Structural Types, Recursive Modules, and the Expression Problem

Jacques Garrigue
Nagoya University, Grad. Sch. of Mathematics

Synopsis

- The Expression Problem
- Polymorphic variant solution
- Recursive modules
- Private row types
- Solution using recursive modules and private row types
- Other applications of private row types

The Expression Problem

Extending a language of expressions, by adding both new cases and new operations, without recompilation, but with static safety. Studied in the 1990s by Cook, Felleisen, ..., formalized by Wadler.

– adding new cases is easy with OO, but hard with algebraic datatypes

– adding new operations is easy with algebraic datatypes, but hard with OO

It looks like an expressiveness benchmark for object-oriented languages, but functional programming languages might actually be better.

Flasback to WG2.8 2000

I proposed a solution based on polymorphic variants and a delayed fixpoint.

```
type 'a eplus = [`Num of int | `Plus of 'a * 'a]
let eval_plus eval_rec : 'a eplus -> int = function
    `Num n -> n
  | `Plus(e1,e2) -> eval_rec e1 + eval_rec e2
val eval_plus : ('a -> int) -> 'a eplus -> int
let rec eval1 e = eval_plus eval1 e
val eval1 : ('a eplus as 'a) -> int

eval1 (`Plus(`Num 3, `Num 4))
- : int = 7
```

This requires recursion at both type and value level.

Solution (cont.)

Extension uses special “expanded” patterns.

```
type ’a emult = [’a eplus | ‘Mult of ’a * ’a]
let eval_mult eval_rec : ’a emult -> int = function
    #eplus as e -> eval_plus eval_rec e
  | ‘Mult(e1,e2) -> eval_rec e1 * eval_rec e2
val eval_mult : (’a -> int) -> ’a emult -> int
let rec eval2 e = eval_mult eval2 e
val eval2 : (’a emult as ’a) -> int

eval2 (‘Plus(‘Mult(‘Num 3,‘Num 4), ‘Num 5))
- : int = 17
```

```
[ ‘Mult of ’a * ’a | ‘Num of int | ‘Plus of ’a * ’a ]
(‘Num _ | ‘Plus _)
```

Advanced operation: simplification

Simplification illustrates the need to pattern-match on unknown types, and to produce values of these types.

```
let simp_plus simp_rec : 'a eplus -> ([> 'a eplus] as 'a) =
  function `Num _ as e -> e
  | `Plus(e1,e2) -> match simp_rec e1, simp_rec e2 with
        `Num m, `Num n -> `Num (m+n)
      | e12 -> `Plus e12
val simp_plus : (([> 'a eplus ] as 'a) -> 'a) -> 'a eplus -> 'a
let rec simp1 e = simp_plus simp1 e
val simp1 : ('a eplus as 'a) -> 'a

simp1 (`Plus(`Num 3, `Num 4))
- : 'a eplus as 'a = `Num 7
```

means [`Num _ | `Plus _ | ..]

A good first solution

– No simple object-oriented solution at that time

– Shorter than any solution presented later

– Allows pattern-matching

Yet,

– Relies heavily on type inference

– Not very scalable

Hard to solve in core language

```
type ('a,'b) ops = { eval: 'a -> int; simp: 'a -> 'b }
let ops_plus ops : ('a eplus, [> 'a eplus] as 'a) ops =
  { eval = (function `Num n -> n
                   | `Plus(e1,e2) -> ops.eval e1 + ops.eval e2);
    simp = ... }
val ops_plus :
  ([> 'a eplus ] as 'a, 'a) ops -> ('a eplus, 'a) ops
let rec plus = ops_plus plus
This kind of expression is not allowed
as right-hand side of `let rec'
```

– value-recursion problem

– still relies heavily on type inference

Hard to solve in core language

```
type ('a,'b) ops = { eval: 'a -> int; simp: 'a -> 'b }
let ops_plus ops : ('a eplus, [> 'a eplus] as 'a) ops =
  { eval = (function `Num n -> n
                   | `Plus(e1,e2) -> ops.eval e1 + ops.eval e2);
    simp = ... }
val ops_plus :
  ([> 'a eplus ] as 'a, 'a) ops -> ('a eplus, 'a) ops
let lazy_ops ops =
  { eval = (fun x -> (Lazy.force ops).eval x);
    simp = (fun x -> (Lazy.force ops).simp x) }
val lazy_ops : ('a, 'b) ops Lazy.t -> ('a, 'b) ops
let rec lazy_plus = lazy (ops_plus (lazy_ops lazy_plus))
val lazy_plus : ('a eplus as 'a, 'a) ops Lazy.t
let plus = Lazy.force lazy_plus
val plus : ('a eplus as 'a, 'a) ops
```

Scalable fixpoint = Recursive module

Recursive modules seem the way to go, as they provide

- fixpoints of functors, with
- scalability (simultaneous parameterization by multiple types and functions)
- explicit typing (through signatures)

But with no abstraction on type structure, this can be rather unwieldly.

