Introduction to Delimited Continuations

Typing Printf

（継続を使った Printf の型付け）

Kenichi Asai

Ochanomizu University

April 13, 2008
Outline of the talk

- **times** (introduction)
  - (1) in Direct Style with exception
  - (2) in Continuation-Passing Style (CPS)

- **sprintf**
  - (3) in Continuation-Passing Style (CPS)
  - (4) in Direct Style with shift/reset

- Related Work / Summary
Multiply elements of a given list:

\[
\text{let rec times lst = match lst with }
\]
\[
\text{[ ] -> 1 }
\]
\[
\text{ | first :: rest -> first * times rest }
\]

For example,

\[
\text{times [1; 2; 3; 4; 5]}
\]
\[
\leadsto 1 * \text{times [2; 3; 4; 5]}
\]
\[
\leadsto 1 * (2 * \text{times [3; 4; 5]})
\]
\[
\leadsto 1 * (2 * (3 * \text{times [4; 5]}))
\]
\[
\leadsto 1 * (2 * (3 * (4 * \text{times [5]})))
\]
\[
\leadsto 1 * (2 * (3 * (4 * (5 * \text{times [ ]}))))
\]
\[
\leadsto 1 * (2 * (3 * (4 * (5 * 1))))
\]
\[
\leadsto * 120
\]
Multiply elements of a given list:

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

For example,

\[
times [1; 2; 0; 4; 5] \\
t \mapsto 1 * times [2; 0; 4; 5] \\
t \mapsto 1 * (2 * times [0; 4; 5]) \\
t \mapsto 1 * (2 * (0 * times [4; 5])) \\
t \mapsto 1 * (2 * (0 * (4 * times [5])))) \\
t \mapsto 1 * (2 * (0 * (4 * (5 * times [])))) \\
t \mapsto 1 * (2 * (0 * (4 * (5 * 1)))) \\
t \mapsto 0
\]
(1) Times: Direct Style

If 0 is found, we can return 0 immediately:

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

Then, we have:

```
times [1; 2; 0; 4; 5]
⇝ 1 * times [2; 0; 4; 5]
⇝ 1 * (2 * times [0; 4; 5])
⇝ 1 * (2 * 0)
⇝ 1 * 0
⇝ 0
```

We could avoid traversing [4; 5], but the unnecessary multiplication still remains.
To discard the unnecessary multiplication, we use exception:

```ocaml
def times lst = match lst with
  | [] -> 1
  | 0 :: rest -> raise Zero
  | first :: rest -> first * times rest
```

Then, we have:

```
try (times [1; 2; 0; 4; 5]) with Zero -> 0
⇝ try (1 * times [2; 0; 4; 5]) with Zero -> 0
⇝ try (1 * (2 * times [0; 4; 5])) with Zero -> 0
⇝ try (1 * (2 * raise Zero)) with Zero -> 0
⇝ 0
```

- The context (1 * (2 * μ)) is discarded, enabling non-local jump.
- Exception mechanism is required.
Continuation = work to be done after the current computation.

```ml
let rec times lst cont = match lst with
  [] -> cont 1
| first :: rest ->
  times rest (fun r -> cont (first * r))
```

Then, we have:

```
times [1; 2; 0; 4; 5] i
⇝ times [2; 0; 4; 5] (fun r -> i (1 * r))
⇝ times [0; 4; 5] (fun r -> i (1 * (2 * r)))
⇝ times [4; 5] (fun r -> i (1 * (2 * (0 * r))))
⇝ times [5] (fun r -> i (1 * (2 * (0 * (4 * r)))))
⇝ times [] (fun r -> i (1 * (2 * (0 * (4 * (5 * r))))))
⇝ i (1 * (2 * (0 * (4 * (5 * 1)))))
⇝* i 0
```

where \(i\) is an initial continuation (for example, \(\text{fun } x \to x\)).
(2) Times: Continuation-Passing Style (CPS)

Avoid traversing over the list after 0 is found:

```ocaml
let rec times lst cont = match lst with
  [] -> cont 1
| 0 :: rest -> cont 0
| first :: rest ->
  times rest (fun r -> cont (first * r))
```

Then, we have:

```
times [1; 2; 0; 4; 5] i
⇝ times [2; 0; 4; 5] (fun r -> i (1 * r))
⇝ times [0; 4; 5] (fun r -> i (1 * (2 * r)))
⇝ i (1 * (2 * 0))
⇝ i (1 * 0)
⇝ i 0
```

