Modular macros

Jeremy Yallop
University of Cambridge Computer Laboratory
jeremy.yallop@cl.cam.ac.uk

Leo White
leo@lpw25.net

We propose an OCaml language extension for type-safe compile-time code generation.

1. Macros

Programming often involves a choice between efficiency and abstraction. One the one hand, using high-level abstractions can lead to an elegant program with suboptimal performance On the other, optimising for performance often means abandoning reusability and manifest correctness.

As an example, consider the following definition of `Printf`-style format strings:

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

let (%) x y = Cat (x, y)
```

Here is a format string, analogous to `"(%d,%d)"`, for printing out pairs of integers:

```
let p = Lit "(" % Int % Lit "," % Int % Lit ")"
```

And here is `sprintf`, by means of a CPS auxiliary function:

```
let rec printk :
 type a b. (string → b) → (a, b) fmt → a =
  fun k → function
    Int → fun s → k (string_of_int s)
  | Lit s → k s
  | Cat (l, r) →
     printk (fun x → printk (fun y → k (x ^ y)) r) l

let sprintf fmt = printk (fun x → x) fmt

# sprintf p 3 4
- : string = "(3,4)"
```

The `printk` function acts as an interpreter for the language of format strings. Using format strings rather than direct calls to the functions `string_of_int`, `^`, etc., has a number of advantages, including readability and reusability, but is typically less efficient.

If we instead write `printk` using the system of *macros* described in this abstract there is no longer a need to choose between abstraction and performance. Macros support programming with the full OCaml language, including the module system, and make it possible to build low-level code (like a sequence of calls to `string_of_int`) from high-level descriptions (like a value of the `fmt` type), retaining the benefits of abstraction, but eliminating all interpretative overhead. In contrast with existing solutions to compile-time metaprogramming such as Camlp4 and "ppx" AST transformers, macros are fully integrated into the OCaml language, so that code generation itself is well-typed. Whereas existing tools rely on the OCaml compiler to check generated code for well-typedness (and even well-scopedness), macros themselves are guaranteed to never generate ill-typed code.

Here is a second of definition of `printk`, written as a code-generating macro:

```
macro rec printk :
 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 << $(lift_string s) >>
  | Cat (l, r) →
     printk (fun x →
       printk (fun y → k << $x ^ $y >>) r) l
```

This definition of `printk` involves two new expression constructs, borrowed from MetaOCaml. *Quoting* an expression by placing it between brackets `<< >>` delays its evaluation, turning it into a piece of code that can later be used as part of a larger program. *Splicing* a quoted expression into a larger piece of code is performed with the `$` operator. The typing is straightforward: if `e` has type `t` then `<<e>>` has type `t expr`; conversely, if `e'` has type `t expr` then `$e'` has type `e`. As in MetaOCaml, quotations allow type-safe programming with *open code*, but we direct the interested reader to the MetaOCaml literature (e.g. [2]) for the details.

Inspired by Racket [1], we divide the evaluation of programs written using macros into two *phases* (and sometimes more, but we will stick to two here). Expressions are evaluated either at runtime (phase 0) or during compilation (phase 1). In this abstract we use colour to highlight the phase distinction, colouring those expressions in blue which are evaluated in phase 1, and leaving those expressions black whose evaluation is delayed until phase 0. Evidently, in the definition of `printk`, only black expressions will remain in the program after macro expansion. There is a family of functions `lift_string`, `lift_int`, and so on, which turn values into expressions suitable for use at a later phase.

Evaluation phases, macro bindings, quotation and splicing together form a coherent system. Macros (bound with **`macro`**) can only be used directly in expressions at phase 1, whereas functions (bound with **`let`**) can only be used directly in expressions at phase 0. Quoting allows references to phase 0 definitions in phase 1 expressions, whilst splicing allows construction of phase 0 expressions using phase 1 definitions.

The definition of `sprintf` does not need to be changed, but it now generates code rather than printing its arguments directly:

```
# sprintf p
- : (int → int → string) expr =
<< fun s1 s2 → "(" ^ string_of_int s1 ^
                "," ^ string_of_int s2 ^ ")" >>
```

The generated code can be inserted into a larger program using the splicing operator:

```
let print_pair (x, y) =
 $(sprintf (Lit "(" % Int % Lit ","
                    % Int % Lit ")")) x y
```

This definition highlights a difference between macros and MetaOCaml. In MetaOCaml, splices are only permitted within a quotation. With macros, splices can occur at the top level of a program, allowing insertion of generated code into a file which is to be evaluated at phase 0.

2. Modular macros

Like other OCaml program elements — values, types, exceptions, and so on — macros belong to modules, and play a full part in the module system: a macro can be referred to by a path, included within another module, hidden by signature ascription, and so on. The combination of macros and modules raises two questions of particular interest: first, what happens when a macro expands to code which includes identifiers hidden by a signature? and second, what is the meaning of a macro passed via a functor argument?

2.1 Macros out of modules: ascription and path closures

An expression inside a quotation can refer to any in-scope identifier that will be available when the expression is evaluated. However, an identifier that is in scope when a quotation is created need not be visible at the point where the quotation is spliced into the program.

For example, here is an implementation of the classic staged `power` function, which builds a recursion-free exponentiation function for a particular exponent.

