Caml Light + shift/reset = Caml Shift

Moe Masuko Kenichi Asai
Ochanomizu University, Japan
moe@pllab.is.ocha.ac.jp asai@is.ocha.ac.jp

Abstract

We show a direct implementation of `shift/reset` in the Caml Light system. This implementation enables us to program with `shift/reset` in a typed setting easily. The implementation supports the optimization at return time employed in the original ZINC abstract machine. We show various execution examples together with their types. The implementation is expected to promote the use of delimited control operators in practice.

1 Introduction

Although it is widely recognized that the delimited control operators are useful in many situations, implementations of `shift/reset`, especially *direct* implementations in a *typed* setting, have not been popular so far. To break this situation, we presented a direct implementation of `shift/reset` in the MinCaml compiler [Sum05] in our previous work [MA09]. Since it implements a type system that handles answer types explicitly, it enables us to execute various interesting examples. However, restriction of the syntax of MinCaml prohibits us from writing more complex programs. For example, if we want to write a partial evaluator that uses `shift/reset` for let-insertion [Asa07], we need the following data type to represent an abstract syntax tree:

```
type t = Var of string | Lam of string * t | App of t * t
       | Shift of string * t | Reset of t | Let of string * t * t
```

However, since MinCaml does not support user-defined data types, we cannot define such a new type unless we extend the MinCaml compiler itself. Additionally, a lack of garbage collection in MinCaml prevents us from executing large programs.

In this paper, we present a direct implementation of `shift/reset` in the Caml Light system, a lightweight and portable implementation of the Caml language [Ler97]. Caml Light is rich enough to run various interesting programs, yet it is simple enough to add new language features like delimited control operators. We will demonstrate various execution examples and show what it looks like to program with typed delimited-control operators in practice.

Overview In Section 2, we introduce the delimited control operators `shift/reset`. In Section 3, we outline the Caml Light system and the ZINC abstract machine, a core of Caml Light. The direct implementation of `shift/reset` in Caml Light is described in Section 4. Various execution examples are shown in Section 5. We show related work in Section 6. The paper concludes in Section 7.

2 Shift and Reset

The delimited control operators `shift` and `reset` are proposed by Danvy and Filinski [DF90]. Intuitively, `shift` captures the current continuation and `reset` delimits the continuation captured by `shift`.

To show the behavior and the power of `shift/reset`, we consider a search program. Typically, a search program is written with backtracking which makes a program complex. Instead of backtracking, we can write a search program in a straightforward manner if we are given non-deterministic operators.

For example, the following function `queen` solves the N-Queen problem using three non-deterministic operators, `choice_num`, `fail`, and `start`:

```
let queen n =
  let rec loop i solution =
    if i <= 0 then solution
    else let j = choice_num n in
         let solution2 = j :: solution in
         if is_safe solution2 then loop (i - 1) solution2
         else fail ()
  in start (fun () -> loop n []);;
```

A function `is_safe` checks whether the current partial solution satisfies the condition of the N-Queen problem or not. The above function has an extremely simple structure. It chooses a new queen in `j`, checks if it is safe to place the queen in the current partial solution, and if it is, it loops to place more queens to the new partial solution. There is no backtracking. The trick is in the non-deterministic operator `choice_num` which chooses an integer between 1 and `n` non-deterministically. When the choice was not good, the function calls `fail` which aborts the current execution and triggers another choice. In other words, all the backtracking mechanisms are embedded into these operators and we can write a function without thinking about backtracking.

Then, how can we define such operators? One answer is, to use `shift/reset`: [1]

```
let choice_num n = shift (fun k -> fun cont ->
                            let rec loop i =
                              if i <= 0 then cont ()
                              else k i (fun () -> loop (i - 1))
                            in loop n);;
let fail () = shift (fun k -> fun cont -> cont ());;
let start f = reset (fun () -> let r = f () in fun _ -> r)
                    (fun () -> raise Not_found);;
```

The operator `choice_num` saves the current computation in `k` and executes it for all the possible choices by calling `k` in the recursive function `loop`. In this program, `k` and the additional argument `cont` represent the so-called *success continuation* and *failure continuation*, respectively [SICP, Chapter 4.3]. The former is the computation to be backtracked, and the latter keeps other choices to be triggered by `fail`. The failure continuation is passed around by making the context higher-order, which is initialized by `start`.

The non-deterministic operators enable us to separate *how* to search from *what* to search. Such operators can be implemented using `shift/reset`.

