D[\texttt{pushSC}\,\Box e'], q) & e \text{ non-value} \\[6pt]
((\lambda x.\,e)v, D, q) & \mapsto & (e[v/x], D, q) & \\
(\texttt{newP}, D, q) & \mapsto & (q, D, q+1) & \\
(\texttt{pushP}\,pe, D, q) & \mapsto & (e, D[\texttt{pushP}\,p\Box], q) & \\
(\texttt{takeSC}\,pv, D, q) & \mapsto & (vD_1, D_2, q) & D_2[\texttt{pushP}\,pD_1] = D,\ \texttt{pushP}\,pD' \notin D_1 \\
(\texttt{pushSC}\,D'e, D, q) & \mapsto & (e, D[D'], q) & \\[6pt]
(v, D[D_1], q) & \mapsto & (D_1[v], D, q) & D_1 \text{ single frame} \\
(\texttt{pushP}\,pv, D, q) & \mapsto & (v, D, q) &
\end{array}
$$

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

are syntactic forms rather than constants: for example, `newP` evaluates each time to 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 $\Box$ in the 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 $(e, \square, 0)$; the machine terminates when it reaches $(v, \square, q)$.

On one hand, the machine is a standard stack machine: 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 corresponds to the return from a function call, popping a single frame off the top of the stack and passing the return value to it.

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 $\texttt{pushP}\, p D_1$ 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 for storing composite values. These non-standard stack operations (called in §4.2 as `push_stack_fragment` for pushing several frames, `get_ek` and `reset_ek` for locating a frame and splitting the stack, and `copy_stack_fragment` for copying the stack prefix) 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 available feature. As a specification of exception handling we take an abstract machine M_{ex}, Figure 2. The program for M_{ex} is also 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.

The comparison of Figures 1 and 2 shows many similarities. For example, we observe that the expression $\texttt{pushP}\, p\, v$ reduces to v in any evaluation context; likewise, $\texttt{try}_p\, v\, e'$ 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 q serving as prompts and the conditional $\texttt{if}\ (q_1, q_2)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, which branches on equality of two integer prompts q_1 and q_2. These prompts cannot appear in source programs but are generated by an operator `newQ`, evaluating each time to a fresh value.

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

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

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

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

Single Frame $::= \Box e \mid v\Box \mid \texttt{raise}_p\, \Box \mid \texttt{try}_p\, \Box\, e$

Transitions between configurations (e, D)

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

Figure 2: Abstract machine M_{ex} for exception handling

We add unit $()$, pairs (v, v), and pair projections functions `fst` and `snd`. 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 $\texttt{takeSC}\, e\, (\lambda x.\, e')$ where x is not free in e'. The latter restriction assures that contexts D are not substituted into terms; since D cannot appear in source terms by definition, contexts D do not appear in $\mathsf{M}'_{\mathsf{dc}}$ terms at all. Hence we drop D from the syntax of $\mathsf{M}'_{\mathsf{dc}}$ terms and values. The complete definitions for $\mathsf{M}'_{\mathsf{dc}}$ and $\mathsf{M}'_{\mathsf{ex}}$ are given in Appendix B.

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):

$$
\begin{array}{lcl}
\lfloor \texttt{takeSC}\, p\, (\lambda x.\, e) \rfloor & = & \texttt{raise}_{p_0} (\lambda x.\, \lfloor e \rfloor, \lfloor p \rfloor) \\
\lfloor \texttt{pushP}\, p\, e \rfloor & = & \texttt{try}_{p_0} \lfloor e \rfloor\ \texttt{TH}_{\lfloor p \rfloor}
\end{array}
$$

where

$\texttt{TH}_q = \lambda y.\, \texttt{if}\ (\lambda y_2.\, (q, y_2))(\texttt{snd}\, y)\ \texttt{then}\ \texttt{fst}\, y\, ()\ \texttt{else}\ \texttt{raise}_{p_0}\, y$

The translation is a 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. 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 bi-simulation proof is in Appendix B.

We conclude that M_{ex} effectively provides the operation to locate a particular stack frame and split the stack at the frame, discarding 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 ought to be able to recover from it [14]. 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 [14].

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 [18]. When capturing a 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 the scAPI, we identify `pushP` frames with exception handling frames. To our knowledge, the points of exception handling correspond to stable points of concrete machines. The clearest evidence comes from architecture-description files used by the OCaml native-code generator: On all supported architectures, the code generator should assume that 'all physical registers are destroyed by raise.' That is, when an exception is raised, CPU registers other than control registers contain no machine state.

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

$$(\texttt{takeSC}\, pv, D, q) \;\mapsto\; (vD_1, D_2, q+1)$$
$$D_2[\texttt{pushP}\, pD_1] = D[\texttt{pushP}\, p'\square], \quad p' = q, \quad \texttt{pushP}\, pD' \notin D_1$$
$$(\texttt{pushSC}\, D'e, D, q) \;\mapsto\; (e, D[\texttt{pushP}\, p''D'], q+1) \quad p'' = q$$

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 p' and p'' are fresh, are not passed as values and so there cannot be any `takeSC` operations referring to these prompts. Any continuation captured in $\mathsf{M}^{i}_{\mathsf{dc}}$ is delimited by $\texttt{pushP}\, p'$ at one end and $\texttt{pushP}\, p$ at the other: the continuation is captured between two stable points, as desired. The re-instated continuation is again sandwiched between two `pushP` frames: $\texttt{pushP}\, p'\square$ 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. On the other hand, the introduction of the auxiliary `pushP` frames may break tail-call optimization and lead to a memory leak; we discuss how to plug it in §5.

4. Implementation in OCaml

In the previous section, we have introduced the 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 [11, 12]) 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 for `push_prompt p` and `v` does for `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 [19] 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.

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;;
```

We have implicitly assumed that a `push_prompt` receives the `P0` exception raised by `take_subcont`. That assumption is violated if the user-supplied `thunk` contained an expression of the form `try ... with _ -> ...` that intercepts and ignores all exceptions. Our full implementation in §4.3 ensures the assumption always holds, even if the user code intercepts and fails to re-raise exceptions.

We make the code more uniform so that the `try`-ed expression always ends in the `P0` exception, raised either during the evaluation of `thunk` or afterwards.

```
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;;
```

When we come to capturing of delimited continuations in §4.3, we will see that the uniform code gives us the convenient, for cleaning up, invariant that the evaluation of a captured continuation always ends in an exception. 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 reset_ek  : ek -> exn -> 'a
 val rebase_ek : ek -> ek -> ek -> ek

 val copy_stack_fragment : ek -> ekfragment
 val push_stack_fragment : ekfragment -> exn -> 'a
end
```

with two abstract types, `ek` and `ekfragment` representing the relevant parts of the machine state, and the operations to query the state and to alter it. The state altering operations, `reset_ek` and `push_stack_fragment`, reset the machine to a stable point. These functions have the return type `'a` meaning that they do not return.

The abstract type `ek` identifies an exception frame, that is, a particular frame $\mathtt{try}_p \,\square e'$ within $\mathsf{M}'_{\mathsf{ex}}$'s context; we will write the `ek`-identified frame as $\mathtt{try}_{\mathrm{ek}} \,\square$. The function `get_ek ()` returns the identity of the latest exception frame. There are *no* operations to scan the stack looking for a particular frame. The state-altering operation `reset_ek` is a version of `raise`: whereas `raise ex` throws the exception `ex` to the latest exception frame, `reset_ek ek ex` throws the exception to the specific exception frame identified by `ek`, which must be on the stack. We will explain `rebase_ek` shortly.

