On Typing Delimited Continuations: Three New Solutions to the Printf Problem

Kenichi Asai
Ochanomizu University, 2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan
http://pllab.is.ocha.ac.jp/~asai

Abstract

In "Functional Unparsing" (JFP 8(6): 621–625, 1998), Danvy presented a type-safe printf function using continuations and an accumulator to achieve the effect of dependent types. The key technique employed in Danvy's solution is the non-standard use of continuations: not all of its calls are tail calls, *i.e.*, it uses delimited continuations. Against this backdrop, we present three new solutions to the printf problem: a simpler one that also uses delimited continuations but that does not use an accumulator, and the corresponding two in direct style with the delimited-control operators, `shift` and `reset`. These two solutions are the direct-style counterparts of the two continuation-based ones. The last solution pinpoints the essence of Danvy's solution: `shift` is used to change the answer type of delimited continuations. Besides providing a new application of `shift` and `reset`, the solutions in direct style raise a key issue in the typing of first-class delimited continuations and require Danvy and Filinski's original type system. The resulting types precisely account for the behavior of printf.

This is the extended version of the previous technical report OCHA-IS 07-1. It contains an introduction to continuation-passing style and delimited-control operators, `shift` and `reset`.

Keywords Printf, Delimited Continuations, Continuation-Passing Style (CPS), Direct Style, Types

1 Background and Introduction

Over the last decade, type-indexed functions have often been illustrated with the printf problem: writing a printf-like function that takes a formatting directive and several arguments to format and returns a formatted string. Typing this function appears to require dependent types, because the types of the arguments to the function vary according to the formatting directive. Contrary to this intuition, Danvy [3] showed that the printf function can be expressed within the ML type system, without using dependent types.

The key technique employed in Danvy's solution is the non-standard use of continuations. He achieves the dependent behavior of printf by using a control effect that modifies the return type of continuations according to the given pattern directives. Since the technique hinges on continuations, the solution requires to be written either with explicit continuations, *i.e.*, in continuation-passing style (CPS), or with implicit continuations, *i.e.*, in direct style using control operators. Danvy's original solution takes the first approach. The present article explores both approaches and puts them into perspective: we show another continuation-based solution and two more solutions in direct style with delimited-control operators, `shift` and `reset`. The solutions turn out to pinpoint the essence of Danvy's solution. In particular, the final direct-style solution is so concise that it is almost trivial except for one function where `shift` is used to achieve the dependent behavior of the printf function. The use of `shift` in the solutions appears to be natural because the desired behavior for the directives is to change the behavior of its context, not the output string itself.

The direct-style solutions raise a key issue in the typing of delimited-control operators. These solutions do not type check if we use Filinski's implementation [8] of `shift` and `reset`. We show that they are typable with the original type system by Danvy and Filinski [4], and that their type precisely accounts for the dependent behavior of printf. The type system therefore provides an accurate abstraction of the behavior of the printf function and of its control effects.

After describing the printf problem in the next section, we first introduce the basic concepts of CPS in Section 3. We then review Danvy's solution in Section 4. Next, we introduce the delimited-control operators, `shift` and `reset`, in Section 5. Using them, we present our first direct-style solution in Section 6. Section 7 demonstrates that this solution does not type check with Filinski's implementation of delimited continuations in ML, and Section 8 shows that it does with Danvy and Filinski's original type system. In Section 9, we simplify Danvy's solution by eliminating the accumulator to obtain our second continuation-based solution. Applying the same simplification to the direct-style solution yields our third direct-style solution in Section 10. Related work is reviewed in Section 11 and the article concludes in Section 12. The four solutions in the paper are summarized in Table 1.

Table 1: Four Solutions to the Printf Problem

	with an accumulator	without an accumulator
continuation-based	Section 4	Section 9
in direct style with `shift` and `reset`	Section 6	Section 10

Prerequisites and domain of discourse We assume basic familiarity with CPS transformation [6, 11, 12]. We use the call-by-value functional language OCaml[1] as an implementation language, but one could also use SML/NJ.[2]

2 Problem Specification

The printf problem is specified as follows. Assume that the formatting string consists of the following pattern directives:

- `lit` for literal strings,
- `eol` for newlines, and
- a field type specifier `%` for input field arguments that accepts one of the following representations of types:
  - `int` for integers, or

[1] `http://caml.inria.fr/ocaml/`
[2] `http://www.smlnj.org/`

– `str` for strings.

The problem is to write a function `sprintf` together with the above pattern directives and type representations in a type-safe manner. For example, assuming that `++` is an infix operator that connects pattern directives,

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

should have type `string` and evaluate to the value `"Hello world!\n"`;

```
sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol)
```

should have type `string -> string`, and thus

```
sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol) "world"
```

should evaluate to `"Hello world!\n"`; and

```
sprintf (lit "The value of " ++ % str ++ lit " is " ++ % int ++ eol)
```

should have type `string -> int -> string`, and thus

```
sprintf (lit "The value of " ++ % str ++ lit " is " ++ % int ++ eol)
        "x" 3
```

should evaluate to `"The value of x is 3\n"`. The return type of `sprintf` therefore depends on the given pattern.

