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.

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 a printf-like function, which 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 [2] showed that printf can be expressed within the ML type system, without using dependent types. This example has proved popular. For example: Yang [10] showed how to encode types in ML-like languages; Fridlender and Indrika [6] demonstrated how to define a general *zipWith* function in Haskell; Xi, Chen, and Chen [9] showed an implementation based on guarded recursive datatype constructors; Hinze [7] presented how to define generic functions on types within the Haskell type system; and Oliveira and Gibbons [1] generalized these techniques as a design pattern.

The key technique employed in Danvy's solution is the non-standard use of continuations. He achieves the dependent behavior of printf by modifying the return type of continuations according to the given pattern directives. Since the technique mentions continuations, the solution requires to be written either by passing continuations or with first-class continuation constructs. Danvy's original solution takes the first approach. The present article explores both approaches: we show another continuation-based solution and two more solutions in direct style with delimited continuation

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 printf.

The direct-style solutions raise an interesting issue in the typing of delimited continuation constructs. They do not type check if we use Filinski's implementation [5] of `shift` and `reset`. We show that they are typable with the original type system by Danvy and Filinski [3], and that their type precisely accounts for the dependent behavior of printf. The type system therefore provides an accurate abstraction of the behavior of printf and of its control effects.

After describing the printf problem in the next section, we review Danvy's solution in Section 3. We then present the first direct-style solution in Section 4. Section 5 demonstrates that it does not type check with Filinski's implementation of delimited continuations in ML, and Section 6 shows that it does with Danvy and Filinski's original type system. In Section 7, we simplify Danvy's solution by eliminating the accumulator to obtain another continuation-based solution. Applying the same simplification to the direct-style solution yields the second and extremely simple solution in direct style in Section 8. The article concludes in Section 9. The four solutions in the paper are summarized in Table 1.

Table 1: Four Solutions to the Printf Problem

	Accumulator	No Accumulator
Continuation-based	Section 3	Section 7
Direct Style with `shift` and `reset`	Section 4	Section 8

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
  - `str` for strings.

The goal of the printf 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`,

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

should have type `string -> string`, and

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

should have type `string -> int -> string`. The return type of `sprintf` therefore depends on the given pattern.

3 Continuation-based Solution

Danvy's continuation-based solution [2] 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 Danvy's solution re-written in OCaml:

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

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

`lit x` and `eol` are not encoded as strings but as string transformers written in continuation-passing style (CPS). They both receive a continuation `k` and a string `s` and pass their results to `k`. Accordingly, two patterns cannot be connected using a string concatenation operator `^`, but are connected using function composition (*i.e.*, not in CPS):

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

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 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:[1]

[1]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.

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

Given a field type `to_str` (either `int` or `str`), a continuation `k`, and a string `s`, `%` does *not* return a string but a function that receives an additional argument `x`. It then concatenates its string representation to `s` and passes the result to `k`. Notice that the above definition of `%` does not conform to the proper continuation-passing style. The continuation `k` is not applied in tail position, but within a function. It is this non-standard use of continuations that changes the return type of `sprintf`. The construction of the output string for all the directives finishes once the string `(s ^ to_str x)` is passed to `k`. By wrapping it into an abstraction `fun x ->`, the return value of `sprintf` changes from a string to a function, thus making it possible to accept another argument.

Using `%`, we have:

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

Multiple uses of `%` change the return type further to accept yet another argument:

```
# 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"
#
```

4 Solution in Direct Style

Although elegant, the continuation-based solution suffers from the common problem of CPS programs: every function has to pass continuations. Since the cleverness of the solution is encoded into continuations that are scattered around 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 `%`. This calls for the use of continuation-manipulation operations: they would get rid of continuations altogether except for `%`, thus highlighting the essence of the solution. Here, we use `shift` and `reset`, which were introduced by Danvy and Filinski [4], to manipulate delimited continuations in a way compatible with continuation-passing style: `shift` is like `call-with-current-continuation` in Scheme (or `callcc` in ML) but it captures the continuation only up to its enclosing `reset`.