A fragment of the stack between two exception frames is represented by `ekfragment`. Given the stack of the form $D_2[\mathtt{try}_{\mathrm{ek1}}[D_1[\mathtt{try}_{\mathrm{ek2}}\, D']]]$ where D' has no exception frames, `copy_stack_fragment ek1` returns the part of the stack $D_1[\mathtt{try}_{\mathrm{ek2}} \,\square]$ from `ek1` through the latest exception frame. The latest exception frame is captured as part of the returned `ekfragment`, which is a heap-allocated OCaml value. The copied `ekfragment` remains on the stack. To remove the fragment off the stack, up to the exception frame `ek1`, we should execute `reset_ek ek1 ex`.

The operation `push_stack_fragment ekfragment ex` splices-in the previously copied `ekfragment` at the point of the latest exception frame, turning the stack from $D_2[\mathtt{try}_{\mathrm{ek}}\, D']$ to $D_2[\mathtt{try}_{\mathrm{ek}}[D_1[\mathtt{try}_{\mathrm{ek2}}\, D']]]$. After the splicing, the function throws the exception `ex` so the control resumes from the stable point identified by `ek2`. The reset, copy and push operations clearly correspond to the transitions of $\mathsf{M}^i_{\mathsf{dc}}$ in §3. We never capture the top stack

frames D' and never copy onto the top of the stack D' because D' contains ephemeral local data [18].

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, using `rebase_ek`. Suppose we copied an `ekfragment` up to the exception frame `ekbase` and then put the fragment back onto the stack starting with the exception frame `ekbase'`. Then the adjusted `ek` value is given by the expression `rebase_ek ek ekbase ekbase'`. If `ek` represents an address, `rebase_ek` offsets it.

The OCaml byte-code interpreter [20], an elaboration of the abstract machine ZAM [18], 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 stack addresses; `ek` is such an address, relative to the beginning of the stack `caml_stack_high`. The foreign-function `get_ek ()` exposes `trapsp` as `ek`.

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 `copy_stack_fragment` and `push_stack_fragment` are variants of interpreter's stack-copying procedure. These operations along with `get_ek` can be invoked from OCaml code via the foreign-function interface (FFI).

There are further conditions for safely putting copied stack fragments back onto stack, perhaps several times. First of all, OCaml data structures with mutable fields must not be allocated or otherwise stored on stack. The OCaml FFI manual guarantees that all such data structures are heap-allocated. Second, no frame should contain a relative address pointer to data inside other frames. That condition is also satisfied by all OCaml back-ends. (Since all non-integer-valued data in OCaml are heap-allocated, a stack frame has nothing to expose to other frames.)

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 the `takeSC` transition of $\mathsf{M}^i_{\mathsf{dc}}$ 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 the 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) : 'any =
  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 sets 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 and mbox_receive defined in that section.

The code for take_subcont is again an elaboration of the code in §4.1; now it has to capture the continuation rather than discarding 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 pframes 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 pframes 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 field h.pfr_ek identifies the corresponding pushP p frame on the real stack. 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 $\mathsf{M}^i_{\mathsf{dc}}$. First it 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. The function take_subcont then splits the parallel stack at the closest pframe h corresponding to the given prompt p; the assignment ptop := s removes h and the subsequent pframes from the parallel stack. The removed prefix, subcontchain, becomes part of the continuation object. We save in the field subcont_ek the corresponding part of the real stack. Finally we remove the copied part of the real stack, delivering DelimCCE straight to the exception frame ek lying beneath the copied ekfragment. That direct exception delivery, which effectively raises the exception after ekfragment is removed, means that we no longer rely on user's 'good exception-handling behavior', to re-raise our DelimCCE. Exception handlers in user code never get a chance to intercept DelimCCE.

```
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 p' = new_prompt () in
  push_prompt_simple p'
    (fun () ->
      let (h,s,subcontchain) = unwind [] p.mark !ptop in
      let () = ptop := s in
```

```
    let ek = h.pfr_ek in
    let ekfrag = copy_stack_fragment ek in
    p.mbox :=
    f {subcont_ek = ekfrag; subcont_pa = p';
       subcont_pb = p; subcont_ps = subcontchain;
       subcont_bs = ek};
    reset_ek ek DelimCCE)
```

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 p'' = new_prompt () in
  push_prompt_simple p'' (fun () ->
   try
    let base = sk.subcont_bs in
    let ek = get_ek () in
    List.iter (fun pframe ->
       ptop := {pframe with pfr_ek = rebase_ek pframe.pfr_ek base ek} ::
            !ptop) sk.subcont_ps;
    sk.subcont_pa.mbox := m;
    push_stack_fragment sk.subcont_ek DelimCCE
   with DelimCCE ->
     let v = mbox_receive sk.subcont_pb in
        p''.mbox := fun () -> v)
```

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. We can optimize the code by fusing the repeated try expression (one of which is hidden in push_prompt_simple).

5. 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 m)
```

However, that implementation has a memory leak, which we demonstrate. The function push_delim_subcont expresses a common pattern of pushing a *delimited* continuation. The same pattern occurs in implementations of user-level threads or coroutines, 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. After several thousand iterations the loop sched_loop (push_prompt p proc) exhausts all available memory and abnormally terminates.

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 (). Evaluating $\mathtt{pushP}\, p$ $(\mathtt{pushSC}\, D_1\, ())$, which reduces to $\mathtt{pushP}\, p\, (\mathtt{pushP}\, p''\, (\mathtt{loop}\, e_b e_b))$ resumes the thread. Here, p'' is the fresh prompt introduced by the 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 `pushP` p at the top of the stack, the `pushSC` transition of $\mathsf{M}^i_{\mathsf{dc}}$ no longer needs to push the `pushP` p'' frame. With this new primitive, the paused and resumed thread `proc` runs in constant memory, as demonstrated in delimcc's test suite.

6. Implementing delimcc in native-code OCaml

To summarize so far, §3 described a general method of implementing delimited continuations on a system that provides exception handling and the minimalistic scAPI. We have followed that method in §4 to implement delimcc on byte-code OCaml – which has exceptions and does happen to support the scAPI. In this section we describe another implementation of delimcc, in native-code OCaml. To be precise, we describe the difficulties and tricks of implementing just the scAPI in native-code OCaml. The rest of the delimcc code, written in terms of scAPI, stays literally the same.

Native-code OCaml is a different back-end of the OCaml compiler. Whereas the byte-code back-end (which we have dealt with so far) compiles OCaml into code for the OCaml virtual machine, the native-code back-end compiles into assembly code for one of the supported architectures (i386, amd64, arm, etc). The two back-ends are quite distinct owing to the differences between CPU instruction sets and the OCaml byte-code. Notably, whereas the byte-code machine dedicates a separate stack to the execution of byte-code, the native-code program has to share the native stack, or 'the C stack', with foreign functions, primitives and signal handlers. The byte-code interpreter handles stack overflow, by resizing (and hence, copying) the stack; in contrast, stack overflow in native-code programs is non-recoverable. Although the native-code stack is no longer copied, fortunately there are no stumbling blocks for doing so, as OCaml-generated code never uses absolute stack addresses (with the sole exception of linking exception frames; that one case can be accommodated by adjusting the exception frame pointers as we copy the stack). We already discussed in §4.2 that the OCaml FFI specifies that no mutable OCaml data are allocated or stored on stack. Furthermore, delimcc ensures that the captured stack prefix has no C frames, raising a run-time error otherwise. In our experience we have never seen the capturing of a delimited continuation across the OCaml callback invoked from C code.

