Direct Implementation of Shift and Reset in the MinCaml Compiler

Moe Masuko Kenichi Asai

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

Abstract

Although delimited control operators are becoming one of the useful tools to manipulate flow of programs, their direct and compiled implementation in a low-level language has not been proposed so far. The only direct and low-level implementations available are Gasbichler and Sperber's implementation in the Scheme 48 virtual machine and Kiselyov's implementation in the OCaml bytecode. Even though these implementations do provide an insight into how stack frames are composed, they are not directly portable to compiled implementation at the assembly language. This paper presents a direct implementation of delimited control operators `shift` and `reset` in the MinCaml compiler. It shows all the details of how composable continuations can be implemented in the PowerPC microprocessor using the stack strategy. We also show an implementation that copies the stack frames lazily. To our knowledge, this is the first implementation of `shift/reset` in the assembly language. It makes clear at the assembly language level what we have informally described so far, such as "copying and composing stack frames" and "inserting a reset mark when captured continuations are called". We demonstrate various benchmarks to show the performance of the direct implementation and discuss its pros and cons.

1 Introduction

A continuation is a notion of the rest of the work to be done after the current calculation. By manipulating continuations, we obtain control over flow of evaluation. Well-known examples of using continuations include exception handling, non-local jump from nested function calls, coroutines, non-deterministic programming [5], the typed `printf` function [2], dynamic code generation [11], and let-insertion in partial evaluation [1].

To provide users with the ability to manipulate continuations without changing the program globally into continuation-passing style (CPS), various control operators have been proposed, such as `call/cc` found in Scheme [14] (and Ruby [13]), `control/prompt` [8], and `shift/reset` [5].

Among them, this paper focuses on the delimited control operators, `shift` and `reset`, introduced by Danvy and Filinski and shows their direct implementation at the PowerPC assembly language level in the MinCaml compiler [15]. The current goals and contributions of our research are twofold:

1. to establish how to implement composable continuations in the low-level language

It has been long said that composable continuations can be implemented by "cutting and copying stack frames up to a reset mark" and "reinstating the frames back to the stack top". These informal statements help us understand the behavior of composable continuations, but the relationship to the actual implementation in a low-level language is not at all obvious. In particular, a stack in the real implementation typically contains the return address of a functional call. Without showing how to connect the call chains of functions, we cannot implement composable continuations in a low-level language.

Our first contribution is to present all the details of how `shift/reset` can be implemented in the PowerPC assembly language using the stack strategy. Although written in the assembly language, the implementation is simple enough for general audience, showing the essence of the implementation. Because of its simplicity, we think that the presented implementation can serve as a reference implementation of `shift/reset`.

We employ the raw PowerPC assembly language rather than other higher-level assembly languages, such as C-- [12], because another level of abstraction introduced by such languages could hide subtle but important implementation details of `shift/reset`.

2. to provide a testbed on which performance and implementation methods of `shift/reset` can be discussed

Emulated implementation of `shift/reset`, such as Filinski's implementation in terms of `call/cc` [9] suffers from the efficiency problem. Gasbichler and Sperber reported that emulating `shift/reset` using `call/cc` leads to significant performance penalty compared to the direct implementation [10]. Thus, for wider uses of `shift/reset`, it is mandatory to provide an efficient implementation. Furthermore, the lack of a direct implementation prohibits us from discussing on the efficiency of programs using `shift/reset`.

Our second contribution is to provided a testbed on which we can discuss the performance of programs with `shift/reset` as well as implementation methods of `shift/reset`. Only through experiments performed on the direct implementation can we compare if a program with `shift/reset` is faster or slower than its CPS counterpart. We can also investigate various other implementation strategies based on the one shown in the paper for better performance. As an example, we show the lazy implementation in this paper.

We do not intend that the two implementations shown here are the ideal implementations of `shift/reset`. The present work is still ongoing, and lacks some important aspects, such as garbage collection. However, we believe that the paper sets out the basic platform from which various implementations can be built and compared. More sophisticated techniques become only possible through a clear and succinct base implementation, which this paper offers.

Overview

The paper is organized as follows. Section 2 introduces the delimited continuation constructs `shift/reset` and shows some examples. Section 3 presents an outline of the MinCaml compiler, which our direct implementation is based on. Section 4 describes the direct implementation of `shift/reset` in the MinCaml compiler and shows some benchmarks. Section 5 gives the lazy implementation as an alternative implementation together with another benchmarks. Section 6 shows related work, and Section 7 concludes. Appendix A briefly reviews PowerPC's instructions used in the paper.

2 Shift and Reset

Figure 1 shows the continuation semantics of the lambda calculus extended with `shift/reset` [6, 7]. The expression `shift`$(\lambda k.M)$ captures the current continuation κ, binds it to k, and executes the

$$\begin{aligned}
[\![x]\!] &= \lambda\kappa.\kappa\ x \\
[\![\lambda x.M]\!] &= \lambda\kappa.\kappa\ (\lambda x.[\![M]\!]) \\
[\![M\ N]\!] &= \lambda\kappa.[\![M]\!]\ (\lambda m.[\![N]\!]\ (\lambda n.m\ n\ \kappa))
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{shift}(\lambda k.M)]\!] &= \lambda\kappa.([\![M]\!]\ (\lambda m.m))[k \mapsto \lambda a.\lambda\kappa'.\kappa'(\kappa\ a)] \\
[\![\texttt{reset}(\lambda\_.M)]\!] &= \lambda\kappa.\kappa\ ([\![M]\!]\ (\lambda m.m))
\end{aligned}$$

Figure 1: Continuation semantics and `shift`/`reset`.

expression M with an empty continuation—the identity function. The expression `reset`$(\lambda\_.M)$ executes the expression M with an empty continuation and passes the return value to the continuation κ of `reset`. Intuitively, `shift` captures the current continuation and `reset` delimits the scope of the continuation captured by `shift`. For example:

```
1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))
⇒ 1 + (3 + 2 * 4)
⇒ 12
```

In this expression, `shift` captures the continuation `2 * []`.

We can use `shift`/`reset` to write interesting programs. The following function `times` is one of the simplest examples. It calculates a product of a given list of integers. We can define this function as follows:

```
let rec times lst = match lst with
  | [] -> 1
  | a :: rest -> if a = 0 then 0 else a * time rest
```

Thus, `times [1; 2; 3; 4]` evaluates to 24, while `times [0; 1; 2; 3]` immediately evaluates to `0`. However, `times [1; 2; 0; 3]` evaluates to `1 * 2 * 0` and we have to calculate this expression to obtain `0`, although we know the answer will be `0` immediately after we find `0` in the given list.

One of the ways to avoid this unnecessary calculation is to rewrite `times` in CPS:

```
let rec times0 lst k = match lst with
  | [] -> k 1
  | a :: rest -> if a = 0 then 0  (* k is discarded *)
                 else times0 rest (fun x -> k (a * x)) in
let rec times lst = times0 lst (fun x -> x)
```

Since the calculation is saved in the form of continuations, we can throw away the calculation by not using `k` (the `then` clause in the above program). To achieve the same effect without changing whole the program into CPS, we use `shift`/`reset`:

```
let rec times0 lst = match lst with
  | [] -> 1
  | a :: rest -> if a = 0 then shift (fun k -> 0)  (* k is discarded *)
                 else a * times0 rest in
let rec times lst = reset (fun () -> times0 lst)
```

This definition is almost the same as the original definition. The only difference between the two definitions is in the `then` clause. In this definition, if `0` is found in the given list, `shift` captures the current continuation in `k` and discards it. Using `shift/reset`, we can handle continuations freely in direct style.

Although the same effect can be realized by exceptions, there are various applications that cannot be expressed using exceptions but require the full expressiveness of `shift/reset` [2, 5, 6].

3 The MinCaml Compiler

The MinCaml compiler [15], implemented by Eijiro Sumii, is a compiler for educational purposes that achieves two conflicting goals: simplicity (for easy understanding of the internals of compilers) and reasonable efficiency. The compiler consists of only 2000 lines of OCaml code, yet it produces assembly code comparable to that of OCamlOpt and GCC. Because of these features, the MinCaml compiler serves as an ideal platform for implementing and testing new features such as delimited continuations.

The source language of MinCaml is a subset of ML, consisting of integers, floating-point numbers, booleans, tuples, arrays, variable definitions, recursive function definitions, function applications, and so on, in the syntax of OCaml. The original MinCaml compiler generates SPARC assembly, but we have ported it to PowerPC. In this section, we briefly describe the overview of MinCaml compiler, and show the necessary background for the direct implementation of `shift` and `reset` as well as the required extension to the compiler.