3 Caml Light and ZINC

The Caml Light system is a lightweight and portable implementation of the Caml language [Ler97]. It has a relatively simple type system (without modules or objects as was introduced in OCaml) but is sophisticated enough to write complex programs. Therefore, Caml Light serves as a good platform for implementing new features such as `shift` and `reset`.

Let us see how Caml Light executes programs. A program in Caml Light is compiled into a code sequence of the ZINC abstract machine [Ler90]. The ZINC abstract machine uses an argument stack, a return stack, an environment, an accumulator, a heap, and a program counter to execute code. Each component is used as follows:

[1] The definition of these operators is due to Chihiro Kaneko.

Argument stack It stores arguments of function calls and a *mark* (henceforth `MARK`) which indicates a boundary of arguments.

Return stack It mainly stores return frames and temporary values. The latter is called a cache. The components of a return frame include a program counter, an environment, and a cache size (the number of temporary values). The components of a cache include variables introduced by `let` and arguments of function calls. Trap frames which are used to implement exceptions are also stored.

Environment It stores values for variables. When closures are created, a new environment is created from the current cache.

Accumulator It stores a closure when a function is called and the first argument when a primitive is called. The results of function calls are also stored.

Heap It stores various values such as closures, environments, pairs, and floating-point numbers.

The ZINC abstract machine uses two techniques to execute curried functions efficiently by storing a `MARK` (a distinguished value) in the argument stack. The first one is to avoid creation of closures for curried functions when enough arguments are supplied. The second one is to apply a returned function directly to its arguments. Since the introduction of `shift/reset` does not affect the former, we describe the latter in detail.

For example, `(fun x -> x) (fun y -> y + 1) 4` is compiled into the following code sequence:

```
    Pushmark          // push a MARK onto the argument stack
    Quote 4           // push 4 onto the argument stack via the accumulator (accu)
    Push
    Closure 3         // push a closure of Label 3 onto the argument stack via accu
    Push
    Closure 2         // set a closure of Label 2 to accu
    Apply             // call a function stored in accu
  function:
    Label 2:
      Label 5:
        Access 0      // set the 0th element of the environment to accu
        Return        // return to the caller or consume one argument
    Label 3:
      Label 4:
        Quote 1       // push 1 onto the argument stack via accu
        Push
        Access 0      // set the 0th element of the environment to accu
        AddInt        // addition of two integers
        Return        // return to a caller or consume one argument
```

After storing a `MARK` in the argument stack using the `Pushmark` instruction, `4` and a closure `fun y -> y + 1` (`Label 3`) are pushed onto the argument stack. Then, a function `fun x -> x` (`Label 2`) is called. A function application is done by the `Apply` instruction (if the application is not in a tail position). The behavior of `Apply` is defined as follows:

(1) Push a return frame including a program counter (i.e., a return address) onto the return stack.

(2) Move an element stored in the top of the argument stack (i.e., the first argument) to the current cache area.

(3) Invoke a closure stored in the accumulator.

If we do not employ any optimization, a closure `fun y -> y + 1` is returned to a caller as a result of `fun x -> x`, and then it is applied to `4`. However, when an argument is found on the argument stack, we can apply `fun y -> y + 1` to it, shortcutting the return followed by the call. This handling is realized by the `Return` instruction inserted at the end of function bodies. The behavior of `Return` follows:

- If the top of the argument stack *is* a `MARK`: no arguments are applied to the current result (stored in the accumulator); return to a caller.

  (1) Discard the current cache.
  (2) Remove a `MARK` from the top of the argument stack.
  (3) Jump to a caller by restoring a return frame.

- If the top of the argument stack is *not* a `MARK`: at least one argument is found for the current result; execute the application without returning to a caller.

  (1) Discard the current cache.
  (2) Move a value stored in the top of the argument stack (i.e., the first argument) to a new cache area.
  (3) Directly jump to a closure which is stored in the accumulator.

If the top of the argument stack is a `MARK`, it means there is no applicable argument, so we return to a caller by restoring a return frame. If the top of the argument stack is not a `MARK`, on the other hand, it means there is at least one applicable argument. In this case, we can pass the value stored in the top of the argument stack to a closure in the accumulator because the accumulator should hold a closure if a program has passed a type checker. The `Return` instruction enables us to call a function directly without returning to a caller if arguments of a result are given.

The behavior of the `Apply` and `Return` instructions with cache are summarized as follows. (A pair of code and an environment (c,e) represents a closure, ε represents a `MARK`, and a triplet of code, an environment, and a cache size (c,e,m) represents a return frame.)