Thus scAPI – with exactly the interface of §4.2 – is implementable for native code (for the currently supported 32- and 64-bit `x86` architectures).

Therefore, the rest of the delimcc implementation in §4.3, which uses scAPI, applies to native code *as it is.* The only difference between byte- and native-code versions of delimcc is the implementation of scAPI.

Although scAPI is supported for native-code programs, its implementation was not easy. The main difficulty is the sharing of the C stack with primitives and foreign functions. Besides OCaml values the stack therefore may contain unboxed values. Since the garbage collector (GC) in OCaml is precise, the GC needs to know exactly which values on the stack are definitely OCaml heap pointers. The GC gets this information from so-called frame tables, placed into the executable file by the code generator. We must take care to preserve the frame structure as we copy parts of the stack. Mainly, the continuation object, containing a part of the C stack, is not an ordinary OCaml value since it contains a mixture of heap pointers and unboxed values. We have to arrange for a special GC procedure to scan such a mixed value. This custom GC scanning procedure turns out to be possible, *without any modifications to the OCaml system* – albeit not very efficient at the moment.

7. Benchmarks

The library delimcc has been used in a variety of applications and proved to be adequate in performance. The paper [4] details the performance of a probabilistic embedded domain specific language that relies on delimcc for probabilistic choice and failure. Deleuze [21] has compared an old version of delimcc with other OCaml concurrency frameworks on several benchmarks. (The present version of delimcc is about ten per cent faster.) Running micro-benchmarks and the sample code included in the delimcc distribution can also give one some sense of the library performance. In this section we discuss two of the micro-benchmarks, written to experimentally validate the basic theoretical expectations of the delimcc library.

The implementation of delimcc exploits the relation between raising exceptions and capturing delimited continuations. We have seen in §3 that capturing and throwing away a delimited continuation – or, aborting – is equivalent to raising an exception. One would expect then that aborting using delimcc is just as fast as raising a native OCaml exception.

The first benchmark checks that expectation by timing the two operations. The benchmark computes the product of a list of numbers, throwing an exception or aborting upon encountering zero. We intentionally use the non-tail-recursive product computation and make sure zero occurs at the very

end of the list, so that an exception or the abort have a large portion of stack to unwind. In the following code, test1_ex raises the native OCaml exception Zero, whereas test1_abort relies on delimcc, with prompt p playing the role of the exception type.

```
exception Zero
let test1_ex lst =
  let f x acc =  if x = 0 then raise Zero else x * acc
  in
  try List.fold_right f lst 1 with Zero -> 0

let test1_abort lst =
  let p = new_prompt () in
  let f x acc =  if x = 0 then abort p 0 else x * acc
  in
  push_prompt p (fun () -> List.fold_right f lst 1)
```

The function abort p v immediately returns the value v to the closest push_prompt p, skipping the rest of the push_prompt's body. The function can be defined in delimcc as

```
let abort p v = take_subcont p (fun _ -> v)
```

which is wasteful as it throws away the subcont object that take_subcont took time to allocate and build. Since abort turns out practically useful delimcc provides abort as a primitive, which is a version of take_subcont that skips the saving of the captured stack fragment. The primitive abort still has to do the chores of maintaining the parallel stack.

We have run the benchmark on lists as long as 110 000 elements, which is nearly at the edge of stack overflow. The timing showed no perceptible difference in performance between test1_ex and test1_abort. The file bench_exc.ml of the delimcc distribution contains the complete code, which also includes a more involved version of the benchmark, which tests throwing an exception in the presence of very many other exception handlers.

The second theoretical expectation of delimcc is that the implementation deals only with the relevant prefix of the stack, never having to scan, move, or otherwise handle the whole stack. In other words, the performance of a delimcc application that operates on delimited continuations whose size is bounded by a fixed number is not expected to depend on the total size of the stack. The opposite is expected of an application that uses the implementation of call/cc that copies the whole continuation. To experimentally

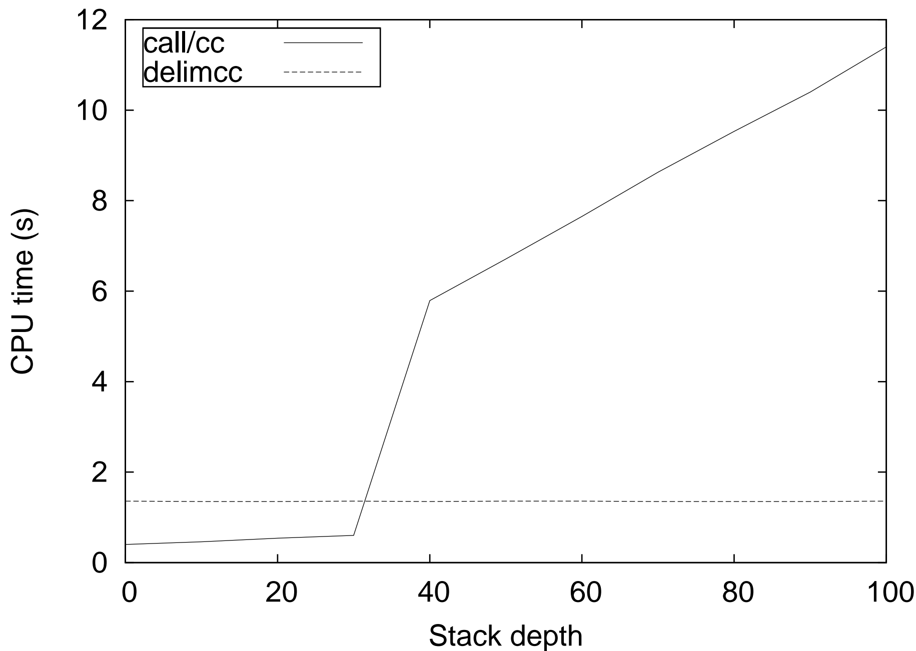

Figure 3: The running time (in seconds) vs the stack depth, for the call/cc and delimcc versions of the coroutine benchmark. The platform: OCaml 3.11 byte-code interpreter, i386 FreeBSD, 2GHz Pentium 4. The shown running time values are the medians of five consecutive runs.

test these expectations, we used the coroutine benchmark included with the call/cc library [22] for byte-code OCaml. We can invoke the benchmark function either as the top-level expression in a module (stack depth 0), or from a non-tail recursive function that has called itself 10, 20, ..., 100 times. The number of non-tail-recursive calls is the measure of the stack depth at which the benchmark is executed. The results are presented in Figure 3, as plots of the running time of the benchmark vs. the stack depth.

