Safe and efficient generic functions with MacoCaml

Dmitrij Szamozvancev
University of Cambridge
ds709@cam.ac.uk

Leo White
Jane Street Capital
leo@lpw25.net

Ningning Xie
University of Toronto
ningningxie@cs.toronto.edu

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

Abstract

We apply MacoCaml, an extension of OCaml with support for compile-time user-specified code generation, to the *generic function* problem. MacoCaml's combination of macros with phase separation and code quotations neatly addresses what is a recurring challenge for OCaml developers: how to write safe and efficient functions over type representations?

Our solution to the challenge also illustrates some recently-established formal guarantees offered by MacoCaml, including soundness and phase distinction.

1. Generic functions in OCaml: two approaches

Generic functions — that is, operations such as equality and pretty-printing defined over the structure of arbitrary data types — are a frequent need in OCaml programs. At the time of writing, the most-viewed post on the OCaml Discourse forum asks about the availability of generic functions[1] and the most-commented thread this year discusses how to write them[2].

There are presently two main approaches to writing generic functions, with complementary advantages and drawbacks. The *static approach* uses preprocessors such as ppx to manipulate parse trees during compilation. Here is an excerpt of a typical example[3], which generates code for an equality function on lists by mapping syntax for types `[%type ...]` to syntax for expressions `[%expr ...]`:

```
| [%type: [%t? typ] list] →
  [%expr let rec loop x y =
          match x, y with
          | [], [] → true ... ]
```

Defining generic functions by code generation has a clear benefit: generated functions can be as efficient as handwritten code. However, using untyped AST manipulation has drawbacks: there are no guarantees that the generated code is well-scoped or well-typed.

The *dynamic approach* involves functions defined over representations of types. Here is a typical type representation:

```
type 'a prim = Int: int prim | Float: float prim | ...
type 'a t = Primitive of 'a prim
          | Var of 'a var
          | Sum: 'a t * 'b t → ('a, 'b) either t
          | Prod: 'a t * 'b t → ('a * 'b) t
          | Iso: ('a,'b) iso * 'a t  → 'b t
type 'a ty = Ty of 'a var * 'a t
```

A value of type `prim` represents a predefined OCaml primitive type such as `int` or `float`. A value of type `t` represents either a primitive type p, a reference x to a type definition, a sum $t_1 + t_2$, a product $t_1 \times t_2$, or a representation of a type that is isomorphic to some other representation. A value of type `ty` attaches a variable x[4] to a representation t, making it possible to build recursive definitions $\mu x.t$. For example, the type list-of-a is represented using the isomorphism to $\mu x.1 + (a \times x)$:

```
let list: type a.a t → a list t =
  fun a → let x = var () in
  Ty (x, Iso {out=outList; in_=inList),
          (Sum (Primitive Unit,
                Prod (Var (tyname a), Var x))))
```

These type representations can be used to define generic functions. For example, `eqt`, defined inductively on type representations `t`, acts as an equality function at the corresponding type:

```
let rec eqt: type a.env → a t → a → a → bool =
  fun env t x y → match t with
  | Primitive p → eqprim p x y
  | Var v → lookup v env x y
  | Sum (l,r) → ...
  | Prod (l,r) → let a,b = x and c,d = y in
                 eqt env l a c && eqt env r b a
  | Iso ({out; in_}, i) → eqt env i (in_ x) (in_ y)
```

The first parameter, of type `env`, maps names to equality functions. The top-level function `eqty` adds an entry to that environment for each recursive type definition, allowing recursively-defined functions:

```
let eqty: type a.env → a ty → a → a → bool =
  fun env (Ty (v,ty)) →
  let rec eq x y = eqt (Bind (v,eq,env)) ty x y) in eq
```

Now `eqty` can be used to compare values of arbitrary type:

```
eqty Nil (Prod (bool, int)) (true, 4) (true, 4)   ⇝  true
eqty Nil (list int32) [3l; 4l] [4l; 5l]           ⇝  false
```

Since `eqty` is a normal function, OCaml ensures that it has no scoping or typing errors. However, inspecting type representations at runtime may introduce unacceptable performance costs.