Let us write `sprintf` in direct style now. We start from `lit` and `eol`. They are string transformers as before, but now written without continuations:

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

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

The definitions of `int` and `str` as well as `++` remain the same. The interesting part is the definition of `%`. Omitting the typing information for now, it is defined as follows:

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

Given a field type `to_str` (either `int` or `str` as before) and a string `s`, `%` captures its current continuation, binds it to `k`, and returns a function. When the function is applied to an argument of the designated type, it concatenates the string representation of that argument to `s` and passes the result to `k`. Observe this non-standard use of the captured continuation again. As in the continuation-based solution, the continuation `k` is not applied in tail position, but within a function. The construction of the output string finishes once the string `(s ^ to_str x)` is passed to `k`. By wrapping it into an abstraction `fun x ->`, the type of whole the expression changes from a string to a function, making it possible to accept another argument.

It is easy to see the correspondence between the previous continuation-based solution and this solution: CPS-transforming the latter yields the former. This new solution shows clearly that continuation passing is not necessary to achieve the dependent behavior of printf. Instead, it exemplifies the non-standard use of continuations that modifies the answer type, which is the crucial point of the solution.

Since `%` uses `shift`, we need to delimit the continuation when we invoke `sprintf`:

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

It corresponds to passing the initial continuation in Danvy's solution.

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.

The above program actually runs if we ignore the typing issue. In fact, the author has implemented it in Scheme and confirmed that the expected results are obtained. However, it turns out that the above program does not type check if we use the available implementation [5] of `shift` and `reset` using `callcc` in ML. We address this typing issue in the following two sections.

5 Typing the Direct-Style Solution

We now see how functions in the direct-style solution could be typed. The types of `int`, `str`, and `++` are already shown in Section 3 and remain the same. The types of `eol` and `lit` for the direct-style solution are presented in Section 4.

Now, what about `%`?

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

To type `%`, we first observe that `k` has a type `string -> 'a` for some `'a` representing the answer type. Since `% to_str` is a string transformer and thus `% to_str s` is expected to be a string, `string -> 'a` is the right type for its continuation. Knowing that `k` has this type, 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? Before the continuation was grabbed, it was assumed

to be `'a`. After the capture, however, it became `'b -> 'a`. In other words, *the final answer type changed*.

This is where a type conflict occurs if we use Filinski's implementation of `shift` and `reset` in terms of `callcc`. This implementation requires that the type of the final answer be chosen first and fixed throughout afterwards.

6 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 [3] 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 final answer type from `A` to `B` when applied. 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`. Since it does not change the answer type, it is called control-effect free or *pure* [8].

Using this type system, the definition of `%` does type check as follows:

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

Given `to_str` of type `'b -> string`, it produces a string transformer of type `string / 'a -> string / ('b -> 'a)`. This type indicates that the string transformer 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 `%`.

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.

7 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 k = k x

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

Since they are no longer string transformers, we cannot reuse function composition but have to define `++` as a CPS function that concatenates two strings:

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

Then, `%` can be defined as follows:

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

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.

Finally, `sprintf` is defined as a function that passes the initial continuation to the given pattern directive:

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

8 Another Solution in Direct Style

By analyzing Danvy's solution, we have seen that it is continuation-based and uses an accumulator. In Section 4, 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 `++`. This is in contrast to the previous three solutions, where we had to encode pattern directives into an appropriate form.

The definitions of `int` and `str` remain the same. The essence of the solution is again in the definition of `%`, where the final 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 final answer type changes from `'a` to `'b -> 'a`, `%` cannot be given a conventional ML type.

As a whole, this solution exactly pinpoints the essence of Danvy's solution. All the functions except for `%` are elementary. The dependent behavior of printf is solely achieved by the non-standard use of continuations in `%` that modifies the answer 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 p = reset p
```

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.

9 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 continuation constructs. 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 is yet 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 for many valuable comments, suggestions, and encouragements.

References

[1] Bruno C. d. S. Oliveira and Jeremy Gibbons. TypeCase: a Design Pattern for Type-Indexed Functions. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, pages 98–109, New York, September 2005. ACM Press.

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

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

[4] Olivier Danvy and Andrzej 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.

[5] Andrzej 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.

[6] Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.

[7] Ralf Hinze. Generics for the Masses. In *ICFP '04: Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 236–243, New York, September 2004. ACM Press.

[8] Hayo 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.

[9] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded Recursive Datatype Constructors. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 224–235, New York, January 2003. ACM Press.

[10] Zhe Yang. Encoding Types in ML-like Languages. *Theoretical Computer Science*, 315(1):151–190, 2004.