Code	Accu.	Env.	Size	Arg. stack	Return stack
`Apply`; c_0	$a = (c_1, e_1)$	e_0	m_0	$v.s$	r
c_1	a	e_1	1	s	$v.(c_0, e_0, m_0).r$
`Return`; c_0	a	e_0	m	$\varepsilon.s$	$v_0 \ldots v_{m-1}.(c_1, e_1, m_1).r_0$
c_1	a	e_1	m_1	s	r
`Return`; c_0	$a = (c_1, e_1)$	e_0	m	$v.s$	$v_0 \ldots v_{m-1}.r_0$
c_1	a	e_1	1	s	$v.r_0$

We support the optimization realized by the `Return` instruction in our implementation when the body of `shift` or `reset` returns a function and its argument is available at the return time. See Section 4.3.

4 Implementation

In this section, we describe the implementation of `shift/reset` in Caml Light in detail.

First, we extended the syntax of intermediate languages with `shift` and `reset`. We employed Asai and Kameyama's polymorphic type system [AK07] and slightly modified it to adapt to the evaluation order of Caml Light (i.e., call-by-value, right-to-left). We did not use the purity restriction but used the value restriction employed by Caml Light.

In the bytecode interpreter, we added four instructions `Shift`, `Reset`, `EndSR`, and `CallK` as primitives to support `shift/reset`. Both `shift` and `reset` are compiled into code that sets the argument to the accumulator as a closure and executes `Shift` and `Reset` instructions, respectively. The body of `shift` and `reset` are compiled into code where `EndSR` is inserted before `Return` to enable the optimization shown in the previous section. The `CallK` instruction represents the work to be done when a captured continuation is invoked. The definition of the behavior of these instructions is at the heart of the implementation. Before describing them, however, let us summarize the direct implementation of `shift/reset` in the MinCaml compiler [MA09]:

- When `reset` is executed, set the `reset` mark on the stack.
- When `shift` is executed, move a sequence of stack frames down to the nearest `reset` mark to the heap.
- Maintain the invariant that a return address is always stored immediately under the `reset` mark.

The `reset` mark introduced here (not to be confused with the `MARK` in the ZINC abstract machine) is used to delimit a stack. In our implementation in Caml Light, we basically follow the same implementation method, which is described below.

4.1 The Implementation of Reset

The implementation of the `Reset` instruction is as follows:

(1) Push a return frame onto the return stack.

(2) Store the `reset` mark in both the argument stack and the return stack.

(3) Invoke a closure stored in the accumulator (this closure corresponds to the argument of `reset`).

The invariant of our implementation in the Caml Light system is "in the return stack, a return frame is always stored immediately under the `reset` mark." This invariant is almost the same as the one in the implementation in MinCaml, because a return frame corresponds to a return address. Since the Caml Light system uses two stacks (the argument stack and the return stack), we store the `reset` mark in both of them. The `reset` mark in both the argument stack and the return stack are global pointers (the `reset` pointers). It is stored by `Reset` and `CallK` instructions. We update the value of the `reset` pointer when storing it in the stacks.

4.2 The Implementation of Shift and Continuation Invocation

Based on the above implementation of `Reset`, we implemented the `Shift` instruction as follows:

(1) Allocate a frame in heap to create a closure of a captured continuation.

(2) Move the stack frames down to the nearest `reset` mark (excluding the `reset` mark itself) from both the argument stack and the return stack to the closure allocated in (1).

(3) Store the following information into the closure allocated in (1): the program counter to be executed when this closure is applied, an environment, sizes of the two copied stack frames, a program counter (corresponding to the captured continuation), a cache size, a position of the argument stack, and a trap pointer to the heap (first two information is stored as a standard closure in ZINC, and last two information is required to be compatible with exceptions).

(4) To treat the closure as a first argument of the argument of `shift`, store it into the accumulator.

(5) Invoke the closure stored in the accumulator (this closure corresponds to the argument of `shift`).

In contrast to the standard closures in ZINC that contain a program counter and an environment, closures created for captured continuations contain other information such as copied stack frames and a cache size. They are used to restore the context in which `shift` was called, when a captured continuation is invoked. The instruction to be executed when the closure is applied is `CallK`. In other words, all the closures for captured continuations have the same program counter, which is set at (3). Their behavior is different, however, because copied stack frames are different.

A trap frame which is used to implement exceptions includes two pointers to the stack: one is to the previous trap frame and the other is to the argument stack. A trap frame is stored in the return stack and a pointer to the current trap frame (the trap pointer) is updated when `try` is invoked. When an exception is raised, the return stack and the argument stack are shifted to the trap pointer and a position pointed to by a trap frame, respectively. If frames of the return stack captured by `shift` include a trap frame pointed to by the trap pointer, we have to change the value of the trap pointer to point to the nearest remaining trap frame. Moreover, we have to correctly connect pointers included in the copied stack frames when a captured continuation is invoked.

