A new implementation of OCaml formats based on GADTs

Benoît Vaugon, ENSTA-ParisTech

June 7, 2013

1 Introduction

OCaml provides a `Printf` module inherited from Caml-light more than a decade ago and inspired from the C `printf` function. Its implementation overrules the OCaml type system by using “magic”, which makes it very hard to maintain and evolve. We have implemented a new safe version of modules `Printf`, `Format` and `Scanf` that takes advantage of GADTs, which is a feature only recently introduced in OCaml. This new version improves the safety and performance of the code while preserving an almost complete compatibility with the old one.

The talk will justify the interest of a new implementation of formats in OCaml, show how GADTs are useful and efficient to implement it and discuss issues encountered during the implementation.

There exist different styles to convert or print values. For instance:

- In Java: `System.out.println("n␣=␣"+ n);`
- In PHP: `echo "n␣=␣$n";`
- In C: `printf("n␣=␣%d", n);`

The latter style, from which OCaml takes its inspiration, has the advantage of separating the format (or structure) from the data.

In OCaml, the `Printf` module provides means to format data in order to output the result into streams, extensible buffers or strings. The `Format` module is a superset of `Printf`. In addition to the latter, it provides boxes that are useful to pretty-print code as in the OCaml compiler. Formats may also be used with the `Scanf` module to extract data (by converting them on the fly) from strings or streams.

It is important to note that the syntax for formats is shared with the syntax for literal strings. To manage this, the OCaml type-checker uses a hack to recognize a format and computes its type.

2 The current implementation

In the current implementation, at run time, formats are represented by strings, which are parsed at each printing and scanning function call. More precisely, the printing functions wait for arguments, read the format character by character searching for a `'%'` or an `'@'`, extract the sub-format, call the C `sprintf` function to convert arguments into strings, then print in a stream or store in a buffer the resulting strings.

The scanning function similarly parses the format and consumes the stream, converts and accumulates extracted values onto a stack. Then, if the conversions succeed, it calls the callback function with extracted values as arguments.

The implementation of modules `Printf`, `Format` and `Scanf` uses `Obj.magic` to bypass the OCaml type-checker. This approach may lead to execution crashes. Number of use cases of formats generating segmentation faults have been found during the development.

In addition, this technique allocates lots of intermediate closures and strings. It slow-downs the execution and it is an issue in OCaml projects for which allocations are critical.

Finally, the parsing of formats is written multiple times in the library and the OCaml type-checker, bringing formats incompatibilities (which also cause crashes) between the type-checker and the library.

3 Which new implementation?

There are different ways to implement a safe and more efficient version of printing and scanning functions, that may or may not use GADTs.

An interesting idea is to implement formats as a triplet containing a printing function, a scanning function and a string representing the format. For instance, format "Hello␣%d%!" may be implemented by:

```
((fun out print_s box_f flush k n ->
    print_s out "Hello␣";
    print_s out (string_of_int n);
    flush out;
    k out),
 (fun stream callback ->
    check_string stream "Hello␣";
    let n = read_int stream in
    check_end_of_stream stream;
    callback n),
 "Hello␣%d%!")
```

With this implementation of formats, the printing and scanning functions just have to call a member of the format with the right parameters. For instance, the Printf.bprintf function would be implemented as follows:

```
let bprintf buf (print, _, _) =
  let out     = buf
  and print_s = Buffer.add_string
  and box_f _ = ()
  and flush _ = ()
  and kont _  = () in
  print out print_s box_f flush kont
```

We notice that, for compactness of the code, generic printing, formatting and flushing functions are passed as arguments. This slightly causes a slow-down. Lots of variants of this code may be implemented to improve efficiency, for instance by specialising the format functions for streams, buffers and strings.

In practice, all implementations of this kind generate huge codes and, in my opinion, are less elegant and enjoyable than GADTs approaches.