Still, 1 * (2 * 0) is computed.
(2) Times: Continuation-Passing Style (CPS)

By not using cont, 0 is returned to the original call site.

```
let rec times lst cont = match lst with
  [] -> cont 1
| 0 :: rest -> 0 (* cont is discarded! *)
| first :: rest ->
    times rest (fun r -> cont (first * r))
```

Then, we have the following (where id = fun x -> x):

```
times [1; 2; 0; 4; 5] id
~\Rightarrow times [2; 0; 4; 5] (fun r -> id (1 * r))
~\Rightarrow times [0; 4; 5] (fun r -> id (1 * (2 * r)))
~\Rightarrow 0
```

- Non-local jump is realized through writing a program in CPS.
- We have to write whole the program in CPS.
(2) Times: Direct Style with shift/reset

Write a program in direct style with:

- **shift**: captures the current continuation (up to reset)
- **reset**: installs the empty (identity) continuation

```ocaml
let rec times lst = match lst with
    | [] -> 1
    | 0 :: rest -> shift (fun cont -> 0)
    | first :: rest -> first * times rest
```

Then, we have:

```
reset (fun () -> times [1; 2; 0; 4; 5])
⇝ reset (fun () -> 1 * times [2; 0; 4; 5])
⇝ reset (fun () -> 1 * (2 * times [0; 4; 5]))
⇝ reset (fun () -> 1 * (2 * shift (fun cont -> 0)))
⇝ 0
```

where `cont = fun r -> reset (fun () -> (1 * (2 * r)))`. 
Printf

Goal:

\[
\text{printf (lit "Hello world!")}
\Rightarrow "Hello world!"
\]

\[
\text{printf (lit "Hello " ++ lit "world!")}
\Rightarrow "Hello world!"
\]

\[
\text{printf (% str ++ lit "world!") "Hello "}
\Rightarrow "Hello world!"
\]

\[
\text{printf (% str ++ lit " is " ++ % int) "t" 3}
\Rightarrow "t is 3"
\]

The types of the boxes depend on the first argument of `printf`. 
\[ \Rightarrow \text{Do we need dependent types? – No!} \]
Observation

The occurrence of % changes the type of the box, in other words, the type of the context (=continuation!).

```
sprintf(\% str ++ lit " is " ++ \% int) "t" 3
```

\[ \leadsto "t is 3" \]

If the current continuation is available at hand, we could write a type-safe sprintf without using dependent types.

Danvy [JFP 1998] did this by writing the boxed parts in CPS.
The continuation is initialized by `sprintf`:

```ocaml
let sprintf pattern = pattern id
```

The string literal is simply passed to the continuation:

```ocaml
let lit s cont = cont s
```

For example, we have:

```ocaml
let lit s cont = cont s

sprintf (lit "Hello world!")
```

```ocaml
⇝ lit "Hello world!" id
```

```ocaml
⇝ id "Hello world!"
```

```ocaml
⇝ "Hello world!"
```
Two patterns, p1 and p2, are concatenated in a CPS manner:

```ml
let (++) p1 p2 cont =
  p1 (fun x -> p2 (fun y -> cont (x \^ y)))
```

Then, we have:

```ml
sprintf (lit "Hello " ++ lit "world!")
⇝ (lit "Hello " ++ lit "world!") id
⇝ lit "Hello " (fun x -> lit "world!" (fun y ->
    id (x \^ y)))
⇝* "Hello world!"
```
Let `int` and `str` be functions that return string representation of their argument:

```ml
let int x = string_of_int x
let str (x : string) = x
```

Then, `%` can be defined as follows:

```ml
let % to_str cont = fun z -> cont (to_str z)
```

We have:

```ml
sprintf (% str ++ lit "world!") "Hello "
⇝ (% str ++ lit "world!") id "Hello "
⇝ % str (fun x -> lit "world!" (fun y -> id (x ^ y)))
   "Hello "
⇝ (fun z -> (...) (str z)) "Hello "
⇝* "Hello world!"
```
Printf: Continuation-Passing Style (CPS)

Multiple uses of \% leads to accepting more arguments.

