Modular macros (demo proposal)

Olivier Nicole
Paris-Saclay University
nicole@ensta.fr

Leo White
Jane Street Capital
leo@lpw25.net

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

1 Modular macros

Modular macros extend OCaml with support for generative, typed, compile-time metaprogramming. A previous presentation [5] introduced the high-level design of modular macros, which we have since implemented in a fork of the OCaml compiler.

Modular macros inherit the basic MetaOCaml design [2], with primitives for quoting code and combining code via splicing:

```
<< e >>     $e
```

Following MetaOCaml, modular macros combine strong guarantees with expressive power: they are generative, hygienic, and typed, and the full OCaml language, with effects, modules, data types, and polymorphism, can be used to generate quoted programs.

Departing from MetaOCaml, and inspired by Racket [1], modular macros execute entirely during the compilation phase, expanding into macro-free OCaml programs.

This change in execution model leads to a cascade of additional differences in the design and implementation. Modular macros cannot support cross-stage persistence (CSP) of arbitrary values, since the compile-time execution context in which macros are expanded no longer exists when the fully-expanded program is executed. However, modular macros do support a more limited form of CSP that restricts quoted identifiers to top-level bindings in modules. Full support for this feature involves a closure conversion transformation to ensure that identifiers are accessible even after module signature ascription. (See [5] for details.)

Despite the differences, many substantial MetaOCaml programs, such as the Strymonas library [3], can be ported to modular macros with only local syntactic changes.

2 The demo

We propose demonstrating the modular macros implementation by interactive construction of a small library for typed `printf`-style printing with compile-time optimizations.

Modular macros integrate harmoniously with the OCaml toolchain; our library uses the standard `ppx` preprocessor to transform a convenient syntax for format strings into a typed GADT representation:

```
  [%fmt "(%b,%b)"]
⇝
  Cat (Lit "(", Cat (Bool,
   Cat (Lit ",", Cat (Bool, Lit ")"))))
```

After this untyped syntactic transformation, typed modular macros perform the heavy lifting, using partially-static data [4] and control effects to generate efficient programs specialized to particular format strings.

```
  $(sprintf [%fmt "(%b,%b)"])
⇝
  let f b = if b then "(false,true)"
            else "(false,false)" in
  let g b = if b then "(true,true)"
            else "(true,false)" in
  let h b = if b then g else f in h
```

Along the way we will show how modular macros interact with OCaml's advanced module system.

If time permits we will show larger examples, such as the Strymonas library for stream fusion (Figure 1).

```
$(of_arr << [| 0;1;2;3 |] >>
  |> filter (fun x   → << $x mod 2 = 0 >>)
  |> fold   (fun z a → <<$a::$z>>) <<[]>>)
⇝
let s = ref []  in
(let arr = [|0;1;2;3|]  in
 for i = 0 to Array.length arr - 1 do
  let el = Array.get arr i in
  if el mod 2 = 0 then s := el :: !s
 done);
!s
```

Figure 1. Strymonas ported to modular macros

References

[1] Matthew Flatt. 2002. Composable and compilable macros: you want it when?. In *ICFP 2002*.
[2] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml – System Description. In *FLOPS 2014*.
[3] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL 2017*. ACM.
[4] Jeremy Yallop, Tamara von Glehn, and Ohad Kammar. 2018. Partially static data as free extension of algebras. (January 2018). Presented at PEPM 2018.
[5] Jeremy Yallop and Leo White. 2015. Modular Macros. (September 2015). OCaml Users and Developers Workshop 2015.

Conference'17, July 2017, Washington, DC, USA