The invocation of captured continuations is treated as an ordinary function application. When invoked, they execute the `CallK` instruction. The behavior of `CallK` is the following:

(1) Store the `reset` mark in both the argument stack and the return stack.

(2) Set the first argument that is stored in the top of a cache area to the accumulator.

(3) Copy the two stack frames stored in the closure to the top of the argument stack and the return stack.

(4) Set the program to the one counter preserved in the closure (i.e., the program counter when this continuation was captured).

In the implementation, we do not store a `MARK` at (1) because we force storing a return frame before a captured continuation is invoked. We maintain the invariant that a return frame is always stored in the top of the return stack when storing the `reset` mark in this way.

When the execution of the body of `reset` or `shift` is finished, i.e., the `EndSR` instruction is executed, we perform the optimization of `Return`: if a function is returned as a result of `reset` or invocation of a captured continuation and its arguments have been already given, we can call the returned function directly. So, the behavior of `EndSR` is to discard the frames so that `Return` optimization becomes possible:

(1) Discard the frames down to the nearest `reset` mark (including the `reset` mark itself) in both the argument stack and the return stack.

(2) Execute `Return`.

Let us examine execution of expressions including `reset` and `shift` to see how they are handled.

The Return instruction is executed at the end of reset

For example, let us consider an expression `reset (fun () -> fun x -> x) 3`. First, a `MARK` and `3` are pushed onto the argument stack and the argument of `reset` is set to the accumulator. Then, `Reset` is executed and the `reset` mark is stored in the two stacks. By invoking the argument of `Reset`, a closure

`fun x -> x` is set to the accumulator. Next, `EndSR` is executed to remove the `reset` marks from the two stacks, and `Return` is executed. Since `3` is on the top of the argument stack, an application `(fun x -> x) 3` is directly executed.

In this way, we maintain the optimization of `Return` in the presence of `reset`.

The two stack frames are moved by shift

As another example, let us consider an expression `let f x = shift (fun k -> k x) in reset (fun () -> f 3 + 2)`. First, a closure `f` is stored in a cache area. After setting the argument of `reset` to the accumulator, `Reset` is executed and the `reset` marks are stored in the two stacks. Next, `2`, a `MARK` and `3` are pushed onto the argument stack, and `f` is called. Then, a return frame (corresponding to `+`) is stored in the return stack, `3` (an argument of `f`) is moved from the top of the argument stack to a cache area (the return stack), and `Shift` is executed. `Shift` moves frames of the argument stack (`2` and the `MARK`) and frames of the return stack (the return frame and `3`) to the heap.

The execution moves on to the body of `shift`.

The captured continuation copies two stack frames

In the execution of the body of `shift`, a `MARK` and the argument `3` (`x`) are pushed onto the argument stack. `3` is moved to the top of the return stack when `k x` is invoked. This value is regarded as the result of the execution of `shift` and is set to the accumulator since `k` is a captured continuation. Then, `CallK` pushes the `reset` marks onto the top of the two stacks, copies back two stack frames (`2` and the `MARK` for the argument stack, and the return frame and `3` for the return stack) from a clousure, and executes a continuation of `shift`.

The argument stack stores necessary values to execute captured continuations, and the return stack stores temporary values and the calling chain until a captured continuation is invoked. We restore the necessary values to execute a captured continuation by copying back this information to the two stacks.

4.3 Summary of the Implementation

Let us summarize the implementation of `shift/reset` in the ZINC abstract machine. The definition of the four instructions, `Reset`, `Shift`, `CallK`, and `EndSR` is as follows. (rp_a and vs_a represent a `reset` pointer and a stack frame of an argument stack, respectively, while rp_r and vs_r are for a return stack.)

