# Introduction to Programming with Shift and Reset

Kenichi Asai Oleg Kiselyov

September 23, 2011

**Abstract**

The concept of continuations arises naturally in programming: a conditional branch selects a continuation from the two possible futures; raising an exception discards a part of the continuation; a tail-call or 'goto' continues with the continuation. Although continuations are implicitly manipulated in every language, manipulating them explicitly as first-class objects is rarely used because of the perceived difficulty.

This tutorial aims to give a gentle introduction to continuations and a taste of programming with first-class delimited continuations using the control operators shift and reset. Assuming no prior knowledge on continuations, the tutorial helps participants write simple co-routines and non-deterministic searches. The tutorial should make it easier to understand and appreciate the talks at CW 2011.

We assume basic familiarity with functional programming languages, such as OCaml, Standard ML, Scheme, and Haskell. No prior knowledge of continuations is needed. Participants are encouraged to bring their laptops and program along.

## 1 Introduction

Continuations represent *the rest of the computation*. Manipulation of continuations is a powerful tool to realize complex control flow of programs without spoiling the overall structure of programs. It is a generalization of exception handling, but is far more expressive.

Traditional way to handle continuations explicitly in a program is to transform a program into continuation-passing style (CPS) [13]. However, it requires us to transform whole the program (or at least the part of programs where we want to manipulate continuations) into CPS. If we want to manipulate continuations in a more intuitive direct-style program, we need control operators.

The most famous control operator is `call/cc` found in Scheme and Standard ML. However, continuations captured by `call/cc` is the whole continuation that includes all the future computation. In practice, most of the continuations that we want to manipulate are only a part of computation. Such continuations are called *delimited* or *partial* continuations.

There are several control operators that capture delimited continuations, such as Felleisen's `control`/`prompt` [6], Danvy and Filinski's `shift`/`reset` [4], and Gunter, Rémy, and Riecke's `cupto`/`prompt` [8]. Among them, we use `shift`/`reset` in this tutorial, because their foundation is established the most, having sound and complete axiomatization [9] as well as polymorphism [2]. Furthermore, because of its clear connection to CPS, many applications of `shift`/`reset` have been proposed.

In this tutorial note, we introduce programming with `shift` and `reset` with various examples to experience what it looks like to program with `shift` and `reset`.

### 1.1 Implementation of `shift` and `reset`

There are a number of ways to use `shift` and `reset` in various programming languages.

- Filinski [5] showed that `shift`/`reset` can be emulated using `call/cc` and a reference cell. One can use `shift`/`reset` using this method in Scheme and Standard ML. In this emulation, one has to fix the answer type beforehand for each context.
- Gasbichler and Sperber implemented `shift`/`reset` directly on Scheme48 [7]. It is reported that the implementation is much more efficient than emulation using `call/cc`.
- Racket supports various control operators including `shift`/`reset`.
- Kiselyov implements delimited control in Delimcc library [10] for OCaml, Scheme, and Haskell.

- Masuko and Asai implemented `shift/reset` directly in MinCaml compiler [11]. It supports type system with answer-type modification and polymorphism. The same idea is used to implement OchaCaml, an extension of Caml Light with `shift/reset` [12].

In this tutorial note, we use OchaCaml to explain and demonstrate programming with `shift/reset`.

### 1.2 Organization

The next section explains programming with `shift` and `reset` with examples and exercises. The section is almost self-contained and does not require base theory on delimited continuations. For those who are interested in the foundations of `shift/reset`, Section 3 shows the brief overview of the base theory of `shift/reset`. For more details, however, readers are referred to read technical papers.

### 1.3 Prerequisite

We assume general knowledge on functional programming languages, such as OCaml, Standard ML, Scheme, and/or Haskell, but no prior knowledge on continuations is required. In Section 3, we assume general knowledge on the $\lambda$-calculus with let polymorphism, its evaluation rules and type system.

## 2 Programming with Shift and Reset

### 2.1 What are continuations?

Continuations represent *the rest of the computation*. When a complex program is executed, a next expression to be evaluated (called *redex*, a reducible expression) is selected, the redex is evaluated, and the result is used to execute "the rest of the computation". This last "rest of the computation" is the continuation of the redex. Thus, the concept of continuations arises in any programming languages regardless of whether control operators are supported or not.

Continuations are relative to where the current expression being evaluated is. To clarify the currently executed expression, we enclose the current expression with a bracket written as `[...]`. For example, consider a simple arithmetic expression $3 + 5 * 2 - 1$. If we are about to execute $5 * 2$ of this expression, the current expression is $5 * 2$. To indicate it, we write $3 + [5 * 2] - 1$. At this moment, the continuation (or the *current* continuation) is $3 + [\cdot] - 1$, in other words, "given a value for $[\cdot]$ (called a *hole*), add three to it and subtract one from the sum." A continuation is similar to a function in that it receives a value for a hole and evaluates the rest of the computation using the received value.

We can also understand the current continuation as the discarded computation when the current expression is replaced with an aborting expression, such as `raise Abort`. In the above example, the current continuation of $3 + [5 * 2] - 1$ is $3 + [\cdot] - 1$, because it is the discarded computation when executing $3 + $ `[raise Abort]` $- 1$.

The current continuation changes as the computation proceeds. After the evaluation of $5 * 2$ finishes, the expression becomes $3 + 10 - 1$. Since the next expression to be evaluated is $3 + 10$, the current expression becomes $3 + 10$ written as $[3 + 10] - 1$. At this moment, the current continuation is $[\cdot] - 1$, namely, "subtract one." After the evaluation of $3 + 10$ finishes, the expression becomes $13 - 1$. The only remaining expression to be evaluated is $13 - 1$ written as $[13 - 1]$. The current continuation at this point is the empty context $[\cdot]$, namely, "do nothing."