Lexical analysis and parsing. We extended a parser and a lexer to accept `shift/reset` expressions (and lists): `shift (fun <var> -> <exp>)` and `reset (fun () -> <exp>)`.

Type inference. The original MinCaml compiler uses a monomorphic type system for simplicity. In the presence of `shift/reset`, however, the answer type polymorphism is critical for many applications. Thus, we implemented Asai and Kameyama's type system [3], which supports let-polymorphism, answer type polymorphism, and answer type modification.

K-normalization and optimizations. After the type inference, the compiler transforms expressions into K-normal form[1] to assign unique names to all the subexpressions. For example, `shift (fun` k `->` M`)` is changed to `let rec` s k `=` M `in shift` s and `reset (fun () ->` M`)` is changed to `let rec` r `() =` M `in reset` r[2], where s and r are fresh variables. After this transformation, the two identifiers `shift` and `reset` are treated as ordinary external functions.

The compiler performs various optimizations, such as constant folding, useless variable elimination, and inline expansion. Although the MinCaml compiler supports rather simple optimizations only, their repeated application yields reasonably optimized code, achieving simplicity and efficiency at the same time. The addition of `shift/reset` does not affect the optimization phase.

Closure conversion. Nested function definitions are converted into closures holding free variables. Each closure consists of a code pointer and a list of free variables. For a function with no

[1]K-normal form is like A-normal form but allows nested `let`-expressions. See [15] for their comparison in the context of MinCaml.

[2]We use `let rec` instead of `let`, because MinCaml supports recursive function definitions only.

free variables, a closure is not constructed but a code pointer itself is used if the function is not used as a value.

After this, expressions are converted to virtual machine code.

Register allocation. The PowerPC microprocessor has 32 general-purpose 32-bit registers, among which two registers are reserved by the operating system (e.g., as a system stack pointer). Thus, there are 30 registers available. Among them, the MinCaml compiler reserves: `R_sp` (a stack pointer) and `R_hp` (a heap pointer). The remaining 28 registers are referred to as `R_i`, starting from `R_0`.

The calling convention of the MinCaml compiler is as follows:

- Arguments to a function are stored in `R_0`, `R_1`, $\cdots$, where `R_0` holds the first argument, and so on.
- The result of a function is stored in `R_0`.
- For closure applications, the address of the closure is stored in `R_cl` (chosen arbitrarily from 28 registers).

Assembly generation. This is the final phase of the compiler. Assembly code is generated from the virtual machine code in a straightforward manner except for function applications and closure creations.

To call a function, live variables that are required after the call as well as the return address have to be saved in the stack. When the function returns, they are restored back from the stack. In addition, if a closure is called, the address of the closure is stored in `R_cl` to satisfy the calling convention.

To create a closure, memory is allocated in the heap to store the free variables and the code pointer. Because tuples, arrays, and closures are allocated in the heap, values allocated in the stack are either immediate values, pointers to the heap, or return addresses, which are all immutable. Thus, it is always safe to copy the contents of the stack.

To realize subroutine calls, PowerPC makes use of a special-purpose register called the link register (LR). Whenever a function is called (using `bctrl`[3]), the next address is stored in the link register as a return address. By executing `blr` (Branch Link Register) in the called function, the control is transferred back to the address stored in the link register. Since the value of the link register is updated at `bctrl`, the old value needs to be saved in the stack.

Tail calls are detected in this phase. If a function call is a tail call, the compiler yields code that does not save the return address but jumps to the function directly.

4 Direct Implementation

By interpreting a program using the continuation semantics, we can regard the state of the program as a continuation stack. Then, `reset` can be thought of as marking the continuation stack, and `shift` capturing the continuation stack up to the nearest mark created by `reset`.

Here is the overview of our implementation:

- When calling `reset`, set a reset mark to the stack.

[3]See Appendix A for the PowerPC assembly language.

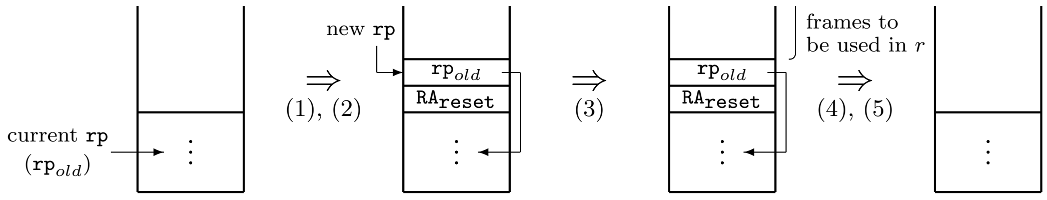


Figure 2: The behavior of the stack when executing `reset`.

- When calling `shift (fun` k `->` M`)`, move a part of the stack frames up to the nearest reset mark to the heap.
- When calling a continuation k, set a reset mark to the stack and copy the corresponding frames from the heap to the stack top.

A reset mark is inserted when k is called, because captured continuations are executed in an empty continuation.

In our implementation, we reserve one register `R_rp` as a reset pointer (`rp`). Reserving a dedicated register for a reset pointer could be costly in the Intel processor where the number of available registers is severely restricted. In PowerPC and SPARC, however, there are relatively many general-purpose registers, and we expect that the effect of using a reset pointer is not prohibitive.

In the rest of this section, we describe the implementation in detail.

4.1 Reset

When `reset` is called with an argument function r (a thunk), a reset mark is set in the stack. Setting the reset mark is realized by pushing the return address to the stack (to preserve the context around `reset`), and storing and updating the reset pointer by the stack pointer (to execute the argument function of `reset` in an empty continuation). We regard the stack address `rp` points to as the reset mark. In our implementation, the return address and the old reset pointer always reside at the bottom of delimited continuations. We maintain this invariant when we manipulate the stack in the following subsections.

The external function `reset` is implemented as follows (Figure 2):

(1) Push the return address (`RA`) to the stack.

(2) Store `rp` to the stack and update `rp`.

(3) Call the argument function r of `reset`.

When r returns, the continuation in which `reset` was executed is restored.

(4) Restore the return address and `rp` from the stack.

(5) Jump to the restored return address.

Notice that the steps (4) and (5) might *not* be executed, because the argument function r can capture the continuation and discard it. Thus, we have to make sure that the correct continuation is always restored when the argument function r finishes its execution.

The PowerPC code for `reset` taken verbatim from the implementation follows (`R_tmp` is chosen arbitrarily from 28 general-purpose registers):

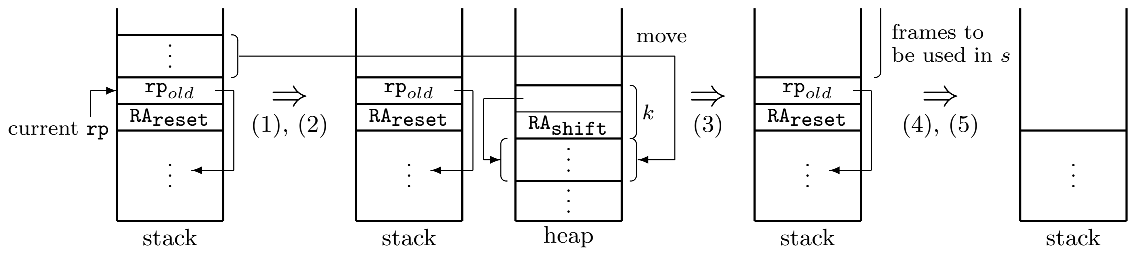


Figure 3: The behavior of the stack and the heap when executing `shift`.

```
1          .globl  min_caml_reset
2  min_caml_reset:
3          mflr    R_tmp            ; (1)
4          stw     R_tmp, 0(R_sp)   ; store RA
5          stw     R_rp, 4(R_sp)    ; (2) store rp
6          addi    R_rp, R_sp, 4    ; update rp
7          addi    R_sp, R_sp, 8
8          mr      R_cl, R_0        ; (3)
9          lwz     R_tmp, 0(R_cl)
10         mtctr   R_tmp
11         bctrl                    ; call r
12         subi    R_sp, R_sp, 8    ; (4)
13         lwz     R_tmp, 0(R_sp)
14         mtlr    R_tmp            ; restore RA
15         lwz     R_rp, 4(R_sp)    ; restore rp
16         blr                      ; (5)
```

This code exactly corresponds to the above five steps. At line 3, the link register (which holds the return address of `reset`) is extracted and pushed on to the stack (line 4). At lines 5 and 6, the reset pointer is stored and updated to the top of the stack. At line 8, `R_0` (the first argument of `reset`) holds the argument closure r of `reset` and moves that value to `R_cl` (to satisfy the calling convention), which is called at line 11. When the function returns, the return address and the reset pointer are restored and the execution goes back to the restored return address.

