CW 2011 Tutorial: Introduction to Programming with Shift and Reset

Kenichi Asai Oleg Kiselyov

September 23, 2011

Thanks to: Kazu Yamamoto (IIJ)

Overview

- Basics
  - What are continuations?
  - What are delimited continuations?
  - How to discard/extract continuations.
- How to use delimited continuations in Haskell
- Challenge 1: co-routine
- Challenge 2: printf
- Challenge 3: search

What are continuations?

Continuation

The rest of the computation.

- The current computation: $\cdots$ inside [ ]
- The rest of the computation: $\cdots$ outside [ ]

For example: $3 + [5 * 2] - 1$.

- The current computation: $5 * 2$
- The current continuation: $3 + [\,\cdot\,] - 1$.

"Given a value for $[\,\cdot\,]$, add 3 to it and sbtract 1 from the sum." i.e., `fun x -> 3 + x - 1`

What are continuations?

Continuation

The rest of the computation.

Continuations are the computation that is discarded when the current computation is aborted.

For example: $3 + [5 * 2] - 1$.

- Replace $[\cdot]$ with `raise Abort`:

$$3 + [\texttt{raise Abort}] - 1$$

- The discarded computation $3 + [\cdot] - 1$ is the current continuation.

What are continuations?

As computation proceeds, continuation changes.

$3 + [5 * 2] - 1$:

- The current computation: $5 * 2$
- The current continuation: $3 + [\cdot] - 1$.

$[3 + 10] - 1$:

- The current computation: $3 + 10$
- The current continuation: $[\cdot] - 1$.

$[13 - 1]$:

- The current computation: $13 - 1$
- The current continuation: $[\cdot]$.

Exercise

Identify the current expression, continuation, and their types.

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))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` :
   `5 * ([·] + 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))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int ->`

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))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
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))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` :
   `(if [·] ...) ^ " world"` :
3. `fst (let x = 1 + 2 in (x, x))`
4. `string_length ("x" ^ string_of_int (3 + 1))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool ->`
3. `fst (let x = 1 + 2 in (x, x))`
4. `string_length ("x" ^ string_of_int (3 + 1))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`

2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`

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

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

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`

2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`

3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` :
   `fst (let x = [·] in (x, x))` :

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

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`

2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`

3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` : `int`
   `fst (let x = [·] in (x, x))` : `int ->`

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

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`
3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` : `int`
   `fst (let x = [·] in (x, x))` : `int -> int`
4. `string_length ("x" ^ string_of_int (3 + 1))`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`

2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`

3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` : `int`
   `fst (let x = [·] in (x, x))` : `int -> int`

4. `string_length ("x" ^ string_of_int [3 + 1])`
   `[3 + 1]` :
   `string_length ("x" ^ string_of_int [·])` :

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`
3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` : `int`
   `fst (let x = [·] in (x, x))` : `int -> int`
4. `string_length ("x" ^ string_of_int [3 + 1])`
   `[3 + 1]` : `int`
   `string_length ("x" ^ string_of_int [·])` : `int ->`

Exercise

Identify the current expression, continuation, and their types.

1. `5 * ([2 * 3] + 3 * 4)`
   `[2 * 3]` : `int`
   `5 * ([·] + 3 * 4)` : `int -> int`
2. `(if [2 = 3] then "hello" else "hi") ^" world"`
   `[2 = 3]` : `bool`
   `(if [·] ...) ^ " world"` : `bool -> string`
3. `fst (let x = [1 + 2] in (x, x))`
   `[1 + 2]` : `int`
   `fst (let x = [·] in (x, x))` : `int -> int`
4. `string_length ("x" ^ string_of_int [3 + 1])`
   `[3 + 1]` : `int`
   `string_length ("x" ^ string_of_int [·])` : `int -> int`

What are delimited continuations?

Delimited Continuation

The rest of the computation up to the delimiter.

Syntax

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

For example:

`reset (fun () ->` $\boxed{\texttt{3 + [5 * 2]}}$ `) - 1`

- The current computation: $5 * 2$
- The current delimited continuation: $3 + [\,\cdot\,]$.

What are delimited continuations?

The delimiter `reset` is like an exception handler.

For example:

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

- Replace `reset` with `try ... with`:

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

- The discarded computation $3 + [\,\cdot\,]$ is the current delimited continuation.