**Exercise 1** In the following expressions, identify the next expression to be evaluated and its continuation by marking the former with $[\cdot]$. What is the type of the former? Given a value of this type, what is the type of the value returned by the continuation?

(1) `5 * (2 * 3 + 3 * 4)`

(2) `(if 2 = 3 then "hello" else "hi") ^ " world"`

(3) `fst (let x = 1 + 2 in (x, x))`

(4) `string_length ("x" ^ string_of_int (3 + 1))`

### 2.2 What are delimited continuations?

Delimited continuations are continuations whose extent is delimited. In the expression $3 + [5 * 2] - 1$, we implicitly assumed that the rest of the current expression spans whole the expression, that is, $3 + [\cdot] - 1$. Rather than taking

whole the rest of the computation, however, we sometimes want to capture only a part of it. Such continuations are called *delimited* continuations.

The extent to which the current delimited continuation spans is designated by an explicit *delimiter* $\langle \cdots \rangle$. For example, in the expression $\langle 3 + \texttt{[}5 * 2\texttt{]} \rangle - 1$, the current delimited continuation at `[5 * 2]` is only $\langle 3 + \texttt{[} \cdot \texttt{]} \rangle$ and does not contain $-1$.

## 2.3 Delimiting continuations: `reset`

In OchaCaml, continuations are delimited by the `reset` construct:

`reset (fun () ->` $M$`)`

It receives a thunk `(fun () ->` $M$`)` and executes its body $M$ in a delimited context. If a continuation is captured during the execution of $M$, it is delimited up to this `reset`. When $M$ evaluates to a value, it becomes the value of this whole `reset` expression.

For example, in the following expression:

```
reset (fun () -> 3 + [5 * 2]) - 1
```

the current delimited continuation at `[5 * 2]` becomes "add three" and does not contain "subtract one." In this example, however, because continuations are not captured, the execution goes without any surprise: $5 * 2$ is reduced to 10, 3 is added, 13 becomes the result of `reset`, and the final result is 12.

The delimited continuation is the discarded computation when `reset` is replaced with `try ... with Abort -> ...` and the current expression with `raise Abort`. For example, the current delimited continuation of

```
reset (fun () -> 3 + [5 * 2]) - 1
```

is $3 + \texttt{[} \cdot \texttt{]}$, because it is the discarded computation when executing

```
(try 3 + [raise Abort] with Abort -> 0) - 1
```

**Exercise 2** In the following expressions, identify the delimited continuation and its type.

(1) `5 * reset (fun () -> [2 * 3] + 3 * 4)`

(2) `reset (fun () -> if [2 = 3] then "hello" else "hi") ^ " world"`

(3) `fst (reset (fun () -> let x = [1 + 2] in (x, x)))`

(4) `string_length (reset (fun () -> "x" ^ string_of_int [3 + 1]))`

## 2.4 Capturing continuations: `shift`

To capture the current delimited continuation in OchaCaml, we use `shift` construct:

`shift (fun k ->` $M$`)`

The execution of this expression proceeds in three steps:

(1) The current delimited continuation is cleared.

(2) The cleared continuation is bound to `k`.

(3) The body $M$ is executed.

We will see how to use `shift` in the subsequent sections in detail.

## 2.5 How to discard delimited continuations

The first use of `shift` is to discard delimited continuations by:

`shift (fun _ ->` $M$`)`

Here, `_` is a variable that does not occur anywhere else in the program. (The above program is the same as `shift (fun k ->` $M$`)` where where `k` is not mentioned in $M$ at all.) The execution of `shift (fun _ ->` $M$`)` proceeds as follows:

(1) The current delimited continuation is cleared.

(2) The cleared continuation is passed to `fun _ ->` $M$ as an argument, but it will never be used because it is not mentioned in $M$.

(3) The body $M$ is executed.

Since the body of `shift` does not mention the captured continuation, it is discarded and the current context up to the enclosing `reset` is replaced with $M$. In other words, we can discard (or abort) the computation up to the enclosing `reset`.

For example, to discard the continuation of $3 + [5 * 2] - 1$, we enclose whole the expression with `reset` and replace $[\cdot]$ with `shift (fun _ -> M)`.

```
# reset (fun () -> 3 + shift (fun _ -> 5 * 2) - 1) ;;
- : int = 10
#
```

In this case, $M$ is $5 * 2$, so the result becomes 10. We can even return a value of different type.

```
# reset (fun () -> 3 + shift (fun _ -> "hello") - 1) ;;
- : string = "hello"
#
```

Note that the discarded continuation $3 + [\cdot] - 1$ had type `int -> int`. Even though the current context delimited by `reset` was going to return a value of type `int`, what is actually returned is a string. Still, the expression is well-typed. We will come back to this type modification later.

The discarded continuation is only up to the enclosing `reset`. In the following example, only "add three" is discarded:

```
# reset (fun () -> 3 + shift (fun _ -> 5 * 2)) - 1 ;;
- : int = 9
#
```

In this case, we cannot return a string, because the value of `reset` is subtracted by one afterwards:

```
# reset (fun () -> 3 + shift (fun _ -> "hello")) - 1 ;;
Toplevel input:
> reset (fun () -> 3 + shift (fun _ -> "hello")) - 1 ;;
> ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type string,
but is used with type int.
#
```

**Exercise 3** Discard the current continuations and return some values in the following expressions by replacing $[\cdot]$ with `shift (fun _ -> M)` for some $M$.