4.2 Shift

When `shift` is called with an argument s (typically of the form `(fun` k `->` M`))`, stack frames are cleared (up to `rp`) and moved to the heap to create a closure for the captured continuation. The external function `shift` is implemented as follows (Figure 3):

(1) Move the stack frames up to `rp` (excluding `rp`) to the heap.

Since the frame `rp` points to is not moved, the cleared stack has the old reset pointer and the return address at the top. In other words, the stack is in the same state as when the `reset` was executed.

(2) Make the closure for the continuation function k with the information on the frames and the return address of `shift`.

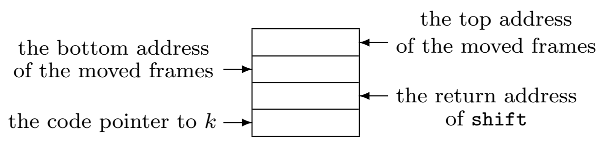


Figure 4: The closure for k.

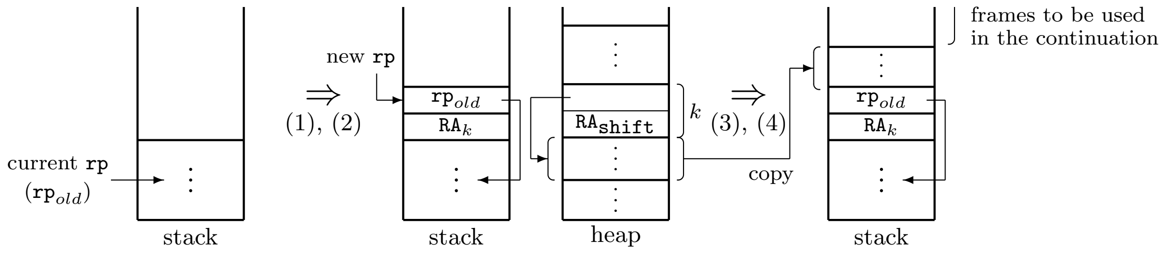


Figure 5: The behavior of the stack when executing the continuation k.

As depicted in Figure 4, the closure for the continuation function k consists of the top and the bottom addresses of the moved frames, the return address of `shift` (to be executed when the captured continuation k is called), and the code pointer to k (see Section 4.3). The closure for k represents the continuation up to the enclosing `reset`. In particular, the last instructions k will execute are the steps (4) and (5) of `reset`. Thus, when we call k, we have to properly set the return address and the reset pointer at the call to k to compose k with the surrounding context (see Section 4.3).

(3) Set k as the first argument and call the argument function s of `shift` on the cleared stack (i.e., the empty continuation).

When s returns, it means that the work to be done in the current context is finished and the reset mark is at the top of the stack. To restore the outer context, we repeat the steps (4) and (5) of `reset` here.

(4) Restore the return address and `rp` from the stack.

(5) Jump to the restored return address.

The PowerPC code for `shift` follows:

```
        .globl  min_caml_shift
min_caml_shift:
        mr      R_cl, R_0
        ;; (1) move the frames from the stack to the heap
        mr      R_0, R_rp       ; current rp
        mr      R_2, R_hp
        subi    R_hp, R_hp, 4
to_heap_loop:
        lwzu    R_tmp, 4(R_0)   ; from the stack
```

```
10         cmpw    cr7, R_0, R_sp
11         beq     cr7, to_heap_exit
12         stwu    R_tmp, 4(R_hp)  ; to the heap
13         b       to_heap_loop
14 to_heap_exit:
15         addi    R_hp, R_hp, 4
16         addi    R_sp, R_rp, 4
17         mr      R_0, R_hp       ; (2) closure k
18         addi    R_hp, R_hp, 16
19         lis     R_1, ha16(min_caml_k)
20         addi    R_1, R_1, lo16(min_caml_k)
21         mflr    R_tmp
22         stw     R_1, 0(R_0)     ; code pointer to k
23         stw     R_tmp, 4(R_0)   ; RA of shift
24         stw     R_2, 8(R_0)     ; bottom address
25         stw     R_0, 12(R_0)    ; top address
26         lwz     R_tmp, 0(R_cl)  ; (3)
27         mtctr   R_tmp
28         bctrl                   ; call s
29         subi    R_sp, R_sp, 8   ; (4)
30         lwz     R_tmp, 0(R_sp)
31         mtlr    R_tmp           ; restore RA
32         lwz     R_rp, 4(R_sp)   ; restore rp
33         blr                     ; (5)
```

Again, this code exactly corresponds to the above five steps. At lines 5 to 16, frames between the reset pointer (`R_rp`) and the stack top (`R_sp`) are copied to the heap. At line 18, the closure for k is allocated in the heap and set properly in the following lines. Since the closure for k is already placed in `R_0` (the first argument position), the argument function s is called at line 28. Lines 29 to 33 are exactly the same as the lines 12 to 16 of the code for `reset`.

4.3 Captured Continuation k

When the captured continuation k is called with an argument, a reset mark is set in the stack to delimit the context and the frames corresponding to k are copied back from the heap to the stack top. The external function k (common to all the captured continuations) is implemented as follows (Figure 5):

(1) Push the return address (`RA`) to the stack.

(2) Store `rp` to the stack and update `rp`.

Since k expects the return address and the reset pointer in the stack, we set them so that k is composed with the current continuation. These two steps are exactly the same as the first two steps of `reset`.

(3) Copy the frames for the captured continuation k to the stack top.

(4) Jump to the return address (code for the captured continuation, $\mathtt{RA}_{\mathtt{shift}}$) preserved in the closure for k.

The PowerPC code for k is as follows:

```
1  min_caml_k:
2          mflr    R_tmp           ; (1)
3          stw     R_tmp, 0(R_sp)  ; store RA
4          stw     R_rp, 4(R_sp)   ; (2) store rp
5          addi    R_rp, R_sp, 4   ; update rp
6          addi    R_sp, R_sp, 8
7          lwz     R_1, 8(R_cl)    ; bottom address
8          lwz     R_2, 12(R_cl)   ; top address
9          ;; (3) copy the frames from the heap to the stack
10         subi    R_sp, R_sp, 4
11         subi    R_1, R_1, 4
12 from_heap_loop:
13         lwzu    R_tmp, 4(R_1)   ; from the heap
14         cmpw    cr7, R_2, R_1
15         beq     cr7, from_heap_exit
16         stwu    R_tmp, 4(R_sp)  ; to the stack
17         b       from_heap_loop
18 from_heap_exit:
19         addi    R_sp, R_sp, 4
20         lwz     R_tmp, 4(R_cl)  ; (4)
21         mtlr    R_tmp
22         blr                     ; jump to the preserved RA
```

First, the link register is extracted and pushed on to the stack (lines 2 and 3) and the reset pointer is stored and updated (lines 4 and 5). These four lines are exactly the same as the lines 3 to 6 of the code for `reset`. At lines 10 to 19, the frames for k are copied to the stack top. Finally, the control is transferred to the preserved return address (lines 20 to 22).

4.4 Benchmarks

In this section, we measure the performance and memory consumption of our implementation of `shift/reset` using several programs and compare them to the performance of running their CPS counterparts. All timings were obtained on a PowerPC G4 system with 500MHz and 1.28 GB SDRAM.

4.4.1 Reverse

The following function is one of the classic examples [5] of `shift/reset` that reverses a given list:

```
let rec visit lst = match lst with
  | [] -> []
  | a :: rest -> shift (fun k -> a :: k (visit rest)) in
let rec reverse lst = reset (fun () -> visit lst)
```

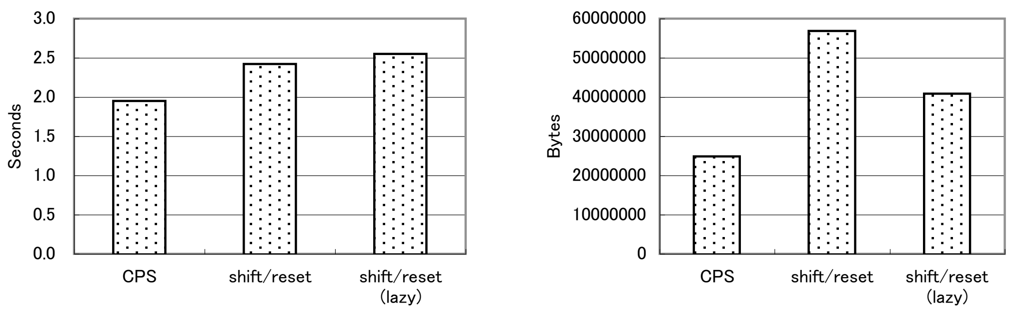


Figure 6: Execution time and used memory of `reverse`.