Modules without private rows (1/2)

```
(* Types involved in our recursion *)
module type ET = sig type exp end
(* Recursive operations on our types *)
module type E = sig
  module T : ET
  val eval : T.exp -> int
  val simp : T.exp -> T.exp
end
(* A functor building the type of the
   conversion module between two types *)
module CW(U : ET)(L : ET) = struct
  module type C = sig
    val inj : L.exp -> U.exp
    val proj : U.exp -> L.exp option
  end
end
(* The identity conversion module *)
module CI = struct
  let inj x = x
  let proj x = Some x
end
```

```
module PlusT(T : ET) = struct
  type exp = [`Num of int | `Plus of T.exp * T.exp]
end
module Plus(E : E)(C : CW(E.T)(PlusT(E.T)).C) =
struct
  module T = PlusT(E.T)
  let eval : T.exp -> int = function
      `Num n -> n
    | `Plus(e1,e2) -> E.eval e1 + E.eval e2
  let simp : T.exp -> E.T.exp = function
      `Num _ as e -
    | `Plus(e1,e2) -
        let e1 = E.simp e1 and e2 = E.simp e2 in
        match C.proj e1, C.proj e2 with
          Some(`Num m), Some(`Num n) ->
            C.inj (`Num(m+n))
        | _ -> C.inj (`Plus(e1,e2))
end
module rec PlusF : E with module T = PlusT(PlusF.T)
  = Plus(PlusF)(CI)
let e1 = PlusF.simp (`Plus(`Num 3, `Num 4))
val e1 : PlusF.T.exp = `Num 7
```

E.T.exp

Modules without private rows (2/2)

```
(* The Mult language *)

module MultT(T : ET) = struct
  type exp =
    [PlusT(T).exp | `Mult of T.exp * T.exp]
end
module Mult(E : E)(C : CW(E.T)(MultT(E.T)).C) =
struct
  module T = MultT(E.T)
  module CPlus = struct
    type t = PlusT(E.T).exp
    (* All conversion modules use the same
       code, but we need to know the concrete
       types to build our coercions *)
    let inj = (C.inj :> t -> _)
    let proj x = match C.proj x with
      Some #t as y -> y
    | _ -> None
  end
  module LPlus = Plus(E)(CPlus)
```

```
  let eval : T.exp -> int = function
      #LPlus.T.exp as e -> LPlus.eval e
    | `Mult(e1,e2) -> E.eval e1 * E.eval e2
  let simp : T.exp -> E.T.exp = function
      #LPlus.T.exp as e -> LPlus.simp e
    | `Mult(e1,e2) ->
        let e1 = E.simp e1
        and e2 = E.simp e2 in
        match C.proj e1, C.proj e2 with
          Some(`Num m), Some(`Num n) ->
            C.inj (`Num(m*n))
        | _ -> C.inj (`Plus(e1,e2))
end
module rec MultF
  : E with module T = MultT(MultF.T)
  = Mult(MultF)(CI)

let e2 = MultF.simp
  (`Plus(`Mult(`Num 3,`Num 4), `Num 5))
val e2 : MultF.T.exp = `Num 17
```

Private row types

Object or variant types whose “row-variable” is kept abstract.

```
type basic = [‘Int of int | ‘String of string]
module Prop(X : sig type t = private [> basic] end) = struct
  let empty : X.t = ‘String ""
  let to_string (v : X.t) = match v with
      ‘Int n    -> string_of_int n
    | ‘String s -> s
    | _ -> "other" (* required by the abstract row *)
end
type extra = [basic | ‘Bool of bool]
module MyProp = Prop(struct type t = extra end)
module MyProp :
  sig val empty : extra val to_string : extra -> string end
```

What are private rows?

Assume an over-simplified model of structural polymorphism, where the polymorphism is expressed by a row variable.

```
[ ‘Int of int | ‘String of string | ’a ]
```

Then a private row type is just a pair of type declarations

```
type t_row
type t = [ ‘Int of int | ‘String of string | t_row ]
```

and we can just use the normal behaviour of functors.