3 Continuation-Passing Style

Since Danvy's solution to the printf problem [3] involves continuations, let us first remind the reader of the essence of CPS. Readers familiar with CPS can skip this section.

A program is said to be written in CPS when all its functions receive a continuation as an additional argument and when all transfers of control are made through:

- a tail-call to another function, or
- a tail-call to a continuation.

The only functions that may not receive continuations are pure and total—*e.g.*, primitive functions [7].

In contrast, ordinary non-CPS programs are called to be in direct style. The terms 'direct semantics' and 'continuation semantics' originate in the area of denotational semantics [13]. The terms 'direct style' and 'continuation-passing style' then migrated to the world of functional programming. The term 'CPS,' in particular, is due to Steele [12].

For example, consider the following direct-style program written in OCaml that multiplies all the elements of a given list:

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

(*  main : int list -> int  *)
let main lst = times lst
```

The corresponding CPS version of the same program is expressed as follows:

```
(*  times2 : int list -> (int -> 'a) -> 'a  *)
let rec times2 lst k = match lst with
    [] -> k 1
  | 0 :: rest -> k 0
  | n :: rest -> times2 rest (fun r -> k (n * r))

(*  main2 : int list -> int  *)
let main2 lst = times2 lst (fun r -> r)
```

Returning a value is expressed as passing the value to a continuation `k`, and nested function calls are flattened by packaging the "rest of the computation" into the continuation argument.

The transformation from a direct-style program into a CPS program is called CPS transformation [11, 12]. There is a mechanical way to do it; we recommend Danvy and Filinski's article [6] on this topic.

Since the new function requires a continuation as its second argument, we need to pass an initial continuation to execute it. For example, if we supply an empty continuation (an identity function) as in `main2`, we obtain the result as is:

```
# times2 [1; 2; 3; 4] (fun r -> r);;
- : int = 24
#
```

whereas if we supply `string_of_int`, we obtain the string representation of the result:

```
# times2 [1; 2; 3; 4] string_of_int;;
- : string = "24"
#
```

It is important to notice that the *answer type* of the supplied continuation (*i.e.*, co-domain of `k`) is arbitrary and it becomes the type of the final result (*i.e.*, the type of `times2 lst k`). In the definition of `times2` above, both the types are shown as `'a`. We call this property *answer type polymorphism* [15].

When we write a program using continuations, we usually do *not* strictly follow the rule of CPS. If a program is strictly in CPS, it is always possible to transform it back to direct style [2]. The obtained direct-style program would then be simpler, easier to understand, and run faster than the CPS program. We write a program using continuations because by deviating from the rule of CPS, we obtain the additional expressive power over the control structure of the program.

For example, in the definition of `times2`, we obtain non-local jump (or abortion) by *not* calling the continuation `k`:

```
(*  times3 : int list -> (int -> int) -> int  *)
let rec times3 lst k = match lst with
    [] -> k 1
  | 0 :: rest -> 0  (* k is discarded! *)
  | n :: rest -> times3 rest (fun r -> k (n * r))

(*  main3 : int list -> int  *)
let main3 lst = times3 lst (fun r -> r)
```

When `0` is found in `lst`, the control is transferred directly to the caller of `times3` (*i.e.*, `main3`), discarding all the multiplications stacked in `k`. This kind of behavior was impossible in a direct-style program unless some additional language mechanism is incorporated, such as exception handling.

In this article, we describe such deviation from the standard CPS style as *non-standard.* When continuations are used in a non-standard way, it typically has an impact on the answer type polymorphism. For example, whereas `times2` could accept a continuation of type `int -> 'a` for an arbitrary `'a`, the revised `times3` accepts only a continuation of type `int -> int`. Since the continuation may be discarded during the execution of `times3`, the answer type of the continuation (the type of `k 1` in the definition of `times3`) and the final answer type (the type of `0`) have to be the same. Thus, we can supply an initial continuation as in `main3`:

```
# times3 [1; 2; 3; 4] (fun r -> r);;
- : int = 24
#
```

We cannot, however, supply `string_of_int`:

```
# times3 [1; 2; 3; 4] string_of_int;;
This expression has type int -> string
but is here used with type int -> int
#
```

Since the continuation may be discarded, it was not a good idea after all to pass `string_of_int` to `times3` if we always want to turn the result into a string. Instead, we 'close' the CPS world by supplying an initial continuation and apply `string_of_int` afterwards (in direct style):

```
# string_of_int (times3 [1; 2; 3; 4] (fun r -> r));;
- : string = "24"
#
```

To this end, we discover that the continuation passed around in `times3` does not represent the whole rest of the computation but only a part of it, up to when the empty continuation was initially supplied. Such continuations are called *delimited* continuations.

When programs are written in CPS to exploit the power of continuations, it is typically the case that the answer type polymorphism is lost and continuations are delimited. Therefore, the key to understanding a program written with continuations is to recognize when the answer type polymorphism is lost and how delimited continuations are used in a non-standard way.

4 Continuation-based Solution

Having reviewed the main characteristics of CPS, let us now examine Danvy's solution [3] to the printf problem. His solution is mostly in CPS and makes a non-standard use of continuations. Since it is not strictly in CPS, we call his solution 'continuation-based' in this article.

Danvy's continuation-based solution involves two techniques. First, it encodes all the necessary information into pattern directives, not in the `sprintf` function itself. Second, it uses continuations in a non-standard way to change the return type of `sprintf` according to the given pattern. Let us review this solution:

```
(*  lit : string -> (string -> 'a) -> string -> 'a  *)
let lit x = fun k -> fun s -> k (s ^ x)

(*  eol : (string -> 'a) -> string -> 'a  *)
let eol = fun k -> fun s -> k (s ^ "\n")
```

The pattern directives `lit x` and `eol` are not encoded as strings but as string transformers written in CPS. Accordingly, two patterns cannot be connected using a string concatenation operator `^`, but are connected using function composition:

```
(*  (++) : ('b -> 'a) -> ('c -> 'b) -> 'c -> 'a  *)
let (++) f1 f2 = fun k -> f1 (f2 k)
```

Although this definition appears to be not in CPS, if we η-expand it as follows:

```
(*  (++) : (('a -> 'b) -> 'c -> 'd) -> ('e -> 'a -> 'b) ->
           'e -> 'c -> 'd  *)
let (++) f1 f2 = fun k -> fun s -> f1 (fun s -> f2 k s) s
```

we see that `(++)` is in fact written in CPS. Danvy's solution cleverly places the continuation argument `k` before the string argument `s`, so that the definition of `(++)` is simplified to a function composition. This contributes to the good performance of his solution.

Forgetting about `%` for now, `sprintf` can be defined by passing the initial continuation (the identity function) and the initial string (the empty string) to the pattern:

```
(*  sprintf : ((string -> string) -> string -> 'a) -> 'a  *)
let sprintf p = p (fun (s : string) -> s) ""

# sprintf (lit "Hello world!" ++ eol);;
- : string = "Hello world!\n"
#
```

Now, the remaining task is to define `%`, `int`, and `str`. Let us define `int` and `str` as primitive functions that return the string representation of their inputs:

```
(*  int : int -> string  *)
let int x = string_of_int x

(*  str : string -> string  *)
let str (x : string) = x
```

The definitions of the above two functions remain the same throughout the four solutions in this article.

Then, the field type specifier `%` can be defined as follows:[3]

```
(*  % : ('b -> string) -> (string -> 'a) -> string -> 'b -> 'a  *)
let % to_str = fun k -> fun s -> fun x -> k (s ^ to_str x)
```

[3]The identifier `%` cannot be actually used because it is an infix symbol in OCaml. One has to use either `(%)` or other prefix identifier instead. In this article, we simply assume it is a prefix symbol.

Given a field type `to_str` (either `int` or `str`) together with the usual arguments `k` and `s`, `%` does *not* pass a string to `k` but instead returns a function that receives an additional argument `x`. This is where the rule of CPS is violated. Instead of passing a result to `k` at a tail position, a function is returned directly. Since `k` is not applied immediately, the current continuation is effectively discarded (until the function is applied). The control is transferred non-locally to the place where the initial continuation was supplied, *i.e.*, where `sprintf` was called. The result of applying `sprintf` becomes this function rather than a string, thus making it possible to accept another argument. So if there is one occurrence of `%`:

```
# sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol);;
- : string -> string = <fun>
#
```

we obtain a function of one argument as a result of applying `sprintf`. It might be helpful to see the following syntactic trace (*i.e.*, reduction sequence) to understand the internals of the resulting function. Let `id` be an initial continuation.

```
   sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol)
→  (lit "Hello " ++ % str ++ lit "!" ++ eol) id ""
→  (lit "Hello " (% str (lit "!" (eol id)))) ""
→  (% str (lit "!" (eol id))) "Hello "
→  fun x -> (lit "!" (eol id)) ("Hello " ^ str x)
```

At the last step, the execution of the continuation `lit "!" (eol id)` is suspended[4] and a function is returned. When this function is applied to a string (*e.g.*, `"world"`), the continuation hidden in the function is activated and the final string is returned:

```
# sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol) "world";;
- : string = "Hello world!\n"
#
```

Observe how the non-standard use of a continuation in `%` changes the answer type from a string to a function. If `k` in `%` followed the rule of CPS and was applied in a tail position, the result would have had the same type as the answer type of `k`; since the initial continuation is the identity function on `string`, the answer type polymorphism would then have ensured that the result be of type `string`. However, by not applying `k` at a tail position but embedding `k` into a function for the later use, what is returned as a result became a function.

This is also clear from the type of `%`:

```
(*  % : ('b -> string) -> (string -> 'a) -> string -> 'b -> 'a  *)
let % to_str = fun k -> fun s -> fun x -> k (s ^ to_str x)
```

Although the answer type of `k` is `'a` (instantiated to `string` in the above example), the final answer type after receiving the string accumulator is `'b -> 'a` where `'b` is dependent on the type of the first argument `to_str`.

The above story is not constrained to the case where `'a` is `string`. Multiple uses of `%` change the return type further to accept yet another argument:

[4] To be more precise, the continuation obtained by evaluating `lit "!" (eol id)` is suspended, *i.e.*, `fun s -> (fun s -> id (s ^ "\n")) (s ^ "!")`.