\[
\text{sprintf (\% str ++ \% int) "t" 3} \\
\text{\rightarrow (\% str ++ \% int) id "t" 3} \\
\text{\rightarrow \% str (fun x \rightarrow \% int (fun y \rightarrow id (x ^ y))) "t" 3} \\
\text{\rightarrow* (fun z \rightarrow (\ldots) (str z)) "t" 3} \\
\text{\rightarrow \% int (fun y \rightarrow id ("t" ^ y)) 3} \\
\text{\rightarrow (fun z \rightarrow (\ldots) (int z)) 3} \\
\text{\rightarrow* id ("t" ^ "3")} \\
\text{\rightarrow id "t3"} \\
\text{\rightarrow "t3"}
\]
Printf: Continuation-Passing Style (CPS)

Complete program (executable in OCaml):

```
let sprintf pattern = pattern (fun (s : string) -> s)
let lit s cont = cont s
let (++) p1 p2 cont =
  p1 (fun x -> p2 (fun y -> cont (x ^ y)))

let int x = string_of_int x
let str (x : string) = x

(* % : ('b -> string) -> (string -> 'a) -> 'b -> 'a *)
let (%) to_str cont = fun z -> cont (to_str z)
```

▸ Practical note: Because % in OCaml is an infix operator, we have to write (%) instead of %.
By transforming the CPS solution back to direct style (with shift and reset), we obtain:

```ocaml
let sprintf pattern ≡ reset (fun () -> pattern)
let lit s = s
let (++) p1 p2 = p1 ^ p2

let int x = string_of_int x
let str (x : string) = x

let (%) to_str =
    shift (fun cont -> fun z -> cont (to_str z))
```

To run this program, one requires implementation of shift/reset that supports answer type modification.
Abbreviating \((\text{reset } (\text{fun } () \to \ldots))\) as \(\langle \ldots \rangle\), we have:

\[
\begin{align*}
\text{printf } (\% \text{str } \wedge \text{"world!"}) & \rightarrow \langle \% \text{str } \wedge \text{"world!"}\rangle \text{"Hello "} \\
\langle \text{shift } (\text{fun cont } \to \text{fun z } \to \text{cont } \text{(str z)})) \wedge \text{"world!"}\rangle & \rightarrow \langle \text{fun z } \to \langle \text{str z } \wedge \text{"world!"}\rangle\rangle \text{"Hello "} \\
\langle \text{fun z } \to \langle \text{str z } \wedge \text{"world!"}\rangle\rangle & \rightarrow* \text{"Hello world!"}
\end{align*}
\]

In CPS, it was:

\[
\begin{align*}
\text{printf } (\% \text{str } \mathbin{++} \text{lit } \text{"world!"}) & \rightarrow (\% \text{str } \mathbin{++} \text{lit } \text{"world!"}) \text{id } \text{"Hello "} \\
\% \text{str} (\text{fun x } \to \text{lit } \text{"world!"} (\text{fun y } \to \text{id } (\text{x } \wedge \text{y}))) \text{"Hello "} & \rightarrow (\text{fun z } \to (\ldots) \text{(str z)}) \text{"Hello "} \\
\langle \text{fun z } \to (\ldots) \text{(str z)}\rangle & \rightarrow* \text{"Hello world!"}
\end{align*}
\]
Related Work

Printf problem:

- Hinze [JFP 2003] solved the same problem in Haskell using type classes.

The implementation of shift/reset:

- shift/reset can be implemented using call/cc with a mutable cell [Filinski, POPL 1994].
- Direct implementation of shift/reset for Scheme48 by Gasbichler and Sperber [ICFP 2002].
- Kiselyov implements shift/reset for OCaml and Haskell [2007]. They support answer type modification and polymorphism, and thus can execute direct-style printf program.
Summary

- Introduction to shift/reset using times and sprintf.
- Exact correspondence between CPS programs and direct-style programs with shift/reset. With shift/reset, we obtain the power of CPS without converting the program into CPS.

Q: Are shift/reset necessary if we can always simulate them by writing programs in CPS?

- Yes. Long time ago when higher-order functions are introduced, people must have argued that they are unnecessary because we can always write a program without using higher-order functions.
- Now, we know higher-order functions are useful. They provide us with a more abstract view.
- Likewise, control operators such as shift/reset provide us with higher level of abstraction.

I warmly invite you to the world of delimited continuations!