In practice however, row variables would not be sufficient to expressed all the forms of structural polymorphism we support. The real formalization uses kinds rather than rows, and some non trivial modifications are needed.

Recursive modules with private rows (1/2)

```
module type E =
  sig type exp  val eval : exp -> int  val simp : exp -> exp end
type 'a eplus = [`Num of int | `Plus of 'a * 'a]
module PF(X: E with type exp = private [> 'a eplus] as 'a) = struct
  type exp = X.exp eplus
  let eval : exp -> int = function
      `Num n -> n
    | `Plus(e1,e2) -> X.eval e1 + X.eval e2
  let simp : exp -> X.exp = function
      `Num _ as e -> e
    | `Plus(e1,e2) -> match X.simp e1, X.simp e2 with
          `Num m, `Num n -> `Num(m+n)
        | e12 -> `Plus e12
end
module rec Plus : (E with type exp = Plus.exp eplus) = PF(Plus)
let e1 = Plus.simp (`Plus(`Num 3, `Num 4))
val e1 : Plus.exp = `Num 7
```

Recursive modules with private rows (2/2)

```
type 'a emult = ['a eplus | `Mult of 'a * 'a]

module MF(X: E with type exp = private [> 'a emult] as 'a) = struct
  type exp = X.exp emult
  module Plus = PF(X)
  let eval : exp -> int = function
      #Plus.exp as e -> Plus.eval e
    | `Mult(e1,e2) -> X.eval e1 * X.eval e2
  let simp : exp -> X.exp = function
      #Plus.exp as e -> Plus.simp e
    | `Mult(e1,e2) -> match X.simp e1, X.simp e2 with
          `Num m, `Num n -> `Num(m*n)
        | e12 -> `Mult e12
end
module rec Mult : (E with type exp = Mult.exp emult) = MF(Mult)
let e2 = Mult.simp (`Plus(`Mult(`Num 3,`Num 4), `Num 5))
val e2 : Mult.exp = `Num 17
```

About this solution

– Types are clearer: no core level polymorphism

– Still a rather complex idiom: fixpoint of functor

– Using functors does not introduce so much boilerplate:
being partly concrete helps a lot

– Small weakness: type recursion has to be done at the core level

– Hopefully anybody can write this code

Other applications of private rows

Independently of their used in functorial polymorphism, private rows have a number of other applications.

– Private types (i.e. protected constructors.)
  They were introduced for records and variants in ocaml 3.07. Private rows can do the same thing for polymorphic variants and object types.

– Private and friend methods in classes
  Objective Caml notion of private method is local to an object. With private rows we define methods local to a module.

– Ensure extensibility of variant types
  Useful for code versioning for instance.

Private types (protected constructors)

```
module Relative : sig
  type t = private [< ‘Zero | ‘Pos of int | ‘Neg of int]
  val inj : int -> t
end = struct
  type t = [‘Zero | ‘Pos of int | ‘Neg of int]
  let inj n =
    if n = 0 then ‘Zero else
    if n > 0 then ‘Pos n else ‘Neg (-n)
end
let one : Relative.t = ‘Pos (-1)
This expression has type [> ‘Pos of int ] but is here used
with type Relative.t
let to_int (x : Relative.t) =
  match x with ‘Zero -> 0 | ‘Pos n -> n | ‘Neg n -> (-n)
val to_int : Relative.t -> int
```

Module-private methods (1/2)

Structural typing allows breaking the usual encoding of friend methods.

```
module OSet : sig
  type 'a t
  class ['a] c : object ('b)
    method contents : 'a t
    method add : 'a -> 'b
    method elements : 'a list
    method union : 'b -> 'b
  end
end
class ['a] broken (s : 'a OSet.c) : ['a] OSet.c = object
  inherit ['a] OSet.c
  method contents = s#contents
end
```

Module-private methods (2/2)

With private row types we can (partially) encode private methods.

```
module OSet : sig
  type 'a c = private
    < add : 'a -> 'b; elements : 'a list;
      union : 'b -> 'b; .. > as 'b
  val create : unit -> 'a c
end
```

– The `contents` method is completely hidden

– `OSet.c` is now a nominal type, whose values can only be created through `OSet.create`

– Pitfall: inheritance cannot be used, as the class is not exported.

Conclusion

- Private row types compined with recursive modules provide a scalable solution to the expression problem
- This solution is easier to compare to the one in Scala for instance
- Modules need not be all that verbose
- As always, abstraction has many (unexpected) applications

See draft paper for technical details and more examples.