(1) `5 * reset (fun () -> [·] + 3 * 4)`

(2) `reset (fun () -> if [·] then "hello" else "hi") ^ " world"`

(3) `fst (reset (fun () -> let x = [·] in (x, x)))`

(4) `string_length (reset (fun () -> "x" ^ string_of_int [·]))`

**Exercise 4** Given a list of integers, the following function returns the product of all the elements.

```
(* times : int list -> int *)
let rec times lst = match lst with
    [] -> 1
  | first :: rest -> first * times rest ;;
```

For example, `times [2; 3; 5]` evaluates to 30. Suppose we apply `times` to `[2; 3; 0; 5]`. Because 0 exists in the list, we know that the result becomes 0 without performing any multiplication. Such a behavior can be realized by discarding the current continuation and returning 0 when 0 is found in the list. Modify the above definition of `times` to include the following clause

```
  | 0 :: rest -> ...
```

to implement this behavior. How can we invoke the modified `times`?

## 2.6 How to extract delimited continuations

The second use of `shift` is to extract delimited continuations by:

`shift (fun k -> k)`

The execution of `shift (fun k -> k)` proceeds as follows:

(1) The current delimited continuation is cleared.

(2) The cleared continuation is bound to `k`.

(3) The body `k` is executed.

Since the body of `shift` is just a variable representing the captured continuation, the last step finishes immediately and returns the captured continuation. Since the current delimited continuation is cleared, the value returned by the enclosing `reset` becomes the captured continuation. Thus, by executing `shift (fun k -> k)`, we can extract the current delimited continuation.

For example, if we want to capture the continuation of $3 + [5 * 2] - 1$, we enclose whole the expression with `reset`, replace $[\cdot]$ with `shift (fun k -> k)`, and bind the result to a variable for later use.

```
# let f x = reset (fun () -> 3 + shift (fun k -> k) - 1) x ;;
f : int -> int = <fun>
#
```

Because the returned continuation is a function, we $\eta$-expand it and bind it to a function `f` with an explicit argument `x` above. We could have written:

```
# let f = reset (fun () -> 3 + shift (fun k -> k) - 1) ;;
f : int => int = <fun>
#
```

but then, because what is bound to `f` is not textually a value, we obtain a weakly polymorphic continuation that can be used only in a context with a specific answer type. In OchaCaml, such a special status of a function is indicated by a new arrow type `=>`. See Section 3.4 for details on `=>`. Here, we avoid it by $\eta$-expanding the definition of `f`.

Now, `f` is bound to a function that, when applied, invokes the captured continuation "add three and subtract one" with the applied value.

```
# f 10 ;;
- : int = 12
#
```

In this case, `f` behaves the same as `fun x -> reset (fun () -> 3 + x - 1)`.

**Exercise 5** Extract delimited continuations of the following expressions by replacing $[\cdot]$ with `shift (fun k -> k)` and give names to them. Figure out what the extracted continuations do by actually applying them to values. What are their types?

(1) `reset (fun () -> 5 * ([·] + 3 * 4))`

(2) `reset (fun () -> (if [·] then "hello" else "hi") ^ " world")`

(3) `reset (fun () -> fst (let x = [·] in (x, x)))`

(4) `reset (fun () -> string_length ("x" ^ string_of_int [·]))`

**Exercise 6** The following program traverses over a given list, and returns the list as is without any modification. In other words, it is an identity function over a list.

```
(* id : 'a list -> 'a list *)
let rec id lst = match lst with
    [] -> []
  | first :: rest -> first :: id rest ;;
```

Suppose we call this function with an argument list `[1; 2; 3]` as follows:

```
reset (fun () -> id [1; 2; 3]) ;;
```

The function traverses over this list, and at last it reaches an empty list. What is the continuation at this moment? Extract it by replacing `[]` with `shift (fun k -> k)`. What is the type of the extracted continuation? What does it do?

## 2.7 How to preserve delimited continuations

We can not only extract/discard continuations but also save them temporarily in a data structure, and resume them later. For example, let us consider traversing over a tree defined as follows:

```
type tree_t = Empty
            | Node of tree_t * int * tree_t ;;
```

The following function traverses over a tree from left to right in a depth-first manner and prints all the values found in the tree.

```
(* walk : tree_t -> unit *)
let rec walk tree = match tree with
    Empty -> ()
  | Node (t1, n, t2) ->
      walk t1;
      print_int n;
      walk t2 ;;
```

For example, we have:

```
# let tree1 = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty)) ;;
tree1 : tree_t = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty))
# walk tree1 ;;
123- : unit = ()
#
```

The function `walk` traverses all the nodes in one go. But sometimes, we want to see each value one at a time and process it before receiving the next value (if any). To realize such a behavior, we replace `print_int n` with a `shift` expression as follows:

```
(* walk : tree_t => unit *)
let rec walk tree = match tree with
    Empty -> ()
  | Node (t1, n, t2) ->
      walk t1;
      yield n;
      walk t2 ;;
```

where `yield` is defined as follows:

```
(* yield : int => unit *)
let yield n = shift (fun k -> Next (n, k)) ;;
```

When a node is found, the function `walk` calls `yield` to abort and return the value `n` together with the current continuation in a suitable constructor `Next` (to be defined soon). The caller of `walk` will thus receive the first value `n` immediately, suspending traversal over other nodes. When the caller wants another value, it resumes the traversal by passing a unit `()` to the continuation. This unit becomes the value of `yield` and the traversal continues to `walk t2`. This way, by returning a value with the current continuation, one can stop and resume execution of a function.