```
# sprintf (lit "The value of " ++ % str ++ lit " is " ++ % int ++
           eol);;
- : string -> int -> string = <fun>
# sprintf (lit "The value of " ++ % str ++ lit " is " ++ % int ++
           eol) "x" 3;;
- : string = "The value of x is 3\n"
#
```

Every time `%` is used, the number of arguments of the returned function increases.

5 Back to Direct Style

Although elegant, the continuation-based solution suffers from the common problem of CPS programs: every function has to pass a continuation. Since the cleverness of the solution is encoded into continuations that are scattered all over the functions, it may give the wrong impression that passing continuations is the essence of the solution.

A closer look at the continuation-based solution, however, reveals that the only non-standard use of continuations occurs in the definition of `%`. In all the other functions, continuations are simply passed around without any effect.

The situation is the same for our `times` example. By writing it in CPS (as in `times2`) and then changing it a little bit (`times3`), we achieved non-local jump to avoid useless multiplication. However, the true point of `times3` is not in writing the program in CPS (which makes the program less readable) but in the discarding of continuations. In such a case, we usually want to stay in a direct-style program to keep the clarity of the program and use a special language mechanism, such as exception handling, to achieve non-local jump:

```
exception Zero

(*  times4 : int list -> int  *)
let rec times4 lst = match lst with
    [] -> 1
  | 0 :: rest -> raise Zero
  | n :: rest -> n * times4 rest

(*  main4 : int list -> int  *)
let main4 lst = try times4 lst with Zero -> 0
```

In this program, discarding of continuations is represented as throwing an exception. All other places are written naturally in direct style. With exception handling, we obtained both: we avoided converting a program globally into CPS and sacrificing clarity, and yet we achieved non-local jump (with the cost of an additional language mechanism). The question is: can we do the same for `sprintf`?

Although exception handling is a useful tool to discard a part of computation, it is not sufficient to simulate all the control effects CPS achieves. In particular, it is impossible to write `%`. To cover more complicated uses of continuations than discarding, we introduce continuation-manipulation operations, `shift` and `reset`, due to Danvy and Filinski [5]. Readers familiar with these operations can safely proceed to the next section.

Intuitively, `shift` is like `call-with-current-continuation` in Scheme and captures the current continuation, but unlike `call-with-current-continuation`, the continuation is captured only up to its enclosing `reset`— the captured continuation is delimited.

To understand `shift` and `reset` in a more rigorous way, it is best to make correspondence between a direct-style program using `shift` and `reset` and a CPS program with non-standard uses of continuations. There exists a one-to-one correspondence between them, and it is always possible to mechanically transform the former to the latter [6]. Thus, one can regard the former as a useful surface language with the ability to manipulate continuations and the latter as the implementation of the former.

The precise correspondence of the two operations is as follows. Capturing of continuations and using them in a non-standard way is done in direct style by:

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

where one can use the current continuation `k` in *`body`* in an arbitrary way. The corresponding CPS program (where the continuation argument is named `k`) is *`body'`* `(fun r -> r)`, where *`body'`* is the CPS transform of *`body`*.

For example, consider:

```
(*  times5 : int list -> int  *)
let rec times5 lst = match lst with
     [] -> 1
   | 0 :: rest -> shift (fun k -> 0)
   | n :: rest -> n * times5 rest
```

This program is the direct-style counterpart of `times3`. When this program is transformed into CPS, we obtain `times3`. In particular, `shift (fun k -> 0)` in `times5` corresponds to 0[5] in `times3`. In both cases, the captured continuation `k` is discarded.

To execute `times5`, we need to reset its continuation to the initial one, just the same as we needed to supply the initial continuation to `times3` in `main3`. Resetting the current continuation is done in direct style by:

```
reset (fun () -> body)
```

(Because the continuation should be reset before *`body`* is evaluated, *`body`* is wrapped in a thunk. This syntax is consistent with Filinski's implementation of `shift` and `reset` mentioned below.) The CPS program corresponding to the above expression is *`body'`* `(fun r -> r)`, where *`body'`* is the CPS transform of *`body`*. For example, `times5` is executed as follows:

```
(*  main5 : int list -> int  *)
let main5 lst = reset (fun () -> times5 lst)
```

There are several ways to actually run a program with `shift` and `reset` in OCaml (and other languages). First, one can always transform it into CPS, which is an ordinary OCaml program and runs on standard OCaml. Second, one can easily write a CPS interpreter that supports `shift` and `reset`. Third, if the host language supports `call-with-current-continuation` and a mutable cell as in Scheme and SML/NJ, there is a way to simulate `shift` and `reset` in terms of them [8]. Finally, Kiselyov implements the delimcc library[6] for OCaml, which includes direct implementation of `shift` and `reset`. Using one of these methods, one can execute `times5`:

[5]To be more precise, `(fun k1 -> k1 0) (fun r -> r)`, because the CPS transform of `0` is `fun k1 -> k1 0`.

[6]Available from `http://okmij.org/ftp/Computation/Continuations.html#caml-shift`.

