Delimited Continuations for Everyone

Kenichi Asai

Ochanomizu University, Japan

September 28, 2017
Overview

Basics:

- What are continuations?
- What are delimited continuations?

Examples:

- How to discard continuations: \texttt{times}
- How to extract continuations: \texttt{append}
- How to reorder continuations: \texttt{take}, \texttt{A-normalize}
- How to wrap continuations: \texttt{printf}, state monad

Speculation:

- Toward delimited continuations in theorem proving
Early papers on control operators

- Control/prompt

- Shift/reset
What are continuations?

**Continuation**

The rest of the computation.

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

For example: \(3 + [5 \times 2] - 1\).

- The current computation: \(5 \times 2\)
- The current continuation: \(3 + [\cdot] - 1\).

“Given a value for \([\cdot]\), add 3 to it and subtract 1 from the sum.” i.e., \(\text{fun } x \rightarrow 3 + x - 1\)
What are continuations?

As computation proceeds, continuation changes.

3 + [5 * 2] − 1:
- The current computation: 5 * 2
- The current continuation: 3 + [ · ] − 1.

[3 + 10] − 1:
- The current computation: 3 + 10
- The current continuation: [ · ] − 1.

[13 − 1]:
- The current computation: 13 − 1
- The current continuation: [ · ].
Examples

Identify the current expressions, continuations, and their types.

1. $5 \times (2 \times 3 + 3 \times 4)$

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

Identify the current expressions, continuations, and their types.

1. \[ 5 \times ([2 \times 3] + 3 \times 4) \]
   
   \[ [2 \times 3] : 5 \times ([\cdot] + 3 \times 4) : \]

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

Identify the current expressions, continuations, and their types.

1. \(5 \times ([2 \times 3] + 3 \times 4)\)
   
   \([2 \times 3] : \text{int}\)
   
   \(5 \times ([\cdot] + 3 \times 4) : \text{int} \rightarrow\)

2. \((\text{if } 2 = 3 \text{ then } "hello" \text{ else } "hi")\)
   
   \(^" \text{world}"\)
Identify the current expressions, continuations, and their types.

1. $5 \times ([2 \times 3] + 3 \times 4)$
   
   $[2 \times 3]: \text{int}$
   
   $5 \times ([\cdot] + 3 \times 4): \text{int} \rightarrow \text{int}$

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

Identify the current expressions, continuations, and their types.

1. $5 \times ([2 \times 3] + 3 \times 4)$
   - $[2 \times 3] : \text{int}$
   - $5 \times ([\cdot] + 3 \times 4) : \text{int} \rightarrow \text{int}$

2. (if $[2 = 3]$ then "hello" else "hi")
   - ^ " world"
   - $[2 = 3] :$
   - (if $[\cdot] \ldots$) ^ " world" :
Examples

Identify the current expressions, continuations, and their types.

1. \(5 \ast ([2 \ast 3] + 3 \ast 4)\)
   - \([2 \ast 3]\) : int
   - \(5 \ast ([\cdot] + 3 \ast 4)\) : int \(\rightarrow\) int

2. (if \([2 = 3]\) then "hello" else "hi") ^ "world"
   - \([2 = 3]\) : bool
   - (if \([\cdot] \ldots\) ^ "world") : bool \(\rightarrow\)
Examples

Identify the current expressions, continuations, and their types.

1. \( 5 \times ([2 \times 3] + 3 \times 4) \)
   - \([2 \times 3]\): int
   - \(5 \times ([\cdot] + 3 \times 4):\) int \(\rightarrow\) int

2. (if \([2 = 3]\) then "hello" else "hi") ^ "world"
   - \([2 = 3]\): bool
   - (if \([\cdot]\) ...) ^ "world": bool \(\rightarrow\) string
What are delimited continuations?

**Delimited Continuation**

The rest of the computation up to the delimiter.

**Syntax**

\[
\text{reset} \ (\text{fun} \ () \rightarrow \ M)
\]

For example:

\[
\text{reset} \ (\text{fun} \ () \rightarrow \ 3 + [5 \ast 2]) - 1
\]

- The current computation: \(5 \ast 2\)
- The current delimited continuation: \(3 + [\cdot]\).
Examples

Identify the delimited continuations, and their types.

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

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

Identify the delimited continuations, and their types.

