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

Outline

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

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>
```

‘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)
```

'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.

‘Typed’ Hybrid Numbers

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

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
```

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
```

‘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)
```

‘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>
```

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>
```

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>
```

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
```

Hybrid-scalar Addition

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

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)
```

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

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
```

Outline

GADT Introduction

▶ **Leibniz Equality**

Rising up the ranks

Injectivity?

Implementation of Leibniz Equality

Generic Programming

Conclusions

Leibniz Equality Wanted

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

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
```

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)
...
```

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
```

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>
```

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.|]
```

Outline

GADT Introduction

Leibniz Equality

▶ **Rising up the ranks**

Injectivity?

Implementation of Leibniz Equality

Generic Programming

Conclusions

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
```

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`

Outline

GADT Introduction

Leibniz Equality

Rising up the ranks

▶ **Injectivity?**

Implementation of Leibniz Equality

Generic Programming

Conclusions

Injectivity

```
('a,'b) eq ⟹ ('a tc, 'b tc) eq
```

Injectivity

```
('a,'b) eq ⟹ ('a tc, 'b tc) eq

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

Injectivity

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

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

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

Injectivity

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

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

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)
```

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
```

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
```

Outline

GADT Introduction

Leibniz Equality

Rising up the ranks

Injectivity?

Implementation of Leibniz Equality

Generic Programming

▶ **Conclusions**

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

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/`