Code	Accu.	Env.	Size	Arg. stack	Return stack
`Reset`; c_0	$a = (c, e)$	e_0	m_0	s	r
c	a	e	0	$rp_a.s$	$rp_r.(c_0, m_0, e_0).r$
`Shift`; c_1	(c, e)	e_1	m_1	$vs_a.rp_a.s$	$vs_r.rp_r.r$
c	$((\texttt{CallK}, e_1), c_1, m_1, vs_a, vs_r)$	e	0	$rp_a.s$	$rp_r.r$
`CallK`; c_0	$((\texttt{CallK}, e_1), c_1, m_1, vs_a, vs_r)$	e	m	s	$v.r$
c_1	v	e_1	m_1	$vs_a.rp_a.s$	$vs_r.rp_r.r$
`EndSR`; `Return`; c	a	e	m	$vs_a.rp_a.s$	$vs_r.rp_r.(c_0, m_0, e_0).r$
`Return`; c	a	e	0	s	$(c_0, m_0, e_0).r$

The compiler $\mathscr{C}[\![-]\!]$ is defined as follows. (The `Cur` instruction creates a closure from an argument and an environment.)

$$\begin{aligned}
\mathscr{C}[\![\texttt{shift}\ (\lambda M)]\!] &= \texttt{Cur}(\mathscr{C}[\![M]\!]; \texttt{EndSR}; \texttt{Return}); \texttt{Shift} \\
\mathscr{C}[\![\texttt{reset}\ (\lambda M)]\!] &= \texttt{Pushmark}; \texttt{Cur}(\mathscr{C}[\![M]\!]; \texttt{EndSR}; \texttt{Return}); \texttt{Reset} \\
\mathscr{C}[\![\texttt{reset}\ (\lambda M)\ N_1 \ldots N_k]\!] &= \\
&\quad \texttt{Pushmark}; \mathscr{C}[\![N_k]\!]; \texttt{Push}; \ldots; \mathscr{C}[\![N_1]\!]; \texttt{Push}; \texttt{Cur}(\mathscr{C}[\![M]\!]; \texttt{EndSR}; \texttt{Return}); \texttt{Reset}
\end{aligned}$$

The compilation of `shift` is simply defined as the creation of the argument closure followed by the `Shift` instruction. At the end of the body of `shift`, the`EndSR` instruction is inserted before the `Return` instruction to remove the `reset` mark. The `Pushmark` instruction is not required for `shift`. This is similar to the compilation of a primitive function application:

$$\mathscr{C}[\![p(M_1,\ldots,M_k)]\!] \quad = \quad \mathscr{C}[\![M_k]\!];\ \texttt{Push};\ \ldots;\ \mathscr{C}[\![M_1]\!];\ \texttt{Push};\ \texttt{Prim}(p)$$

where the number of arguments is fixed and a `MARK` is not inserted.

The compilation of `reset` is split into two cases depending on whether additional arguments are (syntactically) available. In either case, `Pushmark` is required at the beginning to achieve the optimization of `Return` correctly. When the execution of the body of `reset` finishes, `EndSR` is executed to remove the `reset` mark (set by `Reset` or `CallK` instruction) followed by `Return`. At this point, a `MARK` is used to check whether arguments to the current result are available or not. The compilation of `reset` is similar to the compilation of the standard function application:

$$\mathscr{C}[\![(M\ N_1\ \ldots\ N_k)]\!] \quad = \quad \texttt{Pushmark};\ \mathscr{C}[\![N_k]\!];\ \texttt{Push};\ \ldots;\ \mathscr{C}[\![N_1]\!];\ \texttt{Push};\ \mathscr{C}[\![M]\!];\ \texttt{Apply}$$

Here, `Pushmark` is used to indicate available arguments.

4.4 Garbage Collection

The Caml Light system supports generational garbage collection (GC). The garbage collector searches pointers stored in the argument stack and the return stack when the GC is invoked. Frames pointed to by these pointers are reallocated onto the heap and the pointers are rewritten to hold the address of the reallocated frames. We call this process recursively according to header information (a size of a frame, a color used in the GC, and a kind of a frame) of reallocated frames. The frames included in the closure created by `Shift` are handled correctly because they come with header information. The only addition to the GC system is a correct traversal of the `reset` pointers stored in the stacks.

5 Examples

In this section, we demonstrate various execution examples. We give relatively many examples to show what arises in a use of `shift/reset` in a typed setting. Because almost all the examples require knowledge on answer types, we first review the types of `shift/reset` [AK07, DF89].

5.1 The Type System for Shift and Reset

A judgment of the type system for `shift/reset` is defined as $\Gamma;\ \alpha \vdash e : \tau;\ \beta$, which means that the expression e has the type τ under the type context Γ, and the execution of e changes the answer type from α to β. Intuitively, the answer type is a type of the value that surrounding `reset` returns.

On the other hand, if $\Gamma;\ \alpha \vdash e : \tau;\ \alpha$ holds for any type variable α, i.e., the execution of the expression e does not affect the answer type, e is called *pure*, and this judgment is represented as $\Gamma \vdash_p e : \tau$. Pure expressions do not have any control effect and do not change the answer type. Constant, functions, and expressions surrounded by `reset` are pure.

