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: `times`
- How to extract continuations: `append`
- How to reorder continuations: `take`, A-normalize
- How to wrap continuations: `printf`, state monad

Speculation:

- Toward delimited continuations in theorem proving

Early papers on control operators

- Control/prompt

M. Felleisen [POPL 1988]
"The Theory and Practice of First-Class Prompts"

- Shift/reset

O. Danvy and A. Filinski [LFP 1990]
"Abstracting Control"

O. Danvy and A. Filinski [MSCS 1992]
"Representing Control,
a Study of the CPS Transformation"

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 subtract 1 from the sum." i.e., `fun x -> 3 + x - 1`

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: $[\,\cdot\,]$.

Examples

Identify the current expressions, continuations, and their types.

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

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

Examples

Identify the current expressions, continuations, and their types.

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

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

Examples

Identify the current expressions, continuations, 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"`

Examples

Identify the current expressions, continuations, 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"
```

Examples

Identify the current expressions, continuations, 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" :
```

Examples

Identify the current expressions, continuations, 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 ->
```

Examples

Identify the current expressions, continuations, 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
```

What are delimited continuations?

Delimited Continuation

The rest of the computation up to the delimiter.

Syntax

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

For example:

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

- The current computation: $5 * 2$
- 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 * reset (fun () -> [2 * 3] + 3 * 4)
```
$[\cdot] + 3 * 4$:

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

Examples

Identify the delimited continuations, and their types.

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

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

Examples

Identify the delimited continuations, and their types.

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" :
```

Examples

Identify the delimited continuations, and their types.

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`
   ```

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:

`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 the body M in the empty context.

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 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:

```
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
```

Examples

Replace [·] with `shift (fun _ -> ` M`)` for some M.

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

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

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

Examples

Replace [·] with `shift (fun _ ->` M`)` for some M.

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

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

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

Examples

Replace `[ · ]` with `shift (fun _ -> ` M`)` for some M.

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

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

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

Examples

Replace `[·]` with `shift (fun _ -> ` M`)` for some M.

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"

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

```
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)
```

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

```
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 () ->
                                  k      )
   where k = reset (fun () -> 3 + [...] - 1)
   f 10
-> 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

```
# 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: `take`

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

```
take [0; 1; 2; 3; 4] 0 = [0; 1; 2; 3; 4]
take [0; 1; 2; 3; 4] 3 = [3; 0; 1; 2; 4]
take [0; 1; 2; 3; 4] 5 = [0; 1; 2; 3; 4]
```

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)
```

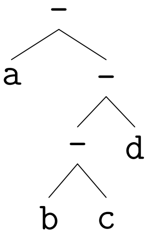
```
take [0; 1; 2; 3; 4] 3 = [3; 0; 1; 2; 4]
```

A-normalization

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

`a - (b - c - d)` becomes:

```
let e1 = b - c in
let e2 = e1 - d in
let e3 = a - e2 in e3
```



- 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

```
| Minus (e1, e2) ->                                    (reshown)
  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)))
```

$$
\begin{aligned}
& \langle \mathrm{loop}[\![\mathtt{a - (b - c - d)}]\!]\rangle \\
\to\ & \langle g(\mathrm{loop}[\![\mathtt{a}]\!]\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{b - c - d}]\!])\rangle \\
\to\ & \langle g(\mathtt{a}\ \mathtt{-}\ g(\mathrm{loop}[\![\mathtt{b - c}]\!]\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{d}]\!]))\rangle \\
\to\ & \langle g(\mathtt{a}\ \mathtt{-}\ g(g(\mathrm{loop}[\![\mathtt{b}]\!]\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{c}]\!])\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{d}]\!]))\rangle \\
\to\ & \langle g(\mathtt{a}\ \mathtt{-}\ g(\boxed{g(\mathtt{b - c})}\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{d}]\!]))\rangle \\
\to\ & \langle \mathtt{let\ e1 = b - c\ in}\ \langle g(\mathtt{a}\ \mathtt{-}\ g(\mathtt{e1}\ \mathtt{-}\ \mathrm{loop}[\![\mathtt{d}]\!]))\rangle\rangle
\end{aligned}
$$

A-normalization

P. Thiemann "Cogen in Six Lines," ICFP 1996.

- The paper describes how to write a compiler generator ("cogen") for λ-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

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

↓ Abort

```
reset (fun () ->
          [fun () -> k "hello"]) ()
     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:

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

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

Can you fill in the following hole:

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

so that it returns `"It's 8 o'clock!"`? (%d)

Solution

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

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

```
reset (fun () ->                                    (%d)
  "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`.

O. Danvy "Functional Unparsing," JFP 1998.

- 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` retrives a value from the mutable cell.

For example, the following expression evaluates to 11.

```
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

A. Filinski "Representing Monads," POPL 1994.

- 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	Intuitionistic logic
$\Gamma \vdash \texttt{e} : A$	$\Delta \vdash A$
"Under type environment Γ, e has type A."	"Under assumption Δ, A holds."
$\dfrac{}{\Gamma, \texttt{x} : A \vdash \texttt{x} : A}$	$\dfrac{}{\Delta, A \vdash A}$
$\dfrac{\Gamma, \texttt{x} : B \vdash \texttt{e} : A}{\Gamma \vdash \texttt{fun x -> e} : B \to A}$	$\dfrac{\Delta, B \vdash A}{\Delta \vdash B \supset A}$
$\dfrac{\Gamma \vdash \texttt{f} : B \to A \quad \Gamma \vdash \texttt{a} : B}{\Gamma \vdash \texttt{f a} : A}$	$\dfrac{\Delta \vdash B \supset A \quad \Delta \vdash B}{\Delta \vdash A}$
e has type A if $\vdash \texttt{e} : A$ can be derived.	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:

impure `A -> B @cps[C, D]`

pure $\forall\alpha$`.A -> B @cps[`α, α`]` $\cong$ `A -> B`

What does this type mean logically?

T. Griffin "A Formulae-as-Types Notion of Control,"
POPL 1990.

- `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 -> B @cps[C, D]`?

Conjecture

`Shift` is intuitionistic: even if we use `shift`, we cannot construct a term having a classic type.

Summary

- `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 transforamtion. 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`!