First-class modules: hidden power and tantalizing promises

to GADTs and beyond

Jeremy Yallop   Oleg Kiselyov
Applicative Ltd

ML 2010 – ACM SIGPLAN Workshop on ML
September 26, 2010

Honestly there was no collusion with Jacques and Alain. In fact we have learned of each other's presentation from the list of accepted papers. It is our luck that we have just heard the introduction to first-class modules from the implementors themselves. In this talk, which is joint work with Jeremy Yallop, we want to show a couple of applications of this cool new feature of OCaml. Our first application is Generally Awesome Data Types, or GADTs, which seem to have fascinated people since inception.

Outline

Monomorphic Addition

```
let add_int x y = x + y
↪ val add_int : int -> int -> int = <fun>

let add_flo x y = x +. y
↪ val add_flo : float -> float -> float = <fun>
```

We start by introducing GADTs using OCaml. We start very slowly. In OCaml, we can add integers, using regular plus, and floats, using a *different* operation, dotted plus. For reference, here are these operations and their signatures. As I said, we start very slow. Suppose my program manipulates lots of integers and floats, regarding a float as a bigger integer with sometimes less precision. Most of my algorithms don't care if they operate on ints or on floats. So, I would like to define a data type that is either an int, or a float. I would like to add such hybrid numbers, among other things.

‘Untyped’ Hybrid Numbers

```
type uif = Int of int | Flo of float
```

```
let add_uif x y =
   match (x,y) with
   | (Int x, Int y) -> Int (x + y)
   | (Flo x, Flo y) -> Flo (x +. y)
```

That is what the variant data type is for. We define a variant `uif` with two alternatives, `Int` and `Flo`. We write our addition function thusly.

‘Untyped’ Hybrid Numbers

```
type uif = Int of int | Flo of float
```

```
let add_uif x y =
   match (x,y) with
   | (Int x, Int y) -> Int (x + y)
   | (Flo x, Flo y) -> Flo (x +. y)
   | (Flo x, Int y) -> ???
```

Wish

The compiler preventing mixing up ints and floats in generic numeric algorithms, ensuring that an int can only be added to an int.

But we have missed case alternatives, as the compiler tells us. We haven't defined adding a float and an int, for example. Suppose we don't want to do that. My algorithm should deal only with integers, or only with floats, but it should not mix them. So, I would like this extra match clause to be in error. Furthermore, I would like it to be a statically determined error. I want the compiler prevent me from mixing ints and floats in generic numeric algorithms, ensuring that an int can only be added to an int. Alas, we can't obtain such a static guarantee from a variant data type.

‘Typed’ Hybrid Numbers

```
type 'a sif = Int of (int,'a) eq   * int
            | Flo of (float,'a) eq * float
```

We need a variant data type with an attribute, a type parameter, that tells us which variant is present in any instance of the data type. We need a data type like `'a sif`. It has two variants as in the untyped `uif` shown earlier. It also has a parameter `'a`. The first variant says that `'a` is actually `int`; in the second variant, `'a` is actually `float`. It says so by including the evidence, the witness `eq`.

Constructive Type Equality

```
module type EQ = sig
  type ('a, 'b) eq
  val refl : unit -> ('a, 'a) eq
  val cast : ('a, 'b) eq -> 'a -> 'b
end
```

The value of the type `('a,'b) eq` is the witness of the type equality between two types `'a` and `'b`. It is the *constructive* witness: if we obtain the witness that the type `'a` is equal to the type `'b`, we can convert any value of the type `'a` to the value of the type `'b`. After all, if these types are really the same, then the conversion is just the identity.

Constructive Type Equality

```
module type EQ = sig
  type ('a, 'b) eq
  val refl : unit -> ('a, 'a) eq
  val cast : ('a, 'b) eq -> 'a -> 'b
end

module SomeEq : EQ = struct
  type ('a, 'b) eq = 'a -> 'b
  let refl ()    = fun x -> x
  let cast eq a = eq a
end
```

Here is one, very primitive way of implementing the signature `EQ`. It gives the right idea of the implementation. But we will need more sophistication soon. For now, let us assume an implementation of `EQ` and see what we can do with it.

‘Typed’ Hybrid Numbers

```
type 'a sif = Int of (int,'a) eq * int
            | Flo of (float,'a) eq * float
```

```
let make_int (x : int) : int sif = Int (?? : (int,'a) eq,x)
```

First, we need a way to create hybrid numbers. For example, we want a function taking an integer `x` and returning the corresponding hybrid number tagged appropriately as `int sif`. Obviously we choose the `Int` alternative. What about the witness, of the type `(int,'a) eq`? Well, in our case, `'a` is int. So, we need a witness of the type `(int,int) eq`. We happen to have the value of that type: it's an instance of `refl ()`.