We have re-implemented the benchmark using delimited continuations. (The file `bench_coroutine.ml` of the delimcc distribution contains the complete code of both implementations.) Again, we plot the running time vs. the depth of the stack at the point the benchmark was invoked. At stack depth zero, call/cc is more efficient than delimcc: the size of the captured continuation is roughly the same, but delimcc has an administrative overhead of maintaining the parallel stack; delimcc invokes more FFI (scAPI) functions and incurs more FFI overhead. The advantage of delimcc becomes apparent as the stack depth increases. The coroutine benchmark creates two coroutines, which invoke each other two hundred thousand times. We only need to capture the continuation of the current coroutine up to the scheduling point, at the start of the benchmark. The delimcc implementation does exactly that. The size of the captured continuation is bounded by the size of a coroutine, which is fixed. The performance of delimcc benchmark stays constant too, regardless of the total size of the control stack. The more

extensive benchmark study of delimcc, call/cc and other lightweight concurrency frameworks comes to the same conclusions [21]. The benchmarks thus validate the theoretical expectations of the delimcc library.

8. Persistent Delimited Continuations

Recall from §2 that a captured delimited continuation is a 'restart object'; it can later be used, by `push_prompt`, to restart the computation interrupted by `take_subcont`. If we save the captured delimited continuation on disk, we can restart the computation not only later but in a different process or even a different computer. Making captured continuations persistent – serializing and deserializing them – lets us implement checkpointing of computations [5] or process migration [33]. The library delimcc supports persistent delimited continuations (for byte-code only). This section describes the challenges and their resolutions – not only because persistent delimited continuations are so practically useful but also because their implementation is unusually tricky.

At first blush, the implementation should be trivial: OCaml's standard library has a function `Marshal.to_channel` to serialize OCaml values, chasing the referred values and writing them too, preserving sharing. The function `Marshal.from_channel` de-serializes. Applying `Marshal.to_channel` to a captured delimited continuations leads however to a run-time error.

The error is fortunate: otherwise, we would have obtained a huge value giving subtle problems upon deserialization. All three problems have the same cause: extensive data dependencies of captured continuations. The smallest, identity, continuation captured by the following code

```
let krepr =
  let p = new_prompt () in
  push_prompt p (fun () -> take_subcont p (fun sk () -> Obj.repr sk))
```

contains only 18 stack words. However, it transitively refers to a large part of the core library. Serializing such a continuation has to serialize, along with it, almost entire global data. The global data include IO channels like `stdin`, which are not serializable. That is the cause of the run-time error when attempting to marshal a captured delimited continuation.

The global data reachable from a captured delimited continuation also include `ptop`, the top of the parallel stack, §4.3. The marshaled continuation will have its own copy of `ptop`. After deserialization, we end up with two

copies of `ptop`, which will cause insidious errors. We come across the general problem of serializing any global mutable data.

The problem of serializing global data – which are large, contain non-serializable values such as IO channels and contain mutable globals of delimcc – is solved by getting the OCaml marshaling functions to serialize some values by reference rather than by value. Code pointers are already serialized by reference: `Marshal.to_channel` does not write the whole code segment; it merely emits the offset from the beginning of the code segment to the pointed code location. We should arrange for the similar treatment of global data. Unlike code, which is immutable and unmovable in memory, global data are loaded into the heap upon start-up, and hence are movable by the garbage collector. Our solution is to 'relativitize' the captured continuation before serializing it, and 'absolutize' it after deserializing. The standard marshaling functions can be used as they are. The relativitization procedure replaces references to seemingly global data with relative indices, in the global array `global_data`, which is not serialized. We determine the seemingly global data as all data reachable from the identity continuation captured by delimcc upon its initialization. The library populates `global_data` at that moment then. The library lets users register their own global data to be serialized by reference.

The serialized delimited continuation is thus twice delimited: with respect to the whole continuation (the whole stack) and with respect to the global environment.

9. Related Work

Paper [11] introduced multi-prompt delimited control and presented its implementation in SML/NJ, relying on local exceptions and call/cc. Later the same authors offered a byte-code–only OCaml implementation [15], using "a very naive experimental brute-force version of callcc that copies the stack", along with `Obj.magic`, or unsafe coerce. The copying of the entire control stack to and from the heap on each use of control operators is not the only problem. 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 [22] is not obvious as it copies the stack regardless of whether the byte-code interpreter is at a stable point or not. Since some of the interpreter state is maintained in registers

(such as `extra_args` register), copying the stack may not necessarily preserve all the data needed for restarting the interpreter. The implementation of call/cc attempts to force saving of `extra_args` by writing code in a way so to defeat the tail-call optimization. This technique is not robust with respect to compiler improvements.

Multi-prompt delimited control was further developed and formalized in [12], which 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 [23], specifically for the Scheme48 interpreter. 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, specifically, to mark `reset`'s frames. The GC also had to be modified. On many benchmarks, the paper [23] 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; one of the reasons, mentioned in [24], was that the interactions of shift/reset with the rest of the Scheme48 system (in particular, dynamic binding, exceptions and dynamic-wind) have not been worked out. The paper [23] specifically left to future work relating the implementation to the specification of shift/reset.

Flatt et al. [24], picking up where [23] left off, worked out the interactions of delimited control with the standard Scheme features (such as dynamic-wind) as well as with many extensions of PLT Scheme (e.g., continuation marks). We share with the authors of [24] the goal of adding delimited control to the 'production' rather than an idealized environment, ensuring the new features interact with the rest of the system in well-defined and useful ways, and maintaining, hopefully, backwards compatibility. This goal has been achieved; admittedly adding delimited control to OCaml was simpler since OCaml does not have dynamic-wind, which is the main source of complexity [24]. Flatt et al. give few details about their implementation; the correctness is argued for only extensionally, by comparing test suite results with the results of the executable specification. The authors of [24] are the implementers of PLT Scheme, who could make (and it seemed, have made) changes to the system to accommodate new features. Our strategy was exactly the opposite.

The motivation to add delimited continuations to an existing language as

it is puts us within the approach pioneered by Kumar et al. [25], who were the first to constructively prove, in the untyped setting, that a language system supporting threads supports *one-shot* delimited continuations. One-shot delimited continuations suffice for many applications of delimited control except for non-determinism and probabilistic programming. The implementation [25] was simplified by their choice of control operators, spawn/controller. Our operators require more effort since `take_subcont` finds the corresponding `push_prompt` essentially from the dynamic environment, which we would have to emulate. Since OCaml supports threads, it is possible to use the (extended) technique of [25] to implement a one-shot version of delimcc. It will be slow: the study [21] showed that lightweight concurrency via delimcc is notably more efficient than OCaml threads (especially system threads, the only choice for native-code OCaml).

Recently there has been interest in direct implementations (as compared to the call/cc-based one [26] in SML/NJ) of the single-prompt shift/reset in the typed setting [27, 28]. 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. [14]. Clinger et al. [29] is a comprehensive survey. Their lessons hold for delimited control as well.

Sekiguchi et al. [30] 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 [31], 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 also has difficulties interfacing third-party libraries.

10. 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 since 2006. The abstract implementation has related delimited control to exception handling and distilled scAPI, a minimalistic API sufficient for the implementation of delimited control. Any

language system accommodating exception handling and stack-overflow recovery is likely to support scAPI. The OCaml byte- and native-code systems do support scAPI, and thus permit, *as they are*, 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, until recently 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 [14]. Preliminary results of porting delimcc to Haskell point out towards the derivation of the (hitherto ad hoc) stack segmentation technique from $\mathsf{M}^{i}_{\mathsf{dc}}$.