1. \[5 \cdot \text{reset (fun () \to [2 \cdot 3] + 3 \cdot 4)}\]
   \[\cdot] + 3 \cdot 4:

2. \[\text{reset (fun () \to if [2 = 3] then "hello" else "hi")} \wedge " \text{world}"\]
Examples

Identify the delimited continuations, and their types.

1. \(5 \cdot \text{reset} \ (\text{fun} \ () \rightarrow \ [2 \cdot 3] + 3 \cdot 4)\)
   \(\cdot [\cdot] + 3 \cdot 4 : \text{int} \rightarrow \text{int}\)

2. \(\text{reset} \ (\text{fun} \ () \rightarrow \)
   \(\quad \text{if} \ [2 = 3] \ \text{then} \ "\text{hello}" \text{ else} \ "\text{hi}"\)
   \(\wedge " \text{world}"\)
Examples

Identify the delimited continuations, and their types.

1. \[ 5 \times \text{reset } (\text{fun } () \rightarrow [2 \times 3] + 3 \times 4) \]
   \[ \cdot + 3 \times 4 : \text{int } \rightarrow \text{int} \]

2. \text{reset } (\text{fun } () \rightarrow
   \text{if } [2 = 3] \text{ then } "\text{hello}" \text{ else } "\text{hi}")
   \text{^ } "\text{ world}"
   \text{if } [\cdot] \text{ then } "\text{hello}" \text{ else } "\text{hi}":


Examples

Identify the delimited continuations, and their types.

1. \(5 \times \text{reset} (\text{fun} () -> [2 \times 3] + 3 \times 4)\)
   
   \([\cdot] + 3 \times 4 : \text{int} \rightarrow \text{int}\)

2. \(\text{reset} (\text{fun} () ->\)

   \(\text{if} [2 = 3] \text{ then } "hello" \text{ else } "hi" )\)

   \(^"\text{ world}\)"

   \(\text{if} [\cdot] \text{ then } "hello" \text{ else } "hi" : \)

   \(\text{bool} \rightarrow \text{string}\)
shift

Syntax

\[ \text{shift} \left( \text{fun } k \rightarrow M \right) \]

- It **clears** the current continuation,
- **binds** the cleared continuation to \( k \), and
- **executes** the body \( M \) in the empty context.

For example:

\[
\text{reset} \left( \text{fun } () \rightarrow 3 + \left[ \text{shift} \left( \text{fun } k \rightarrow M \right) \right] \right) - 1
\]

We will see a number of examples today.
**shift**

**Syntax**

\[
\text{shift} \ (\text{fun } k \rightarrow M) \\
\]

- It **clears** the current continuation,
- **binds** the cleared continuation to \( k \), and
- **executes** the body \( M \) in the empty context.

For example:

\[
\text{reset} \ (\text{fun } () \rightarrow \ [\text{shift} \ (\text{fun } k \rightarrow M)]) - 1 \\
\]

We will see a number of examples today.
**Syntax**

`shift (fun k -> M)`

- It **clears** the current continuation,
- **binds** the cleared continuation to `k`, and
- **executes** the body `M` in the empty context.

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** the body `M` in the empty context.

For example:

```plaintext
reset (fun () -> M) - 1

k = reset (fun () -> 3 + [\cdot])
```

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:

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

```
2 - 1
1
```
Examples

Replace \([\_\_\_]\) with \(\text{shift(} \text{fun } _\_ \text{ -> } M \text{)}\) for some \(M\).

1. \(5 * \text{reset(} \text{fun } () \text{ -> } [\_] + 3 * 4\)\)

2. \(\text{reset(} \text{fun } () \text{ -> if } [\_] \text{ then "hello" else "hi"})\)
   \(\text{^ " world"}\)

We need the type of the context to fill in the body.
Examples

Replace \([\cdot]\) with \(\text{shift}\ (\text{fun } _ \rightarrow M)\) for some \(M\).

1. \(5 \ast \text{reset}\ (\text{fun } () \rightarrow [\cdot] + 3 \ast 4)\)
   \(\text{shift}\ (\text{fun } _ \rightarrow ?)\)

2. \(\text{reset}\ (\text{fun } () \rightarrow\)
   \(\text{if } [\cdot] \text{ then } "\text{hello}" \text{ else } "\text{hi}"\)
   
   \(\wedge " \text{world}"
   \(\text{shift}\ (\text{fun } _ \rightarrow ?)\)

We need the type of the context to fill in the body.
Examples

Replace \([\_\_]\) with \(\text{shift} \ (\text{fun } \_\_ \rightarrow M)\) for some \(M\).