2. A new approach: generic macros in MacoCaml

MacoCaml, an extension of OCaml that we are developing, supports a third approach to the generic function problem. Like the static approach above, it uses quotations to generate code; however, MacoCaml's quotations come with strong type safety guarantees. Like the dynamic approach above, it involves defining functions inductively on type representations; however, MacoCaml ensures that the type representations are not used at runtime.

Figure 1 shows the `eqty` example, transformed to use MacoCaml's macros. There are two key changes to `eqt` and `eqty`: they are defined using **macro** rather than **let**, and they have been annotated with quotes `<< e >>` and splices `$e` to turn them into code generators.

The **macro** keyword is an example of MacoCaml's support for *phases*, inspired by Racket (Flatt 2002). Phases are times, such as *compile time* or *run time*, when expressions may be evaluated. Definitions bound with **macro** make expressions available for evaluation at compile time, while definitions bound with **let** make expressions available for evaluation at run time. MacoCaml also supports controlling the phase of code using the module system; we refer the reader to our recent paper (Xie et al. 2023) for details.

MacoCaml's *quotations* support safe generation of typed code. Inspired by quotations in MetaOCaml (Kiselyov 2014), they enjoy the same strong guarantees: generated code is guaranteed to be well-typed and well-scoped. As in MetaOCaml, a quoted expression `<< e >>` builds a representation of `e` (with type `t` **expr** if `e` has type `t`) rather than evaluating it, and a splice `$e` evaluates `e` (of type `t` **expr**) to generate code that is inserted at the splice location.

---

[1] *How does one print any type?*, https://discuss.ocaml.org/t/-/4362

[2] *Idea: Standard OCaml runtime type representation*, https://discuss.ocaml.org/t/-/12051

[3] from ppx_deriving https://github.com/ocaml-ppx/ppx_deriving

[4] Variables, and corresponding functions to resolve them in environments, are defined using *type identifiers*, which Daniel Bünzli recently added to OCaml (*Add Type.Id*, ocaml/ocaml#11830); we omit the details here.

```
macro rec eqt env t x y = match t with
 | Primitive p → eqprim p x y
 | Var v → << $(lookup v env) $x $y >>
 | Sum (l,r) → (* ... *)
 | Prod (l,r) → << let a,b = $x and c,d = $y in
                  $(eqt env l <<a>> <<c>>)
               && $(eqt env r <<b>> <<d>>) >>
 | Iso ({out; in_}, i) → eqt env i (in_ x) (in_ y)
```

erase → *(Erased!)*


```
macro rec eqt env t x y = match t with
 | Primitive p → eqprim p x y
 | Var v → << $(lookup v env) $x $y >>
 | Sum (l,r) → (* ... *)
 | Prod (l,r) → << let a,b = $x and c,d = $y in
                  $(eqt env l <<a>> <<c>>)
               && $(eqt env r <<b>> <<d>>) >>
 | Iso ({out; in_}, i) → eqt env i (in_ x) (in_ y)
```


```
macro eqty: type a.env → a ty → (a → a → bool) expr
 = fun env (Ty (v,ty)) →
   << let rec eq x y =
        $(eqt (Bind (v,<<eq>>,env)) ty <<x>> <<y>>)
      in eq >>
```

erase → *(Erased!)*


```
macro eqty: type a.env → a ty → (a → a → bool) expr
 = fun env (Ty (v,ty)) →
   << let rec eq x y =
        $(eqt (Bind (v,<<eq>>,env)) ty <<x>> <<y>>)
      in eq >>
```


```
let eq_list_int32: int32 list → int32 list → bool =
  $(eqty (list int32))
```

expand →

```
let eq_list_int32: int32 list → int32 list → bool =
  let rec eq x y = match inList x, inList y with
  | Left l, Left r → true
  | Right l, Right r → let a,b = l and c,d = r in
                       Int32.eq a c && eq b d
  | _ → false
  in eq
```

Figure 1. Compilation: preserve types, erase static computations, expand splices and discard the static heap

However, there are also important differences between MacoCaml's quotations and MetaOCaml's. First, MacoCaml supports *top-level splices*, which allow code generated by a macro to be inserted at program top level, as illustrated in the definition of `eq_list_int32` in Figure 1. Second, MacoCaml's careful phase management means that cross-stage persistence (quoting values in scope within a code generator so that they appear in generated code) is not allowed; only identifiers that are bound either in the generated code (such as `eq` in the definition of `eqty`) or in top-level **let** bindings can be quoted.