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.

References

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

[2] O. Kiselyov, C.-c. Shan, A. Sabry, Delimited dynamic binding, in: ICFP, ACM, 2006, pp. 26–37.

[3] O. Kiselyov, C.-c. Shan, Embedded probabilistic programming, in: Proc. IFIP Working Conf. on DSL, volume 5658 of *LNCS*, Springer, 2009, pp. 360–384.

[4] O. Kiselyov, C.-c. Shan, Monolingual probabilistic programming using generalized coroutines, in: Uncertainty in Artificial Intelligence, AUAI Press, 2009.

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

[6] Y. Kameyama, O. Kiselyov, C.-c. Shan, Shifting the stage: Staging with delimited control, in: PEPM, ACM, 2009, pp. 111–120.

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

[8] K. Anton, P. Thiemann, Towards deriving type systems and implementations for coroutines, in: APLAS, volume 6461 of *LNCS*, Springer, 2010, pp. 63–79.

[9] J. Donham, Mixing monadic and direct-style code with delimited continuations, http://ambassadortothecomputers.blogspot.com/2010/08/mixing-monadic-and-direct-style-code.html, 2010.

[10] O. Kiselyov, Delimited control in OCaml, abstractly and concretely: System description, in: Proc. FLOPS 2010: 10th International Symposium on Functional and Logic Programming, number 6009 in LNCS, Springer, 2010, pp. 304–320.

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

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

[13] V. Balat, R. Di Cosmo, M. P. Fiore, Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums, in: POPL, ACM, 2004, pp. 64–76.

[14] R. Hieb, R. K. Dybvig, C. Bruggeman, Representing control in the presence of first-class continuations, in: PLDI, ACM, 1990, pp. 66–77.

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

[16] O. Danvy, A. Filinski, Abstracting control, in: LFP, ACM, 1990, pp. 151–160.

[17] M. Felleisen, The theory and practice of first-class prompts, in: POPL, ACM, 1988, pp. 180–190.

[18] X. Leroy, The ZINC Experiment: An Economical Implementation of the ML Language, Technical Report 117, INRIA, 1990.

[19] N. Glew, Type dispatch for named hierarchical types, in: ICFP, ACM, 1999, pp. 172–182.

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

[21] C. Deleuze, Light weight concurrency in OCaml: continuations, monads, events, and friends, 2010.

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

[23] M. Gasbichler, M. Sperber, Final shift for call/cc: Direct implementation of shift and reset, in: ICFP, ACM, 2002, pp. 271–282.

[24] M. Flatt, G. Yu, R. B. Findler, M. Felleisen, Adding delimited and composable control to a production programming environment, in: ICFP, ACM, 2007, pp. 165–176.

[25] S. Kumar, C. Bruggeman, R. K. Dybvig, Threads yield continuations, Lisp and Symbolic Computation 10 (1998) 223–236.

[26] A. Filinski, Representing monads, in: POPL, ACM, 1994, pp. 446–457.

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

[28] T. Rompf, I. Maier, M. Odersky, Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform, in: ICFP, ACM, 2009, pp. 317–328.

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

[30] T. Sekiguchi, T. Sakamoto, A. Yonezawa, Portable implementation of continuation operators in imperative languages by exception handling, in: Advances in Exception Handling Techniques, volume 2022 of *LNCS*, Springer, 2001, pp. 217–233.

[31] G. Pettyjohn, J. Clements, J. Marshall, S. Krishnamurthi, M. Felleisen, Continuations from generalized stack inspection, in: ICFP, ACM, 2005, pp. 216–227.

[32] D. Biernacki, O. Danvy, K. Millikin, A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations, Report RS-05-16, BRICS, Denmark, 2005.

[33] E. Sumii, An implementation of transparent migration on standard Scheme, in: Proc. Workshop on Scheme and Functional Programming, number 00-368 in Tech. Rep., Dept. Computer Science, Rice University, 2000, pp. 61–63.

Appendix 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 [32].

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 $(e, \square, [], 0)$ and terminates when it reaches $(v, \square, [], q)$.

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: $\widehat{e} \sim_e e$ is $\widehat{e} = e$ extended with

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

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

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

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

Figure A.4: Definitional machine $\mathsf{M}_{\mathsf{defn}}$ for multi-prompt delimited control from [12, Figure 1] (adjusted for style). Prompts p and sequences E may not appear in source programs.

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* $\widehat{E} = \widehat{E_1} +\!\!+ \widehat{E_2}$ *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* $\widehat{E_1}$ *and* $\widehat{E_2}$ *such that* $\widehat{E} = \widehat{E_1} +\!\!+ \widehat{E_2}$ *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 $\widehat{E_1}$ (in one direction) or $\widehat{E}$ (in the converse direction), using Lemma 1 and Lemma 2.

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

1. *If* $\widehat{C} \mapsto \widehat{C'}$ *for some* $\widehat{C'}$ *then there exists* $\widehat{C''}$ *and* C' *such that* $\widehat{C'} \mapsto^* \widehat{C''}$, $C \mapsto^* C'$*, 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* $\widehat{C'} \sim_c C$*. Conversely, if* $\widehat{C}$ *is terminal, so is* C*.*

Only the cases where $\widehat{C}$ includes $\widehat{\texttt{pushP}\,pe}$, $\widehat{\texttt{takeSC}\,pv}$, $\widehat{\texttt{pushSC}\,E'e}$, 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 $\widehat{\texttt{pushP}\,pe}$, $\widehat{\texttt{takeSC}\,pv}$ (seen from Lemma 3) and $\widehat{\texttt{pushSC}\,E'e}$ (proved using Lemma 2).

The remaining case is of the first component of $\widehat{C}$ being a value (the first component of the related C must be a value too, by the definition of $\sim$). There are three sub-cases. First, C and $\widehat{C}$ are both terminal configurations. The lemma (part 3) clearly holds then. Second, C is the terminal configuration $(v, \Box, q)$, but the related $\widehat{C}$ is not. The definition of $\sim$ implies that $\widehat{C}$ must have the form $(\widehat{v}, \widehat{\Box}, \widehat{E}, \widehat{q})$ where $\widehat{E}$ is the list made entirely of $\Box$. In the number of steps equal to the length of the list, the machine reaches the terminal configuration that is related to C, as part 3 of the lemma requires.

The final sub-case deals with a non-terminal $C = (v, D, q)$. The related $\widehat{C}$ can have one of the following three forms: $(\widehat{v}, \widehat{D}, \widehat{E}, \widehat{q})$ with $\widehat{D} \neq \widehat{\Box}$, $(\widehat{v}, \widehat{\Box}, \widehat{p : E}, \widehat{q})$, or $(\widehat{v}, \widehat{\Box}, \widehat{D : E}, \widehat{q})$.

The lemma holds for the second sub-sub-case, with $C' = (\texttt{pushP}\,pv, D', q)$ and $C'' = (v, D', q)$ where $D = D'[\texttt{pushP}\,p\Box]$. The lemma also holds for the third sub-sub-case: we apply the last rule in Figure A.4, maybe more than once if $\widehat{D}$ is empty.