A function type is expressed as `S / A -> T / B`. It is a type of a function from the type `S` to `T`, and the answer type is changed from `A` to `B` when the function is invoked. If a function is pure having the same type variable as its answer types, its type is simply written as `S -> T`. We also introduce a notation `S => T` to indicate an *impure* function. It is a function from `S` to `T` whose answer type are not the same type variable but are hidden for the ease of readability.

5.2 N-Queen

Let us execute the queen program shown in Section 2. Their types are inferred as follows:

```
choice_num : int => int
fail : unit => 'a
start : (unit => 'a) => 'b
queen : int -> int list
# queen 4;;
- : int list = [2; 4; 1; 3]
```

Since `choice_num` receives an integer and returns an integer non-deterministically, `int => int` properly describes the behavior of `choice_num`. The use of `=>` instead of `->` indicates that an execution of this function may incur control effects. In fact, since `choice_num` uses `shift` in its definition, passing an integer to `choice_num` causes the current continuation to be captured.

The type of `fail` resembles that of `raise`: both return an arbitrary type 'a. Because `fail` stops the current computation and triggers backtracking, the `fail` expression itself can have an arbitrary type.

In this example, we can regard `=>` as an ordinary function type, forgetting about the answer type. However, we often have to consider the answer type if we use `shift` and `reset` in a program. To make the answer type explicit, our implementation supports a directive to change the way types are shown. Although our implementation omits the answer type by default and uses `=>` when control effects are used, if they are required, answer types can be displayed explicitly.[2]

5.3 Times

The function `times` receives a list of integers and returns their product. Since the result will be `0` if the given list contains `0`, we can throw away the current computation and return `0` whenever we encounter `0`.

```
# let rec times0 = function
   | [] -> 1
   | 0 :: _ -> shift (fun k -> 0)
   | a :: rest -> a * times0 rest;;
times0 : int list => int = <fun>
# let times lst = reset (fun () -> times0 lst);;
times : int list -> int = <fun>
# times [1; 2; 3];;
- : int = 6
# times [1; 2; 0; 3];;
- : int = 0
```

Because `0` is immediately returned when `0` is found, it constrains the type of its context to be `int`. With explicit answer types, the type of `times0` is actually `int list / int -> int / int`. Therefore, a program such as `reset (fun () -> print_int (times0 [1; 2; 3]))`, is not typable because the type of surrounding `reset` is not `int`. On the other hand, the type of `times` is pure. The answer type of `times` is not changed even though impure `times0` is executed inside, because its effect is delimited within the `reset`.

[2] When all the answer types are explicit, the types of `choice_num`, `fail`, and `start` become as follows:

```
choice_num :  int / ((unit / 'a -> 'b / 'c) / 'a -> 'b / 'c) ->
              int / ((unit / 'a -> 'b / 'c) / 'a -> 'b / 'c)
fail :  unit / 'a -> 'b / ((unit / 'c -> 'd / 'e) / 'c -> 'd / 'e)
start :  (unit / ('a -> 'b) -> 'b / ((unit -> 'c) / 'd -> 'e / 'f)) / 'd -> 'e / 'f
```

5.4 Append and Sprintf

As an example of answer type modification, we show the append function, with the printing of answer types enabled:

```
# let rec append = function
   | [] -> shift (fun k -> k)
   | a :: rest -> a :: append rest;;
append : 'a list / 'b -> 'a list / ('a list -> 'b) = <fun>
# let app123 = reset (fun () -> append [1; 2; 3]);;
app123 : int list / '_a -> int list / '_a = <fun>
# let app123' lst = (reset (fun () -> append [1; 2; 3])) lst;;
app123' : int list -> int list = <fun>
# app123 [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Suppose that append is invoked under the answer type 'b. The type of a continuation captured by shift (fun k -> k) is 'a list -> 'b because append itself returns a value of type 'a list. This continuation is returned to the surrounding reset, so the answer type of append is modified from 'b to 'a list -> 'b. It means this example cannot be typed in a system that does not allow the answer type modification.

Because the Caml Light system employs the value restriction, the answer type of the let-bound value app123 is a weak polymorphic type '_a.[3] In this case, we can avoid the weak polymorphic type by η-expanding app123 and defining app123 as a function that receives a list of integers, as in app123'.

As another example, we show sprintf function [Asa09].