Exercise

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]))
```

Exercise

Identify the delimited continuation, and its type.

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

Exercise

Identify the delimited continuation, and its type.

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

   `[·] + 3 * 4 : int -> int`

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]))
   ```
   ```

Exercise

Identify the delimited continuation, and its type.

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

   `[·] + 3 * 4` : `int -> int`

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

   `if [·] then "hello" else "hi"` :

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

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

Exercise

Identify the delimited continuation, and its type.

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

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

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

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

Exercise

Identify the delimited continuation, and its type.

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

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

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

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

Exercise

Identify the delimited continuation, and its type.

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

   `[·] + 3 * 4` : `int -> int`

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

   `if [·] then "hello" else "hi"` : `bool -> string`

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

   `let x = [·] in (x, x)` : `int -> int * int`

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

Exercise

Identify the delimited continuation, and its type.

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

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

3. ```
   fst (reset (fun () ->
                 let x = [1 + 2] in (x, x)))
   let x = [·] in (x, x) : int -> int * int
   ```

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

Exercise

Identify the delimited continuation, and its type.

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

   `[·] + 3 * 4` : `int -> int`

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

   `if [·] then "hello" else "hi"` : `bool -> string`

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

   `let x = [·] in (x, x)` : `int -> int * int`

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

   `"x" ^ string_of_int [·]` : `int -> string`
   ```

shift

Syntax

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

- It clears the current continuation,
- binds the cleared continuation to `k`, and
- executes M.

For example:

```
reset (fun () -> 3 + [shift (fun k -> M)]) - 1
```

We will see a number of examples today.

`shift`

Syntax

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

- It clears the current continuation,
- binds the cleared continuation to `k`, and
- executes M.

For example:

`reset (fun () ->      [shift (fun k ->` M`)]) - 1`

We will see a number of examples today.

shift

Syntax

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

- It clears the current continuation,
- binds the cleared continuation to `k`, and
- executes M.

For example:

```
reset (fun () ->     [shift (fun k -> M)]) - 1
                  k = reset (fun () -> 3 + [·])
```

We will see a number of examples today.

shift

Syntax

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

- It clears the current continuation,
- binds the cleared continuation to `k`, and
- executes M.

For example:

```
reset (fun () ->                         M   ) - 1
                     k = reset (fun () -> 3 + [·])
```

We will see a number of examples today.

How to discard continuations

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

- Captured continuation is discarded.
- The same as raising an exception.

For example:

```
reset (fun () -> 3 + shift (fun _ -> 2)) - 1
reset (fun () ->                    2 ) - 1
                k = reset (fun () -> 3 + [·])
2 - 1
1
```

Exercise

Replace $[\cdot]$ with `shift (fun _ ->` M`)` for some M.
Try out in your computer to see what happens.

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

Replace [·] with `shift (fun _ -> ` M`)` for some M.
Try out in your computer to see what happens.

1. ```
   5 * reset (fun () -> [·] + 3 * 4)
   shift (fun _ -> ?)
   ```
2. ```
   reset (fun () ->
             if [·] then "hello" else "hi")
   ^ " world"
   shift (fun _ -> ?)
   ```
3. ```
   fst (reset (fun () ->
                 let x = [·] in (x, x)))
   shift (fun _ -> ?)
   ```
4. ```
   string_length (reset (fun () ->
                     "x" ^ string_of_int [·]))
   shift (fun _ -> ?)
   ```

Exercise

Replace $[\cdot]$ with `shift (fun _ -> ` M`)` for some M.
Try out in your computer to see what happens.

1. ```
   5 * reset (fun () -> [·] + 3 * 4)
   shift (fun _ -> 3)
   ```
   $\rightsquigarrow$ 15

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

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

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

Exercise

Replace $[\,\cdot\,]$ with `shift (fun _ -> ` M`)` for some M.
Try out in your computer to see what happens.

1. ```
   5 * reset (fun () -> [·] + 3 * 4)
   shift (fun _ -> 3)
   ```
   $\rightsquigarrow$ 15

2. ```
   reset (fun () ->
             if [·] then "hello" else "hi")
   ^ " world"
   shift (fun _ -> "chao")
   ```
   $\rightsquigarrow$ "chao world"

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

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

Exercise

Replace [·] with `shift (fun _ -> ` M`)` for some M.
Try out in your computer to see what happens.