The only complex sub-sub-case is when $C = (v, D, q)$ and the related $\widehat{C} = (\widehat{v}, \widehat{D}, \widehat{E}, \widehat{q})$ with $\widehat{D} \neq \widehat{\Box}$. The fact that $\widehat{C} \sim_c C$ shows that D is not $\Box$; furthermore, it must have the form $D'[D_1]$ where D_1 is one of

$$\Box e, v\Box, \texttt{pushP}\,\Box\, e, \texttt{pushSC}\,\Box\, e, \texttt{takeSC}\,\Box\, e, \texttt{takeSC}\,p\,\Box.$$

In turn, that implies that $\widehat{D}$ must have the form $\widehat{D'}[\widehat{D_1}]$ where $\widehat{D_1}$ is, respectively, one of

$$\widehat{\Box e}, \widehat{v\Box}, \widehat{\texttt{pushP}\,\Box}\, e, \widehat{\texttt{pushSC}\,\Box}\, e, \widehat{\texttt{takeSC}\,\Box}\, e, \widehat{\texttt{takeSC}\,p}\,\Box.$$

The machine M_{dc} transitions to $C' = (D_1[v], D', q)$ and the definitional machine transitions to $\widehat{C'} = (\widehat{D'[D_1[v]]}, \widehat{\Box}, \widehat{E}, \widehat{q})$. If $\widehat{D'} = \widehat{\Box}$ then $\widehat{C'} \sim_c C'$ by Lemma 1. Otherwise, $\widehat{D'} = \widehat{D_2[D'']}$ where $\widehat{D_2}$ is single-frame (that is, it has the same general structure as $\widehat{D_1}$). We observe that $\widehat{D_1[v]}$ is not a value, and hence, neither is $\widehat{D''[D_1[v]]}$. Therefore, $D_2[\widehat{D''[D_1[v]]}]$ has the structure such that one of the first six transitions of the definitional machine applies, giving

us the configuration $(\widehat{D''[D_1[v]]}, \widehat{D_2}, \widehat{E}, \widehat{q})$. By repeating the process finitely many times we obtain $\widehat{C''} = (\widehat{D_1[v]}, \widehat{D'}, \widehat{E}, \widehat{q})$. Using Lemma 1 we can show that $\widehat{C''} \sim_c C'$.

Appendix B. Proving the equivalence of $\mathsf{M}'_{\mathsf{dc}}$ and $\mathsf{M}'_{\mathsf{ex}}$

In this section we formally relate exception handling and the restricted form of capturing a delimited continuation, justifying the conclusion in §3.

The machine $\mathsf{M}'_{\mathsf{dc}}$ (Figure B.5) is M_{dc} with a restriction on source programs: there is no `pushSC` and all `takeSC` expressions must be of the form $\texttt{takeSC}\, e\, (\lambda_.\, e')$ (where the notation $\lambda_.\, e'$ stands for $\lambda x.\, e'$ such that x is not free in e'). Therefore, contexts D are not values of $\mathsf{M}'_{\mathsf{dc}}$. The machine $\mathsf{M}'_{\mathsf{ex}}$ (Figure B.6) is an extended version of the exception machine M_{ex}. We add integer identifiers q and the conditional $\texttt{if}\ (q_1, q_2)\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, which branches on equality of two identifiers q_1 and q_2. These identifiers cannot appear in source programs but are generated by an operator `newQ`, evaluating each time to a fresh value. We add unit $()$, pairs (v, v), and pair projections functions `fst` and `snd`.

$$
\begin{array}{llll}
\text{Variables} & x, y, \ldots & \text{Prompts} & p, q \in N \\
\text{Expressions} & e & ::= & v \mid e\,e \mid \texttt{newP} \mid \texttt{pushP}\,e\,e \mid \texttt{takeSC}\,e\,\lambda_.\,e \\
\text{Values} & v & ::= & x \mid \lambda x.\,e \mid p \\
\text{Contexts} & D & ::= & \Box \mid De \mid vD \mid \texttt{pushP}\,D\,e \mid \texttt{takeSC}\,D\,\lambda_.\,e \\
 & & & \mid \texttt{pushP}\,p\,D \\
\text{Single Frame} & & ::= & \Box e \mid v\Box \mid \texttt{pushP}\,\Box\,e \mid \texttt{takeSC}\,\Box\,\lambda_.\,e \\
 & & & \mid \texttt{pushP}\,p\,\Box
\end{array}
$$

Transitions between configurations (e, D, q)

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

Figure B.5: Restricted version $\mathsf{M}'_{\mathsf{dc}}$ of the abstract machine M_{dc} that discards the captured continuation. The initial configuration is $(e, \Box, 0)$, the terminal is $(v, \Box, q)$.

$$
\begin{array}{llll}
\text{Variables} & x, y, \ldots \quad \text{Exceptions} \quad p, \ldots \quad \text{Id} \quad q \in N \\
\text{Expressions} & e ::= v \mid e\,e \mid \texttt{raise}_p\, e \mid \texttt{try}_p\, e\, e \mid \texttt{newQ} \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Values} & v ::= q \mid x \mid \lambda x.\, e \mid () \mid (v, v) \mid \texttt{fst} \mid \texttt{snd} \\
\text{Contexts} & D ::= \Box \mid De \mid vD \mid \texttt{raise}_p\, D \mid \texttt{try}_p\, D\, e \mid \texttt{if}\ D\ \texttt{then}\ e\ \texttt{else}\ e \\
\text{Single Frame} & \quad ::= \Box e \mid v\Box \mid \texttt{raise}_p\, \Box \mid \texttt{try}_p\, \Box\, e \mid \texttt{if}\ \Box\ \texttt{then}\ e\ \texttt{else}\ e
\end{array}
$$

Transitions between configurations (e, D, q)

$$
\begin{array}{rcll}
(ee', D, q) & \mapsto & (e, D[\Box e'], q) & e \text{ non-value} \\
(ve, D, q) & \mapsto & (e, D[v\Box], q) & e \text{ non-value} \\
(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2, D, q) & \mapsto & (e, D[\texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2], q) & e \text{ non-value} \\
(\texttt{raise}_p\, e, D, q) & \mapsto & (e, D[\texttt{raise}_p\, \Box], q) & e \text{ non-value} \\
((\lambda x.\, e)v, D, q) & \mapsto & (e[v/x], D, q) & \\
(\texttt{try}_p\, ee', D, q) & \mapsto & (e, D[\texttt{try}_p\, \Box e'], q) & \\
(\texttt{raise}_p\, v, D, q) & \mapsto & (e'v, D_2, q) & \\
 & & \multicolumn{2}{r}{D_2[\texttt{try}_p\, D_1 e'] = D,\ \texttt{try}_p\, D'e \notin D_1} \\
(\texttt{newQ}, D, q) & \mapsto & (q, D, q+1) & \\
(\texttt{fst}\,(v_1, v_2), D, q) & \mapsto & (v_1, D, q) & \\
(\texttt{snd}\,(v_1, v_2), D, q) & \mapsto & (v_2, D, q) & \\
(\texttt{if}\ (q_1, q_2)\ \texttt{then}\ e_1\ \texttt{else}\ e_2, D, q) & \mapsto & (e_1, D, q) & q_1 = q_2 \\
(\texttt{if}\ (q_1, q_2)\ \texttt{then}\ e_1\ \texttt{else}\ e_2, D, q) & \mapsto & (e_2, D, q) & q_1 \neq q_2 \\
(v, D[D_1]) & \mapsto & (D_1[v], D) & D_1 \text{ single frame} \\
(\texttt{try}_p\, ve', D) & \mapsto & (v, D) &
\end{array}
$$

Figure B.6: Abstract machine $\mathsf{M}'_{\mathsf{ex}}$ for exception handling extended with more data types and operations on them. The initial configuration is $(e, \Box, 0)$, the terminal is $(v, \Box, q)$.

