Final Shift for Call/cc: Direct Implementation of Shift and Reset

Martin Gasbichler Michael Sperber
Universität Tübingen
{gasbichl,sperber}@informatik.uni-tuebingen.de

Abstract

We present a direct implementation of the `shift` and `reset` control operators in the Scheme 48 system. The new implementation improves upon the traditional technique of simulating `shift` and `reset` via `call/cc`. Typical applications of these operators exhibit space savings and a significant overall performance gain. Our technique is based upon the popular incremental stack/heap strategy for representing continuations. We present implementation details as well as some benchmark measurements for typical applications.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*

General Terms

Languages, Performance

Keywords

Continuations, Implementation, Scheme

1 Introduction

`Call/cc` (or `call-with-current-continuation`) [4] is becoming an established part of the vocabulary of the programming language community. `Call/cc` *captures* or *reifies* the current continuation—the control state of the underlying machine—into an escape procedure and passes it to its argument, which must be a one-argument procedure. The escape procedure, when called, discards its current continuation and *throws* or *reflects* the original continuation back into the machine state.

The translation provided by the Danvy/Filinski CPS transformation [9] gives a definition of the semantics of `call/cc`. The meat of this transformation has a direct counterpart as a semantic specification of the λ calculus; Figure 1 shows such a semantic specification for the bare λ calculus: each ρ is an environment mapping variables to values, and each k is a continuation—a function from values to values. An abstraction denotes a function from an environment to a function accepting an argument and a continuation.

$$\begin{aligned}
[\![x]\!]\rho &= \lambda k.(k\ (\rho\ x)) \\
[\![\lambda x.M]\!]\rho &= \lambda k.k\ (\lambda v.\lambda k'.([\![M]\!])(\rho[x \mapsto v])\ k') \\
[\![E_1\ E_2]\!]\rho &= \lambda k.[\![E_1]\!]\rho\ (\lambda f.[\![E_2]\!]\rho\ (\lambda a.f\ a\ k)
\end{aligned}$$

Figure 1. Continuation semantics

In the context of the semantics, the rule for `call/cc` is this:

$$[\![\mathbf{call/cc}\ E]\!]\rho = \lambda k.[\![E]\!]\rho\ (\lambda f.f\ (\lambda v.\lambda k'.k\ v)\ k)$$

Call/cc E evaluates E and calls the result f; it then applies f to an escape function which discards the continuation k' passed to it and replaces it by the continuation k of the **call/cc** expression.

The applications of `call/cc` are numerous and include the implementation of interpreters and compilers, exception systems, coroutines, multi-threading, and non-deterministic computation. Because of the central role of `call/cc` in these applications, implementors of functional languages have developed representations for continuations enabling fast implementations of `call/cc` [5].

The `control`, `shift` and `reset` operators are a more recent invention [11, 8]. In contrast to `call/cc` that always captures the entire control state, `control`, `shift` and `reset` allow capturing only a portion of one—a so-called *composable continuation*—thus allowing for more flexibility and expressiveness. We focus on `shift` and `reset` in the high-level presentation. We later show how to implement `shift` in terms of the slightly more primitive `control`.

`Reset` introduces a context delimiter; `shift` turns the context between the `shift` construct and the enclosing delimiter into a procedure and removes that part of the context. `Shift` and `reset` are best understood by some examples:

A sole `reset` has no effect:

```
(+ 1 (reset 3)) ⤳ 4
```

`Shift` deletes the context up to the next enclosing `reset`:

```
(+ 1 (reset (* 2 (shift k 4)))) ⤳ 5
```

Here is an single invocation of a composable continuation:

```
(+ 1 (reset (* 2 (shift k (k 4))))) ⤳ 9
```

Multiple invocations are also possible:

```
(+ 1 (reset (* 2 (shift k (k (k 4)))))) ⤳ 17
```

Semantically, `shift` and `reset` have the following definitions in the context of the semantics in Figure 1 [9]:

$$\begin{aligned}
[\![\textbf{reset}\ E]\!]\rho &= \lambda k.k([\![E]\!]\rho\ (\lambda v.v)) \\
[\![\textbf{shift}\ c\ M]\!]\rho &= \lambda k.[\![M]\!](\rho[c \mapsto \lambda v.\lambda k'.k'\ (k\ v)])\ (\lambda v.v)
\end{aligned}$$

Reset seeds the evaluation of E with an empty continuation—the identity function. It then passes the result of evaluating E to k. This does not affect evaluation as long as there is no **shift**: after all, k would be applied to the final result of evaluating E anyway. However, **call/cc** and **shift** expressions evaluated as part of E now see a truncated continuation, as k has been moved out of their view. **Shift** reifies this truncated (or *delimited*) continuation into a function that composes it with its own continuation and binds that function to c. It then evaluates the body expression M within the empty continuation.

A number of applications make inherent use of `shift` and `reset`, among them direct-style formulations of monads [13, 14], continuation-based partial evaluation [21, 22, 26], and type-directed partial evaluation [6]. However, implementations of these control operators are usually formulated in terms of `call/cc`. This is a reasonable and quite portable approach. However, these *indirect* implementations of `shift` and `reset` exhibit suboptimal performance.

Therefore, we have implemented `shift` and `reset` directly. As it turns out, the direct implementation provides significant speedup of applications using these control operators.

Contributions

Here are the contributions of our work:

- We show how to implement `control`, `shift`, and `reset` directly in the context of the popular incremental stack/heap strategy for implementations of continuations, avoiding some of the overhead associated with indirect implementations of these operators via `call/cc`.
- We show that most typical applications of `shift` and `reset` benefit significantly from the performance improvements gained by their direct implementations. We evaluate a number of standard benchmarks. In particular, we show that the direct implementation is an enabling technology for `shift/reset`-based thread systems, avoiding some of the pitfalls of `call/cc`-based implementations.
- We explain how the results of this paper apply to other programming language implementations.

Overview

The paper is organized as follows: The next section reviews the traditional, indirect implementation technique of `shift/reset` via `call/cc`. The following Section 3 explains briefly the general idea of implementing `shift/reset` directly. The two sections after that explain a direct implementation in more depth: Section 4 gives a brief overview of the architectural elements of Scheme 48 involved in the implementation. Section 5 details the direct implementation of `shift/reset` in Scheme 48. Section 6 presents results from a number of benchmarks. Section 7 reviews alternative implementation strategies for continuations, and if and how they may benefit from a direct implementation of `shift/reset`. Section 8 surveys some related work, and Section 9 concludes.

2 `Shift/Reset` and `Call/cc`

The traditional implementation of `shift/reset` [13] involves managing a *meta-continuation*. Meta-continutations arise from CPS-transforming the continuation semantics. The result then gives rise to an implementation of `shift/reset` in terms of `call/cc`. This section reviews this indirect implementation technique.

CPS-transforming the continuation semantics is a natural step as the rule for **reset** in the continuation semantics does not conform to CPS; it contains a non-tail call. Thus, with the rule for **shift** in place, the semantics manages both the usual explicit continuation as well as a new implicit continuation. Their composition yields the usual intuition of the *entire* future of the computation.

$$\begin{aligned}
[\![x]\!]\rho &= \lambda k.\lambda m.k\ (\rho\ x)\ m \\
[\![\lambda x.M]\!]\rho &= \lambda k.\lambda m.k\ (\lambda v.\lambda k'.\lambda m'.[\![M]\!](\rho[x \mapsto v])\ k'\ m')\ m \\
[\![E_1\ E_2]\!]\rho &= \lambda k.[\![E_1]\!]\rho\ (\lambda f.\lambda m'.[\![E_2]\!]\rho\ (\lambda a.\lambda m''.f\ a\ k\ m'')\ m')\ m
\end{aligned}$$

Figure 2. Meta-continuation semantics

CPS-transforming the semantics makes this new implicit continuation, now called the meta-continuation, explicit again. Figure 2 shows the result—a meta-continuation semantics for the λ calculus [8]: the new meta-continuation is called m, and the "old" continuation, k, accepts a meta-continuation as an argument. For standard λ terms, nothing fundamental is changed compared to the continuation-semantics—the new semantics simply threads an additional continuation through the evaluation process.