‘Typed’ Hybrid Numbers

```
type 'a sif = Int of (int,'a) eq * int
            | Flo of (float,'a) eq * float
```

```
let make_int x = Int (refl (), x)
↪ val make_int : int -> int sif = <fun>

let make_flo x = Flo (refl (), x)
↪ val make_flo : float -> float sif = <fun>
```

Thus, we can write `make_int` as shown. No type annotations are necessary; the desired type is inferred. We likewise inject floats.

Typed Hybrid Addition

```
let add_sif (x : 'a sif) (y : 'a sif) : 'a sif =
  match (x,y) with
  | (Int (eq,x), Int (_,y)) -> Int (eq, x + y)
  | (Flo (eq,x), Flo (_,y)) -> Flo (eq, x +. y)

↪ val add_sif : 'a sif -> 'a sif -> 'a sif = <fun>
```

We can now write the hybrid addition, roughly the way we did it for the untyped case.

Typed Hybrid Addition

```
let add_sif (x : 'a sif) (y : 'a sif) : 'a sif =
  match (x,y) with
  | (Int (eq,x), Int (_,y)) -> Int (eq, x + y)
  | (Flo (eq,x), Flo (_,y)) -> Flo (eq, x +. y)
  | (Flo ((eqf : (float,'a) eq),x),
     Int ((eqi : (int,'a) eq),y)) -> failwith "impossible"

↪ val add_sif : 'a sif -> 'a sif -> 'a sif = <fun>
```

But what about this case alternative, attempting to add a float to an integer? The argument `eqf` of the type constructor `Flo` is a witness that the type `'a` is the same as `float`. The first argument `eqi` of the `Int` data constructor likewise witnesses the equality between `'a` and `int`. But it is the same `'a`. So, in this case alternative we would have that `int` is equal to `float`, which is not possible. So, this alternative cannot actually occur. Alas, OCaml does not see this impossibility and warns about the inexhaustive pattern-match anyway. GHC, at least version 6.10 and below issues a similar warning too. So, OCaml is in a good company. So far, only dependently-typed systems like Twelf, Coq, or Agda can deduce that omitted pattern-match clauses are impossible given the type of the expression.

Twomorphic Addition

```
↪ val add_sif : 'a sif -> 'a sif -> 'a sif = <fun>

add_sif (make_int 1) (make_int 2)
↪ - : int sif = Int (<abstr>, 3)

add_sif (make_flo 1.) (make_flo 2.)
↪ - : float sif = Flo (<abstr>, 3.)

add_sif (make_int 1) (make_flo 2.)
Error: This expression has type float sif
but an expression was expected of type int sif
```

Here are a few examples of using the hybrid addition. We can add two integers or two floats. Attempting to mix ints and floats leads to a type error, as we always wanted.

The operation `add_sif` looks, from its type, and acts, from the examples below, like truly polymorphic addition. However, the polymorphism is not over all types; only over two types, int and float. We have gained something like a bounded, enumerable polymorphism: n-morphism.

Hybrid-scalar Addition

```
let scalar_add_sif (x : 'a sif) (y : 'a) : 'a sif
```

This was all too simple. One may almost think that phantom types would have sufficed. They won't. To see that, let's look at even simpler example: adding to a hybrid number an unboxed number, so to speak. We want the function `scalar_add_sif` of the shown signature.

Hybrid-scalar Addition

```
let scalar_add_sif (x : 'a sif) (y : 'a) : 'a sif =
  match x with
  | Int (eqi,x) -> Int (eqi, x + y)
  | Flo (eqf,x) -> Flo (eqf, x +. y)
```

We may attempt to write it as before.

Hybrid-scalar Addition

```
let scalar_add_sif (x : 'a sif) (y : 'a) : 'a sif =
  match x with
  | Int (eqi,x) -> Int (eqi, x + y:int)
  | Flo (eqf,x) -> Flo (eqf, x +. y:float)
```

Phantom types would not do

But we’ve got a problem. In the first clause, `y`, as an argument of the integer addition, must be of the type `int`. Likewise, in the second clause, `y` must be a float. But it is the same `y`, the second argument of the function. It can’t be both an int and a float. A phantom-type solution would grind to a halt.