```
# reset (fun () -> times5 [1; 2; 3; 4]);;
- : int = 24
#
```

6 Solution in Direct Style

Equipped with delimited-control operators, `shift` and `reset`, we can now transform the continuation-based solution into direct style.[7] Since the only non-standard use of continuations is in `%`, it is straightforward to transform functions other than `%`:

```
(*  lit : string -> string -> string  *)
let lit x = fun s -> s ^ x

(*  eol : string -> string  *)
let eol = fun s -> s ^ "\n"
```

The definitions of `int` and `str` remain the same. The definition of `++` becomes the direct-style counterpart of the η-expanded definition of continuation-based `++`:

```
(*  (++) : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c  *)
let (++) f1 f2 = fun s -> f2 (f1 s)
```

Note that `(++) f1 f2` now receives `s` (a string) rather than a continuation, and the order of function application is reversed. This order correctly reflects the order of evaluation in the η-expanded definition of continuation-based `++`.

The interesting part is the definition of `%`. Omitting the typing information for now, it is defined as follows:

```
let % to_str = fun s -> shift (fun k -> fun x -> k (s ^ to_str x))
```

The correspondence to the continuation-based solution:

```
(*  % : ('b -> string) -> (string -> 'a) -> string -> 'b -> 'a  *)
let % to_str = fun k -> fun s -> fun x -> k (s ^ to_str x)
```

is clear. Except for the order of arguments, `k` and `s`, the direct-style solution is obtained by simply replacing `fun k -> ...` with `shift (fun k -> ...)`.

Even though the definition of `%` is now written in direct style with `shift`, the essence of the solution is the same as before: the non-standard use of the captured continuation. As in the continuation-based solution, the continuation `k` is not applied in tail position, but within a returned function. This function accepts an additional argument and then continues the rest of the computation captured in `k`.

Finally, we need to supply initial continuation to execute it:

```
let sprintf p = reset (fun () -> p "")
```

Again, the correspondence to the continuation-based solution is clear:

[7] This transformation can be done mechanically to some extent [10]. Since the resulting code typically contains many redundant `shift`, however, we performed the transformation by hand.

```
(*  sprintf : ((string -> string) -> string -> 'a) -> 'a  *)
let sprintf p = p (fun (s : string) -> s) ""
```

Supplying the initial continuation is transformed into resetting the context.

We could now write:

```
sprintf (lit "Hello world!" ++ eol)
sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol) "world"
sprintf (lit "The value of " ++ % str ++ lit " is " ++ % int ++ eol)
        "x" 3
```

to obtain `"Hello world!\n"` for the first two and `"The value of x is 3\n"` for the last. For those who want to see how capturing continuations actually goes, here is the syntactic trace for the second invocation:[8]

```
   sprintf (lit "Hello " ++ % str ++ lit "!" ++ eol)
→  reset (fun () -> (lit "Hello " ++ % str ++ lit "!" ++ eol) "")
→  reset (fun () -> eol (lit "!" (% str (lit "Hello " ""))))
→  reset (fun () -> eol (lit "!" (% str "Hello ")))
→  fun x -> reset (fun () -> eol (lit "!" ("Hello " ^ str x)))
```

At the last step, the captured continuation is `reset (fun () -> eol (lit "!" □))`. The continuation (or the context) when `%` was executed up to the enclosing `reset` is captured by `shift` in `%`. In the continuation-based solution, the captured continuation was actually a function. In the direct-style solution, it is a context with a hole □. Still, the correspondence is maintained: by CPS transforming the context `fun s -> reset (fun () -> eol (lit "!" s))`, we obtain the captured function in the continuation-based solution.

The above program actually runs in untyped languages such as Scheme. In a typed language, however, a typing issue arises. We address it in the following two sections.

7 Typing the Direct-Style Solution

We now see how functions in the direct-style solution could be typed. The types of `int` and `str` are already shown in Section 4 and remain the same. The types of `eol`, `lit`, and `++` for the direct-style solution are presented in Section 6.

For functions without non-standard uses of continuations, there is an exact correspondence between types of a direct-style program and a CPS program: a function of type `S -> T` in a direct-style program corresponds to `(T -> `*`Ans`*`) -> S -> `*`Ans`* in a CPS program where *`Ans`* is the answer type. For example, `lit x` in the direct-style solution has type `string -> string` while its CPS counterpart has type `(string -> 'a) -> string -> 'a`. Because of the answer type polymorphism, we do not have to mention the answer type in the direct-style solution, and hence we could transform the latter into the former without losing any information.

Now, what about `%`?