The meta-continuation does not yet do anything—it is `shift` and `reset` that manipulate it. Here are the rules for `shift/reset`:

$$\begin{aligned}
[\![\textbf{reset}\ E]\!]\rho &= \lambda k.\lambda m.[\![E]\!]\rho\ (\lambda x.\lambda m'.m'\ x)\ (\lambda r.k\ r\ m) \\
[\![\textbf{shift}\ c\ M]\!]\rho &= \lambda k.\lambda m.[\![M]\!](\rho[c \mapsto \lambda v.\lambda k'.\lambda m''.k\ v\ (\lambda w.k'\ w\ m'')]) \\
&\qquad (\lambda w.\lambda m'.m'\ w)\ m
\end{aligned}$$

Just like in the continuation semantics, **reset** seeds the evaluation of E with an empty continuation $(\lambda x.\lambda m'.m'\ x)$ that passes its argument to the meta-continuation, ignoring k. Whereas the continuation semantics applies k "directly," the meta-continuation semantics does this in the meta-continuation. **Shift** also seeds with an empty continuation, and binds c to a function that applies the currently delimited continuation k to its parameter v, calls the result w and passes that to its continuation k'.

Call/cc also has a place in the new semantics:

$$[\![\textbf{call/cc}\ E]\!]\rho = \lambda k.\lambda m.[\![E]\!]\rho\ (\lambda f.\lambda m'.f\ (\lambda v.\lambda k'.\lambda m''.k\ v\ m'')\ k\ m')\ m$$

Note that, just like **shift**, it only reifies the k part of the entire continuation. This means, that **reset** influences **call/cc**.

The meta-continuation semantics is the key for implementing `shift/reset` in terms of `call/cc` and assignment. Olivier Danvy contributed an implementation of `shift/reset` to the Scheme 48 distribution [20], which is closely based on the original SML formulation by Andrzej Filinski [13].

First, macros for reset and shift package up their expression operands as procedures and pass them to procedural cousins *reset and *shift that do the actual work:

```
(define-syntax reset
  (syntax-rules ()
    ((reset ?e) (*reset (lambda () ?e)))))

(define-syntax shift
  (syntax-rules ()
    ((shift ?k ?e) (*shift (lambda (?k) ?e)))))
```

The *meta-continuation* variable holds the current meta-continuation:

```
(define (*meta-continuation* v)
  (error "You forgot the top-level reset..."))
```

Note that *meta-continuation* will be assigned to procedures which *replace* the current continuation. Hence, the *abort procedure has the effect of discarding the current continuation (*k* in the semantics), leaving only the meta-continuation:

```
(define (*abort thunk)
  (let ((v (thunk)))
    (*meta-continuation* v)))
```

*Reset is next. In the meta-continuation semantics, **reset** extends the meta-continuation by the current continuation. *Reset uses the escape procedure created by call/cc as the representation of the current continuation:

```
(define (*reset thunk)
  (let ((mc *meta-continuation*))
    (call-with-current-continuation
      (lambda (k)
        (begin
          (set! *meta-continuation*
                (lambda (v)
                  (set! *meta-continuation* mc)
                  (k v)))
```

*Reset replaces the meta-continuation by one that calls the current continuation after restoring the old meta-continuation. As the meta-continuation is single-threaded through the semantics [13], the assignments have the same effect as the composition in the semantics.

Finally, *reset discards the current continuation; this is exactly what *abort does:

```
          (*abort thunk))))))
```

*Shift must call its argument with a procedure that composes the continuation of the *shift call with its own continuation:

```
(define (*shift f)
  (call-with-current-continuation
    (lambda (k)
      (*abort (lambda ()
                (f (lambda (v)
                     (reset (k v))))))))))
```

The call (k v) discards its own continuation. Therefore, it is surrounded by a reset that moves this continuation into *meta-continuation*, effectively protecting it. Again, *shift must discard its own current continuation with *abort.

Filiniski gives a rigorous derivation of this implementation of shift/reset—it is extensionally equivalent to the meta-continuation semantics. However, intensionally, the implementation is different from the semantics: whereas the semantics has the continuation proper parameterized over the meta-continuation, the implementation uses the underlying call/cc which always reifies the entire machine-level continuation. This is too large, sometimes by a sizable amount, and, as Section 6 demonstrates, causes a significant performance penalty.

It is possible to reduce the dead storage taken up by the representations of composable continuations by having the *abort procedure apply thunk with an empty or very small continuation, thus potentially reclaiming some dead storage early. (In fact, the Scheme 48 version does just that.) However, as it turns out, this optimization does not affect run-time performance in a significant way. (See Section 6.5.)

3 Direct Implementation of shift/reset

Consider shift and reset in the context of a representation for continuations using linked frames. Reset marks a place in the continuation chain which delimits the context later reified by shift. Shift reifies the continuation section up to the enclosing mark created by reset, and creates a procedure that will add this section to the chain. This procedure must also set a mark at the link point in the chain which corresponds to the statically most recent reset for the reified section. Figure 3 displays the evaluation of (shift k e) in terms of linked continution frames: It cuts off the section of the chain consisting of frames Top and Reset and binds the reified procedure to k. The continuation for the evaluation of the body e starts with frame C2.

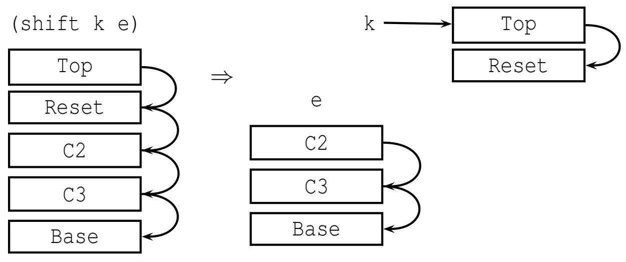

Figure 3. Evaluation of a shift expression

If the continuation chain resides in the heap, this strategy involves copying of continuation frames both during reification and reflection. However, most implementations of Scheme use a representation for continuations involving a stack cache [5] which contains the most recent continuation frames. In these implementations, capturing the current continuation via call/cc flushes the stack cache and moves the continuation frames to the heap.

A direct implementation of shift/reset can exploit the presence of a stack cache in a simple manner: Instead of copying continuation frames one-by-one, shift simply block-copies a slice from the stack into the heap starting at the current continuation up to the frame marked by reset. During reflection, the slice is copied back on the stack. This strategy yields a fast and simple implementation of shift/reset.

Of course, shift and reset are not always this lucky: The marked continuation frame might reside in the heap instead of the stack cache. Also, the reified slice may be bigger than the stack cache. An implementation has to consider these complications, along with several others.

4 Architecture of Scheme 48

We consider a direct implementation of `shift`/`reset` for a pre-release of Scheme 48 1.0 [20], a byte-code implementation of Scheme. Scheme 48 is attractive because its VM is written in Pre-Scheme [19], a low-level dialect of Scheme; this simplifies presenting actual code. This section gives a short overview of the architecture of Scheme 48, highlighting the most important aspects.

4.1 The Incremental Stack/Heap Strategy

Scheme 48 uses the *incremental stack/heap strategy* for representing the current continuation; it stores the most recent continuation frames on the stack, which is thus effectively a cache. Scheme 48 pushes new continuation frames on the stack upon procedure calls and pops them upon return. If the continuation does not fit completely in the stack cache, the early frames reside in the heap.

