This document is also available in Postscript.
Table of modules

- Exp: data-centric encoding.
- Exp1: operation-centric encoding.
- Exp2: another operation-centric encoding.

This is another solution in OCaml to the *Independently Extensible Solutions to the Expression Problem.* as described by Zenger and Odersky in their Technical Report Nr. 200433, March 2004.

This code has been check with OCaml 3.08. It uses anonymous classes in *DblePlusNegTest*. This could have been expanded into toplevel class definitions in an earlier version of OCaml, which allowed closed classes (classes where self had a closed object type). The recent versions of OCaml do not allow this but for anonymous classes).

The code is then a mere translation of the original code in Scala, where type annotations have been removed. Hence, the code is in general shorter than in scala with the only exception of using private methods for sharable instance variables discussed above.

Jacques Garrigue and Didier Rémy.

Module Exp

3 Object-oriented style

3.2 Framework

```
module Base = struct
  class exp = object end

  class num v = object
    inherit exp
    val value : int = v
    method eval = value
  end
end
```

```
module BaseTest = struct
  let e = new Base.num 7
  let _ = Printf.printf "e␣is␣%d\n" (e#eval)
end
```

3.3 Data extensions

```
module BasePlus = struct
  class plus l r = object
    inherit Base.exp
    val left = l
    val right = r
    method eval = left#eval + right#eval
  end
end
```

```
module BaseNeg = struct
  class neg t = object
    inherit Base.exp
    val term = t
    method eval = 0 − term#eval
  end
end
```

Combining Independent Extensions

```
module BasePlusNeg = struct
  include Base
  include BasePlus
  include BaseNeg
end
```

3.4 Operation Extensions

```
module Show = struct
  class num v = object
    inherit Base.num v
    method show = string_of_int value
  end
end
```

Linear extensions

```
module ShowPlusNeg = struct
  include Show
  class plus l r = object
    inherit BasePlusNeg.plus l r
    method show =
      Printf.sprintf "(%s + %s)" (left#show) (right#show)
  end
  class neg t = object
    inherit BasePlusNeg.neg t
    method show = Printf.sprintf "-%s" (term#show)
  end
end
```

```
module ShowPlusNegTest = struct
  open ShowPlusNeg
  let e = new neg (new plus (new num 7) (new num 6))
  let _ = Printf.printf "%s = %d\n" (e#show) (e#eval)
end
```

Tree-transformer extensions

```
module DblePlusNeg = struct
  class virtual num v = object (self : 'a)
    inherit BasePlusNeg.num v
    method private virtual num : int -> 'a
    method dble = self#num (value * 2)
  end
```

```
  class virtual plus l r = object (self : 'a)
    inherit BasePlusNeg.plus l r
    method private virtual plus : 'a → 'a → 'a
    method dble = self#plus (left#dble) (right#dble)
  end
  class virtual neg t = object (self : 'a)
    inherit BasePlusNeg.neg t
    method private virtual neg : 'a → 'a
    method dble = self#neg (term#dble)
  end
end

module DblePlusNegTest = struct
  open DblePlusNeg
  let rec num v = object inherit num v method private num = num end
  and plus l r = object inherit plus l r method private plus = plus end
  and neg t = object inherit neg t method private neg = neg end

  let e = plus (neg (plus (num 1) (num 3))) (num 2)
  let _ = Printf.printf "e␣*␣2␣is␣-4␣?␣%d\n" (e#dble#eval)
end
```

It is useless to check for type errors... OCaml is sound!

Combining independent extensions

```
module ShowDblePlusNeg = struct
  class virtual num v = object
      inherit ShowPlusNeg.num v
      inherit DblePlusNeg.num v
  end
  class virtual plus l r = object
      inherit ShowPlusNeg.plus l r
 inherit DblePlusNeg.plus l r
  end
  class virtual neg t = object
      inherit ShowPlusNeg.neg t
      inherit DblePlusNeg.neg t
  end
end
```

ShowDblePlusNeg uses multiple inheritance of two classes built from a common ancestor. This is a known difficulty when using multiple inheritance, since the state of the common ancestor is being dupplicated in the two subclasses. For instance, objects of the class *ShowDblePlusNeg.num* will contained two occurrences of field *value*. This is fine here because instance variables are not mutable and fields are not updated, so they can be freely dupplicated: the two fields are filled with (and retain) the same initial value v.

OCaml does not offer any primitive construct to deal with this situation. However, a simple solution is to make all read, write and update to instance variables of the shared class go indirectly through private methdods. Since methods definitions are overridden during inheritance, all methods will then refer to the same instance variable—the one defined last—and unused dupplicates will passively sit in the state of the object. Ths is not very elegant, but it works well. A small extension of the language with annotations on instance variables could be used to drive the inheritance of instance variables and avoid the use of private methods.

We have not used this schema here because objects are purely functional.