Hybrid-scalar Addition

```
let scalar_add_sif (x : 'a sif) (y : 'a) : 'a sif =
  match x with
  | Int ((eqi : (int,'a) eq), x) ->
        Int (eqi, x + cast ((symm eqi) : ('a,int) eq) y)

  | Flo (eqf,x) -> Flo (eqf, x +. cast (symm eqf) y)
↪ - : 'a sif -> 'a -> 'a sif = <fun>

scalar_add_sif (make_int 1) 2
↪ - : int sif = Int (<abstr>, 3)
scalar_add_sif (make_flo 1.) 2.
↪ - : float sif = Flo (<abstr>, 3.)
scalar_add_sif (make_flo 1.) 2
Error: This expression has type int but an expression was
expected of type float
```

Let’s look at the problem again. The annotation says that the argument `y` is of some type `'a`. If the `Int` pattern-match succeeds, we obtain the proof that `'a` is `int`. We can use this proof to convince the type checker that `y` is an integer in that case. First we use the symmetry property of equality (which I haven’t shown you yet but hopefully you believe in it) and then we use the cast to constructively say that `y` is an integer and can be added with other integers. Likewise, in the second clause we use a different proof to show that `y` is a float, and can be added to a float. As you can see, `y` can be both an int and a float – only not at the same time. This code does type-check, and we obtain the desired polymorphic, twomorphic type. None of the type annotations are actually needed. The function works as expected.

Outline

Leibniz Equality Wanted

```
let incr_arr (x : 'a sif) (y : 'a array)
```

But we want more. We not only wish to add a hybrid number to an unboxed number; we also wish to add a hybrid number to any collection of unboxed numbers, for example, an array. We want the function `inrc_arr` of the shown signature.

Leibniz Equality Wanted

```
let incr_arr_typeclass
      (plus : 'a -> 'a -> 'a) (x : 'a) (y : 'a array) =
  for i = 0 to pred (Array.length y) do
    y.(i) <- plus y.(i) x
  done



let incr_arr (x : 'a sif) (y : 'a array) =
  match x with
  | Int (eq,x) ->
     incr_arr_typeclass (+) x (cast (symm eq) y)
...

Error: This expression has type 'a array
       but an expression was expected of type 'a
```

We try to write it like the `scalar_add_sif` before. But now the type checker complains. Indeed, the evidence `eq` says that the type `'a` is equal to `int`. But we need to prove that `'a array` is equal to `int array`. By the way we also illustrate another approach to bounded polymorphism: type classes with a dictionary (evidence) passing.

Leibniz Equality Still Wanted

```
let incr_arr (x : 'a sif) (y : 'a array) =
  let cast_array (type s) (type t)
                 (eq: (s,t) eq) (x: s array) : t array
     = Array.map (cast eq) x ???
  in match x with
  | Int (eq,x) ->
     incr_arr_typeclass (+) x (cast_array (symm eq) y)
...
```

Instead of just `cast` we need a function `cast_array` of the shown signature. At first blush, it seems easy: we just map the conversion function. First of all, this is slow: we have to rebuild the entire array doing essentially nothing. Second: we have to rebuild the entire array. The result is a physically different array! To update the original array in-place we need two more copying operations. This solution becomes less and less satisfactory. Finally, we may be given a collection with no mapping function. We need truly the Leibniz equality. First-class modules give it to us.

Constructive Type Equality (in full)

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

module type EQ = sig
  type ('a, 'b) eq
  val refl : unit -> ('a, 'a) eq          Reflexivity Axiom

  module Subst (TC : TyCon) : sig         Leibniz Substitution
    val subst : ('a, 'b) eq -> ('a TC.tc, 'b TC.tc) eq
    (* ∀tc : (∗ → ∗). α = β implies α tc = β tc *)
  end

  val cast : ('a, 'b) eq -> 'a -> 'b      Constructive type eq.

end
```

Now, this is the complete definition. As before, `cast` is the constructive proof of the type equality, letting us convert the value of the type `'a` to the value of the type `'b`. The value `refl` constructively expresses the reflexivity axiom, and `Subst` expresses the Leibniz substitution principle: equals can be substituted for equals. The module type `TC` used by `Subst`, is, as you have guessed it, a type *constructor*. So, `Subst` says that given the proof of equality of `'a` and `'b` we can compute the proof of equality of `'a tc` and `'b tc` for any type constructor tc. Type constructor is to be understood a bit loosely, as we shall see. Essentially, `tc` is a function on types, not necessarily injective.

Leibniz Equality Apprehended

```
let incr_arr (x : 'a sif) (y : 'a array) =
  let cast_array (type s) (type t)
                 (eq: (s,t) eq) (y: s array) : t array