The bottom frame in the stack cache is always an *underflow* continuation frame. The code of this frame copies its parent from the heap into the stack and invokes the parent afterwards. Hence, every continuation frame can safely return to its parent: the underflow frame will mediate between stack and heap.

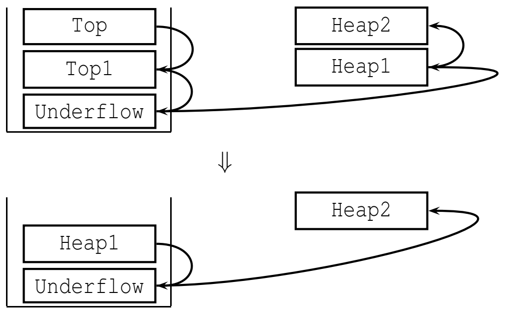

Figure 4. Incremental stack/heap strategy

The top half of Figure 4 shows a typical setup for the incremental stack/heap strategy: The topmost frame of the current continuation `Top` is on the stack with one of its parents `Top1`. Two frames `Heap1` and `Heap2` have been migrated to the heap and linked to the underflow continuation frame (`Underflow`). The bottom half of Figure 4 shows the situation after `Top` and `Top1` have returned: The underflow frame has copied `Heap1` from the heap to the stack.

In Scheme 48, stack continuation frames usually consist of three parts:

1. Optional pointers to the current environment and the current template. The template contains descriptors for constants the code may need.
2. The operands of the currently active procedure call.
3. A code pointer. Scheme 48 maintains the size of the continuation frames in headers in the byte code.

The topmost frame does not contain an explicit code pointer because it is currently executing in the VM. The frames need not be explicitly linked on the stack, as the compiler computes their sizes statically, and places the sizes inside the code.

Scheme 48 creates fresh, flattened environments upon closure creation. It maintains sharing of state by creating heap-allocated cells for shared mutable variables.

A heap continuation object consists of a copy of a stack continuation frame, along with three added values:

1. the code of the continuation—represented as a *code vector*,
2. an offset within the code vector, and
3. a descriptor for the next continuation object.

Heap continuations occur upon `call/cc` or when the stack overflows. In these cases, the VM walks down the stack and creates heap objects for the continuation frames, computing code vector and offset along the way, and chaining the frames together.

Among the various strategies for implementing continuations in the presence of `call/cc`, the incremental stack/heap strategy compares quite favorably [5]. The compact stack frame representation used by Scheme 48 makes the implementation particularly fast.

4.2 The Virtual Machine

The VM is essentially a stack machine. The stack grows towards lower addresses and holds parameters for procedure calls as well as continuation frames and environment frames. The VM manages a number of registers in global variables:

- `*code-pointer*` points to the current instruction
- `*cont*` points to the continuation frame on the stack just beneath the current one
- `*bottom-of-stack*` points to the underflow continuation
- `*heap-continuation*` is the part of the current continuation which resides in the heap
- `*val*` contains the first parameter or a return value

The Scheme 48 VM represents all data objects as 32-bit *descriptors*. Its lower 2 bits indicate the type of the descriptor; the remaining 30 bits are data which are either an immediate representation of a small value or an encoded pointer into the heap.

5 Implementation

This section describes our direct implementation of `shift`/`reset`. All code examples are taken verbatim from our implementation atop the Scheme 48 1.0 Virtual Machine. Our implementation builds straightforwardly upon the stack/heap strategy. However, a few technical complications arise, and therefore our presentation is fairly detailed.

We start with accounts of `reset` and `shift` for the stack-bound case and then show how to extend the approach for continuations residing in the heap.

5.1 `Reset`

`Reset` delimits the section of the continuation that `shift` may capture later. `Reset` evaluates its argument under the empty continuation, and applies its current continuation to the result. Thus, `reset` effectively erects a barrier in the continuation frame chain.

An easy way to erect the barrier is to set a flag in the current continuation frame, and ensure that subsequent calls to `shift` will only reify the part of the continuation chain up to the most recent mark. The compiler merges the continuations of differnet calls into a single frame on the stack whenever possible. Therefore, `reset` has to make sure that the last continuation frame to be reified is represented by a separate frame by creating a thunk and evaluating an unknown call to it.

As a place for the mark, continuation objects receive a further field which is set to zero in normal continuations. A new VM operation `mark-reset-cont` sets the mark in the topmost frame of the current continuation. The `mark-reset-cont` operator is implemented by the `add-reset-marker!` procedure:

```
(define (add-reset-marker!)
  (if (address= *cont* *bottom-of-stack*)
      (if (not (= *heap-continuation* false))
          (set-heap-continuation-marker! *heap-continuation* 1))
      (set-stack-continuation-marker! *cont* 1)))
```

`Add-reset-marker!` checks whether the current continuation is an ordinary frame on the stack or if it is `*bottom-of-stack*` which means that the the current continuation resides in the heap. Due to the different representation of stack and heap continuations, different procedures to mark the frames must be used.

Here is the definition of `reset`:

```
(define-syntax reset
 (syntax-rules ()
   ((reset body ...)
    (call-thunk (lambda () (mark-reset-cont) body ...)))))
```

5.2 **Shift**

`Shift` must do two things: First, it must reify the section of the continuation up to the last `reset` into a heap object. Second, `shift` must construct a procedure that composes the current continuation with the saved section and applies the result to its argument. `Shift` must ensure that the context of the reified continuation is itself delimited. Making this explicit corresponds to implementing `shift` via Felleisen's `control` [11] and is also known as the F- calculus [8].

`Control` is a macro which calls the procedure `control*` with the body wrapped in procedure. `Control*` takes the one-argument procedure as its argument and passes it the composition procedure. `Create-stack-slice` is responsible for turning the section of the continuation into a value. `Copy-slice-to-stack` composes the saved section with the current continuation:

```
(define (control* proc)
  (let ((slice (create-stack-slice)))
    (proc
     (lambda (val)
       (copy-slice-to-stack slice)
       val))))
```

`Control` implements the `(control` *v e*`)` binding form which binds the continuation to *v* and evaluates *e* under that binding:

```
(define-syntax control
  (syntax-rules ()
    ((control c body)
     (control* (lambda (c) (reset body))))))
```

The macro for `shift` relies upon `control` and inserts the needed `reset` to get F-:

```
(define-syntax shift
  (syntax-rules ()
    ((shift c body)
     (control cc (let ((c (lambda (x) (reset (cc x)))))
                   body))))))
```

In our actual implementation we depart from the F- approach for efficiency reasons: Properly supporting multiple-value returns would incur too much overhead converting reifying multiple return values into lists and vice versa. We inline `control` into `shift`:

```
(define-syntax shift
  (syntax-rules ()
    ((shift c body)
     (shift* (lambda (c) body)))))
```

This definition omits the `reset` because `create-stack-slice` will leave the marked continuation on the stack.

Here is the definition of `shift*`:

```
(define (shift* proc)
  (let* ((slice (create-stack-slice))
         (c (lambda vs
              (copy-slice-to-stack slice)
              (apply values vs))))
    (proc c)))
```

The reset omitted here is performed by `copy-slice-to-stack`.

5.2.1 Reification

The following code implements the primitive `create-stack-slice` within the VM:

```
(define-primitive create-stack-slice ()
  (lambda ()
    (let ((v (copy-slice-to-heap)))
      (set! *val* v)
      (goto continue 0))))
```

It calls `copy-slice-to-heap` which returns the slice, loads the value register with it and jumps back into the main interpreter loop via `continue`.