1. \(5 \times \text{reset} \ (\text{fun } () \rightarrow [\_\_] + 3 \times 4)\)
   \(\text{shift} \ (\text{fun } \_\_ \rightarrow 3)\) \(\rightsquigarrow 15\)

2. \(\text{reset} \ (\text{fun } () \rightarrow\)
   \(\text{if } [\_] \text{ then } "\text{hello}" \text{ else } "\text{hi}"\)
   \(\wedge "\text{world}"\)
   \(\text{shift} \ (\text{fun } \_ \rightarrow ?)\)

We need the type of the context to fill in the body.
Examples

Replace \([\cdot]\) with \(\text{shift } (\text{fun } _ \rightarrow M)\) for some \(M\).

1. \(5 \times \text{reset} (\text{fun } () \rightarrow [\cdot] + 3 \times 4)\)
   \(\text{shift } (\text{fun } _ \rightarrow 3)\) \(\leadsto 15\)

2. \(\text{reset } (\text{fun } () \rightarrow\)
   \(\text{if } [\cdot] \text{ then } "\text{hello}" \text{ else } "\text{hi}"\)
   \(^"\text{ world}\)
   \(\text{shift } (\text{fun } _ \rightarrow "\text{chao}")\) \(\leadsto "\text{chao world}"\)

We need the type of the context to fill in the body.
times

The following function multiplies elements of a list:

(* times : int list -> int *)
let rec times lst = match lst with
  [] -> 1
| 0 :: rest -> ???
| first :: rest -> first * times rest

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

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

(* times : int list -> int *)

let rec times lst = match lst with
  | [] -> 1
  | 0 :: rest -> 0
  | first :: rest -> first * times rest

It avoids traversing the rest of the list once 0 is found, but it still multiplies elements up to 0.

  times [1; 2; 0; 4]  
  -> 1 * times [2; 0; 4]  
  -> 1 * 2 * times [0; 4]  
  -> 1 * 2 * 0
Solution: discard the continuation

(* times : int list => int *)
let rec times lst = match lst with
  [] -> 1
| 0 :: rest -> shift (fun _ -> 0)
| first :: rest -> first * times rest
reset (fun () -> times [1; 2; 0; 4])
-> reset (fun () -> 1 * times [2; 0; 4])
-> reset (fun () -> 1 * 2 * times [0; 4])
-> reset (fun () -> 0)
-> 0
How to extract continuations

\[ \text{shift} \ (\text{fun } k \rightarrow k) \]

- Captured continuation is returned immediately.

For example:

\[
\begin{align*}
\text{reset} \ (\text{fun } () \rightarrow 3 + [...]) - 1 \\
\text{let } f &= \text{reset} \ (\text{fun } () \rightarrow \\
& \quad 3 + \text{shift} \ (\text{fun } k \rightarrow k) - 1)
\end{align*}
\]
How to extract continuations

shift (fun k -> k)

- Captured continuation is returned immediately.

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

let f = reset (fun () ->
  3 + shift (fun k -> k) - 1)

-> let f = reset (fun () ->
  shift (fun k -> k) )
How to extract continuations

shift (fun k -> k)

- Captured continuation is returned immediately.

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

let f = reset (fun () -> 3 + shift (fun k -> k) - 1)

-> let f = reset (fun () ->

    shift (fun k -> k) )

where k = reset (fun () -> 3 + [... - 1)
How to extract continuations

\[ \text{shift} \ (\text{fun} \ k \to \ k) \]

- Captured continuation is returned immediately.

