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

(1) Times: Direct Style

Multiply elements of a given list:

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

For example,

$$
\begin{array}{ll}
 & \texttt{times [1; 2; 3; 4; 5]} \\
\rightsquigarrow & \texttt{1 * times [2; 3; 4; 5]} \\
\rightsquigarrow & \texttt{1 * (2 * times [3; 4; 5])} \\
\rightsquigarrow & \texttt{1 * (2 * (3 * times [4; 5]))} \\
\rightsquigarrow & \texttt{1 * (2 * (3 * (4 * times [5])))} \\
\rightsquigarrow & \texttt{1 * (2 * (3 * (4 * (5 * times []))))} \\
\rightsquigarrow & \texttt{1 * (2 * (3 * (4 * (5 * 1))))} \\
\rightsquigarrow^{*} & \texttt{120}
\end{array}
$$

(1) Times: Direct Style

Multiply elements of a given list:

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

For example,

$$
\begin{array}{ll}
& \texttt{times [1; 2; 0; 4; 5]} \\
\leadsto & \texttt{1 * times [2; 0; 4; 5]} \\
\leadsto & \texttt{1 * (2 * times [0; 4; 5])} \\
\leadsto & \texttt{1 * (2 * (0 * times [4; 5]))} \\
\leadsto & \texttt{1 * (2 * (0 * (4 * times [5])))} \\
\leadsto & \texttt{1 * (2 * (0 * (4 * (5 * times []))))} \\
\leadsto & \texttt{1 * (2 * (0 * (4 * (5 * 1))))} \\
\leadsto^{*} & \texttt{0}
\end{array}
$$

(1) Times: Direct Style

If 0 is found, we can return 0 immediately:

```
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.

(1) Times: Direct Style with Exception

To discard the unnecessary multiplication, we use exception:

```
let rec 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 $(1 * (2 * \square))$ is discarded, enabling non-local jump.
- Exception mechanism is required.

(2) Times: Continuation-Passing Style (CPS)

Continuation = work to be done after the current computation.

```
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, fun x -> x).

(2) Times: Continuation-Passing Style (CPS)

Avoid traversing over the list after 0 is found:

```
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
⇝  times [2; 0; 4; 5] (fun r -> id (1 * r))
⇝  times [0; 4; 5] (fun r -> id (1 * (2 * r)))
⇝  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 contiuation (up to `reset`)
- `reset` : installs the empty (identity) continuation

```
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:

$\boxed{\texttt{sprintf (lit "Hello world!")}}$

$\rightsquigarrow$ `"Hello world!"`

$\boxed{\texttt{sprintf (lit "Hello " ++ lit "world!")}}$

$\rightsquigarrow$ `"Hello world!"`

$\boxed{\texttt{sprintf (\% str ++ lit "world!")}}$ `"Hello "`

$\rightsquigarrow$ `"Hello world!"`

$\boxed{\texttt{sprintf (\% str ++ lit " is " ++ \% int)}}$ `"t" 3`

$\rightsquigarrow$ `"t is 3"`

The types of the boxes depend on the first argument of `sprintf`.

$\Longrightarrow$ 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!).

$$\boxed{\texttt{sprintf (\% str ++ lit " is " ++ \% int)}}\ \texttt{"t" 3}$$
$$\rightsquigarrow\ \texttt{"t is 3"}$$

If the current contiuation 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.

(3) Printf: Continuation-Passing Style (CPS)

The continuation is initialized by `sprintf`:

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

The string literal is simply passed to the continuation:

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

For example, we have:

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

(3) Printf: Continuation-Passing Style (CPS)

Two patterns, p1 and p2, are concatenated in a CPS manner:

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

Then, we have:

```
     sprintf (lit "Hello " ++ lit "world!")
⇝    (lit "Hello " ++ lit "world!") id
⇝    lit "Hello " (fun x -> lit "world!" (fun y ->
                                         id (x ^ y)))
⇝*   "Hello world!"
```

(3) Printf: Continuation-Passing Style (CPS)

Let `int` and `str` be functions that return string representation of their argument:

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

Then, % can be defined as follows:

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

We have:

```
    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!"
```

(3) Printf: Continuation-Passing Style (CPS)

Multilpe uses of % leads to accepting more arguments.

```
     sprintf (% str ++ % int) "t" 3
⇝    (% str ++ % int) id "t" 3
⇝    % str (fun x -> % int (fun y -> id (x ^ y))) "t" 3
⇝*   (fun z -> (...) (str z)) "t" 3
⇝    % int (fun y -> id ("t" ^ y)) 3
⇝    (fun z -> (...) (int z)) 3
⇝*   id ("t" ^ "3")
⇝    id "t3"
⇝    "t3"
```

(3) 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 %.

(4) Printf: Direct Style with shift/reset

By transforming the CPS solution back to direct style (with shift and reset), we obtain:

```
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.

(4) Printf: Direct Style with shift/reset

Abbreviating `(reset (fun () -> ...))` as $\langle \ldots \rangle$, we have:

```
     sprintf (% str ^ "world!") "Hello "
~>   ⟨% str ^ "world!"⟩ "Hello "
~>   ⟨(shift (fun cont -> fun z -> cont (str z)))
      ^ "world!"⟩ "Hello "
~>   ⟨fun z -> ⟨str z ^ "world!"⟩⟩ "Hello "
~>*  "Hello world!"
```

In CPS, it was:

```
     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!"
```

Related Work

Printf problem:

- Danvy [JFP 1998] presented a type-safe printf in ML. It is written in CPS and uses an accumulator parameter. Incorporated in Standard ML of New Jersey.
- 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!