First-class modules: hidden power and tantalizing promises

Jeremy Yallop
Applicative Ltd

Oleg Kiselyov
FNMOC, Monterey, CA

Abstract

First-class modules introduced in OCaml 3.12 make type constructors first-class, permitting type constructor abstraction and polymorphism. It becomes possible to manipulate and quantify over types of higher kind. We demonstrate that as a consequence, full-scale, efficient generalized algebraic data types (GADTs) become expressible in OCaml 3.12 as it is, without any further extensions. Value-independent generic programming along the lines of Haskell's popular "Generics for the masses" become possible in OCaml for the first time. We discuss extensions such as a better implementation of polymorphic equality on modules, which can give us intensional type analysis (aka, type-case), permitting generic programming frameworks like SYB.

1. Introduction

OCaml 3.12 introduced three major extensions to OCaml: notation for polymorphic recursion, scoped type variables, and first-class modules. Whereas the former is mainly syntactic sugar (polymorphic recursion being already possible using polymorphic record fields), the latter two are genuine extensions. First-class modules – first-class functors – permit type constructor abstraction and polymorphism. Type constructor polymorphism makes it possible to encode genuine Leibniz equality. The latter, along with existentials and polymorphic recursion (both of which can also be encoded using first-class modules), let us implement GADTs. We can now work with "real" GADTs in OCaml, without the need for extensions.

The complete code that implements and illustrates GADTs, generic programming, and other uses of first-class modules is given as supplemental material. Section titles refer to the corresponding code by the file name, given in parentheses.

2. Leibniz equality (leibniz.ml)

The power of GADTs derives from their behaviour during pattern matching. Each branch of a pattern match potentially exposes a proof of equality between types, allowing the compiler to refine the type of the scrutinee. In order to emulate this behaviour in OCaml, we supply to each constructor an additional argument representing an equality proof. This proof is represented as a value of type `eq`, which has constructors corresponding to familiar equality axioms:

```
type ('a, 'b) eq
val refl : ('a, 'a) eq
val symm : ('a, 'b) eq -> ('b, 'a) eq
val trans : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
```

In order to make use of these proof values, we also need a destructor that turns an proof of type equality into a coercion between types:

```
val cast : ('a, 'b) eq -> 'a -> 'b
```

All versions of OCaml allow an implementation of `eq` as a straightforward isomorphism which supports all these operations:

```
type ('a, 'b) eq = ('a -> 'b) * ('b -> 'a)
```

However, for full generality we need a further axiom, corresponding to Leibniz's subsitutivity law. For every type constructor `tc` we need a function that turns a proof that `'a` is equal to `'b` into a proof that `'a tc` is equal to 'b tc:

```
val subst = ('a, 'b) eq -> ('a tc, 'b tc) eq
```

It is possible to write this function for particular instances of `tc` using the isomorphism implementation of `eq`. For example, if `tc` is the `list` constructor then the `map` function is all we need:

```
let subst_list (f, g) = (map f, map g)
```

However, it is not possible to write such `subst` for every instance of `tc`. There is no satisfactory way to write `subst` for abstract or mutable types, for example. There is also no way to write a single definition that suffices for all type constructors. Further, even the definitions that are possible to write are inefficient: a cast based on `subst_list` traverses and copies its argument.

The only way to abstract over a parameterised type constructor in OCaml is to use the module language. For example, in

```
module type TC = sig type 'a tc end
```

`tc` has the kind `* -> *`. But modules are not first class—until OCaml 3.12.

With genuine Leibniz equality, we do not copy the list, we return the original list as it was. No memory needs to be allocated.

Furthermore, `refl`, `subst` and `cast` are sufficient to implement `symm` and `trans`. Here is a definition of `trans`:

```
let trans (type a) (type b) (type c) a_eq_b b_eq_c =
 let module TC = Subst(struct type 'x tc = (a, 'x) eq end)
  in cast (TC.subst b_eq_c) a_eq_b
```

We regard the type `('a,'b) eq` as an instance of the type `'b tc` where the type "constructor" tc (of the kind `* -> *`) is a *type-level* abstraction λ`x.('a,x)eq`. We then apply Leibniz's law to such interpreted `'b tc`, substituting `'b` with `'c` as licensed by the equality witness `('b,'c) eq`. The result is the term of the type `('a,'c) eq`, witnessing the equality of `'a` with `'c`. The expression for `trans` demonstrates that we take the notion of type constructor quite broadly, as an arbitrary type-level abstraction.

As we show in accompanying code, `symm` can be defined analogously.

3. Polymorphic recursion (polyrec.ml)

OCaml has supported polymorphic recursion for some time, via record fields, object methods with polymorphic types, and recursive modules. OCaml 3.12 adds a more convenient notation.

First-class modules give yet more encodings of polymorphic recursion. One encoding uses functors to emulate the "big lambda" of System F, similar to the treatment of polymorphism in Harper and Stone's semantics of Standard ML. Making functors first class naturally leads to first-class polymorphism, which is sufficient to

encode polymorphic-recursive functions. Less verbosely, we can simply wrap a polymorphic function in a module, which then behaves much the same as a record with a polymorphic field.