Here is the first part of the relevant procedure, `copy-slice-to-heap`:

```
(define (copy-slice-to-heap)
  (ensure-*cont*-in-stack!)
  (receive (cont is-reset-cont?)
      (find-next-stack-reset-cont)
    (if (address= cont (integer->address 0))
        (create-empty-slice)
        (if is-reset-cont?
            (create-slice-from-stack cont)
            ⟨continued in next section⟩
```

To avoid excessive special casing in the rest of the code, `copy-slice-to-heap` first calls `ensure-*cont*-in-stack` to make sure there is at least one continuation frame on the stack. Then, it calls `find-next-stack-reset-cont` to obtain the last continuation frame to be reified.[1] If the section of the continuation to be reified is empty, `copy-slice-to-heap` simply creates a heap object representing an empty slice via the `create-empty-slice` procedure. Otherwise, the code determines whether the section to be reified indeed resides entirely within the stack. For this section, the

[1]`Receive` is syntactic sugar for `call-with-values`, as described in SRFI 8 (`http://srfi.schemers.org/srfi-8/`). In this example, it binds `cont` and `is-reset-cont?` to the return values of `find-next-stack-reset-cont`.

presentation assumes that it does indeed fit; Section 5.3 discusses the situation when this is not the case.

The find-next-reset-cont procedure walks down the continuation frames on the stack until it encounters a reset mark. It then returns the parent frame or 0 if the current continuation is marked. If find-next-reset-cont does not find a marked continuation on the stack, it returns the last continuation frame on the stack (the parent of *bottom-of-stack*). The second return value indicates the case which occurred by a boolean flag:

```
(define (find-next-stack-reset-cont)
  (let lp ((cont *cont*) (prev (integer->address 0)) (i 0))
    (if (address= cont *bottom-of-stack*)
        (values prev
                (not (heap-continuation-marker-zero?
                       *heap-continuation*)))
        (if (not (= (stack-continuation-marker cont) 0))
            (values prev #t)
            (lp (stack-cont-continuation cont) cont (+ i 1))))))
```

Create-slice-from-stack performs the actual work by copying the stack slice represention the section of the continuation to a heap object *en bloc*.

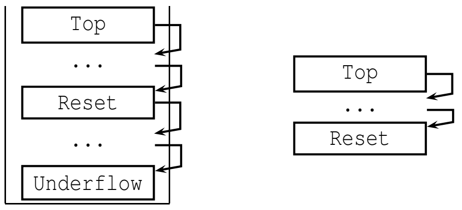

Figure 5. Representing continuation chains with offsets

Scheme 48 uses relative offsets rather than absolute addresses for the intra-stack references as shown in Figure 5. Offsets are represented by arrows with angles. The continuation reset in the heap has no parent in the copied slice.

Stack slices present a challenge to the garbage collector; the Scheme 48 GC ordinarily only handles heap objects consisting entirely of descriptors or entirely of bitmap data; stack slices contain both. Accordingly, we have extended the garbage collector by a custom trace procedure. The first continuation frame of a stack slice is the root of the slice.

Here is the code for create-slice-from-stack:

```
(define (create-slice-from-stack reset-cont)
  (let* ((len (stack-slice-size reset-cont))
         (key (ensure-space (needed-bytes-for-stack-slice len)))
         (slice (make-stack-slice len key)))
    (copy-slice! slice len)
    (set! *cont* (stack-cont-continuation reset-cont))
    slice))
```

Create-slice-from-stack allocates a slice object and calls copy-slice! to copy the stack section into it. After this, it sets the current continuation to the parent of reset-cont, also setting the reset mark. This corresponds to the deletion of the section of the continuation reified by shift. Create-slice-from-stack returns the created slice.

Copy-slice! calls the Pre-Scheme primitive copy-memory! which is translated into C's memcpy:

```
(define (copy-slice! slice len)
  (let ((to-address (address-after-header slice))
        (start *cont*))
    (copy-memory! start to-address len)))
```

5.2.2 *Reflection*

Copy-slice-to-stack is comparatively simple:

```
(define (copy-slice-to-stack! slice)
  (add-reset-marker!)
  (if (double-slice? slice)
      (install-double-slice! slice)
      (if (not (empty-slice? slice))
          (set! *cont* (really-copy-slice-to-stack slice)))))
```

This first adds a reset marker to the current continuation frame as described in Section 5.2. The first conditional determines if the slice about to be reflected will not fit within the stack cache; this exceptional case is described in the next section. For empty slices, nothing needs to be done. For normal slices, the work is relegated to the really-copy-slice-to-stack procedure:

```
(define (really-copy-slice-to-stack slice)
  (save-temp0! slice)
  (let* ((len (b-vector-length slice)))
    (ensure-stack-space! (bytes->cells len))
    (let ((slice (recover-temp0!))
          (new-cont (address-
                      (address+ *stack*
                                (cells->a-units
                                  (operands-on-stack)))
                      (bytes->a-units len))))
      (copy-args-above-incoming-cont! new-cont
                                      (operands-on-stack))
      (copy-slice-bytes slice new-cont)
      new-cont)))
```

Ensure-stack-space! ensures there is enough place on the stack, flushing the stack cache to the heap if necessary. If this triggers a garbage collection, save-temp0! tells the collector that slice is live, and recover-temp0! recovers it if the collector has moved the slice. Next, the code computes the target address for the topmost continuation frame of the slice: This is len bytes below the current top of the stack plus the operands that are currently lying on the stack. These operands must be moved from their current location to the new top of the stack which is just above new-cont. Copy-args-above-incoming-cont! is responsible for this. Now there is room on the stack, and the continuation current prior reflection starts len bytes away from new-cont. Copy-slice-bytes then copies the actual data.

The copy-slice-bytes computes the source address and the number of bytes to be copied from the slice and simply calls copy-memory! to perform the copying.

```
(define (copy-slice-bytes slice to-start)
  (let* ((from-start (address-after-header slice))
         (len (b-vector-length slice)))
    (copy-memory! from-start to-start len)))
```

5.3 Reifying Heap Continuation Frames

The previous discussion assumed that the continuation section to be reified resides entirely within the stack. As described in Section 4.1, continuation frames are migrated to the heap upon call/cc or in the case of a stack overflow. The direct implementation of shift must cope with two problems in this situation:

1. The block-copy strategy is no longer sufficient. Instead, the VM needs to to copy heap continuation frames sequentially into the slice.
2. The resulting slice may be larger than the stack cache which means a simple block copy is not enough to implement reflection.[2]

The remaining code of the `copy-slice-to-heap` procedure deals with these cases; it first searches for the reset continuation in the heap and then branches on the condition of the second problem:

```
(receive (reset-cont
          required-size-on-stack
          required-size-on-heap)
    (find-next-heap-reset-cont)
  (if (< (+ (current-stack-size)
            required-size-on-stack) ; approximation
         (maximum-stack-size))
      (create-slice-in-two-steps cont
                                 reset-cont
                                 required-size-on-stack)
      (create-double-slice cont
                           reset-cont
                           required-size-on-heap)))))))
```

`Find-next-heap-reset-cont` returns the continuation along with the amount of space required to copy these continuations to the stack or to the heap, respectively. Due to the different representations of continuations these numbers differ.

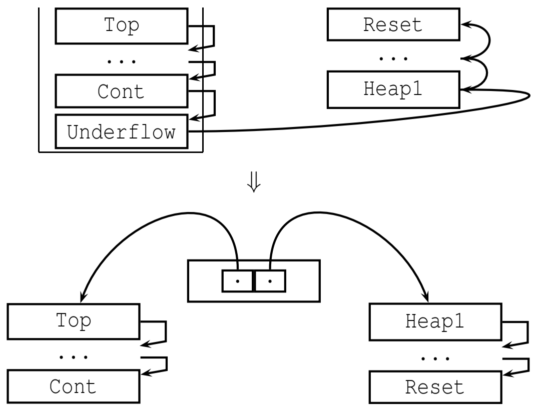

Figure 6. Creating a double slice

If a part of the section resides in the heap but the entire section is not larger than the maximum size of the stack, `create-slice-in-two-steps` creates a slice big enough to hold the chain and fills it. If the section is larger than the stack, `create-double-slice` creates a special object (a *double slice*) that holds two slices: one for the stack, the other for the continuation frames that must go to the heap. Figure 6 shows the creation of a double slice. During reflection of a double slice, only the first slice is copied on the stack, while a copy of the second is linked to the bottom frame. Note that copying the heap frames (and thus breaking sharing) is necessary to modify the last continuation's parent during reflection.