Now that `walk` captures continuations, we need to enclose it with `reset` to invoke it. However, we cannot simply enclose `walk` with `reset` as follows:

```
reset (fun () -> walk tree1) ;;
```

because it results in a type error. Remember that whenever `walk` finds a node, it returns `Next (n, k)` to the enclosing `reset`. On the other hand, if the list passed to `walk` is an empty list, `walk` returns a unit. Thus, if the call to `walk` is directly enclosed by `reset`, the `reset` can return both `Next (n, k)` and a unit, leading to a type error.

To avoid the type error, we need another constructor `Done` (to be defined soon) indicating that there are no more nodes in a tree. Rather than finishing traversal with a unit, we return `Done` as in the following definition:

```
(* start : tree_t -> 'a result_t *)
let start tree =
  reset (fun () -> walk tree; Done) ;;
```

Observe how the returned value in the body of `shift` in `yield` called from `walk` affects the type of values returned by the enclosing `reset`. This implies that typing a `shift` expression requires information on the type of the enclosing `reset`.

The remaining task is to define the two constructors, `Next` and `Done`. They are defined as follows:

```
type 'a result_t = Done
                 | Next of int * (unit / 'a -> 'a result_t / 'a) ;;
```

The constructor `Done` has no arguments, while `Next` has two arguments, one for an integer and the other for a continuation. The type of a captured continuation is basically a function from `unit` to `'a result_t`: given a unit, it returns either `Done` or `Next`. In addition, however, one has to specify their *answer types*. Since answer types of

captured continuations are polymorphic, we added a type parameter `'a` and use it for both the answer types. We will come back to answer types in detail in the subsequent sections.

We can now traverse over a tree by calling `start`. For example, the following function prints out all the integers in the nodes of a tree:

```
(* print_nodes : tree_t -> unit *)
let print_nodes tree =
  let rec loop r = match r with
      Done -> ()        (* no more nodes *)
    | Next (n, k) ->
        print_int n;    (* print n *)
        loop (k ()) in  (* and continue *)
  loop (start tree) ;;
```

At the last line, traversal over `tree` is initiated by calling `start`. The inner function `loop` examines the result: if it is `Done`, there are no more nodes in `tree`; if the result is `Next`, it processes the current value `n` and continues (resumes) traversal by passing `()` to `k`. By calling `print_nodes` with `tree1`, We have:

```
# print_nodes tree1 ;;
123- : unit = ()
#
```

Likewise, the following function adds up all the integers in a tree:

```
(* add_tree : tree_t -> int *)
let add_tree tree =
  let rec loop r = match r with
      Done -> 0
    | Next (n, k) -> n + loop (k ()) in
  loop (start tree) ;;
```

We have:

```
# add_tree tree1 ;;
- : int = 6
#
```

Using this idea, we can implement a kind of co-routines where two processes execute alternately. The following exercise demonstrates it in the simplest form.

**Exercise 7** Write a function `same_fringe`. Given two trees of type `tree_t`, it traverses over the two trees from left to right in a depth-first manner, and checks if the encountered sequences of numbers are the same for the two trees. For example, fringes of the following two trees are the same:

```
let tree1 = Node (Node (Empty, 1, Empty), 2, Node (Empty, 3, Empty)) ;;
let tree2 = Node (Empty, 1, Node (Empty, 2, Node (Empty, 3, Empty))) ;;
```

We can implement `same_fringe` by first converting the trees into lists and comparing the resulting lists. However, this implementation requires to traverse all the trees even if the first elements of the two trees are already different. Instead, write a function that can return `false` at the first point where two numbers differ without further traversing the rest of the trees.

## 2.8 How to wrap delimited continuations: printf

Suspended computation captured in `k` can be resumed by applying `k` to a value. Rather than applying `k` immediately, if we wrap the application with `fun`, we can defer the resumption of captured computation. Furthermore, it enables us to access an argument of the enclosing `reset`.

For example, consider the following expression:

```
shift (fun k -> fun () -> k "hello")
```

It captures the current continuation, binds it to `k`, aborts the current computation, and returns a thunk `fun () -> k "hello"` as a result of the enclosing `reset`. Namely, it temporarily suspends the current computation and waits for a unit to be passed outside the enclosing `reset`. When it receives a unit, it resumes computation with the value `"hello"`.

Suppose the above expression is placed in the context `[...] ^ " world"` as follows:[1]

[1]The definition of `f` is $\eta$-expanded to make `f` answer-type polymorphic.

```
# let f x = reset (fun () ->
              shift (fun k -> fun () -> k "hello") ^ " world") x ;;
f : unit -> string = <fun>
#
```

We then obtain a thunk `f`, which will resume the computation with the value `"hello"` when `()` is passed:

```
# f () ;;
- : string = "hello world"
#
```

Observe that the thunk is placed (deep) under `reset`, while its argument `()` is passed outside of `reset`. By wrapping continuations, we can access the information outside of the enclosing `reset` while staying within `reset` lexically. By applying this idea, we can implement a (typed) printf function [1].

**Exercise 8** By plugging `"world"` into the hole `[...]` of the following expression, we can obtain `"hello world!"`.

```
reset (fun () -> "hello " ^ [...] ^ "!")
```

Instead of plugging `"world"` directly, can you fill in the hole with an expression so that the *argument* to the enclosing `reset` is plugged into the hole? In other words, we want the following interaction, by replacing `[...]` with a suitable expression:

```
# reset (fun () -> "hello " ^ [...] ^ "!") "world" ;;
- : string = "hello world!"
#
```

The expression in `[...]` behaves like `%s` directive. Instead of a string, can you achieve the behavior of `%d` such that an integer argument is embedded into the string? (Use `string_of_int` to convert an integer to a string.) Can you pass multiple arguments? (Be aware of the evaluation order of OchaCaml.)