Figure 1 also illustrates compilation in MacoCaml. During compilation, expressions in top-level splices are evaluated to generate code which is inserted in place, and macros are erased, producing the standard OCaml program shown on the right of the figure.

The generated code for `eq_list_int32` illustrates a key advantage of MacoCaml's approach: it is manifestly free of the overhead of matching type representations[5]. It is easy to verify that the macros on the left of the figure never generate code involving type representations, since the constructors of `t` do not appear within quotes.

3. MacoCaml's safety and efficiency properties

The example in Figure 1 illustrates two key guarantees provided by MacoCaml: type soundness and phase distinction. We briefly describe them here; our recent paper (Xie et al. 2023) gives details.

3.1 Safety

As with other languages in the MetaML family, like MetaML itself (Taha et al. 1998) and Typed Template Haskell (Xie et al. 2022), MacoCaml enjoys type soundness. For a language with quotations, soundness guarantees that generated code is never ill-typed or ill-scoped. For programmers, type soundness is an important guarantee: it means that MacoCaml guarantees basic correctness properties of code generators such as `eqty`, so that it is never necessary to debug type errors in the generated code.

3.2 Efficiency

MacoCaml also enjoys a distinctive *phase separation* guarantee that has not been established for related languages such as MetaML or Typed Template Haskell. Phase separation says that if a program P evaluates to V, then the *erasure* of P evaluates to the erasure of V; it justifies discarding the compile-time heap (used to evaluate macros) and erasing compile-time **macro** bindings after compilation. For programmers, phase separation is also valuable, since it ensures a clean separation between code that manipulates generation-time values such as type representations, and generated code, where those values cannot appear. For generic functions such as `eqty`, phase separation represents an efficiency guarantee, since run-time inspection of type representations carries a performance cost.

[5] The code is, however, somewhat less efficient than it could be due to various naivities in the type representation; more sophisticated approaches to representing types (e.g. Yallop (2017)) do not suffer from this drawback.

4. Status and future plans

Since our last OCaml Users and Developers Workshop presentation (Yallop and White 2015) we've formalised our core design and implemented MacoCaml[6]. As Xie et al. (2023) outline, in the current implementation compilation overhead appears to be fairly modest.

In the future we plan to extend the formalism to support OCaml's full module system, including functors and signatures with abstract types and subtyping, and to bring the implementation to a state where it can be merged into the main OCaml distribution.

References

M. Flatt. Composable and compilable macros: You want it when? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, page 72–83, 2002. .

O. Kiselyov. The design and implementation of BER MetaOcaml. In M. Codish and E. Sumii, editors, *Functional and Logic Programming*. Springer International Publishing, 2014. .

W. Taha, Z. Benaissa, and T. Sheard. Multi-stage programming: Axiomatization and type safety. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming, 25th International Colloquium*, volume 1443 of *Lecture Notes in Computer Science*. Springer, 1998. .

N. Xie, M. Pickering, A. Löh, N. Wu, J. Yallop, and M. Wang. Staging with class: a specification for Typed Template Haskell. *Proc. ACM Program. Lang.*, 6(POPL):1–30, 2022. .

N. Xie, L. White, O. Nicole, and J. Yallop. MacoCaml: Staging composable and compilable macros. *Proc. ACM Program. Lang.*, 7(ICFP), 2023. *Conditionally accepted.*

J. Yallop. Staged generic programming. *Proc. ACM Program. Lang.*, 1(ICFP): 29:1–29:29, 2017. .

J. Yallop and L. White. Modular macros. OCaml Users and Developers Workshop, September 2015.

[6] The implementation was largely developed by Olivier Nicole, and is available here: https://github.com/modular-macros/