Its CPS counterpart is given as follows:

```
let rec visit lst k = match lst with
  | [] -> k []
  | a :: rest -> visit rest (fun x -> a :: k x) in
let rec reverse lst = visit lst (fun x -> x)
```

The use of `shift` in this program is important because the same technique can be used to realize one of the most well-known uses of `shift/reset`, namely, let-insertion in partial evaluation [1]. It also serves as a good example to measure basic overhead of our implementation since the program is simple.

Figure 6 shows the running time and the used memory when reversing a list of 100000 elements 10 times. (For "shift/reset (lazy)", see Section 5.7.) Compared to the CPS version, our implementation is 24 % slower and consumes 129 % more memory. Since both the programs create closures of a similar size, the difference appears to come from:

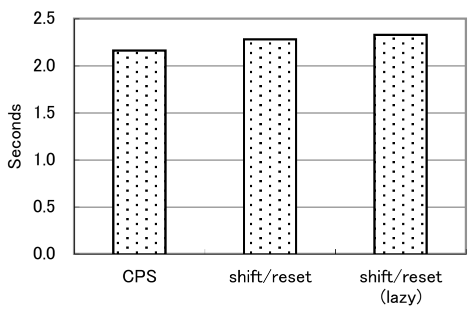
- the tail-call optimization of recursive calls are effective only for the CPS version,
- copying of stack frames, and
- creation of closures for k.

Because the tail-call optimization affects the performance considerably, the overhead of our implementation does not seems to be large, considering that the amount of copied frames is not small.

4.4.2 Prefix

The function `prefix` returns a list of prefixes of a given list. For example, `prefix [1; 2; 3]` $\Rightarrow$ `[[1]; [1; 2]; [1; 2; 3]]`. We can write this function using `shift/reset`:

```
let rec visit lst = match lst with
  | [] -> shift (fun k -> [])
  | a :: rest ->
      shift (fun k -> (k [a]) ::  (reset (fun () -> k (a :: visit rest)))) in
let rec prefix lst = reset (fun () -> visit lst)
```



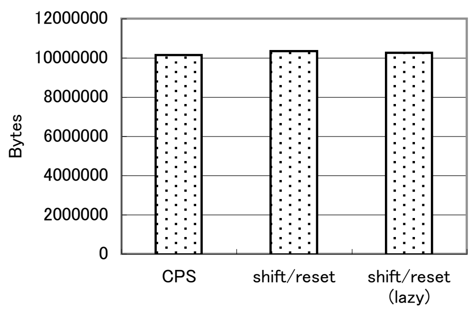


Figure 7: Execution time and used memory of `prefix`.

Its CPS counterpart is given as follows:

```
let rec visit lst k = match lst with
  | [] -> []
  | a :: rest -> (k [a]) :: (visit rest (fun x -> k (a :: x))) in
let rec prefix lst = visit lst (fun x -> x)
```

In the resulting list, the first element is cons'ed many times since it appears in all prefixes. This behavior is realized by capturing the continuation that cons'es the element and using it multiple times.

The running time and the used memory for constructing prefix of 500 elements 10 times are shown in Figure 7. Compared to `reverse`, we observe that the overhead of `shift/reset` is smaller. One of the reasons is that the recursive call of the CPS version is no longer a tail call.[4] Even though the `shift/reset` version creates more closures and copies stack frames, they are negligible in the presence of a large result list of prefixes.

4.4.3 Monads

Filinski showed that an arbitrary monad can be implemented using `shift/reset` [9]. Following Gasbichler and Sperber [10], we use the following two functions to implement a list monad:

```
let rec reflect meaning =
  shift (fun k -> extend k meaning)
let rec reify thunk =
  reset (fun () -> eta (thunk ()))
```

where `eta` and `extend` are the unit and bind operations of a list monad:

```
let rec eta x = [x]
let rec extend f l = flatten (map f l)
```

Using them, we calculated the possible result for adding three numbers each of which has 6 or 7 possible values. The result of execution is found in Figure 8.

We observe that writing a program in direct style can be faster than its CPS counterpart. Notice that to obtain the CPS version, we have to transform all the functions into CPS, which is tedious and

[4]The result of CPS transformation contains non-tail calls because `shift/reset` is used in the original program.

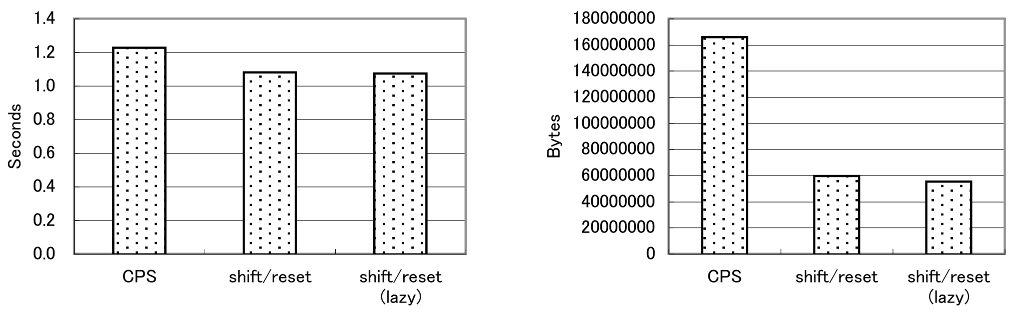


Figure 8: Execution time and used memory of monads.

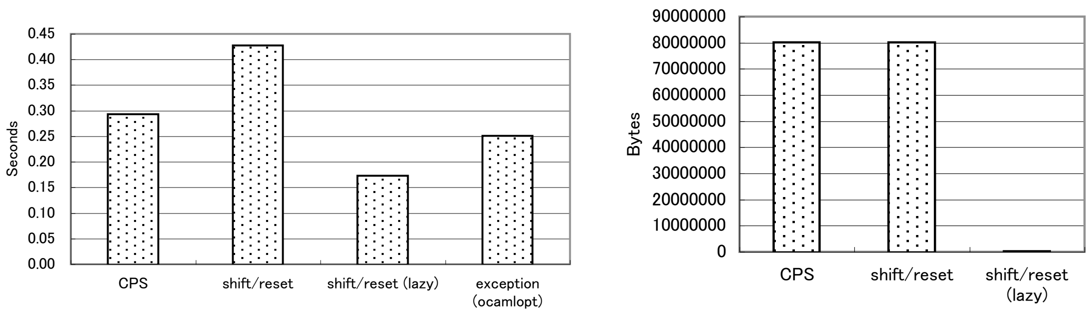


Figure 9: Execution time and used memory of `times`.

error-prone. Furthermore, in the CPS program, having an additional argument for continuations and creating closures for them becomes costly as the program becomes bigger. If `shift/reset` is directly implemented and is used properly, we obtain the expressiveness of `shift/reset` without loosing performance.

4.4.4 Times

We show the execution time and the used memory of `times` (described in Section 2) for multiplying 1000 elements whose last element is 0 repeated 5000 times in Figure 9. In contrast to `prefix`, the difference of the CPS and `shift/reset` versions is not small. It is because in expression `shift (fun k -> 0)`, the large stack frames bound to k are copied despite the fact that they are not necessary to evaluate `0` and can be discarded without copying. This suggests that the implementation could be improved in such a case.

4.5 Issue

The implementation in this section copies the stack frames every time `shift` is called. However, we need not copy the frames in some cases. The above `times` function is one of the examples. In this case, we need not move the frames to the heap but only need to discard it. Moreover, consider the execution of `shift (fun k -> k 3)`. In the present implementation, we move the stack frames from the stack to the heap at `shift`, but immediately copy the same frames back from the heap

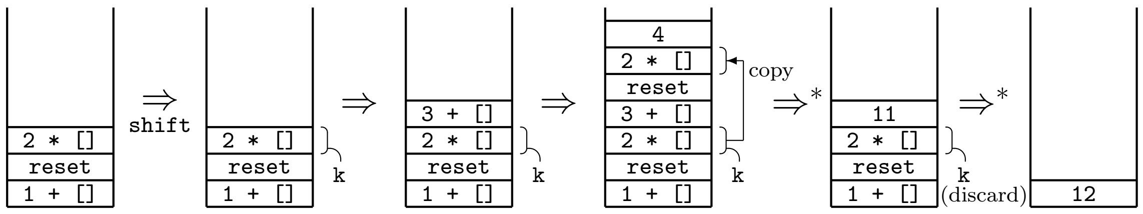


Figure 10: The behavior of the stack for `1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))`. The `reset` frame contains the reset pointer and the return address.

to the stack. In this case, we can obtain the correct answer if we keep the frames in the stack and reuse them. We will tackle these issues in the next section.