## 2.9 Answer type modification

Now is a good time to consider how expressions with `shift`/`reset` are typed. In the following context,

```
reset (fun () -> [...] ^ " world")
```

the value returned by `reset` appears to be a string, because `^` returns a string. How can we pass an argument to this expression without a type error?

To understand how the printf example is typed, we need to know what an *answer type* is. An answer type is a type of the enclosing `reset`. For example, in the expression `reset (fun () -> 3 + [5 * 2])` (where the current expression is `5 * 2`), the answer type is `int`, while in the expression `reset (fun () -> string_of_int [5 * 2])`, the answer type is `string`.

If an expression does not contain `shift`, it can be placed in a context of any answer type. In the above example, the expression `5 * 2` can be placed both in `reset (fun () -> 3 + [...])` and `reset (fun () -> string_of_int [...])`, because the type of the hole is `int` in both cases. In other words, the answer type of continuations of `5 * 2` is arbitrary. This property is called *answer type polymorphism*.

When an expression contains `shift`, the story becomes different. Because `shift` captures the current continuation and installs a new one (*i.e.*, the body of `shift`), it can change the answer type. Back to the printf example:

```
reset (fun () -> [...] ^ " world")
```

the original answer type is `string`. Thus, the type of this context (to be captured below) is `string -> string`. Suppose that in `[...]`, we execute:

```
shift (fun k -> fun () -> k "hello")
```

Since the type of the captured continuation `k` is `string -> string`, what is returned by `shift` to the enclosing `reset` is a thunk `fun () -> k "hello"` of type `unit -> string`. In other words, the answer type of the enclosing `reset` changed from `string` to `unit -> string` due to the execution of the `shift` expression. This phenomenon is called *answer type modification*. This is how the above `reset` expression can accept `()` as an argument.

Because execution of `shift` expressions can change the answer type of the enclosing context, it is necessary to keep track of the answer type all the time to properly type check expressions with `shift` and `reset`. This is how programs in OchaCaml are type checked. See Section 3.4 for technical details.

## 2.10 How to wrap delimited continuations: state monad

By wrapping continuations, we can access information that resides outside the enclosing `reset`. Using this idea, we can implement a mutable state by passing the current value of the state outside the enclosing `reset`.

For example, let us consider having one integer as a state. Like "world" in the printf example, we pass around an integer as an argument of the context as follows:

```
reset (fun () -> M) 3
```

In this example, the initial value of the state is 3. In the body $M$, we can access this state using the following function:

```
# let get () =
    shift (fun k -> fun state -> k state state) ;;
get : unit => 'a = <fun>
#
```

After capturing the current continuation in k, the function get aborts it and returns a function fun state -> k state state to the enclosing context. Thus, the current value of the state is passed to state. After that, the computation in k is resumed with state. Namely, the value of get () becomes the value of the current state.

In the above definition, k is passed state twice. The first one becomes the value of get () while the second one becomes the value of the new state after get () is executed. Continuations captured by shift include the outermost enclosing reset. Thus, application of k to state constitutes a new context. To supply the value of the state during the execution of k, we pass the value of the state after the execution of get. Since get is supposed not to alter the value of the state, we pass state as a new value for the state. If we want to define a function that alters the state, we pass the new value there. For example, the following function adds one to the current state (and returns a unit):

```
# let tick () =
    shift (fun k -> fun state -> k () (state + 1)) ;;
tick : unit => unit = <fun>
#
```

To start computation, we use the following function:

```
# let run_state thunk =
    reset (fun () -> let result = thunk () in
                     fun state -> result) 0 ;;
run_state : (unit => 'a) => 'b = <fun>
#
```

It executes thunk with 0 as the initial value for the state. When the execution of thunk finishes, the value of the state is ignored and the result is returned. Using these functions, we have:

```
# run_state (fun () ->
    tick ();            (* state = 1 *)
    tick ();            (* state = 2 *)
    let a = get () in
    tick ();            (* state = 3 *)
    get () - a) ;;
- : int = 1
#
```

**Exercise 9** What would be the value of the following program?

```
run_state (fun () ->
  (tick (); get ()) - (tick (); get ())) ;;
```

Check your result by actually executing it. (Be aware of the evaluation order of OchaCaml.)

**Exercise 10** Similarly, write a function put. It updates the state with the argument of put and returns a unit.

The method presented in this section effectively implements a state monad on top of a continuation monad.

## 2.11 How to reorder delimited continuations (advanced)

If we apply a continuation at the tail position, the captured computation is simply resumed. If we apply a continuation at the non-tail position, we can perform additional computation after resumed computation finishes. Put differently, we can switch the execution order of the surrounding context (captured continuation) and the local (additional) computation. For example, in the following expression:

```
# reset (fun () -> 1 + (shift (fun k -> 2 * k 3))) ;;
- : int = 8
#
```

the execution of `2 * [...]` and the execution of the surrounding context `1 + [...]` are swapped, and the latter is executed before the former. This simple idea generalizes to implementing A-normalization.