For example:

\[ \text{reset} \ (\text{fun} \ () \to \ 3 + [...] - 1) \]

\[ \text{let} \ f = \text{reset} \ (\text{fun} \ () \to \ 3 + \text{shift} \ (\text{fun} \ k \to \ k) - 1) \]

\[ \to \ \text{let} \ f = \text{reset} \ (\text{fun} \ () \to \ k) \]

where \( k = \text{reset} \ (\text{fun} \ () \to \ 3 + [...] - 1) \)

\[ f \ 10 \]

\[ \to \ 12 \]
Somewhat advanced example

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

(* id : ’a list -> ’a list *)

let rec id lst = match lst with
  [] -> shift (fun k -> k)
| first :: rest -> first :: id rest

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

The captured cont. conses 3, 2, and 1 in this order.
Solution

```ocaml
# 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]

# let append lst1 =
    reset (fun () -> id lst1);;
append : 'a list -> 'a list -> 'a list = <fun>

# append [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```
How to reorder continuations: \texttt{take}

Given a list and a number $n$, return the given list where the $n$-th element is moved to the front.

\begin{align*}
\text{take } [0; 1; 2; 3; 4] & \ 0 = [0; 1; 2; 3; 4] \\
\text{take } [0; 1; 2; 3; 4] & \ 3 = [3; 0; 1; 2; 4] \\
\text{take } [0; 1; 2; 3; 4] & \ 5 = [0; 1; 2; 3; 4]
\end{align*}

Seemingly easy:
- The original list is almost reconstructed as is.
- Only the designated element is moved.

but:
- The $n$-th element might not exist.
- When found, it must be carried over to the front.
type found_t = Found of int | NotFound

(* int list -> int -> found_t * int list *)
let rec loop lst n = match lst with
  [] -> (NotFound, [])
| first :: rest ->
  if n = 0 then (Found first, rest)
  else let (found, l) = loop rest (n - 1) in
        (found, first :: l)

(* take : int list -> int -> int list *)
let take lst n = match loop lst n with
  (NotFound, l) -> l
| (Found e, l) -> e :: l
Simpler solution

(* loop : 'a list => int => 'a list *)
let rec loop lst n = match lst with
    []  -> []
  | first :: rest ->
     if n = 0 then
          shift (fun k -> first :: k rest)
    else first :: (loop rest (n - 1))

(* take : 'a list -> int -> 'a list *)
let take lst n = reset (fun () -> loop lst n)

take [0; 1; 2; 3; 4] 3 = [3; 0; 1; 2; 4]
A-normalization

Given an (arithmetic) expression, return the same expression where subexpressions are uniquely named.

\[
a - (b - c - d)\]

becomes:

\[
\begin{align*}
&\text{let } e_1 = b - c \text{ in} \\
&\text{let } e_2 = e_1 - d \text{ in} \\
&\text{let } e_3 = a - e_2 \text{ in } e_3
\end{align*}
\]

- Each ‘-’ expression is uniquely named using let.
- When A-normalizer encounters \(b - c\), it has to insert corresponding let expression at the beginning.
A-normalization

(* loop : expr_t => expr_t *)
let rec loop expr = match expr with
    Var (x) -> Var (x)
| Minus (e1, e2) ->
    let nf1 = loop e1 in
    let nf2 = loop e2 in
    let x = gensym "e" in
    shift (fun k ->
        Let (x, Minus (nf1, nf2), k (Var x)))

(* anf : expr_t -> expr_t *)
let anf expr = reset (fun () -> loop expr)
A-normalization: example execution

\[
\begin{align*}
| \text{Minus} (e_1, e_2) & \rightarrow \quad \text{(reshown)} \\
\text{let } \text{nf1} = \text{loop } e_1 \text{ in} \\
\text{let } \text{nf2} = \text{loop } e_2 \text{ in} \\
\text{let } x = \text{gensym } "e" \text{ in} \\
\text{shift } (\text{fun } k \rightarrow \\
\quad \text{Let } (x, \text{Minus } (\text{nf1}, \text{nf2}), k (\text{Var } x))
\end{align*}
\]

\[
\langle \text{loop}[a - (b - c - d)] \rangle \\
\rightarrow \langle g(\text{loop}[a] - \text{loop}[b - c - d]) \rangle \\
\rightarrow \langle g(a - g(\text{loop}[b - c] - \text{loop}[d])) \rangle \\
\rightarrow \langle g(a - g(g(\text{loop}[b] - \text{loop}[c]) - \text{loop}[d])) \rangle \\
\rightarrow \langle g(a - g(g(b - c) - \text{loop}[d])) \rangle \\
\rightarrow \langle \text{let } e_1 = b - c \text{ in } g(a - g(e_1 - \text{loop}[d])) \rangle
\]
A-normalization


- The paper describes how to write a compiler generator ("cogen") for \( \lambda \)-calculus.
- Three lines for variable, abstraction, and application.
- Six lines because each has static/dynamic variants.
- A-normalization (via shift/reset) is crucially used to serialize expressions.
- The technique also known as "let insertion" in partial evaluation.
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".
How to wrap continuations

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

↓ Abort

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

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

↓ Access

(fun () -> k "hello")

↓ Resume

reset (fun () -> "hello" ^ " world")
```
How to wrap continuations: printf

Fill in the hole so that the following program:

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

would return "hello world!". (The hole acts as %s.)

