J. Functional Programming **8** (6): 621–625, November 1998. Printed in the United Kingdom
© 1998 Cambridge University Press

Functional unparsing

OLIVIER DANVY
BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark
(*e-mail:* danvy@brics.dk)

Abstract

A string-formatting function such as printf in C seemingly requires dependent types, because its control string determines the rest of its arguments. Examples:

```
printf ("Hello world.\n");
printf ("The %s is %d.\n", "answer", 42);
```

We show how changing the representation of the control string makes it possible to program printf in ML (which does not allow dependent types). The result is well typed and perceptibly more efficient than the corresponding library functions in Standard ML of New Jersey and in Caml.

1 The problem

In ML, expressing a printf-like function is not as trivial as in C. For example, we would like that evaluating the expression

```
format "%i is %s%n" 3 "x"
```

yields the string `"3 is x\n"`, as specified by the pattern `"%i is %s%n"`, which tells `format` to issue an integer, followed by the constant string `" is "`, itself followed by a string and ended by the newline character.

What is the type of `format`? In this example, it is

```
string -> int -> string -> string
```

but we would like our printf-like function to handle any kind of pattern. For example, we would like

```
format "%i/%i" 10 20
```

to yield `"10/20"`. In that example, `format` is used with the type

```
string -> int -> int -> string
```

However, we cannot do that in ML: `format` can only have one type.

2 Analysis

The crux of the problem is that the type of `format` *depends* upon the value of its first argument, i.e., the pattern. This has led, for example, Shields, Sheard and Peyton Jones (Shields *et al*., 1998) to propose a dynamic type system that makes it possible to express such a formatting function by delaying type inference until the pattern is available.

The culprit, however, is not ML's type system, but the fact that the pattern is represented as a string, which `format` in essence has to interpret (in the sense of a programming-language interpreter).

3 A solution

Let us pursue this programming-language analogy, i.e., that `format` interprets the pattern. Instead of considering the *concrete syntax* of each pattern – as a string, we can consider its *abstract syntax* – as a data type.

Abstract syntax of patterns: the data type of patterns is composed of the following pattern directives:

- `lit` for declaring literal strings (`" is "` and `"/"` above);
- `eol` for declaring newlines (`%n` above);
- `int` for specifying integers (`%i` above); and
- `str` for specifying strings (`%s` above).

In addition, we provide the user with an associative infix operator `oo` to glue pattern components together.

Cosmetics: for cosmetic value, we could also provide two 'outfix' directives << and >> to delimit a pattern.

We could also define the operator `%` to be the polymorphic identity function, so that, e.g., `%int` (or even `%i` for that matter) would be a valid pattern directive.

Two examples: thus equipped, we can make `format` construct an appropriate (statically typed) higher-order function, as in the following two examples.

```
format (int oo lit " is " oo str oo eol) : int -> string -> string
format (int oo lit "/" oo int) : int -> int -> string
```

The insights: rather than making format interpret the pattern recursively, we make the pattern construct an appropriate higher-order function inductively. In that, we follow Harry Mairson's observation that most of the time, our programs are inductive, not recursive (Mairson, 1991). More concretely, we use continuation-passing style (CPS) to thread the constructed string throughout. We also exploit the polymorphic domain of answers to instantiate it to the appropriately typed function. Formatting a string then boils down to supplying the initial continuation and the initial string.

For example, the type of the `eol` directive reads as follows:

```
(string -> 'a) -> string -> 'a
```

Its first argument is the continuation, which expects a string and yields the final answer. Its second argument is the threaded string, and because it is in CPS, this directive also yields the final answer.

For a second example, the type of the `int` directive reads as follows:

```
(string -> 'a) -> string -> int -> 'a
```

Its first argument is the continuation and its second argument is the threaded string. This directive yields a function expecting an integer and yielding the final answer.

The directives: `lit` and `eol` operate in a similar way:

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

As for `int` and `str`, they also operate in a similar way:

```
fun int k s (x:int) = k (s ^ (makestring x))
(* val int : (string -> 'a) -> string -> int -> 'a *)
fun str k s x = k (s ^ x)
(* val str : (string -> 'a) -> string -> string -> 'a *)
```

Note that one can uncurry the directives and also change the order of their parameters, but the present formulation yields the simplest definition of `oo`.