```
# let int x = string_of_int x;;
int : int -> string = <fun>
# let str (x : string) = x;;
str : string -> string = <fun>
# let percent to_str = shift (fun k -> fun x -> k (to_str x));;
percent : ('a / 'b -> 'c / 'd) / 'e -> 'c / ('a / 'b -> 'e / 'd) = <fun>
# let sprintf p = reset (fun () -> p ());;
sprintf : (unit / 'a -> 'a / 'b) -> 'b = <fun>
# sprintf               (* sprintf ("The value of %s is %d.", "x", 3) *)
    (fun () -> "The value of " ^ (percent str) ^ " is " ^ (percent int) ^ ".") 3 "x";;
- : string = "The value of x is 3."
```

A function percent receives a function to_str that returns the string representation of its argument. It then aborts computation and returns a function to the enclosing context. When the returned function is applied to an argument, it changes its representation into a string and resumes the aborted computation. We can understand this behavior from the type of percent, if we instantiate the type of percent as follows:

```
('a -> string) / 'e -> string / ('a -> 'e)
```

The answer type of percent (the type of its context) is modified from the original 'e to 'a -> 'e, enabling to accept another argument of type 'a.

A function sprintf receives a thunk that represents a format and executes it under reset. The result of applying sprintf can accept as many arguments as the number of percent in the format. In

[3] This type becomes a polymorphic type 'a if we employ the purity condition [AK07].

the above example, it receives two arguments 3 (of type `int`) and `"x"` (of type `string`) and returns a string `"The value of x is 3."` Because the evaluation order of the Caml Light system is right-to-left, `(percent int)` is evaluated before `(percent string)` is evaluated. So the order of arguments is different from a standard `sprintf` function.

In this example, the result of `sprintf (fun () -> ...)` is a function that receives two values of types `int` and `string`, and returns a string. This is the example where execution would fail if we do not consider whether a `MARK` is on the top of the argument stack or not when the execution of the body of `reset` is finished.

5.5 Partial Evaluation

As an example that uses a data type definition, we show an online partial evaluator using `shift/reset` [Asa07]. In the following program, a function `gensym` creates a new variable, `add` adds a new element to an environment, `get` gets a value from an environment using a key, and `empty_env` represents an empty environment. The main function is `peval`. It receives a term of the lambda calculus extended with `shift/reset` and `let` represented as an abstract syntax tree, defined as shown in Section 1, and returns a partially evaluated term.

```
# (* dynamic type and static type *)
  type sval_t = Dyn of t | Sta of t * (sval_t / sval_t -> sval_t / sval_t);;
Type sval_t defined.
# (* get a program (term) *)
  let lift = function Dyn d -> d | Sta (d, s) -> d;;
lift : sval_t -> t = <fun>
# let rec peval term env = match term with                    (* partial evaluator *)
   | Var x -> get x env
   | Lam (x, t) -> let new_x = gensym x in let new_k = gensym "k" in
       Sta (Lam (new_x, Shift (new_k,
              lift (reset (fun () -> Dyn (Reset (App (Var new_k,
                lift (peval t (add env x (Dyn (Var new_x))))))))))),
            fun arg -> peval t (add env x arg))
   | App (t1, t2) -> let f = peval t1 env in let a = peval t2 env in
       (match f with
          | Dyn d -> let new_t = gensym "t" in
              shift (fun cont -> Dyn (Let (new_t, App (d, lift a),
                                           lift (cont (Dyn (Var new_t))))))
          | Sta (d, s) -> s a)
   | Shift (k, t) ->
       shift (fun cont -> let new_v = gensym "v" in
                peval t (add env k
                           (Sta (Lam (new_v, Reset (lift (cont (Dyn (Var new_v))))),
                                 cont))))
   | Reset t -> reset (fun () -> peval t env)
   | Let (x, t1, t2) -> peval (App (Lam (x, t2), t1)) env;;
peval : t => (string => sval_t) => sval_t = <fun>
# let f term = init ();                                      (* initialize gensym *)
    let result = lift (reset (fun () -> peval term empty_env)) in    (* then do PE, *)
    print_string (to_string result); print_newline ();;       (* and print a result *)
f : t -> unit = <fun>
# let e = Lam ("x", Reset (App (Shift ("k", Var "k"), Var "x")));;
e : t = Lam ("x", Reset (App (Shift ("k", Var "k"), Var "x")))
```