The code for handling heap continuation frames is mostly as in the original Scheme 48. We have therefore omitted it here.

[2]Note that this issue does not occur with `call/cc`: The underflow frame simply copies continuation frames residing in the heap into the stack cache one-by-one.

6 Benchmarks

This section describes a number of benchmarks of the direct implementation of `shift`/`reset`, notably:

- Filinski's representation of monads
- combinator-based partial evaluation
- type-directed partial evaluation

We have also used `shift`/`reset` to implement a thread system. The first set of benchmarks shows that all of these applications exhibit absolute performance gains under the direct implementation. The direct implementation of `shift`/`reset` is an enabling technology for implementing efficient thread systems via `shift`/`reset`.

It is difficult to precisely account for the speedup in each separate case, as the indirect and direct implementations of `shift`/`reset` lead to very different access patterns to the stack and to the heap: The indirect implementation tends to flush the stack cache often, keeping only the (non-meta-)continuation within the cache. Each flush is accompanied by a representation change. In contrast, many reifications in the direct implementation do not flush the stack cache, and do not involve a representation change. The direct implementation potentially performs less sharing than the indirect implementation. Moreover, speed-ups necessarily vary with the amount of computation performed between reifications and reflections of continuations. However, across the benchmarks, the direct implementation performs uniformly better than the indirect one.

Note that all performance gains were obtained under a comparatively slow evaluator. These applications, when run under a native-code implementation of Scheme should benefit significantly more, as there is almost no interpretive overhead in executing `shift` and `reset`. We expect to obtain timings from the upcoming native-code version of Scheme 48 soon.

All timings were obtained on a Pentium III system with 666 Mhz and 128 MB RAM, running FreeBSD 4.3. We used our modified version of a prerelease of Scheme 48 1.0 with an initial heap size of 80 MB and the standard stack size of 10 Kb; none of the tests triggered a garbage collection.

6.1 Monads

The essence of Filinski's work on representing monads [13] is the introduction of two combinators for conversion between value-level expressions and meta-level computations. `Reify` uses the monadic constructor `eta` to turn a computation into an expression. `Eta` uses `reset` to limit the extent of the computation. Its dual operation, `reflect`, calls the monadic combination function `extend` to apply a computation (which corresponds to a section of the continuation obtained by `shift`) to an expression:

```
(define (reflect meaning)
  (shift k (extend k meaning)))
(define (reify thunk)
  (reset (eta (thunk))))))
```

We have run applications using the parsing monad and the ambivalence monad. For benchmarking monadic parsing, we constructed a parser for arithmetic expressions similar to Hutton's [18] and applied it to a term with about 450 operators.

The ambivalence monad and the test expressions we used are defined as:

```
(define (eta x) (list x))
(define (extend f l) (apply append (map f l)))

(define-syntax amb (syntax-rules ()
    ((amb x ...) (amb* (lambda () x) ...))))
(define (amb* . t)
  (reflect (apply append (map reify t))))))

(define (www)
  (let ((f (lambda (x) (+ x (amb 6 4 2 8) (amb 2 4 5 4 1)))))
    (reify (lambda () (f (f (amb 0 2 3 4 5 32)))))))
(define (wwww)
  (let ((f (lambda (x) (+ x (amb 6 4 2 8) (amb 2 4 5 4 1)))))
    (reify (lambda () (f (f (f (amb 0 2 3 4 5 32))))))))
```

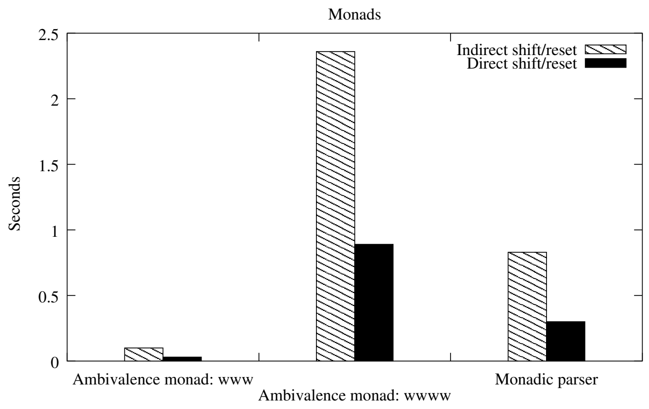

Figure 7. Timings for monads (in seconds)

Figure 7 shows the timings for the monad examples. The bars labeled "Indirect" list the results for the `call/cc`-based implementation of `shift`/`reset`, whereas the column labeled "Direct" contains the timings for our direct implementation of `shift`/`reset`. The speedups for these very `shift`/`reset`-intensive benchmarks are in the range of a factor of three. To see where the direct implementation improves the performance consider Figure 8 which shows the number of bytes copied from the stack to the heap and vice versa. Our direct implementation copies significantly fewer bytes. Another improvement shown in Figure 9 is the number of copy operations. It drops by a factor of three to five from the indirect to the direct implementation, whereas the average number of bytes moved by a single `copy-memory!` increases from 12 to 20 bytes.

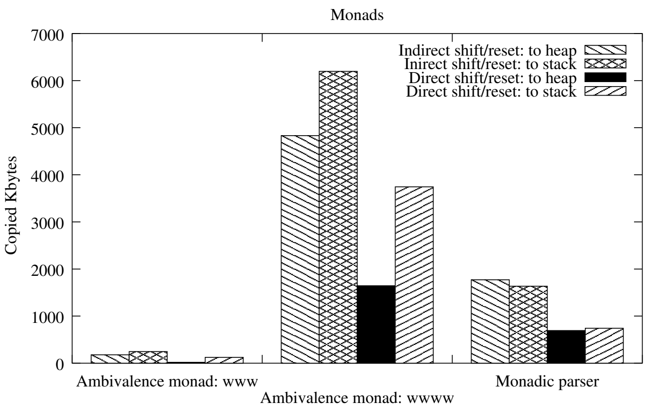

Figure 8. Continuation frames copied for monads (in Kbytes)

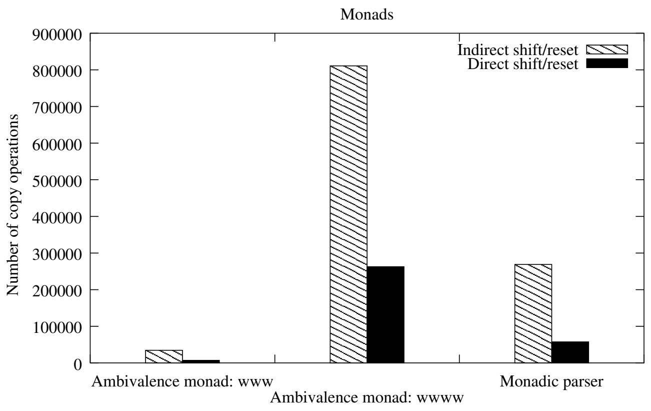

Figure 9. Copy operations for monads

6.2 Combinator-Based Partial Evaluation

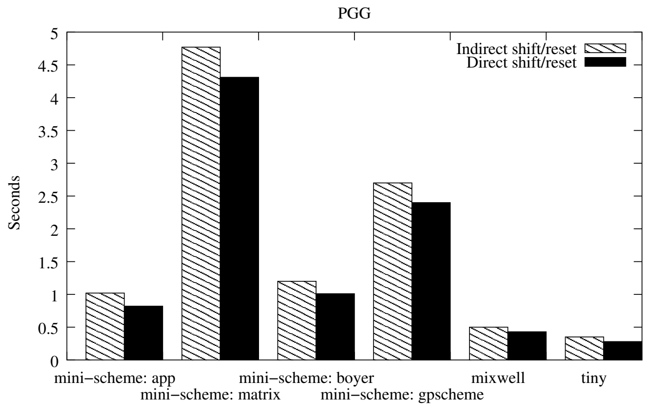

Figure 10. Timings for combinator-based partial evaluation (in seconds)

A partial evaluator takes a program as input and specializes it with regard to some of the arguments of the program. Continuation-based approaches to partial evaluation use composable continuations to represent dynamic context accumulated during specialization [21]. Thiemann's PGG system [27] uses a direct-style version of this transformation implemented via `shift`/`reset`.