```
let % to_str = fun s -> shift (fun k -> fun x -> k (s ^ to_str x))
```

[8] The reduction in the presence of `shift` and `reset` is formally defined in [1]. Here, it is enough to know that the body of the thunk passed to `reset` is evaluated under the delimited context, and `shift` within the thunk captures the continuation up to this `reset`.

To type `%`, we first observe that `k` has type `string -> 'a` for some `'a` representing the answer type. This type is determined by the context in which `%` is used. Looking at the trace shown in the previous section, for example, the captured continuation was `reset (fun () -> eol (lit "!" □))`. The type of the box where `% str "Hello "` appeared is `string`, because it is passed to the second argument of `lit`. Thus, given a string, this continuation would produce a value of type `'a`. (In this particular case, `'a` is `string`, but it could be a function type if `%` is used again.)

Knowing that `k` has type `string -> 'a`, the type of the body of `shift` (that is, `fun x -> k (s ^ to_str x)`) becomes `'b -> 'a` where `'b` is the type of `x`. Now, what is the type of the final answer? Looking at the trace again, the final answer becomes the body of `shift`:

```
fun x -> reset (fun () -> eol (lit "!" ("Hello " ^ str x)))
```

and it has type `'b -> 'a` (or in this particular case, `string -> string`).

To summarize, before the continuation was grabbed, we expected that the result would have type `'a` (since `k` has type `string -> 'a`), but what we actually obtained as the result of execution has type `'b -> 'a`. Because of `shift`, *the answer type changed.*

We could anticipate this conflict of the answer type. The type of continuation-based `% to_str` was `(string -> 'a) -> string -> 'b -> 'a`. Comparing this type to the type of CPS functions `(T ->` *Ans*`) -> S ->` *Ans*, we could easily see that the answer type changed from `'a` to `'b -> 'a`. If we blindly transform the type of `% to_str` into its direct-style counterpart `string -> string`, we lose information on the answer type modification. To correctly type `%` in the direct-style solution, we need to keep track of the answer types.

Unfortunately, Filinski's implementation of `shift` and `reset` as well as Kiselyov's delimcc library do not support answer type modification. They require that the answer type be chosen first and fixed throughout afterwards. Although they could be used to execute `times5` because the answer type of `times5` is always `int` and does not change, they cannot be used to type the direct-style solution.

8 Danvy and Filinski's Type System

The problem of typing `sprintf` arises because we required the final answer type and the return type of continuations to be the same. Although this appears to be natural for the type system like the one for ML, the original type system for delimited continuations introduced by Danvy and Filinski [4] does not have this restriction but allows any expression to alter them. It uses a function type of the form `S / A -> T / B`, which is the type of a function from `S` to `T` that changes the answer type from `A` to `B` when applied. This type corresponds to the CPS type `(T -> A) -> S -> B` (if the continuation is placed before the argument). Because the answer types are mentioned in the type, this new form of a function type can type check any function whose CPS counterpart is typable.

The conventional function type `S -> T` in the type system is considered as polymorphic in the answer type: `S / 'p -> T / 'p` for a new type variable `'p`. When transformed into CPS, it corresponds to `(T -> 'p) -> S -> 'p`, recovering the answer type polymorphism. Since it does not change the answer type, it is called control-effect free or *pure* [14].

Although we do not repeat Danvy and Filinski's type system here, it is simple to infer types of functions in their type system: a function has type `S / A -> T / B` if and only if its CPS counterpart has type `(T -> A) -> S -> B` in the Hindley-Milner type system. For example, because `%` in the continuation-based solution has type `('b -> string) -> (string -> 'a) -> string ->`

`'b -> 'a`, we obtain the following type for `%` in the direct-style solution by mechanically converting the CPS part of this type:

```
(*  % : ('b -> string) -> (string / 'a -> string / ('b -> 'a))  *)
let % to_str = fun s -> shift (fun k -> fun x -> k (s ^ to_str x))
```

Given `to_str` of type `'b -> string`, it produces a function of type `string / 'a -> string / ('b -> 'a)`. This type indicates that it receives a string and produces a string. During this process, however, it alters the answer type from `'a` to `'b -> 'a`, allowing to accept another argument of type `'b`.

Notice how the above type exactly describes the behavior of `%`. The simpler type `string -> string` that ignores the answer types represents only a local behavior of `%`. The true point of `%` is to change the type of its context. By changing the type of the context from `'a` to `'b -> 'a`, additional argument becomes acceptable. Danvy and Filinski's type system enables us to type check direct-style programs without losing any type information that are available in their CPS counterpart.

Kiselyov has implemented this type system[9] (together with `sprintf` as a test case) in Haskell using parameterized monads. With this implementation, one can confirm that the type of `%` is actually inferred as above.

For completeness, we show the type of `sprintf`:

```
(*  sprintf : (string / string -> string / 'a) -> 'a  *)
let sprintf p = reset (fun () -> p "")
```

The type of `sprintf` is interesting. It receives a pattern of the type `string / string -> string / 'a`. It is a function from `string` to `string`, but it changes the return type from `string` to an arbitrary `'a`. Then, `sprintf` *will return a value of this* `'a`. In other words, the return type of `sprintf` is determined by, or dependent on, the final answer type of its argument. This is how the dependent behavior of printf is realized from the perspective of types.

Notice that the function `sprintf` itself does not change the answer type. This is observed by the fact that the outermost `->` is pure. Even though the pattern contains `%` and manipulates its continuations, control effects do not leak out of `sprintf`. Thus, we can consider `sprintf` as a pure function without any control effects.

9 Another Continuation-Based Solution

Looking at the two solutions obtained so far, it becomes clear that they are both written in an accumulator-passing style: by encoding pattern directives as string transformers, all the pattern directives receive a string as an accumulator and add a string to it. However, since the dependent behavior of printf is realized by the non-standard use of continuations, we do not really have to write a solution in accumulator-passing style. In this section, we show another continuation-based solution according to this observation.

Without an accumulator, the CPS version of `lit x` and `eol` are defined as follows:

```
(*  lit : string -> (string -> 'a) -> 'a  *)
let lit x = fun k -> k x
```

[9] Available from `http://okmij.org/ftp/Computation/Continuations.html#genuine-shift`.

```
(*  eol : (string -> 'a) -> 'a  *)
let eol = fun k -> k "\n"
```

Since they receive no longer a string accumulator, we cannot reuse function composition for `++` but use the CPS transform of the string concatenation operator `^`:

```
(*  (++) : ((string -> 'b) -> 'a) ->
           ((string -> 'c) -> 'b) ->
           ((string -> 'c) -> 'a)  *)
let (++) f1 f2 = fun k -> f1 (fun x -> f2 (fun y -> k (x ^ y)))
```

Then, `%` and `sprintf` can be defined as follows:

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

(*  sprintf : ((string -> string) -> 'a) -> 'a  *)
let sprintf p = p (fun (s : string) -> s)
```

The technique used here is again the same: by wrapping `k (to_str x)` with a function, the answer type is changed from `'a` to `'b -> 'a`. This behavior is obtainable without an accumulator.

10 Another Solution in Direct Style

By analyzing Danvy's solution, we have seen that it is continuation-based and uses an accumulator. In Section 6, we have removed continuation-passing to obtain a solution in direct style with an accumulator. In the previous section, we have removed an accumulator to obtain the second continuation-based solution. A natural question to ask, then, is what happens if we remove them both.

In this section, we present a final solution that is written in direct style without using an accumulator. Since it uses neither continuation- nor accumulator-passing, we do not have to encode pattern directives at all. As a result, the solution becomes particularly simple and natural. Here are the definitions of `lit` and `eol`:

```
(*  lit : string -> string  *)
let lit x = x

(*  eol : string  *)
let eol = "\n"
```

In fact, we do not need to use `lit` but a raw string suffices. The same goes for `++`:

```
(*  (++) : string -> string -> string  *)
let (++) s1 s2 = s1 ^ s2
```

We can use `^` directly instead of `++`. Thus, if `%` was not used at all in the pattern, no overhead is required: we don't have to pay for what we don't use. This is in contrast to the previous three solutions, where we had to encode pattern directives into an appropriate form regardless of whether `%` is used or not.

The definitions of `int` and `str` remain the same. The essence of the solution is again in the definition of `%`, where the answer type changes from `'a` to `'b -> 'a`:

```
(*  % : ('b -> string) / 'a -> string / ('b -> 'a)  *)
let % to_str = shift (fun k -> fun x -> k (to_str x))
```

It is easy to see that this definition is the direct-style counterpart of the second continuation-based solution, including its type. By now, the reader can easily figure out its behavior without any further explanation. We only remark that since the answer type changes from `'a` to `'b -> 'a`, `%` cannot be given a conventional ML type.

The solution now has `shift` as a top construct of `% to_str`, and therefore requires us to delimit the context at the outset of `sprintf` before `% to_str` is executed. One way to do this is to wrap the pattern in a thunk:

```
sprintf (fun () -> "Hello world!" ^ eol)
sprintf (fun () -> "Hello " ^ % str ^ "!" ^ eol) "world"
sprintf (fun () -> "The value of " ^ % str ^ " is " ^ % int ^ eol)
        "x" 3
```

We can then define `sprintf` as follows:

```
(*  sprintf : (unit / string -> string / 'a) -> 'a  *)
let sprintf thunk = reset thunk
```

Alternatively, we could define `sprintf p` as a macro for `reset (fun () -> p)`.

Once again, note that the type describes the behavior of `sprintf` accurately: `sprintf` receives a thunk that modifies the answer type to an arbitrary `'a`; the final type depends on this answer type; and `sprintf` itself is pure since the outer `->` is pure.

As a whole, this solution exactly pinpoints the essence of Danvy's solution. Apart from `int` and `str`, it consists of only two lines of code: one for `sprintf` to reset the context and one for `%` to modify the answer type and achieve the dependent behavior of printf through the non-standard use of continuations.

11 Related Work

Hinze [9] presented a general solution to the printf problem using functors together with a function definition that depends on types. In his solution, the type of patterns is decomposed into a functor applied to `string`:

```
(*  lit : string -> Id string  *)
let lit s = s

(*  % : ('a -> string) -> ('a ->) string  *)
let % to_str = to_str
```

Here, `Id` is an identity functor mapping its argument to itself. The type `('a ->) string` in `%` is the same as `'a -> string`, but written explicitly as a functor `('a ->)` (that accepts a type to the second argument of `->`) and an argument `string`.