     = let module S =
            Subst(struct type 'a tc = 'a array end) in
       cast ((S.subst eq) : ('a array, int array) eq) y

  in match x with
  | Int (eq,x) ->
     incr_arr_typeclass (+) x (cast_array (symm eq) y)
  | Flo (eq,x) ->
     incr_arr_typeclass (+.) x (cast_array (symm eq) y)

↪ val incr_arr : 'a sif -> 'a array -> unit = <fun>
```

Now that we have Leibniz substitution, we can use as follows. We instantiate the Leibniz substitution to the particular type constructor: `array`. Then `S.subst eq` is the witness of the type equality between array types. We use the witness to tell that `y` is really an array of integers or an array of floats.

Pop quiz: why do we need this new notation for the polymorphic functions, `(type s)` and `(type t)`. Could we used ordinary type variables, like `'s` and `'t`?

Leibniz Equality Apprehended

```
↪ val incr_arr : 'a sif -> 'a array -> unit = <fun>

let y = [|1;2;3|] in incr_arr (make_int 1) y; y;;
↪ - : int array = [|2; 3; 4|]

let y = [|1.;2.;3.|] in incr_arr (make_flo 1.) y; y;;
↪ - : float array = [|2.; 3.; 4.|]
```

Here are the examples of using our array incrementation. Of course we get a type error if we try to mix up integers and floats.

Outline

Rising up the ranks

```
type ('a,'b) coll = Arr of ('b array, 'a) eq * 'b array
                  | Lst of ('b list, 'a) eq  * 'b list
```

```
let make_coll_arr x = Arr (refl(), x)
↪ val make_coll_arr : 'a array -> ('a array, 'a) coll

let appendcu (x : ('a,'b) coll) (y : 'a) =
  match x with
  | Arr (eq,x) ->
      Arr (eq, Array.append x (cast (symm eq) y))
  | Lst (eq,x) ->
      Lst (eq, List.append x (cast (symm eq) y))
↪ val appendcu : ('a, 'b) coll -> 'a -> ('a, 'b) coll
```

The pattern we've seen generalizes further. We can define a collection of elements abstracting not only over the type of the elements but also over the type of the collection, list or an array, for example. We can build sample collections with `make_coll_arr` or the similar `make_coll_lst` (not shown). We can append a collection to an array or list, `appendcu`.

Rising up the ranks

```
type ('a,'b) coll = Arr of ('b array, 'a) eq * 'b array
                  | Lst of ('b list, 'a) eq  * 'b list
```

```
let add_head (x: ('a,'b) coll) (y:'b sif) =
  let add op eq x y = cast eq (op (cast (symm eq) x) y)
  in match (x,y) with
  | (Lst (eql,xh::xt), Int(eqi,y)) ->
      Lst(eql, (add (+) eqi xh y)::xt)
...
↪ val add_head : ('a, 'b) coll -> 'b sif -> ('a, 'b) coll
```

`(int list, float) coll` is not a populated instance of `('a, 'b) coll`

Or we can increment the element at the head of the collection with the given hybrid number. The types ensure that the increment and the elements of the collection have the same type. We combine the two GADTs, for numbers and for the collections of numbers.

We see a pattern: the type is more polymorphic, but only few alternatives are available. Not all instantiations of a type schema are populated.

Outline

Injectivity

`('a,'b) eq` $\Longrightarrow$ `('a tc, 'b tc) eq`

We have seen that from the equality of two types `’a` and `’b` we can constructively deduce the equality of `’a tc` and `’b tc` for any user-given type constructor `tc`. What about the converse?

Injectivity

```
('a,'b) eq ⟹ ('a tc, 'b tc) eq

type 'a tc = 'a array
('a,'b) eq ⟸ ('a tc, 'b tc) eq ???
```

It could be very useful: if two array types are the same, their element types are the same. We should be able to use that fact.

Injectivity

('a,'b) eq $\Longrightarrow$ ('a tc, 'b tc) eq

```
type 'a tc = int
```

('a,'b) eq $\Longleftarrow$ ('a tc, 'b tc) eq ???

Alas, things are a bit complex: the converse does not always hold. We may need type-checker's internal knowledge about injectivity – which is, unfortunately, not available to us.

Injectivity

`('a,'b) eq` $\Longrightarrow$ `('a tc, 'b tc) eq`

`('a,'b) eq_weak` $\Longleftarrow$ `('a tc, 'b tc) eq`
only for functor `tc`

So far, injectivity holds only for functors `tc` (with the mapping function) and it only gives us a weak Leibniz equality (that is, without the `Subst` operation).

Outline

GADT Introduction

Leibniz Equality

Rising up the ranks

Injectivity?

► **Implementation of Leibniz Equality**

Generic Programming

Conclusions

Implementation of EQ

