BRICS

Basic Research in Computer Science

Formatting Strings in ML

Olivier Danvy

BRICS Report Series RS-98-5
ISSN 0909-0878 March 1998

BRICS RS-98-5 O. Danvy: Formatting Strings in ML

Matematisk Institut
Aarhus Universitet
Trykkeriet

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/98/5/

Formatting Strings in ML

Olivier Danvy
BRICS *
Department of Computer Science
University of Aarhus †

November 1997 (revised in March 1998)

Formatting strings is a standard example in partial evaluation [1]. Indeed, the format is usually specified with a constant "control string," with respect to which the formatting function can be specialized. In this case, partial evaluation removes the overhead of interpreting the control string.

In ML, expressing a printf-like function is not completely trivial. 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 control string `"%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.

*Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
Phone: (+45) 89 42 33 69. Fax: (+45) 89 42 32 55. E-mail: danvy@brics.dk

The crux of the problem is that the type of `format` *depends* on the value of its first argument, i.e., the control string. This has led, for example, Shields, Sheard, and Peyton Jones to propose an extended typing system that makes it possible to express such a formatting function [2].

The culprit, however, is not necessarily ML's typing system: one could say that it is rather the control string, which `format` in essence has to interpret (in the sense of a programming-language interpreter). So rather than representing it as a string, let us represent it as a data type with pattern constructors, namely:

- `i_` for specifying integers (%i above);
- `s_` for specifying strings (%s above);
- `l_` for declaring literal strings (`" is "` and `"/"` above); and
- `n_` for declaring newlines (%n above).

In addition, we provide the user with an associative infix constructor `oo` to construct a complete pattern out of pattern components.[1]

Thus equipped, we can write, e.g.,

```
- format (i_ oo l_ " is " oo s_ oo n_) 3 "foo";
val it = "3 is foo\n" : string
-
```

How does `format` work? By constructing an appropriate (statically typed) higher-order function:

```
format (i_ oo l_ " is " oo s_ oo n_) : int -> string -> string
format (i_ oo l_ "/" oo i_) : int -> int -> string
```

We define the pattern constructors in continuation-passing style, threading the constructed string and with a polymorphic domain of answers. This makes it possible to implement `oo`, e.g., as function composition (`o` in ML).

- `i_` and `s_` work in a similar way:

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

[1] For cosmetic value, we could also provide two "outfix" constructors `<<` and `>>` to delimit a pattern.

So for example, the type of the expression `i_ oo s_ oo i_` reads as follows.

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

The corresponding expression expects a continuation and a string, and returns a function of type `int -> string -> int -> 'a` that matches the "control string" `i_ oo s_ oo i_`.

- `l_` and `n_` work in a similar way:

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

As for `format`, its job reduces to providing an initial continuation and an initial string to trigger the computation specified by the pattern:

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

These definitions are not only interesting from the point of view of the expressive power of ML — they are also perceptibly faster than, e.g., the resident `format` in the New Jersey library `Format` (about 7 times) and the resident `sprintf` function in the Caml library (about 3 times).

Getting back to partial evaluation, specializing a term such as

```
format (i_ oo l_ " is " oo s_ oo n_)
```

yields, as could be expected, the following more efficient residual term.

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

References

[1] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, January 1993. ACM Press.

[2] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 289–302, January 1998. ACM Press.

Recent BRICS Report Series Publications

RS-98-5 Olivier Danvy. *Formatting Strings in ML*. March 1998. 3 pp.

RS-98-4 Mogens Nielsen and Thomas S. Hune. *Deciding Timed Bisimulation through Open Maps*. February 1998.

RS-98-3 Christian N. S. Pedersen, Rune B. Lyngsø, and Jotun Hein. *Comparison of Coding DNA*. January 1998. 20 pp.

RS-98-2 Olivier Danvy. *An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*. January 1998.

RS-98-1 Olivier Danvy. *A Simple Solution to Type Specialization (Extended Abstract)*. January 1998. 7 pp.

RS-97-53 Olivier Danvy. *Online Type-Directed Partial Evaluation*. December 1997. 31 pp. Extended version of an article to appear in *Third Fuji International Symposium on Functional and Logic Programming*, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998).

RS-97-52 Paola Quaglia. *On the Finitary Characterization of π-Congruences*. December 1997. 59 pp.

RS-97-51 James McKinna and Robert Pollack. *Some Lambda Calculus and Type Theory Formalized*. December 1997. 43 pp.

RS-97-50 Ivan B. Damgård and Birgit Pfitzmann. *Sequential Iteration of Interactive Arguments and an Efficient Zero-Knowledge Argument for NP*. December 1997. 19 pp.

RS-97-49 Peter D. Mosses. *CASL for ASF+SDF Users*. December 1997. 22 pp. Appears in *ASF+SDF'97, Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*, http://www.springer.co.uk/ewic/workshops/ASFSDF97. Springer-Verlag, 1997.

RS-97-48 Peter D. Mosses. *CoFI: The Common Framework Initiative for Algebraic Specification and Development*. December 1997. 24 pp. Appears in Bidoit and Dauchet, editors, *Theory and Practice of Software Development. 7th International Joint Conference CAAP/FASE*, TAPSOFT '97 Proceedings, LNCS 1214, 1997, pages 115–137.