We define the translation $\lfloor \cdot \rfloor$ of $\mathsf{M}'_{\mathsf{dc}}$ expressions to the expressions of $\mathsf{M}'_{\mathsf{ex}}$ as follows.

$$
\begin{array}{lcll}
\lfloor \mathtt{takeSC}\, v\, (\lambda_.\, e) \rfloor & = & \mathtt{raise}_{p_0} (\lambda_.\, \lfloor e \rfloor, \lfloor v \rfloor) & \\
\lfloor \mathtt{pushP}\, v\, e \rfloor & = & \mathtt{try}_{p_0} \lfloor e \rfloor\ \mathtt{TH}_{\lfloor v \rfloor} & \\
\lfloor x \rfloor & = & x & \\
\lfloor p \rfloor & = & q & \\
\lfloor \lambda x.\, e \rfloor & = & \lambda x.\, \lfloor e \rfloor & \\
\lfloor e_1\, e_2 \rfloor & = & \lfloor e_1 \rfloor\, \lfloor e_2 \rfloor & \\
\lfloor \mathtt{newP} \rfloor & = & \mathtt{newQ} & \\
\lfloor \mathtt{pushP}\, e\, e' \rfloor & = & \lfloor (\lambda x.\, \mathtt{pushP}\, x\, e') e \rfloor & e \text{ non-value, } x \text{ fresh} \\
\lfloor \mathtt{takeSC}\, e\, \lambda_.\, e' \rfloor & = & \lfloor (\lambda x.\, \mathtt{takeSC}\, x\, \lambda_.\, e') e \rfloor & e \text{ non-value, } x \text{ fresh}
\end{array}
$$

We have introduced a dedicated exception type p_0 and the notation $\mathtt{TH}_q$:
$\mathtt{TH}_q = \lambda y.\, \mathtt{if}\ (\lambda y_2.\, (q, y_2))(\mathtt{snd}\, y)\ \mathtt{then}\ \mathtt{fst}\, y\, ()\ \mathtt{else}\ \mathtt{raise}_{p_0}\, y$

It is easy to see the following properties of the translation:

Lemma 6 (Value classification preservation). *If an expression e is a value of $\mathsf{M}'_{\mathsf{dc}}$, $\lfloor e \rfloor$ is a value of $\mathsf{M}'_{\mathsf{ex}}$, and conversely. If an expression e is not a value of $\mathsf{M}'_{\mathsf{dc}}$, $\lfloor e \rfloor$ is not a value of $\mathsf{M}'_{\mathsf{ex}}$, and conversely.*

Lemma 7 (Substitution). $\lfloor e[v/x] \rfloor = \lfloor e \rfloor [\lfloor v \rfloor / x]$.

Proposition 2 (Equivalence). *For all $\mathsf{M}'_{\mathsf{dc}}$ source programs e, the machine $\mathsf{M}'_{\mathsf{dc}}$ reaches the terminal configuration iff $\mathsf{M}'_{\mathsf{ex}}$ does so for the source program $\lfloor e \rfloor$.*

The proof is by bi-simulation, as follows. We first relate configurations of $\mathsf{M}'_{\mathsf{dc}}$ and $\mathsf{M}'_{\mathsf{ex}}$. To avoid confusion, we place the diacritic mark $\widehat{\cdot}$ over the configurations, contexts and expressions of $\mathsf{M}'_{\mathsf{dc}}$. We define the family of relations $\sim$ as the least relational family satisfying the following:

Relating configurations $\sim_c$

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

Relating expressions: $\widehat{e} \sim_e e$ iff $\lfloor \widehat{e} \rfloor = e$

Relating contexts:

$\widehat{\Box} \sim_d \Box$

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

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

$D[\widehat{\texttt{pushP}\,\Box} e] \sim_d D[(\lambda x.\, \texttt{try}_{p_0}\, e\ \texttt{TH}_x)\Box]$ iff $\widehat{e} \sim_e e,\ \widehat{D} \sim_d D$

$D[\widehat{\texttt{takeSC}\,\Box}\, \lambda_.\, e] \sim_d D[(\lambda x.\, \texttt{raise}_{p_0}(\lambda_.\, e, x))\Box]$ iff $\widehat{e} \sim_e e,\ \widehat{D} \sim_d D$

$D[\widehat{\texttt{pushP}\, p}\, \Box] \sim_d D[\texttt{try}_{p_0}\, \Box\ \texttt{TH}_p]$ iff $\widehat{D} \sim_d D$

By a simple structural induction argument we easily prove the following two lemmas.

Lemma 8. *If $\widehat{D} \sim_d D$ then for each non-value expression $\widehat{e}$ of $\mathsf{M}'_{\mathsf{dc}}$, $\widehat{D[e]} \sim_e D[\lfloor \widehat{e} \rfloor]$.*

Lemma 9. *If $\widehat{D_2[D_1]} \sim_d D$ where $\widehat{D_1}$ is a single frame of $\mathsf{M}'_{\mathsf{dc}}$, then D has the form $D_2[D_1]$ where $\widehat{D_2} \sim_d D_2$ and $\widehat{D_1} \sim_d D_1$ and D_1 is a single frame (of machine $\mathsf{M}'_{\mathsf{ex}}$).*

As before, we write $\mapsto^+$ for the transitive closure of the transition relation, and $\mapsto^*$ for the transitive reflexive closure. The proof of the equivalence of $\mathsf{M}'_{\mathsf{dc}}$ and $\mathsf{M}'_{\mathsf{ex}}$, Proposition 2, is based on the bi-simulation lemma:

Lemma 10. *Let $\widehat{C}$ be the configuration of $\mathsf{M}'_{\mathsf{dc}}$ and let C be the related configuration of $\mathsf{M}'_{\mathsf{ex}}$. Then:*

1. *If $\widehat{C} \mapsto \widehat{C'}$ for some $\widehat{C'}$ then there exists C' such that $C \mapsto^+ C'$, and $\widehat{C'} \sim_c C'$;*
2. *If $C \mapsto C'$ for some C' then either*
   - *(a) there exists C'' and $\widehat{C'}$ such that $C' \mapsto^* C''$, $\widehat{C} \mapsto \widehat{C'}$, and $\widehat{C'} \sim_c C''$; or,*
   - *(b) $\widehat{C}$ is a non-terminal configuration with no further transitions possible (that is, $\mathsf{M}'_{\mathsf{dc}}$ is stuck at $\widehat{C}$) and there exists C'' such that $C' \mapsto^+ C''$ and $\mathsf{M}'_{\mathsf{ex}}$ is stuck at C''.*

3. *If $\widehat{C}$ is a terminal configuration, then so is C, and conversely.*

The third part follows from the definitions of $\sim_c$ and $\sim_d$ and Lemma 6. The first two claims are straightforward when $\widehat{C}$ has the form $(\widehat{ee', D, q})$, $(\widehat{ve, D, q})$ (e is not a value), $(\widehat{\texttt{newP}, D, q})$, $(\widehat{\texttt{pushP}\, pv, D, q})$, and $(\widehat{(\lambda x.\, e)v, D, q})$ (the latter requires the substitution lemma 7). The two machines transition in lock-step in these cases. The machines also transition in lock-step when $\widehat{C}$ has the form $(\widehat{\texttt{pushP}\, ee', D, q})$ $(\widehat{\texttt{takeSC}\, e\lambda_.\, e', D, q})$ and $(\widehat{\texttt{pushP}\, pe, D, q})$ (where e is not a value). We show the proof for the `takeSC` case, the others are analogous.