1. ```
   5 * reset (fun () -> [·] + 3 * 4)
   shift (fun _ -> 3)
   ```
   $\rightsquigarrow$ 15

2. ```
   reset (fun () ->
             if [·] then "hello" else "hi")
   ^ " world"
   shift (fun _ -> "chao")
   ```
   $\rightsquigarrow$ "chao world"

3. ```
   fst (reset (fun () ->
                  let x = [·] in (x, x)))
   shift (fun _ -> (3, 4))
   ```
   $\rightsquigarrow$ 3

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

Exercise

Replace [·] with `shift (fun _ -> ` M`)` for some M. Try out in your computer to see what happens.

1. ```
   5 * reset (fun () -> [·] + 3 * 4)
   shift (fun _ -> 3)
   ```
   $\rightsquigarrow$ 15

2. ```
   reset (fun () ->
            if [·] then "hello" else "hi")
   ^ " world"
   shift (fun _ -> "chao")
   ```
   $\rightsquigarrow$ "chao world"

3. ```
   fst (reset (fun () ->
                 let x = [·] in (x, x)))
   shift (fun _ -> (3, 4))
   ```
   $\rightsquigarrow$ 3

4. ```
   string_length (reset (fun () ->
                            "x" ^ string_of_int [·]))
   shift (fun _ -> "great day!")
   ```
   $\rightsquigarrow$ 10

Advanced Exercise

The following function multiplies elements of a list:

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

Add the following clause:

```
  | 0 :: rest -> ???
```

so that calls like the following will return 0 without performing any multiplication.

```
reset (fun () -> times [1; 2; 0; 4]) ;;
```

Solution

```
# let rec times lst = match lst with
      [] -> 1
    | 0 :: rest -> shift (fun _ -> 0)
    | first :: rest -> first * times rest ;;
times : int list => int = <fun>
# reset (fun () -> times [1; 2; 0; 4]) ;;
- : int = 0
# reset (fun () -> times [1; 2; 3; 4]) ;;
- : int = 24
#
```

How to extract continuations

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

- Captured continuation is returned immediately.
- We can play with the captured contiuation!

For example: `reset (fun () -> 3 + [...] - 1)`

```
# let f   =
    reset (fun () -> 3 + shift (fun k -> k) - 1)   ;;
f : int => int = <fun>
# f 10 ;;
- : int = 12
#
```

How to extract continuations

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

- Captured continuation is returned immediately.
- We can play with the captured contiuation!

For example: `reset (fun () -> 3 + [...] - 1)`

```
# let f x =
    reset (fun () -> 3 + shift (fun k -> k) - 1) x ;;
f : int -> int = <fun>
# f 10 ;;
- : int = 12
#
```

Exercise

Extract the following continuation.
What does it do? Try out in your computer.

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

Extract the following continuation.
What does it do? Try out in your computer.

1. `reset (fun () -> 5 * ([·] + 3 * 4))`
   `f 6` $\rightsquigarrow$ 90

2. `reset (fun () ->`
   `  (if [·] then "hello" else "hi") ^ " world")`
   `f true` $\rightsquigarrow$ `"hello world"`
   `f false` $\rightsquigarrow$ `"hi world"`

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

4. `reset (fun () ->`
   `  string_length ("x" ^ string_of_int [·]))`
   `f 0` $\rightsquigarrow$ 2, `f 10` $\rightsquigarrow$ 3, `f 100` $\rightsquigarrow$ 4

Advanced Exercise

Here is an identity function on a list:

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

By modifying the line `(* A *)`, extract the continuation at `(* A *)` when called as follows:

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

What does the extracted continuation do?

Solution

```
# let rec id lst = match lst with
    [] -> shift (fun k -> k)
  | first :: rest -> first :: id rest ;;
id : 'a list => 'a list = <fun>
# let append123 =
    reset (fun () -> id [1; 2; 3]) ;;
append123 : int list => int list = <fun>
# append123 [4; 5; 6] ;;
- : int list = [1; 2; 3; 4; 5; 6]
#
```

Haskell time

Challenge 1

co-routine

Tree walking

Consider a binary tree of integers:

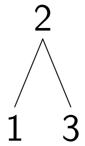
```
type tree_t = Empty
            | Node of tree_t * int * tree_t
```

We can write a function that traverses over a tree:

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

Tree walking

tree1:



For example, we have:

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

Tree walking

Can we write a variant of `walk` that returns integers one by one?

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

`yield` returns `n` and "the way to get more integers"

How to preserve continuations

```
type    result_t =
    Done                       (* no more Nodes *)
  | Next of int *
            (unit      ->    result_t      )
```

We can then define `yield` as follows:

```
let yield n = shift (fun k -> Next (n, k))
```

- Captured continuation is preserved in `Next` and returned to the enclosing `reset`.

How to preserve continuations

```
type 'a result_t =
    Done                          (* no more Nodes *)
  | Next of int *
            (unit / 'a -> 'a result_t / 'a)
```

We can then define `yield` as follows:

```
let yield n = shift (fun k -> Next (n, k))
```

- Captured continuation is preserved in `Next` and returned to the enclosing `reset`.

How to preserve continuations

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

(* 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) ;;
```

Exercise

1. Try `print_nodes` in your computer.
2. Similarly, can you write a function that returns the sum of all the integers in a tree?

```
(* add_tree : tree_t -> int *)
let add_tree tree =
  ...
```

Exercise

1. Try `print_nodes` in your computer.
2. Similarly, can you write a function that returns the sum of 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) ;;
```

Challenge 1: co-routine

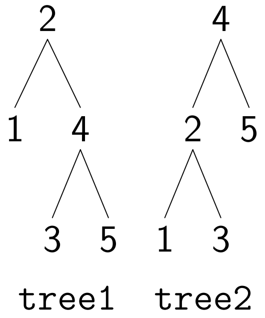
Write a function `same_fringe`.

```
same_fringe tree1 tree2 ;;
```

evaluates to `true` if the 'fringe' of the two trees are the same, and `false` otherwise.

Note:
When mismatch is detected, we want to return `false` without further traversing the trees.
(We do not want to flatten trees.)

For example,



Solution

```
(* same_fringe : tree_t -> tree_t -> bool *)
let same_fringe t1 t2 =
  let rec loop r1 r2 = match (r1, r2) with
      (Done, Done) -> true
    | (Next (n1, k1), Next (n2, k2)) ->
        n1 = n2 && loop (k1 ()) (k2 ())
    | (_, _) -> false in
  loop (start t1) (start t2) ;;
```

Challenge 2

printf

Well, we are not going to use libc library...

How to wrap continuations

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

Abort The current computation is aborted with a thunk.

Access It receives () from outside the context.

Resume The aborted computation is resumed with "hello".

For example,

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

How to wrap continuations

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

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

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

reset (fun () -> "hello" ^ " world")
```

- Code is effectively inserted around `reset`.

Challenge 2: printf

Fill in the hole so that the following program:

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

would return `"hello world!"`. Can you fill in the following hole:

```
reset (fun () -> "It's " ^ [...] ^ " o'clock!") 8 ;;
```

so that it returns `"It's 8 o'clock!"`?

Hint: You can use `string_of_int`.

Solution

```
reset (fun () ->
  "hello " ^ shift (fun k -> fun x -> k x) ^ "!")
"world" ;;
```

or even `shift (fun k -> k)` would do.

```
reset (fun () ->
  "It's " ^
  shift (fun k -> fun x -> k (string_of_int x)) ^
  " o'clock!")
8 ;;
```

The same idea can be used to implement a state monad.

Answer type modification

```
reset (fun () ->
  "hello " ^ shift (fun k -> fun x -> k x) ^ "!")
"world" ;;
```

- The body of reset appears to be a string:
  `reset (fun () -> "hello " ^ [ ] ^ "!")`
- How can we pass an argument `"world"` to it?
- Because `shift` replaces the context with:
  `fun x -> k x`

Answer type changes from: `string`
to: `string -> string`.

Challenge 3

search

How to duplicate continuations

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

- Captured continuation is used twice.
- The caller of `either` receives both `a` and `b`.

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

Generate and test

Is the following logical formula satisfiable?

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

```
# 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 = ()
#
```

Challenge 3: search

1. 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.
2. Using `choice`, define a function that searches for three natural numbers between 1 and 5 that satisfy the Pythagorean theorem:

$$\text{Find: } 1 \le x, y, z \le 5, \text{ s.t. } x^2 + y^2 = z^2.$$

```
(* choice : 'a list => 'a *)
let choice lst =
  let rec loop k lst = match lst with
      [] -> ()
    | first :: rest -> k first; loop k rest in
  shift (fun k -> loop k lst) ;;

(* search : unit => unit *)
let search () =
  let x = choice [1; 2; 3; 4; 5] in
  let y = choice [1; 2; 3; 4; 5] in
  let z = choice [1; 2; 3; 4; 5] in
  if x * x + y * y = z * z
  then (print_int x; print_string " ";
        print_int y; print_string " ";
        print_int z; print_newline ()) ;;
```

How to use `shift`/`reset` in other languages

Scheme	Racket and Gauche support `shift`/`reset`.
Haskell	Delimcc Library.
Scala	Implementation via selective CPS translation.
OCaml	Delimcc Library or emulation via `call/cc`.

Happy programming with
`shift` and `reset`!