Macros for OCaml: internship proposal

Olivier Nicole

June, 2016

The key features of the macro system implemented so far are: execution of static code during compilation, insertion of statically computed values in run-time code, and quoting, already allowing for the implementation of useful examples. A second internship working on it would be the opportunity to make it a usable extension and to develop a proof-of-concept project using it.

1 Nature of the work

The goal of the internship would be to continue implementing the modular macro system proposed by Leo White and Jeremy Yallop[1].

The proposed macro system is:

- homogeneous, i.e. the macro language is OCaml itself
- modular, i.e. seamlessly integrated into OCaml's module language
- type-safe.

2 Current progress

As of today, a subset of the proposal has been implemented, both in the bytecode compiler and the REPL, demonstrating the key features of macros:

2.1 `static` modifier, quoting and splicing

Variables and functions may be declared static (i.e., compile-time) using the `static` modifier. Static declarations can perform side effects, which will be performed during compilation:

```
static x = print_endline "defining␣x"; 42
```

Thanks to Leo White's quoting library, code may be quoted as in MetaOCaml, constructing values of type `'a expr`. For instance:

```
# static quote = << fun x -> x + 42 >>
val quote : (int -> int) expr =
  << fun x_1  -> x_1 + 42 >>
```

Any static value of type `'a expr` may be spliced elsewhere in the code using the `$` operator:

```
# static rec range = function
  | 0 -> << [] >>
  | n ->
    << $(Expr.of_int n) ::
       $(range (^Pervasives.pred n)) >>
val range : int -> int list expr = <fun>
# $(range 5)
- : int list = [5; 4; 3; 2; 1]
```

The `Expr` module contains functions for converting values of type `'a` to type `'a expr`. The `^Pervasives` syntax is an example of module lifting, described in the next section.

These basic features already permit to implement a *printf macro* as described in the "Modular macros" paper:

```
type (_,_) fmt =
    Int : (int -> 'a, 'a) fmt
  | Lit : string -> ('a, 'a) fmt
  | Cat : ('a, 'b) fmt * ('b, 'c) fmt ->
      ('a, 'c) fmt

static (%) x y = Cat (x, y)

static rec printk :
```

[1] "Modular macros", http://www.lpw25.net/ocaml2015-abs1.pdf

```
  type a b. (string expr -> b expr) ->
    (a, b) fmt -> a expr =
  fun k -> function
  | Int -> << fun s ->
      $(k <<string_of_int s>>) >>
  | Lit s -> k (Expr.of_string s)
  | Cat (l, r) ->
      printk (fun x ->
        printk (fun y -> k << $x ^ $y >>)
        r) l

static sprintf fmt = printk (fun x -> x) fmt
```

We can now have the expressiveness of `printf` without any runtime overhead (and without resorting to any compiler magic):

```
# static p = Lit "(" % Int % Lit ","
                     % Int % Lit ")"
# sprintf p
- : (int -> int -> string) expr =
  << fun s_3  ->
       fun s_4  ->
         ((("(" ^
              (Pervasives.string_of_int s_3))
              ^ ",") ^
              (Pervasives.string_of_int s_4))
            ^ ")" >>
# let print_pair (x,y) = $(sprintf p) x y
val print_pair : int * int -> string = <fun>
```

2.2 Integration within modules

An OCaml file may contain both static and runtime declarations.

```
(* file power.mli *)
val square : int -> int
static val power : int -> (int -> int) expr
```

The static part of a module is saved to a `.cmm` file. The static declarations in `Power` can then be used in other modules:

```
(* file a.ml *)
let power_nine = $(Power.power 9)
```

2.3 Phases

To prevent the user from mixing static and runtime code, a separation is enforced between runtime variables (phase 0) and static declarations and splices (phase 1):

```
# static x = 42
val x : int = 42
# let y = x + 1
Error: Attempt to use value x of phase 1 in
an environment of phase 0
```

But we want to be able to use runtime values from external values in static code. For this we have to explicitly lift the module by adding a ^ before its name:

```
# static tbl = Hashtbl.create 17
Error: Attempt to use value Hashtbl.create
of phase 0 in an environment of phase 1
# static tbl = ^Hashtbl.create 17
```

3 What's missing

3.1 Path closures

The natural way to write the `power` macro from above would be the following:

```
let square x = x * x

open ^Pervasives

static power' n x =
  if n = 0 then
    << 1 >>
  else if n mod 2 = 0 then
    << square $(power' (n/2) x) >>
  else
    << $x * $(power' (n-1) x) >>

static power n =
  << fun x -> $(power' n <<x>>) >>
```

However, this is rejected by the current implementation since splicing `power` elsewhere in the code could generate calls to `square` even if this function is no longer in scope (see "Modular macros"). Writing the above requires support for path closures, yet to be implemented.

3.2 Static modules

If a module is only intended for use in static code, it should be possible to declare it static:

```
static module Foo = struct
  (* ... *)
end
```

4 Example use cases

4.1 Build a static Web server

The `mirage-seal` tool creates a unikernel serving the contents of a specific directory over HTTPS. For this purpose it uses `ocaml-crunch`, which converts the contents of a directory into a static module.

Currently, `ocaml-crunch` generates OCaml files as simple strings. Its implementation could be made type-safe using macros.

4.2 Automatic generation of Web service interfaces

Web services may provide descriptions of their interfaces in the WSDL format, an XML-based standard playing quite the same role as that of an OCaml signature.

Macros could be used to automatically generate an OCaml module for accessing a particular Web service from a WSDL file.

4.3 Static ASN.1 combinators

`asn1-combinators` is a library that allows manipulation of ASN.1 grammars as first-class entities and generation of the associated parsers and serializers.

Using macros, ASN.1 parsers could be generated statically (thus improving performance) in a type-safe manner, offering security guarantees that are critical e.g. in OCaml-TLS.

5 What can be done in another 6 months

In a second 6-month, the following achievements can reasonably be expected:

- complete support of the macro system in its existing design, including path closures and static modules
- macro support in all compilers, including `ocamlopt`, `ocamlc.opt` and `ocamlopt.opt`
- polishing of the changes made to the compiler code up to a standard where it could be merged upstream
- development of at least one of the example use cases listed in the previous section.