Safe and efficient generic functions with MacoCaml

Dmitrij Szamovanzve
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’ and the most-commented thread this year discusses how to write them’. 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 example1, which generates code for an equality function on lists by mapping syntax for types [Xtype ...] to syntax for expressions [Xexpr ...]:

```ocaml
| [Xtype: [?t? type] list] ->
| [Xexpr 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:

```ocaml
type 'a var = Int : int var | Float : float var | ...
type 'a ty = Ty

A value of type var represents a predefined OCaml primitive type such as int or float. A value of type ty represents either a primitive type p, a reference x to a type definition, a sum t₁ + t₂, a product t₁ × t₂, or a representation of a type that is isomorphic to some other representation. A value of type ty attaches a variable x' to a representation t, making it possible to build recursive definitions μx.t. For example, the type list-of-a is represented using the isomorphism to μx.1 + (a × x):

```

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/-/12061
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.

let list: type a. a t -> a list t =
  fun a -> let x = var() in
  Ty (x, In, (in_,Nil)),
  (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 -> expr (p x y)
  | Var v -> lookup v env x y
  | Sum (1, r) -> ...
  | Prod (1, r) -> let a, b = x and c, d = y in
eqt env 1 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 = eq (Bind (v, env, ty)) ty x y in
eqy env
```

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) [31; 41] [41; 51] -> 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 | 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 | 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.
### 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, it means that MacoCaml guarantees basic correctness properties of code generators 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.

#### 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.

### References


---

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

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