We have used benchmarks from Helsen's and Thiemann's paper on comparing combinator-based and type-directed partial evaluation [16]. These benchmarks are interpreters; partial evaluation yields compiled versions of the input programs. These are the programming languages accepted by the interpreters:

Mixwell is a first-order purely functional language.

Tiny is a small imperative language with expressions, assignment, `if`, `while` and sequencing.

Mini-Scheme is a large subset of Scheme, lacking only `call/cc` and support for multiple return values.

We used a standard test program for the Mixwell interpreter, and the factorial procedure for the Tiny interpreter. The Mini-Scheme interpreter was specialized to several programs from Andrew Wright's benchmark suite for Scheme: `app`, the `append` procedure, `boyer`, a term-rewriting system, `matrix` that tests a random matrix to be maximal under permutation and negation of rows and columns, and `gpscheme`, an implementation of genetic programming.

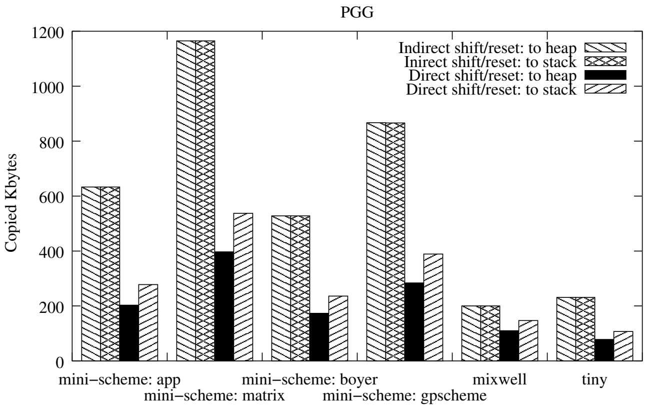

Figure 11. Continuation frames copied for combinator-based partial evaluation (in Kbytes)

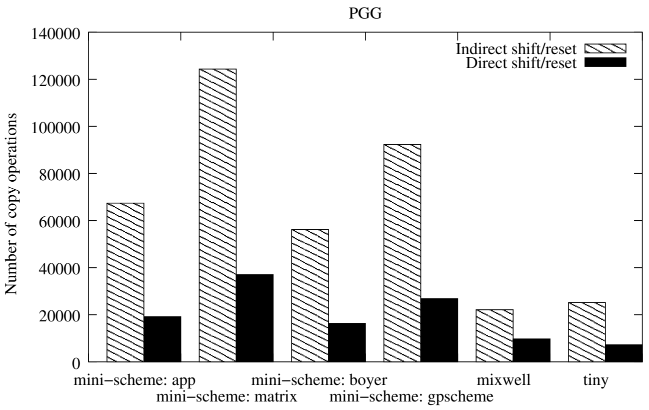

Figure 12. Copy operations for combinator-based partial evaluation

Figure 10 contains the specialization timings. Here, the speedup is quite uniformly between 10% and 20%. Again, Figure 11 shows the summarized size of the moved continuation frames. It is interesting to note that indirect approach copies exactly the same amount of memory from the stack as vice versa whereas our direct implementation moves more memory from the heap to the stack. Figure 12 shows the number of memory copy operations during the benchmarks. Our direct approach typically cuts down the number of operations to a third.

6.3 Type-Directed Partial Evaluation

Danvy's *type-directed partial evaluation* (*TDPE*) [6] is an alternative approach to partial evaluation which operates on the compiled version of the subject program. TDPE also uses `shift`/`reset` to capture context. We have applied TDPE to the same examples as PGG. Figure 13 shows the result. The speedups obtained were similar to those obtained with PGG. Figure 14 shows a peculiarity: For the specialization of the Mixwell interpreter, our implementation copies an order of magnitude more than the implementation based on `call/cc` but is still faster. The speedup can be explained by the number of copying operations: Figure 15 shows that this number drop by a factor of three. This also dramatically increases the bytes per copy operation from 12 in the indirect implementation to 354 in our implementation.

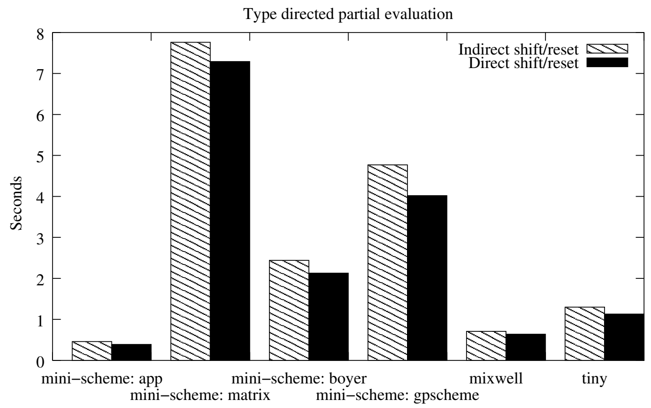

Figure 13. Timings for type-directed partial evaluation (in seconds)

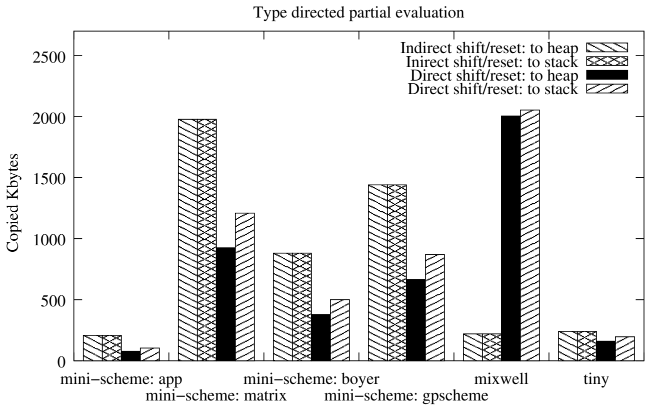

Figure 14. Continuation frames copied for type-directed partial evaluation (in Kbytes)

6.4 Threads

The use of first-class continuations for efficient implementations of thread systems is not new: On a context switch, the system reifies the continuation of the running thread and stores it within the scheduler. Next, the scheduler replaces the current continuation by the stored continuation of another thread, thus invoking that thread. However, `call/cc` reifies not just the continuation of the thread but also the continuation of the scheduler. This also can lead to a space leak [2]. In a thread system using `shift` and `reset`, it is possible to delimit the continuation of the thread with a `reset`.

To measure the effects of this approach, we have modified the thread system of Scheme 48 to use `shift`/`reset`. As the continuation of a thread may receive multiple values it was necessary to extend the implementation. Instead of using variable-arity procedures in the implementation described in Section 5, we built directly on the slice-copying operations provided by the VM just as the stock Scheme 48 system uses VM-level continuation primitives.