4. Existential types (existentials.ml)

It is well known that abstract types (as used in modules) can be used to encode existential types. Indeed, Russo's design for first class modules, from which the OCaml design derives, is based directly on extended forms of the `open` and `pack` forms for existentials.

Russo illustrates the connection with existentials using an implementation of the sieve of Eratosthenes based on an abstract stream type. A functor from streams to streams corresponds to one step of the algorithm. Since the number of applications of the functor depends on a runtime value, this encoding fundamentally depends on first-class modules to eliminate the stratification between the module and core languages.

As is also well known, existentials can be encoded using parametric polymorphism. As we noted above, OCaml supports polymorphism for record or object fields, and Russo's example translates straightforwardly without the need for first-class modules. However, the existentials arising from first-class modules are strictly more powerful than the existentials encoded using polymorphic record fields: the former support higher-kinded existential variables, which cannot be represented in the latter encoding. We include a second example, also based on streams, adding a `map` function which illustrates this additional power.

5. GADTs (gadts.ml)

We now have all the pieces needed to encode GADTs in OCaml. Perhaps the most common example of a GADT is a well-typed evaluator for an algebraic typed object language represented as an algebraic datatype. In Haskell notation we might write it as follows:

```
data Exp :: * -> * where
 Lft :: Int -> Exp Int                    -- Lift constants
 Inc :: Exp (Int -> Int)                  -- Succ function
 Lam :: (Exp u -> Exp v) -> Exp (u -> v) -- Lambda (HOAS)
 App :: Exp (u -> v) -> Exp u -> Exp v    -- Application
```

Internally, GHC represents this as a standard existential datatype with explicit equality proofs :

```
data Exp a =
    a ~ Int => Lft Int
  | a ~ Int -> Int => Inc
  | forall u v. a ~ u -> v => Lam (Exp u -> Exp v)
  | forall u v. a ~ v => App (Exp (u -> v)) (Exp u)
```

Our OCaml encoding represents both the equality proofs and the existential variables explicitly:

```
type 'a exp =
    Lft of ('a,'a) eq * 'a
  | Inc of (int -> int,'a) eq
  | Lam of < m_lam : 'w. ('a,'w) lam_k -> 'w >
  | App of < m_app : 'w. ('a,'w) app_k -> 'w >
and ('a,'w) lam_k =
 {lam_k : 'u 'v. (('u->'v),'a) eq * ('u exp->'v exp) -> 'w}
and ('a,'w) app_k =
 {app_k : 'u 'v. ('v,'a) eq * ('u->'v) exp * 'u exp -> 'w}
```

During evaluation we unpack the existentials and use the equality proofs to coerce the results of each branch. The accompanying file gives the full code, including smart constructors and example terms. We also include a second example: a GADT representing a formatting specification for use by both printf and scanf.

6. Generic programming (generics.ml)

Generic programming systems are typically designed around value-level representations of types. Most work on generic programming in ML uses value-dependent representations of types, in which a type representation consists of one or more generic functions specialized to the type that is represented. For example, the following type representation consists of a printing function and an equality function.

```
type 'a repr = ('a -> string) * ('a -> 'a -> bool)
val unit : unit repr
val int : int repr
val ( * ) : 'a repr -> 'b repr -> ('a * 'b) repr
```

The representation type comes equipped with one constructor value for each constructor of the type language: we can construct a type representation value `int * unit` that can be used to print or compare values of the type that is written in the same way.

In contrast, the more extensive work on generic programming in Haskell is almost entirely based around value-independent type representations, which have no inherent connection with any particular generic function. The modularity benefits of such approaches are obvious, but they typically require advanced type system features, such as GADTs or higher-kinded type variables, that are not available in ML-family languages.

The introduction of first class modules make generic programming based on value-independent type representations possible in OCaml for the first time. We can specify the type representation without committing to any particular implementation:

```
module type Interpretation : sig
  type 'a tc
  val unit : unit tc
  val int : int tc
  val ( * )  : 'a tc -> 'b tc -> ('a * 'b) tc
end
```

Our representation then abstracts over instances of this signature:

```
module type Repr = sig
  type a
  module Interpret (I : Interpretation) :
  sig val result : a I.tc end
end
type 'a repr = (module Repr with type a = 'a)
```

A generic function takes a type representation `s repr` and supplies the Interpretation argument to obtain a function whose type involves `s`. For example, the generic printing function has the type:

```
val show : 'a. 'a repr -> 'a -> string
```

Its implementation unpacks the `repr` argument and applies its `Interpret` functor to a suitable instance of Interpretation:

```
module Show : Interpretation with type 'a tc = 'a -> string
```

7. Proposal: Intensional type analysis

A truly usable OCaml generic programming library will need a complete representation for "representable" types. Using OCaml's own representation of types would avoid the need to define a type representation for every user-defined type—an enticing prospect.

First-class modules would make it possible to tap into OCaml's type representation, if the equality of first-class module values were more principled. Currently, polymorphic equality compares runtime representations of value components of the modules, ignoring type components (which may raise soundness issues).

Also, this would be a chance to specify and compute the canonical type representation for OCaml; currently, deciding if two types are equal requires full-blown unification (and backtracking).

8. Future work (open_gadts.ml)

Tantalizing promises include *open* GADTs, injectivity of GADTs, first-class tagless final encodings and higher-order nested datatypes.