Then, Hinze defined `(++)` as a function that depends on types, as informally shown in the following psuedo code:

```
(*  (++) : F string -> G string -> (F . G) string  *)
let (++) d1 d2 = match (F, G) with  (* dispatch over types *)
    (Id, Id) -> d1 ^ d2
  | (Id, _ ) -> (fun s -> d1 ^ s) . d2
  | (_ , Id) -> (fun s -> s ^ d2) . d1
  | (_ , _ ) -> (fun s1 -> ((fun s2 -> s1 ^ s2) . d2)) . d1
```

Observe the type of `(++)` (where `(F . G)` is a (functor) composition).[10] When `F` and `G` are both `Id` (*i.e.*, the two arguments of `(++)` are both string literals), the type of the result `(F . G) string` is `Id (Id string)` = `string`. But if, for example, `F` has the form `('a ->)`, the type of the result becomes `(F . G) string` = `('a ->) (G string)` = `'a -> G string`. That is, it accepts an argument of type `'a`. By representing the type of patterns using functors, he achieved the dependent behavior by functor composition.

Hinze showed how to encode this type-dependent function in Haskell using type classes. He further abstracted the type `F string` to other types and showed how Danvy's solution can be encoded into his solution.

The solutions presented in this article, developed independently of Hinze's work, show an operational interpretation of the functor composition: the functor `Id` means the type of the context remains the same; the functor `('a ->)` means the type of the context changes from `'b` to `'a -> 'b`; and composing these functors means the type of the context is modified successively according to the component functors. Thanks to this operational interpretation, our solutions do not depend on how they are typed. To type the solutions, however, requires the original type system for `shift` and `reset`.

12 Conclusion

This article presented four solutions to the printf problem: two continuation-based solutions and two direct-style solutions, each using or not using an accumulator. These solutions show that the dependent behavior of printf is realized by the modification of the answer type.

The two direct-style solutions provide new examples of using delimited-control operators. The use of `shift` and `reset` here appears to be natural because the desired behavior for the directives is to change the behavior of its context, not the output string itself.

The solutions in direct style raised an interesting issue about typing programs with `shift` and `reset`. They are not typable in the existing ML type system but require Danvy and Filinski's original type system. Although how to incorporate it into the existing ML type system remains to be seen, the type system seems to provide an accurate abstraction of control effects when these are compatible with ordinary continuation-passing style. In fact, we have seen that the type of `%` accounts for its behavior, much in the same way as conventional types account for the behavior of pure functions.

Acknowledgements

I would like to thank Olivier Danvy, Yukiyoshi Kameyama, Oleg Kiselyov, and Ken Shan for many comments and encouragements. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (C) 18500005.

[10] We assume here that `F` and `G` have the form `('a ->)` for some `'a`, when they are not `Id`.

References

[1] K. Asai and Y. Kameyama. Polymorphic Delimited Continuations. In *5th Asian Symposium on Programming Languages and Systems, APLAS 2007, Lecture Notes in Computer Science 4807*, pages 239–254. Springer-Verlag, November 2007.

[2] O. Danvy. Back to Direct Style. *Science of Computer Programming*, 22(3):183–195, 1994.

[3] O. Danvy. Functional Unparsing. *Journal of Functional Programming*, 8(6):621–625, November 1998.

[4] O. Danvy and A. Filinski. A Functional Abstraction of Typed Contexts. Technical Report 89/12, DIKU, University of Copenhagen, July 1989.

[5] O. Danvy and A. Filinski. Abstracting Control. In *LFP '90: Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 151–160, New York, June 1990. ACM Press.

[6] O. Danvy and A. Filinski. Representing Control, a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[7] O. Danvy and J. Hatcliff. On the Transformation between Direct and Continuation Semantics. In *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science 802*, pages 627–648. Springer-Verlag, April 1993.

[8] A. Filinski. Representing Monads. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 446–457, New York, January 1994. ACM Press.

[9] R. Hinze. Formatting: A Class Act. *Journal of Functional Programming*, 13(5):935–944, September 2003.

[10] Y. Kameyama and M. Hasegawa. A Sound and Complete Axiomatization of Delimited Continuations. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 177–188, New York, August 2003. ACM Press.

[11] G. D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[12] G. L. Steele. RABBIT: A Compiler for SCHEME. Technical Report AI-TR-474, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, May 1978.

[13] C. Strachey and C. P. Wadsworth. Continuations: A Mathematical Semantics for Handling Full Jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England, July 1974. Reprinted in Higher-Order and Symbolic Computation 13 (1/2):135–152, 2000.

[14] H. Thielecke. From Control Effects to Typed Continuation Passing. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 139–149, New York, January 2003. ACM Press.

[15] H. Thielecke. Answer Type Polymorphism in Call-by-Name Continuation Passing. In *13th European Symposium on Programming, ESOP 2004, Lecture Notes in Computer Science 2986*, pages 279–293. Springer-Verlag, March 2004.