5 Lazy Implementation

In this section, we describe the *lazy* implementation which does not copy frames until needed. The technique consists of two parts: keeping stack frames in the stack and reusing stack frames for tail calls.

5.1 Keeping Stack Frames in the Stack

In the implementation in Section 4, we always move frames to the heap when `shift` is called. In the lazy implementation, we do not do so but keep frames in the stack (if k does not escape) and remember that the frames correspond to k. When k is called later, the stack frames (residing in the stack) are copied to the stack top. For example, consider the execution of `1 + reset (fun () -> 2 * shift (fun k -> 3 + k 4))`. The behavior of the stack is depicted in Figure 10. When `shift` is executed, the stack frame for `2 * []` is *not* moved to the heap but kept in the stack. When `k` is called later, the corresponding frame is copied to the stack top. In this case, we can execute the expression by copying the frame only once. In contrast, we had to copy the frame twice in the implementation of Section 4. Notice that when the execution of the body of `shift` finishes, we need to discard the remaining stack frames captured for k.

We cannot keep frames in the stack every time, however. For example, consider the expression `shift (fun k -> k)`. This expression captures the current continuation and returns it. In other words, `k` *escapes* from the lexical scope of `shift`. In this case, we have to copy `k`'s stack frames to the heap, because otherwise, the stack frames will be destroyed in the subsequent execution.

5.2 Reusing Stack Frames for Tail Calls

We can further reduce the number of copies, if the captured continuation k is always called at the tail position. Consider the execution of `1 + reset (fun () -> 2 * shift (fun k -> k 3))`. The behavior of the stack is depicted in Figure 11. In this case, the frame corresponding to `k` already exists in the stack top (the second figure of Figure 11). By reusing this frame, we can omit the copy of the frame entirely. In the implementation of Section 4, we had to copy the frame twice to execute the same expression.

Readers might think that k is typically not called at the tail position, because in that case, the program could have been written without using `shift`. One does not write `shift (fun k -> k`

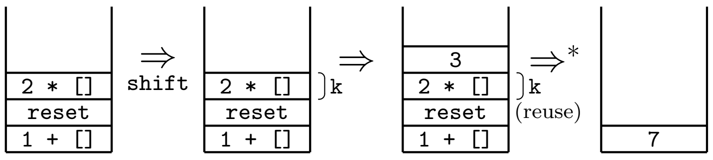


Figure 11: The behavior of the stack for `1 + reset (fun () -> 2 * shift (fun k -> k 3))`

`3)` because he could simply write `3` instead of using `shift`. However, it is known that such `shift` expressions are produced if we apply partial evaluation to programs written with `shift/reset` [1]. Therefore, it is important that such expressions are executed without much penalty.

Besides, there exists a case where k is used multiple times. If there exist both the tail call of k and non-tail calls of k (e.g., `shift (fun k -> k (1 + k 3))`), we only need to copy frames in the non-tail cases. For the tail case, no copy is required.

5.3 Overview

In the lazy implementation, we use a shift mark in addition to the reset mark. Intuitively, the shift mark is used to separate the active stack frames from the captured frames.

The overview of the lazy implementation is as follows:

- When calling `reset`, set a reset mark and a shift mark.
- When calling `shift (fun` k `->` M`)`, we classify the treatment in 3 ways according to how k is used in M (see below).
- When calling a continuation k, set a reset mark and a shift mark and copy the corresponding frames from the heap or the stack to the stack top.

Like a reset mark, we use a dedicated register `R_shp` as a shift pointer (`shp`). In the following subsections, we describe the lazy implementation in detail.

5.4 Reset

The same as before. We only need to save and restore `shp`:

(1) Push the return address (`RA`) to the stack.

(2) Store `rp` and `shp` to the stack and update both of them.

(3) Call the argument function r of `reset`.

(4) Restore the return address, `rp`, and `shp` from the stack after returning from the function call.

(5) Jump to the restored return address.

As in Section 4, these steps exactly correspond to the PowerPC code:

```
        .globl  min_caml_reset
min_caml_reset:
        mflr    R_tmp            ; (1)
        stw     R_tmp, 0(R_sp)   ; store RA
        stw     R_rp, 4(R_sp)    ; (2) store rp
        addi    R_rp, R_sp, 4    ; update rp
        stw     R_shp, 8(R_sp)   ; store shp
        addi    R_shp, R_sp, 8   ; update shp
        addi    R_sp, R_sp, 16
        mr      R_cl, R_0        ; (3)
        lwz     R_tmp, 0(R_cl)
        mtctr   R_tmp
        bctrl                    ; call r
        subi    R_sp, R_sp, 16   ; (4)
        lwz     R_tmp, 0(R_sp)
        mtlr    R_tmp            ; restore RA
        lwz     R_rp, 4(R_sp)    ; restore rp
        lwz     R_shp, 8(R_sp)   ; restore shp
        blr                      ; (5)
```

5.5 Shift

The continuation is delimited not only when `reset` is executed, but also when `shift` is executed—the body of `shift` is executed with an empty continuation. In the implementation of Section 4, we moved the stack frames up to `rp` when calling `shift`. Then, the state of the stack became the same as when the enclosing `reset` was executed. In the lazy implementation, however, we keep frames in the stack. To avoid capturing the unnecessary frames, we use `shp`: it always points to the top of the captured frames, showing the start of the active frames. When `shift` is executed, the stack frames up to `shp` are captured.

When the execution of the body of `shift` finishes, we must discard the remaining captured frames. (We did not have to do this in the implementation of Section 4, because there were no remaining captured frames and the return address and the reset pointer were at the top of the stack.) To discard the remaining captured frames, we use `rp`: we discard frames up to `rp` and jump to the saved return address.

The above description does not take advantage of the tail call of k. To reuse stack frames for tail calls, we do two things. First, we do not update `shp` when `shift` is called. By *not* updating `shp`, the stack frames corresponding to k are included in the active frames. When another `shift` is executed later, the reused frames will be correctly captured. Second, the application of k at the tail position is detected syntactically and is turned into the `blr` (return) instruction, so that the result is passed to the reused frames.

For example, consider the execution of `1 + reset (fun () -> 2 * shift (fun k -> k (k 3)))`. The behavior of the stack is depicted in Figure 12. Since `k` does not escape and the argument function of `shift` calls `k` in the tail position, the frame for `k` is preserved in the stack without updating the shift mark (the second figure of Figure 12) and reused for the tail call of `k`. To call the inner non-tail `k`, we copy the frame from the stack to the stack top (the third figure). To call the outer `k`, we do not copy the frame but pass the result value (6) to the remaining captured frame for `k` (the fourth figure).

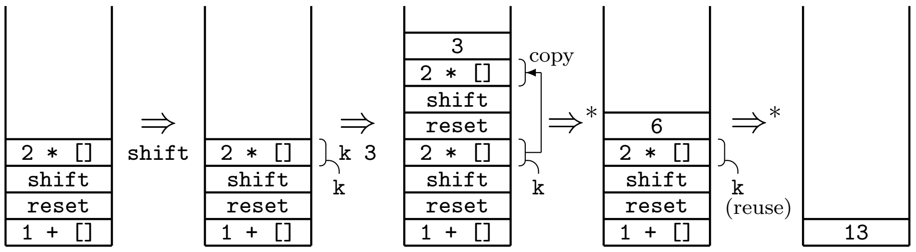


Figure 12: The behavior of the stack for `1 + reset (fun () -> 2 * shift (fun k -> k (k 3)))`. The `shift` frame contains the shift pointer.

