Formatting Strings in ML

Olivier Danvy
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

\[
\text{format "\%i is \%a\n" 3 "x"}
\]

eys yields the string "3 is x\n", as specified by the control string "\%i is \%a\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

\[
\text{string \to int \to string \to string}
\]

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

\[
\text{format "\%i/\%i" 10 20}
\]

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

\[
\text{string \to int \to int \to string}
\]

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

- \texttt{i...} for specifying integers (\texttt{2i} above);
- \texttt{s...} for specifying strings (\texttt{2s} above);
- \texttt{l...} for declaring literal strings (\texttt{"is" and \texttt{/"} above); and
- \texttt{n...} for declaring newlines (\texttt{2n} above).

In addition, we provide the user with an associative infix constructor \texttt{co} to construct a complete pattern out of pattern components.\textsuperscript{1}

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

```ml
- format (i.. oo l.. "is" oo s.. oo n..) 3 "foo"
- val it = "3 is foo\n"
```

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

```ml
fun format (l.. oo l.. "is" oo s.. oo n..) : int -> string -> string
fun format (l.. oo l.. "/" oo l..) : 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 \texttt{co}, e.g., as function composition (\texttt{\circ} in ML).

- \texttt{i... and s...} work in a similar way:

```
fun l.. k s (x:int) = k (s "(makestring x))
(* val l.. : (string -> 'a) -> string -> int -> 'a *)
fun s.. k s x = k (s x)
(* val s.. : (string -> 'a) -> string -> string -> 'a *)
```

So for example, the type of the expression \texttt{i.. oo s.. oo i..} reads as follows.

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

The corresponding expression expects a continuation and a string, and returns a function of type \texttt{int -> string -> int -> 'a} that matches the "control string" \texttt{i.. oo s.. oo i..}.

- \texttt{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 x = 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:

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

\begin{verbatim}
format (i.. oo l.. "is" oo s.. oo n..)
\end{verbatim}

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

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

References


\textsuperscript{1}For cosmetic value, we could also provide two "outfix" constructors \texttt{<<} and \texttt{>>} to delimit a pattern.
Recent BRICS Report Series Publications