Let $\widehat{C}$ be $(\widehat{\texttt{takeSC}\, e\lambda_.\, e', D, q})$ where e is not a value. From the definition of $\sim_c$, the related C must have the form $((\lambda x.\, \texttt{raise}_{p_0}(\lambda_.\, e', x))e, D, q)$ where $\widehat{e} \sim_e e$, $\widehat{e'} \sim_e e'$, and $\widehat{D} \sim_d D$. The machine $\mathsf{M}'_{\mathsf{dc}}$ is able to transition from $\widehat{C}$ to $\widehat{C'} = (e, \widehat{D[\texttt{takeSC}\,\square\lambda_.\, e']}, q)$. The machine $\mathsf{M}'_{\mathsf{ex}}$ also can make a transition, from C to $C' = (e, D[(\lambda x.\, \texttt{raise}_{p_0}(\lambda_.\, e', x))\square], q)$. We observe that $\widehat{C'} \sim_c C'$.

In the case of $\widehat{C}$ being $(v, \widehat{D[D_1]}, q)$, the machines too transition in lock-step except when the single frame $\widehat{D_1}$ is either $\widehat{\texttt{pushP}\,\square e}$ or $\widehat{\texttt{takeSC}\,\square\,\lambda_.\, e}$. The two exceptional cases are analogous; we describe the second one. From the definition of $\sim_c$, the related configuration C must have the form $(v, D[D_1], q)$ where $\widehat{v} \sim_e v$, $\widehat{D} \sim_d D$ and D_1 is $(\lambda x.\, \texttt{raise}_{p_0}(\lambda_.\, e, x))\square$. The machine $\mathsf{M}'_{\mathsf{dc}}$ transitions to $\widehat{C'} = (\widehat{\texttt{takeSC}\, v\,\lambda_.\, e, D, q})$. The machine $\mathsf{M}'_{\mathsf{ex}}$ transitions to C' of the form $((\lambda x.\, \texttt{raise}_{p_0}(\lambda_.\, e, x))v, D, q)$. The machine can make another transition, to C'', which is $(\texttt{raise}_{p_0}(\lambda_.\, e, v), D, q)$ and related to $\widehat{C'}$.

The only non-trivial case is $\widehat{C}$ being $(\widehat{\texttt{takeSC}\, p\lambda_.\, e', D, q})$. The related C must have the form $(\texttt{raise}_{p_0}(\lambda_.\, e', p), D, q)$ where $\widehat{e'} \sim_e e'$ and $\widehat{D} \sim_d D$. Suppose that $\widehat{D}$ has the form $D_2[\widehat{\texttt{pushP}\, pD_1}]$ where $\widehat{\texttt{pushP}\, pD'} \notin \widehat{D_1}$. The machine $\mathsf{M}'_{\mathsf{dc}}$ then transitions to $\widehat{C'} = (\widehat{e', D_2, q})$. The definition of $\sim_d$ shows that D has to have the form $D_2[\texttt{try}_{p_0}\, D_1\ \texttt{TH}_p]$. The context D_1 may well include a frame $\texttt{try}_{p_0}\,\square\ \texttt{TH}_{p'}$. Let us suppose it does. Since $\widehat{\texttt{pushP}\, pD'} \notin \widehat{D_1}$ by assumption, we know that p' is different from p. Thus D has the form $D_2[\texttt{try}_{p_0}\, D_1[\texttt{try}_{p_0}\, D'_1\ \texttt{TH}_{p'}]\ \texttt{TH}_p]$. The machine $\mathsf{M}'_{\mathsf{ex}}$ then transitions to $(\texttt{TH}_{p'}(\lambda_.\, e', p), D_2[\texttt{try}_{p_0}\, D_1\ \texttt{TH}_p], q)$ and eventually (see the definition of $\texttt{TH}_q$) to $(\texttt{raise}_{p_0}(\lambda_.\, e', p), D_2[\texttt{try}_{p_0}\, D_1\ \texttt{TH}_p], q)$, which is just like the starting configuration, but with the shorter D_1. Eventually there will be no frame $\texttt{try}_{p_0}\,\square\ \texttt{TH}_{p'}$ in D_1. Then $\mathsf{M}'_{\mathsf{ex}}$ will transition to $(\texttt{TH}_p(\lambda_.\, e', p), D_2, q)$, followed

by $((\lambda_.\, e')(), D_2, q)$ and finally to (e', D_2, q). The latter is related to $\widehat{C'}$.

Conversely, suppose that $\mathsf{M}'_{\mathsf{ex}}$ can make a transition from C. That implies the existence of a $\mathtt{try}_{p_0}\,\Box\, e'$ frame in D. By the definition of $\sim_d$, all such frames have the form $\mathtt{try}_{p_0}\,\Box\;\mathtt{TH}_{p'}$ for some p'. Thus if D has the form $D_2[\mathtt{try}_{p_0}\, D_1\;\mathtt{TH}_{p'}]$ the related $\widehat{D}$ must have the form $D_2[\widehat{\mathtt{pushP}\, p'} D_1]$. In particular, if D has a frame $\mathtt{try}_{p_0}\,\Box\;\mathtt{TH}_{p}$ then $\widehat{D}$ must have the form $D_2[\widehat{\mathtt{pushP}\, p} D_1]$ and $\mathsf{M}'_{\mathsf{dc}}$ can transition from $\widehat{C}$ to some $\widehat{C'}$. The argument in the previous paragraph shows that $\mathsf{M}'_{\mathsf{ex}}$ eventually reaches a related configuration. If D has frames $\mathtt{try}_{p_0}\,\Box\;\mathtt{TH}_{p'}$ but in none of them p' is equal to p, then the related $\widehat{D}$ has no frame $\widehat{\mathtt{pushP}\, p}\Box$ and so $\mathsf{M}'_{\mathsf{dc}}$ is stuck at $\widehat{C}$. The sequence of transitions described in the previous paragraph shows that $\mathsf{M}'_{\mathsf{ex}}$ eventually reaches $(\mathtt{raise}_{p_0}(\lambda_.\, e', p), D', q)$, where D' is the prefix of D that no longer has any $\mathtt{try}_{p_0}\,\Box\;\mathtt{TH}_{p'}$ frame. The machine $\mathsf{M}'_{\mathsf{ex}}$ gets stuck at that point.