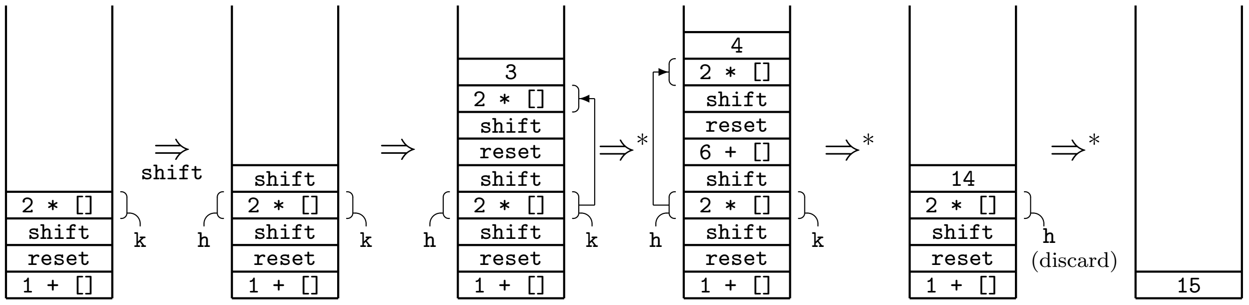


Figure 13: The behavior of the stack for `1 + reset (fun () -> 2 * shift (fun k -> k (shift (fun h -> k 3 + h 4))))`.

The above method to reuse stack frames of tail calls works fine even when `shift` is used multiple times. For example, consider the execution of `1 + reset (fun () -> 2 * shift (fun k -> k (shift (fun h -> k 3 + h 4))))`. The behavior of the stack is depicted in Figure 13. Neither `k` nor `h` escapes. Since the first `shift` calls `k` in the tail position, the shift mark is not updated in this case (the first figure of Figure 13). Thus, the second `shift` correctly captures the right frame (i.e., `2 * []`). Since `h` is not used in the tail position, the shift mark is updated at this point (the second figure). When `k 3` is executed, the frame for `k` is copied to the stack top (the third figure), and when `h 4` is executed, the frame for `h` is copied to the stack top (the fourth figure). Finally, when the execution of the body of the second `shift` is finished (the fifth figure), the frame for `h` is discarded and we get the correct answer `15`. In this case, we cannot reuse the frame for `k`, because it is recaptured by second `shift`.

To implement `shift`, we distinguish three cases. We write the closure pointing to continuation frames in the stack k_{lazy}.

- k does not escape and there may not exist the tail call of k (Figure 14).

  In this case, we can keep the frames in the stack. When k is called, they are copied to the stack top. Since k does not appear in the tail position, we need to update `shp` and discard the remaining captured frames when the execution of `shift` ends.

  (1) Make the closure for the continuation function k_{lazy} with the information on the frames and the return address of `shift`. We do not copy or move the frames to the heap.

  (2) Store `shp` to the stack and update `shp`. We do not save the return address or `rp`.

  (3) Set k_{lazy} as the first argument and call the argument function s of `shift`.

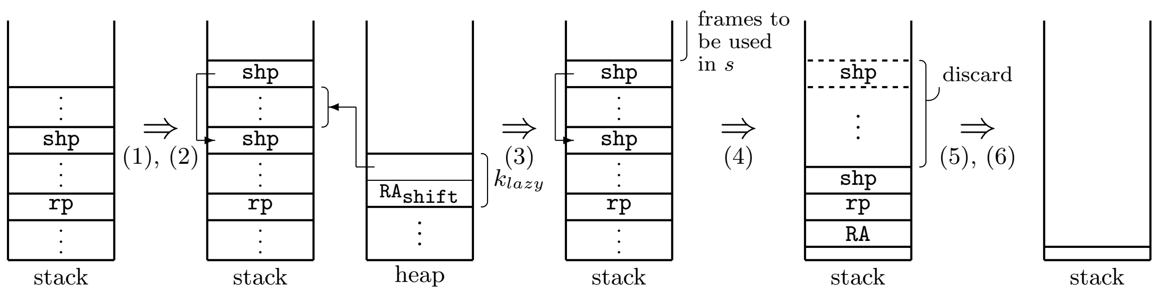


Figure 14: The behavior of the stack and the heap for `shift`: k does not escape; k may not be called at the tail position.

(4) Discard the frames up to `rp` but excluding `rp` and `shp`. after returning from the function call.

(5) Restore the return address and `rp`, `shp` from the stack.

(6) Jump to the restored return address.

```
        .globl  min_caml_break_not_escape_shift
min_caml_break_not_escape_shift:
        mr      R_cl, R_0
        mr      R_0, R_hp       ; (1) closure k
        addi    R_hp, R_hp, 16
        lis     R_1, ha16(min_caml_k_lazy)
        addi    R_1, R_1, lo16(min_caml_k_lazy)
        mflr    R_tmp
        stw     R_1, 0(R_0)     ; code pointer to k
        stw     R_tmp, 4(R_0)   ; RA to shift
        stw     R_shp, 8(R_0)   ; bottom address
        stw     R_shp, 0(R_sp)  ; (2) store shp
        mr      R_shp, R_sp     ; update shp
        stw     R_shp, 12(R_0)  ; top address
        addi    R_sp, R_sp, 8
        lwz     R_tmp, 0(R_cl)  ; (3)
        mtctr   R_tmp
        bctrl                   ; call s
        subi    R_sp, R_rp, 4   ; (4)
        lwz     R_tmp, 0(R_sp)
        mtlr    R_tmp           ; restore RA
        lwz     R_rp, 4(R_sp)   ; restore rp
        lwz     R_shp, 8(R_sp)  ; restore shp
        blr                     ; (5)
```

- k does not escape and there always exists the tail call of k (Figure 15).

  In this case, we can still keep the frames in the stack. We do not update `shp` and reuse the frames of k for the tail call. If k is applied at the non-tail position in s (e.g. the inner `k` of `shift (fun k -> k (k 3))`), we copy the frames from the stack to the stack top.

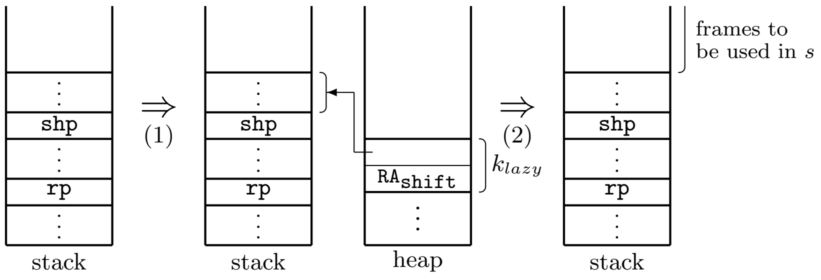


Figure 15: The behavior of the stack and the heap for `shift`: k does not escape; k is always called at the tail position.

(1) Make the closure for the continuation function k_{lazy} with the information on the frames and the return address of `shift`. We do not copy or move the frames to the heap.

(2) Set k_{lazy} as the first argument and jump to the argument function s of `shift`.

We do not renew the return address but jump directly so that the tail-ness of k is preserved.

```
        .globl  min_caml_tail_not_escape_shift
min_caml_tail_not_escape_shift:
        mr      R_cl, R_0
        mr      R_0, R_hp       ; (1) closure k
        addi    R_hp, R_hp, 16
        lis     R_1, ha16(min_caml_k_lazy)
        addi    R_1, R_1, lo16(min_caml_k_lazy)
        mflr    R_tmp
        stw     R_1, 0(R_0)     ; code pointer to k
        stw     R_tmp, 4(R_0)   ; RA to shift
        stw     R_shp, 8(R_0)   ; bottom address
        stw     R_sp, 12(R_0)   ; top address
        lwz     R_tmp, 0(R_cl)  ; (2)
        mtctr   R_tmp
        bctr                    ; jump to s
```

- k may escape (Figure 16).

  In this case, we do not keep continuation frames in the stack but copy them to the heap. The following steps are the same as the ones in Section 4.2 except for the use of `shp` instead of `rp` and discarding of remaining captured frames.

  (1) Move the stack frame up to `shp` (excluding `shp`) to the heap.

  (2) Make the closure of the continuation function k with the information on the frames and the return address of `shift`.

  (3) Set k as the first argument and call the argument function s of `shift`.

  (4) Discard the frames up to `rp` but excluding `rp` and `shp` after returning from the function call.