```
module Power : sig
 macro power : int → (int → int) expr
end = struct
 let square x = x * x

 macro rec spower n x =
   if n = 0 then
     << 1 >>
   else if n mod 2 = 0 then
     << square $(spower (n/2) x) >>
   else
     << $x * $(spower (n-1) x) >>

 macro power n = << fun x → $(spower n << x >>) >>
end
```

With a naive implementation, calling `power` would generate code containing references to the `square` function, which is not in scope outside the `Power` module, having been hidden by the signature:

```
# Power.power 5
- : int expr =
<< fun x → x * square (square (x * 1)) >>
```

How can we ensure that identifiers available in the environment where a quotation was created can be safely used in a different context? There is a clear analogy to the question of how to treat free variables used in local functions, and the issue can be solved using an analogous technique: closures.

A closure in a language with lexical scope consists of the code of a function together with the values of the free variables used by the function. Analogously, a *path closure* for a macro consists of the definition of the macro along with the set of free identifiers used in the macro definition. A macro exported from a module therefore itself behaves as a module which exports a binding for each identifier used by the macro

```
module Power : sig
 module Closure1 :
   val square : int → int
   macro power : int → (int → int) expr
 end
end = (* ... *)
```

Invoking the macro generates code which refers to elements of the module:

```
# Power.power 5 (* expands to Power.Closure1.power 5 *)
- : int expr =
<< fun x → x * (Closure1.square
                 (Closure1.square (x * 1))) >>
```

In order to preserve abstraction, the names in the closure cannot be referred to directly by the program; they are only accessible through the corresponding macro.

2.2 Macros into modules: functor staging

As we have seen, macros belong to modules. Here is a signature `MONOID` of printable monoid values, which contains some macro members:

```
module type MONOID = sig
  type t
  macro one : t expr
  macro mul : t expr → t expr → t expr
  val show : t → string
end
```

OCaml functors are modules which are parameterised by other modules. Here is a functor `F` which is parameterised by a module `M` with type `MONOID`:

```
module F(M : MONOID) =
struct
  let rec mtimes = function
    [] → $(M.one)
  | x :: xs → $(M.mul <<x>> << mtimes xs >>)

  let show_mtimes l = M.show (mtimes l)
end
```

The functor `F` defines two functions, `mtimes` and `show_mtimes`, which use macros and functions from `M`. However, there is an apparent difficulty: in OCaml functor application takes place at runtime (phase 0), which is too late to perform macro expansion.

Once again, there is a known technique which can be generalized to solve the difficulty. A one-parameter functor in regular OCaml can be decomposed into a curried two-parameter functor which accepts static arguments (i.e. the type components) via the first argument and the dynamic arguments (i.e. the value components) via the second. The dynamic arguments may depend on the static arguments, but there are no dependencies in the other direction. Extending this scheme to support macros involves adding the macro components to the static parameter; for example, the functor `F` may be decomposed as follows:

```
module F_staged(M_static : sig
                 type t
                 macro one : t expr
                 macro mul : t expr → t expr → t expr
                end)
               (M_dynamic : sig
                 val show : M_static.t → string
                end) = (* etc. *)
```

Each functor argument is applied in a particular phase. For example, an application `F(Int)` is decomposed into an application `F_staged(Int_static)(Int_dynamic)`, where the first part of the application takes place during phase 1 and the second part takes place during phase 0.

2.3 Module lifting

Macro definitions are one way to construct functions that can be used during compilation (phase 1). A second source of compile-time functions comes from importing compiled modules during compilation, making their values available for use in macros.

Racket supports cross-phase module lifting using the `require-for-syntax` construct. Since OCaml does not use explicit require statements we instead use a command-line argument; for example, the following makes the values from phase 0 of the Power module available for use in phase 1 of `math.ml`:

```
ocamlc -k power.cmo math.ml
```

3. Acknowledgements

We thank Matthew Flatt, Oleg Kiselyov, Jay McCarthy and Carl Eastlund for helpful discussions.

References

[1] Matthew Flatt. Composable and compilable macros: : you want it when? In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, pages 72–83. ACM, 2002.

[2] Oleg Kiselyov. The design and implementation of BER metaocaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014.