Consider the following definition of $\lambda$-terms:

```
type term_t = Var of string
            | Lam of string * term_t
            | App of term_t * term_t ;;
```

Given a $\lambda$-term, we want to obtain its A-normal form, an equivalent term but all the applications in the term is given a unique name. For example, the A-normal form of the $S$ combinator $\lambda x.\lambda y.\lambda z.(xz)(yz)$ is

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = xz\ \mathsf{in}\ \mathsf{let}\ t_2 = yz\ \mathsf{in}\ \mathsf{let}\ t_3 = t_1 t_2\ \mathsf{in}\ t_3$$

To write an A-normalizer, we first define an identity function that traverses all the subterms and reconstruct them:

```
(* id_term : term_t -> term_t *)
let rec id_term term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, id_term t)
  | App (t1, t2) -> App (id_term t1, id_term t2) ;;
```

Because we want to assign a unique name to each application, we slightly change the above function to introduce a let expression for each application as follows:

```
(* id_term' : term_t -> term_t *)
let rec id_term' term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, id_term' t)
  | App (t1, t2) ->
      let t = gensym () in   (* generate fresh variable *)
      App (Lam (t, Var (t)), (* let expression *)
           App (id_term' t1, id_term' t2)) ;;
```

Here, we emulated $\mathsf{let}\ t = M\ \mathsf{in}\ N$ by $(\lambda t.N)M$. Using this new identity function, the $S$ combinator is translated to the following term:

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = (\mathsf{let}\ t_2 = xz\ \mathsf{in}\ t_2)(\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$$

This is not quite what we want, however. What we want is a term where nested let expressions are flattened.

Now, suppose we are traversing the syntax tree for the $S$ combinator using `id_term'`, and are currently looking at the first application $xz$. At this point, the current continuation is "to traverse $yz$, to reconstruct an outer application, and to reconstruct three lambdas":

$$\lambda x.\lambda y.\lambda z.\mathsf{let}\ t_1 = [\,\cdot\,](\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$$

To flatten the nested let expressions, we now reorder the construction of the let expression for $xz$ (*i.e.*, $\mathsf{let}\ t_2 = xz\ \mathsf{in}\ [\,\cdot\,]$) and the rest of the reconstruction up to lambda (*i.e.*, $\mathsf{let}\ t_1 = [\,\cdot\,](\mathsf{let}\ t_3 = yz\ \mathsf{in}\ t_3)\ \mathsf{in}\ t_1$) as follows:

```
(* a_normal : term_t => term_t *)
let rec a_normal term = match term with
    Var (x) -> Var (x)
  | Lam (x, t) -> Lam (x, reset (fun () -> a_normal t))
  | App (t1, t2) ->
      shift (fun k ->
        let t = gensym () in    (* generate fresh variable *)
        App (Lam (t,            (* let expression *)
                  k (Var (t))), (* continue with new variable *)
             App (a_normal t1, a_normal t2))) ;;
```

In the `App` case, after capturing the current continuation in `k`, it is resumed with a newly generated variable `t`. Thus, the application in the original term is translated to this new variable. After translation of the whole term is finished, the definition of `t` is added in front of it.

Because we do not want the definition of variables to extrude the scope of lambdas, the context is delimited in the `Lam` case. With this definition, we achieve A-normalization. The presented method of residualizing let expressions is called *let insertion* in the partial evaluation community.

**Exercise 11** Transform the $S$ combinator into A-normal form using the above function `a_normal`. What can we observe?

## 2.12 How to duplicate delimited continuations

So far, we have used captured continuations only once. If we use them more than once, we can execute the rest of the computation in various ways. It can be used to achieve backtracking.

Consider the following function:

```
(* either : 'a -> 'a -> 'a *)
let either a b =
  shift (fun k -> k a; k b) ;;
```

The function `either` receives two arguments, `a` and `b`, captures the current continuation in `k`, and applies it to both `a` and `b` in this order. Because `k` is applied twice, the rest of the computation that calls `either` is executed twice. One can also understand `either` as returning two values. In fact, if we execute `either` in the following context, we can actually see that the computation after `either` is actually executed twice.

```
# reset (fun () ->
    let x = either 0 1 in
    print_int x;
    print_newline ()) ;;
0
1
- : unit = ()
#
```

Since `either` executes its continuation twice, the value of `x` is printed twice, once for the first argument of `either` and the other for the second argument.

**Exercise 12** Define a recursive function `choice` that receives a list of values and returns all the elements of the list to the continuation one after the other.

Using `either`, we can write a simple generate-and-test style of functions easily. For example, suppose we have two boolean variables, $P$ and $Q$, and want to know if the following boolean formula is satisfiable:

$$(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg Q)$$

Since $P$ and $Q$ take either `true` or `false`, we can write the following program:

```
# reset (fun () ->
    let p = either true false in
    let q = either true false in
    if (p || q) && (p || not q) && (not p || not q)
    then (print_string (string_of_bool p);
          print_string ", ";
          print_string (string_of_bool q);
          print_newline ())) ;;
true, false
- : unit = ()
#
```

Notice that the program looks like a straightline program. It binds two variables `p` and `q`, checks whether the boolean formula is satisfied, and prints values of variables if it is. No loop, no backtracking. In the actual execution, the operator `either` executes its rest of the computation twice. Because there are two occurrences of `either`, the test is executed four times in total, each corresponding to the possible assignment of values to `p` and `q`. We can regard `either` as returning one of its arguments non-deterministically.

**Exercise 13** Using `choice`, define a function that searches for three natural numbers between 1 and 5 that satisfy the Pythagorean theorem. In other words, find $1 \leq x, y, z \leq 5$ such that $x^2 + y^2 = z^2$.

# 3 Foundations of `shift` and `reset`

In this section, we show a brief overview of foundations of `shift` and `reset`. The language we consider is a left-to-right[2] call-by-value (CBV) $\lambda$-calculus extended with polymorphic let expressions and two control operators, `shift` and `reset`.

[2] Unlike OchaCaml.