If the user has access to `shift`/`reset`, a `reset` would prevent the thread system's `shift` from finding the reset mark delimiting the continuation of the thread [2]. This could easily be remedied in our implementation by changing the reset mark to be the thread uid of the thread which set the mark. As the scheduler is a thread by itself, it does not interfere with the other threads.

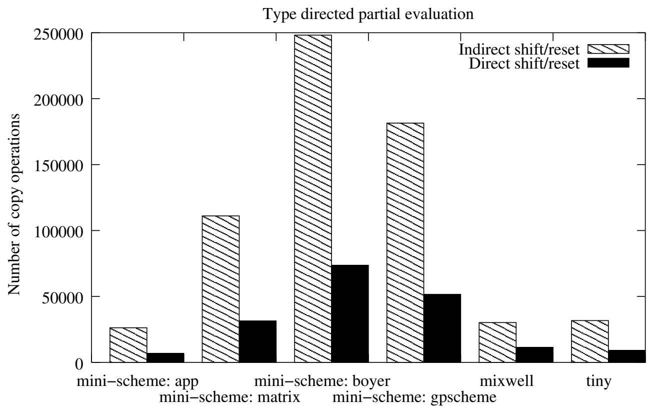

Figure 15. Copy operations for type-directed partial evaluation

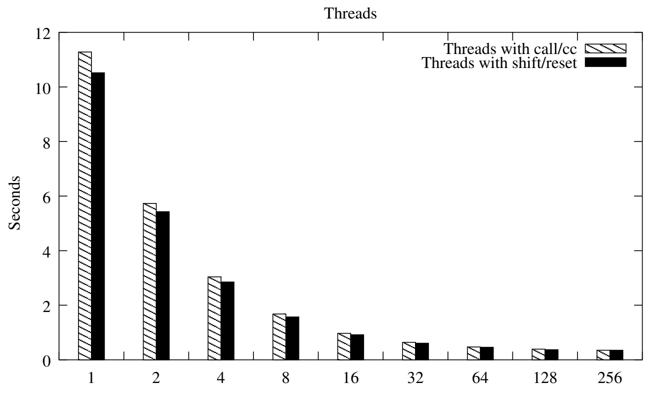

Figure 16. Timings for the thread benchmark (in seconds)

For the actual measurements we adopt the benchmarks by Bruggeman et al. who use one-shot continuations for implementing threads [3]. Each thread computes the 20th Fibonacci number. We vary the number of procedure calls between two context switches for a fixed number of 100 threads. Figure 16 shows the resulting timings. Just like Bruggemann et. al. we observe a speedup only for very frequent context switches whereas the timings are equal when the context switches happen at a lower rate than every 64 procedure calls. Figure 17 shows that the number of copied bytes is much larger in the `shift`/`reset` case than in the `call/cc` implementation. The number of copy operations, however, is linear to the number of context switches in the `shift`/`reset` case as Figure 18 reveals. This figure also shows that the standard `call/cc` approach cannot hold this limit when context switches become less frequent as the running thread is more likely to return to a continuation on the heap the longer it runs. As the context-switch rate decreases, the pure execution time of the threads dominates, yielding similar performance with both approaches. These observations coincide with those of Bruggeman at al.

6.5 Optimizing `*Abort`

As mentioned in Section 2, the indirect implementation of `shift`/`reset` in Scheme 48 uses a version of the `*abort` primitive that discards its entire continuation before calling the meta-continuation. This enables the garbage collector to reclaim more unreachable continuation frames. However, this optimization does not affect the run-time performance of the benchmarks: The dead frames just reside in the heap, and the program never touches them.

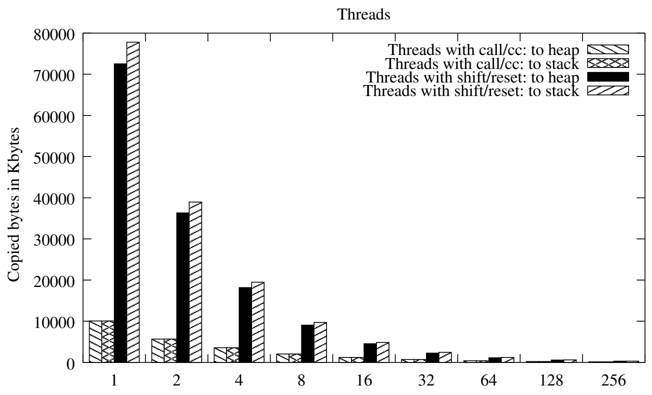

Figure 17. Continuation frames copied for the thread benchmark (in Kbytes)

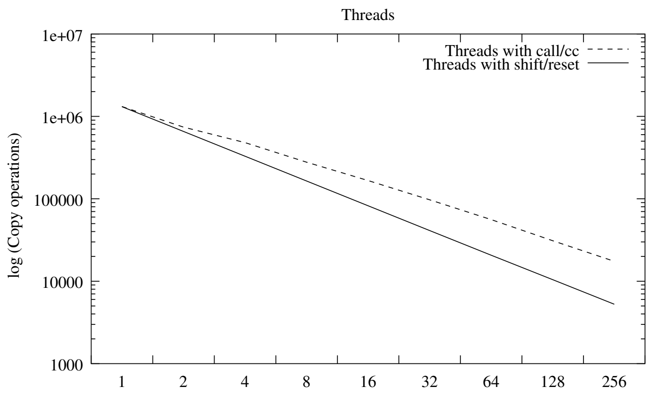

Figure 18. Number of copy operations for the thread benchmark

7 Other Implementation Strategies

The incremental stack/heap strategy employed by Scheme 48 is not the only common implementation strategy for continuations in the presence of control operators. The main attractive alternative implementations are the *gc strategy*, the *Hieb-Dybvig-Bruggeman strategy*, and the *stack/heap strategy*, along with the simple-to-implement *stack strategy* [5]. This section briefly reviews how the results obtained for this paper carry over to these alternatives.

7.1 The GC Strategy

Implementations using the gc strategy keep all continuations in the heap, relying on a fast garbage collector and compile-time optimizations for a fast implementation. The strategy is not a zero-overhead strategy in the sense of Clinger et al. [5], meaning that it incurs overhead for *all* programs over stack-based strategies.

In the gc strategy, `call/cc` as well as reflecting a continuation are especially fast and incur only small constant run time. In particular, `call/cc` involves no copying. In contrast, a direct implementation of `shift`/`reset` would need to copy continuation frames corresponding to a composable continuation, which likely makes a direct implementation more expensive in terms of run time than the indirect implementation. The only potential savings are space savings due to not retaining dead meta-continuations (see Section 2).

7.2 The Hieb-Dybvig-Bruggeman Strategy

The Hieb-Dybvig-Bruggeman strategy [17] represents continuations as linked lists of stack segments. When the current stack segment overflows, the system simply allocates a fresh one, linking it to the previous segment. `Call/cc` creates a small object pointing to the current frame within a stack segment. It then splits the segment by installing an underflow continuation frame, similar to the incremental stack/heap strategy. Reflecting the continuation also works as in the incremental stack/heap strategy, by copying a portion of the captured continuation into the current stack segment. As in the gc strategy, `call/cc` itself is a constant-time operation. On the other hand, reflecting a reified continuation involves overhead similar to the incremental stack/heap strategy.

A direct implementation of `shift`/`reset` would create a stack segment corresponding to the reified composable continuation, terminating the link at the bottom. The system would reflect the continuation by making a copy of the stack segment and setting its link pointer. It is hard to assess the tradeoffs of a direct implementation of `shift`/`reset` in this case; this warrants further research.