Can you fill in the following hole:

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

so that it returns "It’s 8 o’clock!"? (%d)
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 ;;
How to wrap continuations: printf

The shown solution uses shift and reset.


- This paper shows how printf can be written type-safe in the standard functional languages (without dependent types).
- It is written in continuation-passing style (CPS) and uses continuation in a non-trivial way.
State monad

Define the following without using a mutable cell:

- `put` stores a value into a mutable cell, and
- `get` retrieves a value from the mutable cell.

For example, the following expression evaluates to 11.

```plaintext
put 3; (get () + put 4; get ()) + get ()
```

**idea**

Let the context higher-order, and the mutable cell is passed outside the context (just as we did for printf).
State monad

reset (fun () -> ... expression ...) 0

The cell (0) is passed as an argument of the context.

let get () = shift (fun k -> fun v -> k v v)
let put v = shift (fun k -> fun _ -> k () v)

For example,

reset (fun () -> ...[get ()]...) 0
-> reset (fun () -> fun v -> k v v) 0
-> (fun v -> k v v) 0
-> k 0 0
-> reset (fun () -> ...[0]...) 0
State monad


- Any monads can be represented in direct style using shift/reset.
- Includes complete code in SML.
Future: shift/reset in theorem proving?

The current proof assistants do not allow exception (nor shift/reset).

If we could introduce shift and reset into theorem proving, we are liberated from writing monadic proofs.

Questions:

- Curry-Howard isomorphism for shift and reset?
- What is the type of shift?
- What is the logical meaning of shift?
Curry-Howard isomorphism

Typed functional language

\[ \Gamma \vdash e : A \]

"Under type environment \( \Gamma \),
\( e \) has type \( A \)."

\[ \Gamma, x : A \vdash x : A \]

\[ \Gamma, x : B \vdash e : A \]

\[ \Gamma \vdash \text{fun } x \to e : B \to A \]

\[ \Gamma \vdash f : B \to A \quad \Gamma \vdash a : B \]

\[ \Gamma \vdash f \ a : A \]

If \( \vdash e : A \) can be derived.

Intuitionistic logic

\[ \Delta \vdash A \]

"Under assumption \( \Delta \),
\( A \) holds."

\[ \Delta, A \vdash A \]

\[ \Delta, B \vdash A \]

\[ \Delta \vdash B \supset A \]

\[ \Delta \vdash B \supset A \]

\[ \Delta \vdash B \]

\[ \Delta \vdash A \]

\( A \) holds

If \( \vdash A \) can be derived.
What is the type of shift?

We have to take the type of the context into account.

- Pure (non-shift) expression can appear in any context (answer-type polymorphic).
- Shift restricts the type of its context.

The function `put` and `get` can appear only in the higher-order context.

In general, a function type has the form:

\[
\text{impure } A \rightarrow B @cps[C, D] \\
\text{pure } \forall \alpha. A \rightarrow B @cps[\alpha, \alpha] \equiv A \rightarrow B
\]

What does this type mean logically?
call/cc has type \(((\alpha \to \beta) \to \alpha) \to \alpha\), which is classic (Peirce’s law).

It does not take the answer type into account.

What about shift?

- Shift moves around a part of computation.
- Logically, it cuts and pastes a part of proof tree.
- Is this somehow meaning of \( A \to B \ @cps[C, D] \)?

**Conjecture**

Shift is intuitionistic: even if we use shift, we cannot construct a term having a classic type.
Shift and reset are simple, but quite expressive.

We have a type system for shift and reset, but their relationship to logic is unknown.

Q: We can always turn a program with shift/reset into a program without by CPS transformation. Are shift/reset really necessary?

A: Yes, just like higher-order functions whose necessity must have been questioned long time ago. They provide us with higher level of abstraction.
How to use shift/reset

OchaCaml

shift/reset-extension of Caml Light:
http://pllab.is.ocha.ac.jp/~asai/OchaCaml

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!