## 3.1 Syntax

$$\begin{array}{rrcl} \text{(value)} & V & ::= & x \mid \lambda x.\, M \\ \text{(term)} & M & ::= & V \mid M\, M \mid \mathsf{let}\ x = M\ \mathsf{in}\ M \mid \mathcal{S}k.\, M \mid \langle M \rangle \\ \text{(pure evaluation context)} & F & ::= & [\,] \mid F\, M \mid V\, F \mid \mathsf{let}\ x = F\ \mathsf{in}\ M \\ \text{(evaluation context)} & E & ::= & [\,] \mid E\, M \mid V\, E \mid \mathsf{let}\ x = E\ \mathsf{in}\ M \mid \langle E \rangle \end{array}$$

In this calculus, `shift (fun k -> M)` is written as $\mathcal{S}k.\, M$ and `reset (fun () -> M)` as $\langle M \rangle$. The evaluation context is divided into two. The pure evaluation context is an evaluation context where the hole is not enclosed by any `reset`.

## 3.2 Reduction rules

$$\begin{array}{rcll} (\lambda x.\, M)\, V & \rightsquigarrow & M[x := V] & \\ \mathsf{let}\ x = V\ \mathsf{in}\ M & \rightsquigarrow & M[x := V] & \\ \langle V \rangle & \rightsquigarrow & V & \\ \langle F[\mathcal{S}\, V] \rangle & \rightsquigarrow & \langle V\, (\lambda x.\, \langle F[x] \rangle) \rangle & x \text{ is fresh} \end{array}$$

The first two rules are from the ordinary $\lambda$-calculus. The third rule says that `reset` around a value can be dropped. The last rule shows how to capture the current continuation represented as a pure evaluation context $F[\,]$. It is reified as a function $\lambda x.\, \langle F[x] \rangle$ and passed to $V$. Notice that the enclosing `reset` remains in the right-hand side of the rule and that the reified function has an enclosing `reset` in it. (If one or both of these `reset` is removed, we obtain other kinds of control operators.)

## 3.3 Evaluation rule

The evaluation rule is defined as follows:

$$E[M] \quad \rightarrow \quad E[M'] \quad \text{if} \quad M \quad \rightsquigarrow \quad M'$$

The definition of an evaluation context specifies that the evaluation order of this calculus is from left to right (*i.e.*, the function part is evaluated before the argument part).

The evaluation rule can be divided into three steps:

(1) If an expression to be evaluated is already a value, it is the result of evaluation. If it is not, it can be decomposed into the form $E[M]$, where $M$ is a redex.

(2) The redex $M$ is reduced to $M'$ according to the reduction rules.

(3) The result $M'$ is plugged into the original evaluation context $E[\,]$ to give the result $E[M']$. This is the result of one-step reduction.

## 3.4 Typing Rules

Types and type schemes are defined as follows, where $t$ represents a type variable.

$$\begin{array}{rrcl} \text{type} & \tau & ::= & t \mid \tau/\tau \to \tau/\tau \\ \text{type scheme} & \sigma & ::= & \tau \mid \forall t.\sigma \end{array}$$

Here, the type $\tau_1/\alpha \to \tau_2/\beta$ represents a type of a function from $\tau_1$ to $\tau_2$, but during the execution of this function, the answer type changes from $\alpha$ to $\beta$ [3]. If a function does not have any control effects (roughly speaking, the function does not use `shift`), the two answer types $\alpha$ and $\beta$ become both the same type variable that does not appear anywhere else. Such a function is called *pure*. It is also called *answer-type polymorphic*. See the next section for answer types.

In OchaCaml, types of pure functions are written as $\tau_1$ `->` $\tau_2$, omitting answer types. On the other hand, types of impure functions (that contain `shift` expressions) are written as $\tau_1$ `=>` $\tau_2$, indicating that the answer types (also omitted) are not polymorphic. When one wants to see all the answer types, OchaCaml supports a directive `#answer "all";;`. OchaCaml will then show types of impure functions as $\tau_1$ `/` $\alpha$ `->` $\tau_2$ `/` $\beta$.

Typing judgements have one of the following forms:

$$\Gamma \vdash_p M : \tau$$
$$\Gamma, \alpha \vdash M : \tau, \beta$$

The former reads: under a type environment $\Gamma$, an expression $M$ is pure and has type $\tau$. The latter reads: under a type environment $\Gamma$, an expression $M$ has type $\tau$ and execution of $M$ changes the answer type from $\alpha$ to $\beta$. The typing rules are given as follows:

$$\frac{(x:\sigma)\in\Gamma \quad \sigma \succ \tau}{\Gamma \vdash_p x : \tau}\ (\text{var}) \quad \frac{\Gamma, x:\tau_1, \alpha \vdash M : \tau_2, \beta}{\Gamma \vdash_p \lambda x.\, M : \tau_1/\alpha \to \tau_2/\beta}\ (\text{fun})$$

$$\frac{\Gamma, \gamma \vdash M_1 : \tau_1/\alpha \to \tau_2/\beta, \delta \quad \Gamma, \beta \vdash M_2 : \tau_1, \gamma}{\Gamma, \alpha \vdash M_1\, M_2 : \tau_2, \delta}\ (\text{app}) \quad \frac{\Gamma \vdash_p M : \tau}{\Gamma, \alpha \vdash M : \tau, \alpha}\ (\text{exp})$$

$$\frac{\Gamma \vdash_p M_1 : \tau_1 \quad \Gamma, x : \mathsf{Gen}(\tau_1, \Gamma), \alpha \vdash M_2 : \tau_2, \beta}{\Gamma, \alpha \vdash \mathsf{let}\ x = M_1\ \mathsf{in}\ M_2 : \tau_2, \beta}\ (\text{let})$$

$$\frac{\Gamma, k : \forall t.(\tau/t \to \alpha/t), \gamma \vdash M : \gamma, \beta}{\Gamma, \alpha \vdash \mathcal{S}k.\, M : \tau, \beta}\ (\text{shift}) \quad \frac{\Gamma, \gamma \vdash M : \gamma, \tau}{\Gamma \vdash_p \langle M \rangle : \tau}\ (\text{reset})$$

As usual, $\sigma \succ \tau$ represents that a type $\tau$ is an instance of a type scheme $\sigma$, and $\mathsf{Gen}(\tau, \Gamma)$ represents a type scheme obtained by generalizing the type variables in $\tau$ that does not occur free in $\Gamma$.