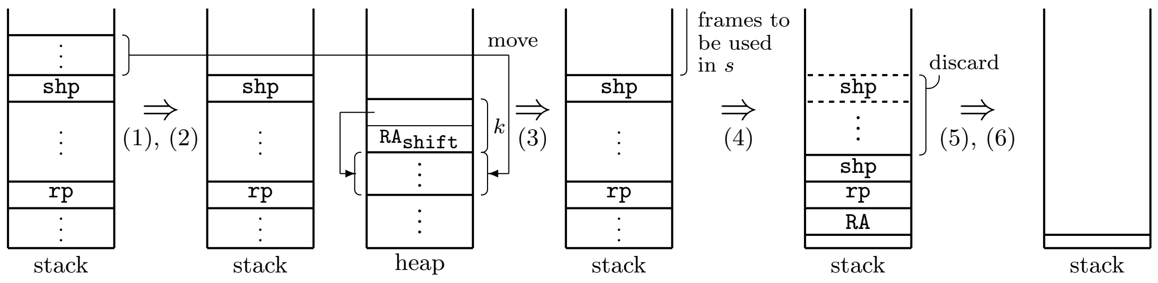


Figure 16: The behavior of the stack and the heap for `shift`: k may escape.

(5) Restore the return address, `rp`, and `shp` from the stack.

(6) Jump to the restored return address.

```
        .globl  min_caml_escape_shift
min_caml_escape_shift:
        mr      R_cl, R_0
        ;; (1) move the frames from the stack to the heap
        addi    R_0, R_shp, 4   ; current shp
        subi    R_2, R_hp, 4
        subi    R_hp, R_hp, 4
to_heap_loop:
        lwzu    R_tmp, 4(R_0)   ; from the stack
        cmpw    cr7, R_0, R_sp
        beq     cr7, to_heap_exit
        stwu    R_tmp, 4(R_hp)  ; to the heap
        b       to_heap_loop
to_heap_exit:
        addi    R_hp, R_hp, 4
        addi    R_sp, R_shp, 8
        mr      R_0, R_hp       ; (2) closure k
        addi    R_hp, R_hp, 16
        lis     R_1, ha16(min_caml_k)
        addi    R_1, R_1, lo16(min_caml_k)
        mflr    R_tmp
        stw     R_1, 0(R_0)     ; code pointer to k
        stw     R_tmp, 4(R_0)   ; RA to shift
        stw     R_2, 8(R_0)     ; bottom address
        stw     R_0, 12(R_0)    ; top address
        lwz     R_tmp, 0(R_cl)  ; (3)
        mtctr   R_tmp
        bctrl                   ; call s
        subi    R_sp, R_rp, 4   ; (4)
        lwz     R_tmp, 0(R_sp)  ; (5)
        mtlr    R_tmp           ; restore RA
        lwz     R_rp, 4(R_sp)   ; restore rp
        lwz     R_shp, 8(R_sp)  ; restore shp
        blr                     ; (6)
```

To decide whether k may escape or not, we employ a simple and standard type-based escape analysis, extended to cope with `shift/reset`. A continuation is regarded as escaping according to the following rules:

- Return values of functions may escape.
- If a function may escape, their free variables may also escape.
- If a tuple or a list may escape, their elements may escape.
- Values assigned in arrays may escape.

The check of whether there always exists the tail call of k is done syntactically. For `if` expressions, we judge that there always exists the tail call of k if the tail call of k exists in both branches.

5.6 Captured Continuation k

The behavior when the captured continuation k or k_{lazy} is called is the same as before except that a shift mark is set. When k is called, the following steps are executed:

(1) Push the return address (`RA`) to the stack.

(2) Store `rp` and `shp` to the stack and update both of them.

(3) Copy the frames for the captured continuation k from the heap to the stack top.

(4) Jump to the return address preserved in the closure of k.

Similarly for k_{lazy}:

(1) Push the return address (`RA`) to the stack.

(2) Store `rp` and `shp` to the stack and update both of them.

(3) Copy the frames for the captured continuation k_{lazy} from the stack to the stack top.

(4) Jump to the return address preserved in the closure of k_{lazy}.

The difference between k and k_{lazy} is whether copying the frame is from the heap or from the stack. In fact, the codes of k and k_{lazy} are exactly the same.

Finally, the tail call of non-escaping k is converted to do nothing and jump to the return address (the `blr` instruction) because necessary frames already exist in the top of the stack.

```
min_caml_k:
min_caml_k_lazy:
        mflr    R_tmp           ; (1)
        stw     R_tmp, 0(R_sp)  ; store RA
        stw     R_rp, 4(R_sp)   ; (2) store rp
        addi    R_rp, R_sp, 4   ; update rp
        stw     R_shp, 8(R_sp)  ; store shp
        addi    R_shp, R_sp, 8  ; update shp
        addi    R_sp, R_sp, 16
        lwz     R_1, 8(R_cl)    ; bottom address
        lwz     R_2, 12(R_cl)   ; top address
        ;; (3) copy the frames to the stack top
        subi    R_sp, R_sp, 4
        addi    R_1, R_1, 4
from_heap_loop:
        lwzu    R_tmp, 4(R_1)   ; from the heap or the stack
        cmpw    cr7, R_2, R_1
        beq     cr7, from_heap_exit
        stwu    R_tmp, 4(R_sp)  ; to the stack top
        b       from_heap_loop
from_heap_exit:
        addi    R_sp, R_sp, 4
        lwz     R_tmp, 4(R_cl)  ; (4)
        mtlr    R_tmp
        blr                     ; jump to the preserved RA
```

5.7 Benchmarks

In this chapter, we discuss the efficiency of the lazy implementation with some benchmark programs. The same as in Section 4, all timings are obtained on a PowerPC G4 system with 500MHz and 1.28 GB SDRAM.

5.7.1 Reverse, Prefix, Monads

Timings and used memory for `reverse`, `prefix`, and monads are already shown in Figures 6, 7, and 8, respectively. In all the examples, the amount of used memory decreases thanks to keeping continuation frames in the stack. However, since none of them call k at the tail position, stack frames are not reused. Even though the number of copies is reduced, the running time somewhat increases due to the overhead of the lazy implementation.

5.7.2 Times

The lazy implementation is the most effective for the `times` example (Figure 9). In the lazy implementation, unused continuation frames are not copied but simply discarded. It outperforms all the other cases, even the native OCaml implementation that uses exception. We cannot directly compare our implementation with the OCaml system, because OCaml supports many more features than MinCaml. However, this result shows that the lazy implementation can be effective in some cases.

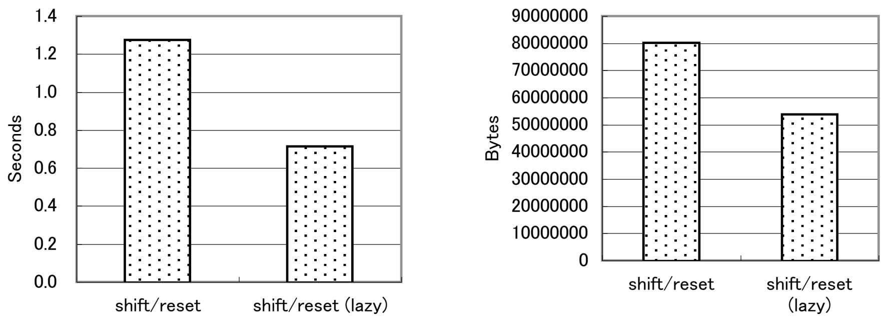


Figure 17: Execution time and used memory of `queen`.

5.7.3 N-Queen

As an example that uses `shift` more extensively, we show the function `queen` that solves the N-Queen problem. Instead of using a double loop to search for a solution, we write the program in non-deterministic style. The following function `choice` captures the current continuation `k` and passes the value `j` to `k`. Moreover, it also passes the result of the recursive call to `k`:

```
let rec choice j = if j = 1 then 1
                   else shift (fun k -> k j; k (choice (j - 1)))
```

Therefore, we can consider `choice` as the function that returns the value from `1` to `j` non-deterministically. In the function `choice`, there is the tail call of k and k does not escape in the argument function of `shift`.

Using this function, the N-Queen problem can be solved by a single loop without backtracking:

```
let rec queen n =
  let rec loop i solution =
    if i = 0 then print_solution solution
    else let j = choice n in
         let solution2 = j :: solution in
           if is_safe solution2
           then loop (i - 1) solution2 in
  reset (fun () -> loop n [])
```

In the program, `print_solution` prints the answer list, and `is_safe` checks whether a queen is safe to place.

We show in Figure 17 the execution time and the consumed memory to solve the 11-Queen problem with this program using the implementation of Section 4 and the lazy implementation. In the implementation of Section 4, we have to copy frames three times for every `shift`. In the lazy implementation, on the other hand, we have to copy frames only once for every `shift`, because one of the calls of `k` is a tail call and does not need copying. This difference of the number of copying affects the performance of the two implementations. Moreover, since the tail frames are concatenated to the active frames, copied frames tend to become larger in the lazy implementation, which could have a positive effect on performance. In the implementation of Section 4, the same frames are copied separately, incurring overhead for each copy.