Glueing the directives: we can implement `oo`, for example, as function composition (`o` in ML). So glueing `int` together with itself, for example, yields a function of the following type:

```
int oo int : (string -> 'a) -> string -> int -> int -> 'a
```

Initializing the computation: the job of `format` reduces to providing an initial continuation and an initial string to trigger the computation specified by the pattern:

```
fun format p = p (fn (s:string) => s) ""
(* val format : ((string -> string) -> string -> 'a) -> 'a *)
```

So given the pattern `int oo int`, the `format` function supplies it with an initial continuation (the identity function over strings) and an initial string (the empty string), yielding a value of the following type, as desired:

```
int -> int -> string
```

4 An alternative solution

Alternatively, and given an end-of-pattern directive (implemented as the identity function), we can implement glueing as function application instead of as function composition. In both cases, the implementation of the directives remains the same, but the definition of `format` need no longer supply an initial continuation, since the initial continuation in effect is already provided by the end-of-pattern directive:

```
fun format' p = p ""
(* val format' : (string -> 'a) -> 'a *)

fun eod (s:string) = s
(* val eod = fn : string -> string *)
```

Therefore, glueing `int` together with itself and the end-of-pattern directive, for example, yields a function of the following type.

```
int oo int oo eod : string -> int -> int -> string
```

More on cosmetics: implementing glueing as function application makes it simple to implement the outfix directives << and >> mentioned in section 3. We can simply define each of them as the polymorphic identity function:

```
fun << x = x
fun >> x = x
```

And then we can write, for example, the following:

```
<< int oo int >> : string -> int -> int -> string
format' (<< int oo int >>) : int -> int -> string
```

5 Assessment

Formatting strings is a standard example in partial evaluation (Consel and Danvy, 1993): the formatting function can be specialized with respect to any given pattern. Partial evaluation then removes the overhead of interpreting each pattern. So, for example, specializing a term such as

```
format (int oo lit " is " oo str oo eol)
```

yields the following more efficient residual term:

```
fn (x1:int) => fn x2 => (makestring x1) ^ " is " ^ x2 ^ "\n"
```

The required partial-evaluation steps can be very mild: for the functional specification described here, mere inlining (β-reduction) suffices. The back end of the ML Kit, for example, provides the specialization just above (Martin Elsmann, personal communication, March 1998).

Independently of partial evaluation, the functional specification is also efficient on its own. For example, besides being type-safer, it appears to be perceptibly faster than the resident `format` function in the New Jersey library `Format`: it is three to four times faster if glueing is implemented as function composition. Ditto for the resident `sprintf` function in the Caml library: the functional specification is two to three times faster if glueing is implemented as function composition. In both cases, making function composition left- or right-associative has little influence on the overall efficiency. Finally, implementing glueing as (right-associative) function application gives another 10% speedup both in Standard ML of New Jersey and in Caml.

Independently of efficiency, this functional specification of `format` further illustrates the expressive power of ML, or for that matter of any functional language based on the Hindley–Milner static type system (Yang, 1998). It also easily scales up to inductive types such as lists (Danvy, 1998).

Acknowledgements

This work is supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

Thanks to Richard Bird for editorial advice.

References

Consel, C. and Danvy, O. (1993) Tutorial notes on partial evaluation. *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, Graham, S. L. (ed.), pp. 493–501. ACM Press.

Danvy, O. (1998) *Functional unparsing (extended version)*. Technical Report BRICS RS-98-12, Department of Computer Science, University of Aarhus, Denmark.

Mairson, H. (1991) Outline of a proof theory of parametricity. *Proc. 5th ACM Conference on Functional Programming and Computer Architecture*, Hughes, J. (ed.), pp. 313–327. *Lecture Notes in Computer Science 523*. Springer-Verlag.

Shields, M., Sheard, T. and Jones, S. P. (1998) Dynamic typing as staged type inference. *Proc. 25th Annual ACM Symposium on Principles Of Programming Languages*, Cardelli, L. (ed.), pp. 289–302. ACM Press.

Yang, Z. (1998) *Encoding types in ML-like languages (preliminary version)*. Technical Report BRICS RS-98-9. Department of Computer Science, University of Aarhus, Denmark.