```
(* data EqTC a b = Cast{cast :: ∀ tc. tc a -> tc b} *)
module type EqTC = sig
  type a and b
  module Cast : functor (TC : TyCon) -> sig
     val cast : a TC.tc -> b TC.tc
  end
end

type ('a, 'b) eq =
   (module EqTC with type a = 'a and type b = 'b)

let refl (type t) () = (module struct
  type a = t and b = t
  module Cast (TC : TyCon) = struct
    let cast v = v end
end : EqTC with type a = t and type b = t)
```

Briefly, here is the real implementation of the signature `EQ`. The witness of the type equality `('a,'b') eq` is a first-class module, of the type `EqTC`, containing the types in question and the ability to convert from `a tc` to `b tc` for *any* type constructor `tc` given by the user. The module `EqTC` is equivalent to this Haskell data type, familiar from Baars and Swierstra's "Typing Dynamic Typing". The function `refl` constructs the first-class module witnessing the equality of the type `'a` with itself. We certainly can convert the value of any `'a tc` to itself, using the identity function.

Substituting

```
let cast (type s) (type t) s_eq_t =
  let module S_eqtc = (val s_eq_t :
         EqTC with type a = s and type b = t) in
  let module C = S_eqtc.Cast(struct type 'a tc = 'a end)
  in C.cast

module Subst (TC : TyCon) = struct
 let subst (type s) (type t) s_eq_t = (module struct
  type a = s TC.tc and b = t TC.tc
  module S_eqtc = (val s_eq_t :
      EqTC with type a = s and type b = t)
  module Cast (SC : TyCon) = struct
    module C = S_eqtc.Cast(struct
        type 'a tc = 'a TC.tc SC.tc end)
    let cast = C.cast
    end
  end : EqTC with type a = s TC.tc and type b = t TC.tc)
end
```

There is a lot of boiler-plate in this code. Yes, we have to repeat this signature `EqTC with type a =` over and over again.

Outline

- GADT Introduction
- Leibniz Equality
- Rising up the ranks
- Injectivity?
- Implementation of Leibniz Equality
- ▶ **Generic Programming**
- Conclusions

Really Generic Programming

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

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

Our examples already hinted at the generic programming. Our hybrid numbers contained the representation of the type of the value, in the constructors `Int` or `Flo`, and the values themselves. But we can separate the two. By “really generic” (as opposed to the use of objects), we specify the type representation without committing to any particular interpretation. There are no methods for `show` or `add` in `Repr`. Rather, `Repr` receives the interpretation from the user and applies it. A generic function such as `show` takes a type representation `s repr` and supplies the `Interpretation` argument to obtain a function whose type involves `s`. Please see the paper and the code for more detail.

Outline

What else

- Existentials via first-class modules, including existentials over higher-kinded types
- Leibniz equality
- Common examples of GADTs: typed formatting, typed interpreter
- A generic programming library (EMGM-like)
- Towards open GADTs: extensible evaluator for a typed object language

What else is there? The accompanying code also talks about these issues.

Conclusions

First-class modules

- bring type constructors, setting the way for $F\omega$
- represent existentials directly
- permit higher-rank and higher-kind polymorphism
- offer “generic programming for OCaml masses”

GADTs in OCaml

+ value-restricted polymorphism
- limited injectivity

Interesting things are possible, but not convenient

`http://okmij.org/ftp/ML/first-class-modules/`

First-class modules make type-constructors almost first-class and permit abstraction over type constructors (e.g., quantification over them). Quantifying over type constructors gives us polymorphism of higher-kind. We get on the road to $F\omega$. We can define the genuine Leibniz equality and so implement GADTs modulo injectivity.
First-class modules directly express existentials, which are often needed for GADT programming. We can instantiate type variables with module types containing existentials and universals, which gives us higher-rank polymorphism.
We see overall pattern: restricted polymorphism, restricted at the value level so to speak: the type may be polymorphic, like our collections, but only some instances of that type can really be created. Not all instantiations of a type schema are populated types.
Injectivity, alas, holds only for 'weak' GADTs.

Overall conclusion: we can do interesting things, at the cost of lots of boilerplate and contorted code. Hopefully the users will notice the first part and the OCaml team will especially notice the second part of the conclusion.