6 Related Work

Gasbichler and Sperber showed the direct implementation of `shift`, `reset`, and `control` and implemented it in the Scheme 48 system [10]. They showed that the direct implementation of `shift/reset` is significantly faster than the indirect implementation using `call/cc`. Their implementation is done in the low-level Scheme called PreScheme, a virtual machine for the Scheme 48 system. Kiselyov implemented the delimcc library[5] that includes native implementation of `shift/reset` at the OCaml bytecode level. His implementation copies only the necessary prefix of the stack and is fully integrated with OCaml exceptions. The present work shares the goal of obtaining fast implementation of `shift/reset` with them. Rather than in a virtual machine or the OCaml bytecode, we achieved the same effect in the PowerPC assembly language, making explicit all the details including how to store the return address and how to represent the reset mark. Gasbichler and Sperber's implementation employs the incremental stack/heap strategy, while we employ the simpler stack strategy.

Clinger et al. presented a comprehensive list of strategies for first-class continuations and defined "zero overhead" criteria for implementation strategies of first-class continuations [4]. An implementation strategy is called to have zero overhead if the support for the first-class continuation does not incur any overhead on programs that do not use them. Strictly speaking, our implementation strategy does not have the zero overhead property according to this criteria, because we reserve two registers for a reset pointer and a shift pointer. The investigation on the effect of this design choice is a future work. If instead we store the reset and shift pointers in memory (with performance penalty for `shift` and `reset`), our strategy obtains the zero overhead property: programs that do not use `shift/reset` are compiled exactly the same as the original MinCaml compiler.

Ugawa et al. proposed lazy stack copying to implement first class continuations in the stack-based implementation [16]. In their lazy stack copying, frames are not copied when calling `call/cc`, but deferred until needed. We employed the same idea to implement `shift/reset` in a typed setting rather than `call/cc` in an untyped setting.

7 Conclusion and Future Work

This paper presented a direct implementation as well as the lazy implementation of delimited control operators `shift` and `reset` in the MinCaml compiler. By showing the implementation in the PowerPC assembly language, we spelled out all the details of how composable continuations are implemented using the stack strategy. We presented various benchmarks and discussed their performance. Because of the simplicity of the implementation, it can serve as a reference implementation of `shift/reset`. We hope that our implementation promotes the use of `shift` and `reset` in programming.

As future work, we plan to compare various implementation strategies of `shift/reset`, to investigate the interaction with garbage collection, and to relate this implementation with the definitional interpreter for `shift/reset` to formally verify the correctness of the implementation.

Acknowledgements

We thank Martin Gasbichler for sending us benchmark programs.

[5]Available from `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`.

References

[1] K. Asai, "Logical Relations for Call-by-value Delimited Continuations", In *Trends in Functional Programming*, Vol. 6, pp. 63–78, Intellect (2007).

[2] K. Asai, "On Typing Delimited Continuations: Three New Solutions to the Printf Problem," to appear in *Higher-Order and Symbolic Computation.* Also available as Technical Report OCHA-IS 08-2, Department of Information Sciences, Ochanomizu University, 17 pages (April 2006).

[3] K. Asai, and Y. Kameyama, "Polymorphic Delimited Continuations", In Z. Shao editor, *5th Asian Symposium on Programming Languages and Systems (APLAS 2007)*, pp. 239–254 (November 2007).

[4] W. Clinger, A. Hartheimer, E. Ost, "Implementation Strategies for First-Class Continuations", *Higher-Oeder and Symbolic Computation*, Vol. 12, No. 1, pp. 7–45, Kluwer Academic Publishers (1999).

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

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

[7] O. Danvy, and A. Filinski, "Representing Control: A Study of the CPS transformation", In *Mathematical Structures in Computer Science*, Vol. 2, No. 4, pp. 361–391 (December 1992).

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

[9] A. Filinski, "Representing Monads", In *Conference Records of the 21st Annual ACM Symposium on Principles of Programming Languages*, pp. 446–457 (January 1994).

[10] M. Gasbichler, and M. Sperber, "Final Shift for Call/cc: Direct Implementation of Shift and Reset", In *International Conferrence on Functional Programming (ICFP 2002)*, pp. 271–281 (October 2002).

[11] Y. Kameyama, O. Kiselyov, and C. C. Shan. "Shifting the Stage: Staging with Delimited Control", In *Proceeding of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation (PEPM 2009)*, pp. 111–120 (January 2009).

[12] S. L. Peyton Jones, N. Ramsey, and F. Reig. "C--: A Portable Assembly Language that Supports Garbage Collection", In *Principles and Practice of Declarative Programming (PPDP '99)*, pp. 1–28 (October 1999).

[13] K. Sasada, "Ruby Continuation", presented at Continuation Fest, Tokyo (April 2008).

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

[15] E. Sumii, “MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language”. In *ACM SIGPLAN Workshop on Functional and Declarative Programming in Education (FDPE 2005)*, pp. 27–38 (September 2005).

[16] T. Ugawa, N. Minagawa, T. Komiya, M. Yasugi, and T. Yuasa. “Lazy Stack Copying and Stack Copy Sharing for the Efficient Implementation of Continuations”. In *Proceeding of the First Asian Symposium on Programming Languages and Systems (APLAS 2003)*, pp. 410–426 (October 2003).

A The PowerPC Assembly Language

In this section, we briefly describe PowerPC assembly language required to understand code for `reset`, `shift`, and the captured continuation k.

The `mr` (Move Register) instruction moves the content of a general register to the other general register.

```
        mr      r1, r2          ; r1 <- r2
```

The `addi` (Add Immediate) instruction adds the value of the second operand and the third operand, and saves the result in the first operand. So,

```
        addi    r1, r1, 4       ; r1 <- r1 + 4
```

increments register `r1` by 4. The `subi` (Subtract Immediate) instruction subtracts the immediate value from the value of a register.

The `lis` (Load Immediate Shifted) instruction loads the 16-bit immediate value of the second operand shifted left 16-bit into the first operand. We use this instruction when we want to get a code pointer to a function, For example,

```
        lis     r1, ha16(f)     ; high 16bits << 16
        addi    r1, r1, lo16(f) ; low 16bits
```

loads the code pointer of `f` to the register `r1`.

The `lwz` (Load Word and Zero) instruction loads a word from memory into a general register. The `stw` (Store Word) instruction stores the content of a general register into memory. The first operand of `lwz` and `stw` instructions is a general register to be loaded or stored. The effective address being loaded or stored is calculated as sum of the immediate value of the second operand and the content of the third operand. So,

```
        lwz     r1, 4(r2)       ; r1 <- mem(4 + r2)
        stw     r1, 4(r3)       ; r1 -> mem(4 + r3)
```

loads a word of memory from the address `4 + r2` and stores that word into memory of the address `4 + r3`. Add the action of updating the general The `lwzu` (Load Word and Zero with Update) instruction is same as `lwz`, but updates the content of the third operand to the effective address. The same for the `stwu` (Store Word with Update) instruction. So,

```
lwzu    r1, 4(r2)        ; r1 <- mem(4 + r2)
                         ; r2 <- 4 + r2
stwu    r1, 4(r3)        ; r1 -> mem(4 + r3)
                         ; r3 <- 4 + r3
```

is same as an above example, but registers `r2` and `r3` are both incremented by 4.

The `mflr` (Move from Link Register) instruction moves the content of the link register to a general register. The `mtlr` (Move to Link Register) instruction moves the content of a general register to the link register. So,

```
mflr    r1               ; r1 <- LR
mtlr    r2               ; LR <- r2
```

moves the content of the link register to the register `r1` and modifies the content of the link register to the register `r2`. These instructions are used to save and restore the return address. The `blr` (Branch Link Register) instruction jumps to the address indicated by the link register.

The `mtctr` (Move to Count Register) instruction moves the content of a general register to the count register; the `bctrl` (Branch Count Register then Link) instruction jumps to the address indicated by the count register and sets the next address of this instruction to the link register. These two instructions are used to realize "computed goto".

The `b` (Branch) instruction jumps to the label unconditionally without changing the content of the link register.

The `cmpw` (Compare Word) instruction compares the content of two general registers and sets the result to a condition register. The `beq` (Branch Equal) instruction uses this condition register: If two values are equal, jumps to the label of the second operand. So,

```
cmpw    cr7, r1, r2      ; r1 = r2 ?
beq     cr7, l1
```

judges whether the register `r1` equals to the register `r2` and if so, jumps to the label `l1`.