7.3 The Stack/Heap Strategy

The stack/heap strategy differs from the incremental stack/heap strategy in that it can return to a continuation residing in the heap directly, instead of first copying portions back into the stack cache. This means that a continuation resides either entirely within the stack or entirely within the heap. The stack/heap strategy is also not a zero-overhead strategy because each return needs to check where the continuation resides. A direct implementation of `shift`/`reset` in this context would be similar to the one presented here, with similar performance tradeoffs as for the implementation of `call/cc`.

The stack/heap strategy incurs an additional overhead for procedure call returns, as it needs to check whether the current continuation resides in the stack or in the heap. Hence, it is not a zero-overhead strategy. The stack/heap strategy seems comparatively rare in contemporary programming language implementations.

7.4 The Stack Strategy

In implementations of programming languages which do not support first-class continuations, the most common strategy is the *stack strategy* that keeps all continuation frames on a global stack. The stack strategy is even used in some systems with `call/cc` (MzScheme, Bigloo and scm, for example.) This strategy effectively discourages the use of `call/cc`. However, a direct implementation of `shift`/`reset` for the stack strategy is quite simple as it can use a contiguous representation for reified continuations. In particular, it does not have to deal with the stack-overflow situation. Hence, we expect the relative performance benefits of a direct implementation with this strategy to be significantly greater than with strategies allowing an efficient implementation of `call/cc`. This in turn might make `shift`/`reset` attractive for programming languages which do not support `call/cc` at all, such as Objective Caml.

8 Related Work

Felleisen et al. originally came up with ideas for control operators for composable continuations [12, 11]. Danvy and Filinski discovered `shift` and `reset` in the course of their investigation of the CPS transformation [8]. The seminal work on the CPS transformation and on `shift`/`reset` is Danvy's and Filinski's 1992 paper [9]. Filinski shows how to implement `shift`/`reset` via `call/cc` and a mutable reference [13]. Danvy and Filinski note that `shift`/`reset` coincides with Sitaram's and Felleisen's F- operationally [24, 8]. Moreau and Queinnec also employ marks on the stack to define `marker`/`call/pc`, another pair of control operators for composable continuations [23]. Gunter, Rémy, and Riecke investigate another alternative approach to composable continuations via named prompts [15]. They also mention the possibility of a direct implementation. To our knowledge, this has not been pursued to date.

Clinger, Hartheimer and Ost offer a comprehensive account of efficient implementation strategies for first-class continuations [5] and present detailed comparative measurements. Danvy formalizes two stack-based implementation strategies for first-class continuations by constructing special abstract machines which are proven equivalent to a standard abstract machine for CPS programs [7].

The performance problems of indirect implementations of `shift`/`reset` have been noted for a while [1]. In particular, implementors of partial evaluators have been trying to replace the use of `shift`/`reset` for performance reasons—this is possible for some but not all applications of `shift`/`reset` in that context [25]. Operating systems research also has a long history of trying to overcome the inefficiencies stemming from capturing and reinstating complete continuations. Draves allows kernel functions to specify a composable continuation explicitly for a context switch instead of forcing the kernel substrate to capture the complete continuation [10]. This corresponds to using `shift`/`reset` instead of `call/cc` for context switching.

9 Conclusion

`Shift` and `reset` have long gained a critical mass of applications to warrant first-class support in modern functional programming languages. However, research on the pragmatics of supplying and efficiently implementing `shift`/`reset` has only just begun. Our work is a first indicator that direct implementations of `shift`/`reset` as opposed to indirect ones using `call/cc` are indeed worthwhile: the efficiency gains for applications of `shift`/`reset` are significant. However, more research is needed, in particular into the pragmatics of having `call/cc`, `shift`/`reset`, and threads in the same system, as well as in alternative implementation strategies, possibly involving more advanced garbage-collection technology or exploiting representations derived from extended continuation-passing style. Another area for future work is relating our direct implementation formally to the semantic specification of `shift`/`reset`. Danvy's work [7] should provide a good starting point for this.

Acknowledgments

We thank Simon Helsen and Peter Thiemann for providing us with their benchmark suite. Richard Kelsey always kept us supplied with the latest development releases of Scheme 48. Olivier Danvy provided us with helpful discussions about `shift`/`reset` as well as excellent suggestions on how to improve our presentation. We also thank Matthias Felleisen for valuable insights and pointers.

10 References

[1] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'98)*, number 1473 in Lecture Notes in Computer Science, Kyoto, Japan, March 1998.

[2] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, December 1998.

[3] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 99–107, Philadelphia, PA, USA, May 1996. ACM Press.

[4] William D. Clinger, Dan P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In John Reynolds and Maurice Nivat, editors, *Algebraic Methods in Semantics: Proceedings of the US-French Seminar on the Application of Algebra to Language Definition and Compilation*, pages 237–250, Cambridge, 1985. Cambridge University Press.

[5] William D. Clinger, Anne Hartheimer, and Eric Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 1(12):7–45, April 1999.

[6] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, number 1706 in Lecture Notes in Computer Science, pages 367–411. Springer-Verlag, Copenhagen, Denmark, 1999.

[7] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Proc. 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 88–103, Berlin, Germany, March 2000. Springer-Verlag.

[8] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, 1990. ACM Press.

[9] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.

[10] Richard P. Draves. *Control Transfer in Operating System Kernels*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, May 1994. CMU-CS-94-142.

[11] Matthias Felleisen. The theory and practice of first-class prompts. In *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pages 180–190, San Diego, California, January 1988. ACM Press.

[12] Matthias Felleisen, Dan P. Friedman, Bruce Duba, and J. Merrill. Beyond continuations. Technical Report 216, Computer Science Department, Indiana University, Bloomington, Indiana, 1987.

[13] Andrzej Filinski. Representing monads. In *Proceedings of the 1994 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 446–457, Portland, OR, January 1994. ACM Press.

[14] Andrzej Filinski. Representing layered monads. In Alexander Aiken, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, USA, January 1999. ACM Press.

[15] Carl Gunter, Didier Rémy, and John Riecke. A generalization of exceptions and control in ML-like languages. In Simon Peyton Jones, editor, *Proc. Functional Programming Languages and Computer Architecture 1995*, pages 12–23, La Jolla, CA, June 1995. ACM Press, New York.

[16] Simon Helsen and Peter Thiemann. Two flavors of offline partial evaluation. In J. Hsiang and A. Ohori, editors, *Advances in Computing Science - ASIAN'98*, number 1538 in Lecture Notes in Computer Science, pages 188–205, Manila, The Philippines, December 1998.

[17] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. Conference on Programming Language Design and Implementation '90*, pages 66–77, White Plains, New York, USA, June 1990. ACM.

[18] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.

[19] Richard A. Kelsey. Pre-Scheme: A Scheme dialect for systems programming. June 1997.

[20] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[21] Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 227–238, Orlando, Florida, USA, June 1994. ACM Press.

[22] Julia Lawall and Olivier Danvy. Continuation-based partial evaluation. Technical Report Technical Report CS-95-178, Brandeis University, Waltham, Massachusetts, 1995. Extended version of [21] from `ftp://ftp.brics.dk/pub/danvy/Papers/lawall-danvy-lfp94-extended.ps.gz`.

[23] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations — a duumvirate of control operators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '94)*, number 844 in Lecture Notes in Computer Science, Madrid, Spain, September 1994. Springer-Verlag.

[24] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, January 1990.

[25] Eijiro Sumii and Naoki Kobayashi. A hybrid approach to online and offline partial evaluation. *Higher-Order and Symbolic Computation*, 14(2/3):101–142, 2001.

[26] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, September 1999.

[27] Peter Thiemann. *The PGG System—User Manual.* Universität Freiburg, Freiburg, Germany, March 2000. Available from `http://www.informatik.uni-freiburg.de/proglang/software/pgg/`.