```
# f e;;
(lam x1. (shift k2. (reset (k2 @ (lam v3. (reset (let t4 = (v3 @ x1) in t4)))))))
- : unit = ()
```

A type `sval_t` represents a symbolic value, which is either a dynamic value or a static value. The former is used to residualize a program, while the latter is used to reduce a program during partial evaluation. A dynamic value `Dyn` brings a program text, while a static value `Sta` brings a static value in addition to a program text. The function `lift` turns a symbolic value into a program text by extracting its dynamic part. In `peval`, `shift` is used for let-insertion of a function application and for the evaluation of `shift` itself.

In the current implementation, if the answer types of a function type is omitted, as in `let f (x : int -> int) = x`, we regard this function as pure and complement them with a polymorphic type variable `'a`. However, we disallow omission of answer types in a type declaration, because complementing type variables leads to unbound type variables:

```
# type t = A of int / 'a -> int / 'a;;
Toplevel input:
> type t = A of int / 'a -> int / 'a;;
>                                 ^^
The type variable a is unbound.
```

Although some programs are typable by mechanically adding type parameters, as in `type 'a t = A of int / 'a -> int / 'a`, this is not always the case. In fact, the following natural definition of `sval_t` for the above example:

```
type sval_t = Dyn of t | Sta of t * (sval_t -> sval_t);;
```

does not work, because neither

```
type 'a sval_t =
  | Dyn of t | Sta of t * ('a sval_t / 'a -> 'a sval_t / 'a);;
```

nor

```
type ('a, 'b) sval_t =
  | Dyn of t | Sta of t * (('a, 'b) sval_t / 'a -> ('a, 'b) sval_t / 'b);;
```

passes the type check. Rather than automatically complementing type variables, we ask the user to write them explicitly, as in the definition of type `sval_t` at the beginnig of this section.

6 Related Work

Gasbichler and Sperber presented a direct implementation of `shift/reset` and `control` in the Scheme 48 system [GS02]. They showed that the direct implementation eases the overhead of the indirect implementation using `call/cc` and improves the execution efficiency. They employed incremental stack/heap strategy and used PreScheme, a virtual machine for the Scheme 48 system.

Rompf et al. implemented `shift/reset` that supports the answer type modification using the pluggable typing architecture of Scala [RMO09]. They discriminate expressions whether they have control effects or not using types, and selectively transform them into CPS. Their method improves the efficiency problem of full CPS transformation and achieves the implementation in a widely-used programming language.

Kiselyov showed a generic approach to implement multi-prompt delimited control operators and applied it to OCaml and Scheme [Kis10]. He implemented `shift/reset` directly without modifying the existing implementation that supports both the exception and the recovery from the stack overflow.

7 Conclusion and Future Work

We described the direct implementation of `shift/reset` in the Caml Light system and demonstrated various examples on it. Although the basic idea is the same as our previous work on the MinCaml compiler, applicability of the result differs significantly. We can now program with `shift/reset` in the typed setting easily and experiment with various programs on it.

We are now trying to establish the formal correctness of our implementation using the functional derivation approach. The comparison with other implementation techniques is remained as future work. We are also interested in writing various applications using `shift` and `reset`. How to tame the verbose answer types, not only in the types of expressions but also in error messages, is another interesting topic.

Acknowledgment We would like to thank anonymous reviewers for their helpful comments.

References

[AK07] Asai, K. and Y. Kameyama. Polymorphic delimited continuations. In *5th Asian Symposium on Programming Languages and Systems, Lecture Notes in Computer Science 4807*, pages 239–254, November 2007.

[Asa07] Asai, K. Logical relations for call-by-value delimited continuations. *Trends in Functional Programming*, 6:64–78, 2007.

[Asa09] Asai, K. On typing delimited continuations: Three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 2009.

[SICP] Abelson, H., G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[DF89] Danvy, O. and A. Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[DF90] Danvy, O. and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, June 1990.

[GS02] Gasbichler, M. and M. Sperber. Final shift for call/cc: direct implementation of shift and reset. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, pages 271–282, October 2002.

[Kis10] Kiselyov, O. Delimited control in OCaml, abstractly and concretely. system description. In *10th International Symposium on Functional and Logic Programming*, pages 304–320, April 2010.

[Ler90] Leroy, X. The Zinc experiment: An economical implementation of the ML language. Technical report, INRIA, February 1990.

[Ler97] Leroy, X. *The Caml Light system release 0.74*, December 1997.

[MA09] Masuko, M. and K. Asai. Direct implementation of shift and reset in the MinCaml compiler. In *Proceedings of the 2009 ACM SIGPLAN workshop on ML*, pages 49–60, August 2009.

[RMO09] Rompf, T., I. Maier, and M. Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 317–328, 2009.

[Sum05] Sumii, E. MinCaml: a simple and efficient compiler for a minimal functional language. In *Proceedings of the 2005 workshop on Functional and Declarative Programming in Education*, pages 27–38, September 2005.