## 3.5 Answer Types

An answer type is a type of the current context. For example, in the following expression:

```
reset (fun () -> 3 + [5 * 2] - 1)
```

the type of the current expression `[5 * 2]` is `int` and the type of a value returned by its surrounding context `3 + [5 * 2] - 1` is `int` (both before and after the execution of `5 * 2`). Thus, the answer type of `[5 * 2]` is `int`. We represent this typing as the following judgement:

$$\Gamma, \texttt{int} \vdash \texttt{5 * 2} : \texttt{int}, \texttt{int}$$

The first `int` is the answer type before the execution of `5 * 2`; the second `int` is the type of `5 * 2`; and the third `int` is the answer type after the execution of `5 * 2`.

The type of an expression and its answer type are not always the same. For example, in the expression:

```
reset (fun -> if [2 = 3] then 1 + 2 else 3 - 1)
```

the current expression `2 = 3` has type `bool`, but the answer type is `int`. Thus, the typing judgement becomes as follows:

$$\Gamma, \texttt{int} \vdash \texttt{2 = 3} : \texttt{bool}, \texttt{int}$$

The two answer types are always the same if the current expression is pure. Because answer types do not affect the type of pure expressions, we can forget about answer types when we are dealing with only pure expressions.

Next, let us consider examples where the answer type changes.

```
reset (fun () ->
  [shift (fun k -> fun () -> k "hello")] ^ " world")
```

It is not immediately clear what the type of the current expression `[...]` is, but it is `string`, because the result of the current expression is passed to `^` which requires two string arguments. What is the answer type? Before the execution of `[...]`, the answer type of the current context is `string` because it is the result of `^` which returns a string value. After the execution of `[...]`, however, the current continuation is captured in `k` and what is actually returned by the enclosing `reset` is a function `fun () -> k "hello"`. Since this function receives a unit `()` and returns a string `"hello world"`, it has roughly type `unit -> string`. To be more precise, because the type of this function must also mention answer types and because the function turns out to be pure, it has type `unit / 'a -> string / 'a`. Thus, the typing judgement becomes as follows:

$$\Gamma, \texttt{string} \vdash \texttt{shift (fun k -> ...)} : \texttt{string},\ \texttt{unit / 'a -> string / 'a}$$

Thus, the use of `shift` can change the answer types in various ways. This phenomenon is called *answer type modification*.

We can understand a type of functions in a similar way. For example, consider `get` introduced in Section 2.8. Since this function receives `()` and returns an integer (the current value of the state), it has type `unit -> int`, if we ignore answer types. To see what the answer types are, we consider the context in which `get` is used. For example, in the following expression:

```
reset (fun () ->
  let result = [get ()] + 2 * get () in
  fun state -> result)
```

since the context returns a function `fun state -> result` as a result, the answer type is (roughly) `int -> int`. Thus, the type of `get` becomes something like `unit / (int -> int) -> int / (int -> int)`. To be more precise, because the type of state is not constrained to `int` from the definition of `get` and the type of context can be more general, the exact type of `get` is:

```
unit / ('a / 'b -> 'c / 'd) -> 'a / ('a / 'b -> 'c / 'd)
```

The introduction of `shift`/`reset` forces us to think about answer types. It amounts to consider what the type of context is and how it changes during execution.

# References


[1] Asai, K. "On Typing Delimited Continuations: Three New Solutions to the Printf Problem," *Higher-Order and Symbolic Computation*, Vol. 22, No. 3, pp. 275–291, Kluwer Academic Publishers (September 2009).

[2] Asai, K., and Y. Kameyama "Polymorphic Delimited Continuations," *Proceedings of the Fifth Asian Symposium on Programming Languages and Systems (APLAS'07), LNCS 4807*, pp. 239–254 (November 2007).

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

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

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

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

[7] Gasbichler, M., and M. Sperber "Final Shift for Call/cc: Direct Implementation of Shift and Reset," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pp. 271–282 (October 2002).

[8] Gunter, C. A., D. Rémy, and J. G. Riecke "A Generalization of Exceptions and Control in ML-Like Languages," *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pp. 12–23 (June 1995).

[9] Kameyama, Y., and M. Hasegawa "A Sound and Complete Axiomatization of Delimited Continuations," *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming (ICFP'03)*, pp. 177–188 (August 2003).

[10] Kiselyov, O. "Delimited Control in OCaml, Abstractly and Concretely: System Description," In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (LNCS 6009)*, pp. 304–320 (April 2010).

[11] Masuko, M., and K. Asai "Direct Implementation of Shift and Reset in the MinCaml Compiler," *Proceedings of the 2009 ACM SIGPLAN Workshop on ML*, pp. 49–60 (September 2009).

[12] Masuko, M., and K. Asai "Caml Light + shift/reset = Caml Shift," *Theory and Practice of Delimited Continuations (TPDC 2011)*, pp. 33–46 (May 2011).

[13] Plotkin, G. D. "Call-by-name, call-by-value, and the $\lambda$-calculus," *Theoretical Computer Science*, Vol. 1, No. 2, pp. 125–159 (December 1975).