4 The new implementation

The new implementation uses GADTs instead of strings to represent formats at run time. The declaration of the format type looks like:

```
type ('a,'b,'c,'d,'e,'f) format6 =
| Char :                           (* %c *)
  ('a,'b,'c,'d,'e,'f) format6 ->
    (char->'a,'b,'c,'d,'e,'f) format6
| Bool :                           (* %B *)
  ('a,'b,'c,'d,'e,'f) format6 ->
    (bool->'a,'b,'c,'d,'e,'f) format6
| Flush :                          (* %! *)
  ('a,'b,'c,'d,'e,'f) format6 ->
    ('a,'b,'c,'d,'e,'f) format6
| String_literal :             (* "..." *)
  string *
  ('a,'b,'c,'d,'e,'f) format6 ->
    ('a,'b,'c,'d,'e,'f) format6
| [...]
| End_of_format
    ('f, 'b, 'c, 'e, 'e, 'f) format6
```

For instance, the GADT value associated to format "Hello␣%d%!" looks like:

```
String_literal ("Hello␣",
  Int (Iconv_d, No_padding,
       No_precision,
       Flush End_of_format))
```

and is statically computed and pre-allocated in the data segment by the OCaml compilers.

At type-check time, the OCaml type-checker recognizes formats as it is done in the current implementation (by matching "literal strings" with the "format type"), but the type-inference of the format is different. The constant string extracted from the source code, representing the format, is converted into a GADT value by a call to the unique parsing function shared with the standard library (remember that format syntax "%{...%}" can be used by scanning functions to extract a format from the stream). This GADT value is then converted into an "Abstract Syntax Sub-Tree" that is re-injected into the type-checker. The type of the format is not "manually" (or "magically") computed as in the current implementation. Instead, it is computed according to the format type definition (see the definition of format6 above) using the standard GADT typing rules.

This mechanism ensures format compatibilities between the type-checker and the library. It also factorizes the implementation of format parsing.

At print time, the format (which is a GADT value) is “converted” to functions. These functions take, as parameters, the heterogeneous values to be printed, and store them in an accumulator implemented by a heterogeneous linked list (thanks to GADTs). When all arguments are received (i.e, `End_of_format` is encountered), the accumulator elements are printed at once.

It is important to notice that the printing must be delayed until all arguments are received. Otherwise, the following code:

```
Array.iteri (printf "tab[%d]␣=␣%d\n")
  [| 41 ; 42 ; 43 |]
```

would print:

```
tab[0] = 41
1] = 42
2] = 43
```

instead of:

```
tab[0] = 41
tab[1] = 42
tab[2] = 43
```

At scan time, the format is read a first time to take extra scanning arguments representing the optional readers (see the semantic of `scanf "%r"`). The readers are stored in a heterogeneous list. The scanning of the stream is then performed according to the format contents. Sub-strings are extracted, converted on the fly and stored in an accumulator implemented by a heterogeneous list. Finally, the callback function is applied to the contents of the accumulator at once.

Let us mention that, as for the printing functions, the execution order is mandatory. However, the reason is different: it is here for the management of errors. Indeed, we must not start scanning until all arguments (the optional readers and the callback function) are received, and we must not start to partially call the callback function until the scanning has successfully finished.

5 Issues

As mentioned above, execution order of printing and scanning functions must be carefully kept to preserve the semantics of the current implementation (which appears to be a good compromise in practice). It implies multiple usages of heterogeneous lists to store printing arguments, readers and scanned values. They are other interesting usages of GADTs.

The compatibility with the current implementation was also a challenge. In particular, `'@'` has a different meaning for `Printf` (for which `'@'` is a standard character), `Format` (for which `'@'` is an escaped character used to manage boxes and separators) and `Scanf` (see the `scanf "%s@X"` syntax). In addition, some format constructions are specific to some modules. For instance, `"%a"` and `"%t"` are used for printing and are meaningless for scanning, `"%r"`, `"%[a-z]"` and `"%_"` are used for scanning but are meaningless for printing. These are the reasons why it was difficult to define a unified type for formats.

Finally, the factorisation of the parsing code for formats between the OCaml type-checker and the scanning functions was a challenge. This implementation required multiple hacks using original forms of GADTs.

6 Conclusion

This new implementation of formats would have a positive impact on the OCaml community. In fact, it does improve performance, fixes a lot of bugs, tidies and stabilises the code. Finally, it is a real usage of GADTs to elegantly solve a complicated problem.