Section 5: Binary methods

Binary methods are rarely a problem in OCaml...

```
module Equals = struct
  class exp = object
    inherit Base.exp
    method isNum (v : int) = false
  end
  class num v = object (self : 'a)
    inherit exp
    inherit Base.num v
    method eql (other : 'a) = other#isNum v
    method isNum v = v = value
  end
end
```

5.1 Data extensions

```
module EqualsPlusNeg = struct
  class exp = object (self : 'a)
    inherit Equals.exp
```

```
    method isNum (v : int) = false
    method isPlus (l : 'a) (r : 'a) = false
    method isNeg (t : 'a) = false
  end

  class num v = object
    inherit exp
    inherit Equals.num v
  end

  class plus l r = object (self : 'a)
    inherit exp
    inherit BasePlusNeg.plus l r
    method isPlus l r = left#eql l ∧ right#eql r
    method eql (other : 'a) = other#isPlus (left) (right)
  end

  class neg t = object (self : 'a)
    inherit exp
    inherit BasePlusNeg.neg t
    method isNeg t = term#eql t
    method eql (other : 'a) = other#isNeg (term)
  end
end
```

5.2 Operation extensions

```
module EqualsShowPlusNeg = struct
  class num v = object
    inherit EqualsPlusNeg.num v
    inherit ShowPlusNeg.num v
  end

  class plus l r = object
    inherit EqualsPlusNeg.plus l r
    inherit ShowPlusNeg.plus l r
  end

  class neg t = object
    inherit EqualsPlusNeg.neg t
    inherit ShowPlusNeg.neg t
  end
```

```
end

module EqualsShowPlusNegTest = struct
  open EqualsShowPlusNeg
  let t1 = new plus (new num 1) (new num 2)
  let t2 = new plus (new num 1) (new num 2)
  let t3 = new neg (new num 2)
  let _ =
    Printf.printf "%s = %s ? %b\n" (t1#show) (t2#show) (t2#eql t2);
    Printf.printf "%s = %s ? %b\n" (t1#show) (t3#show) (t2#eql t3)
end
```

Module Exp1

This module provides a functional decomposition (operation-centric view) of the expression problem.

> *Many class type definitions and type annotations could be omitted. We keep them to provide an early check on the interfaces of the classes we define.*

```
module FBase = struct
  class type ['v] exp = object
    method accept : 'v → unit
  end
  class ['v] num value = object (_ : 'v #exp)
    method accept v = v#visitNum value
  end
  class type visitor = object
    method visitNum : int → unit
  end
  class ['e] eval = object (self : #visitor)
    val mutable result = 0
    method private return x =
      result ← x
    method apply (t : _ #exp as 'e) =
      t#accept self; result
    method visitNum value =
      self#return value
  end
end
```

In visitors (such as eval), we used a private method *#return* to store the result. This way the pair apply/return will work properly even if the field *result* is shadowed.

```
module FBasePlus = struct
  class type ['e] visitor = object
    inherit FBase.visitor
    method visitPlus : 'e → 'e → unit
  end
  class ['v] plus l r = object (_ : ('e #visitor as 'v) #FBase.exp as 'e)
    method accept v =
      v#visitPlus l r
  end
  class ['e] eval = object (self : 'e #visitor)
    inherit ['e] FBase.eval
    method visitPlus l r =
      self#return (self#apply l + self#apply r)
  end
end

module FBaseNeg = struct
  class type ['e] visitor = object
    inherit FBase.visitor
    method visitNeg : 'e → unit
  end
  class ['v] neg t = object (_ : ('e #visitor as 'v) #FBase.exp as 'e)
    method accept v =
      v#visitNeg t
  end
  class ['e] eval = object (self : 'e #visitor)
    inherit ['e] FBase.eval
    method visitNeg t =
      self#return (− (self#apply t))
  end
end

module FBasePlusNeg = struct
  class type ['e] visitor = object
    inherit ['e] FBasePlus.visitor
    inherit ['e] FBaseNeg.visitor
  end
  class ['e] eval = object (self : 'e #visitor)
    inherit ['e] FBasePlus.eval
```

```
    inherit ['e] FBaseNeg.eval
  end
end
```

The definition of class *eval* raises warnings. We can ignore them as *result* is only used by the apply/return pair of methods.

```
module FShowPlusNeg = struct
  open FBasePlusNeg
  class ['e] show = object (self : 'e #visitor)
    val mutable result = ""
    method private return x =
      result <- x
    method apply (t : 'e) =
      t#accept self; result
    method visitNum v =
      self#return (string_of_int v)
    method visitPlus l r =
      self#return ("("^ self#apply l ^"+"^ self#apply r ^")")
    method visitNeg t =
      self#return ("(-" ^ self#apply t ^")")
  end
end

module FShowTest = struct
  open FBase
  open FBasePlusNeg
  open FShowPlusNeg
  let eval =
    let e = new eval in
    (e#apply : ('a eval exp as 'a) -> _ :> ('b visitor exp as 'b) -> _)

  let show =
    let s = new show in
    (s#apply : ('a show exp as 'a) -> _ :> ('b visitor exp as 'b) -> _)

  open FBasePlus
  open FBaseNeg
  let e = new plus (new neg (new plus (new num 1) (new num 2))) (new num 3)
  let () = Printf.printf "%s = %d\n" (show e) (eval e)
end
```

The *eval* and *show* classes above contain the method *#apply*, with different types. Since expressions and vistors have mutually recursive types, this would make it impossible to use

both visitors on the same expression. Fortunately, the recursion is covariant, and we can coerce to forget the apply method in expression types.
Below, a new problem arises in class *dble*, as visitors return expressions. We can no longer use the covariance of the recursion. We choose to make #*apply* private, so that it no longer appears in the object type. We can extract the #*apply* method by using an out parameter.

```
module FDblePlusNeg = struct
  open FBasePlusNeg
  class ['e] dble apply = object (self : 'e #visitor)
    val mutable result = None
    method private apply (t : 'e) : 'e =
      t#accept self;
      match result with Some x → x | None → assert false
    initializer apply := self#apply
    method private return x =
      result ← Some x
    method visitNum v =
      self#return (new FBase.num (v × 2))
    method visitPlus l r =
      self#return (new FBasePlus.plus (self#apply l) (self#apply r))
    method visitNeg t =
      self#return (new FBaseNeg.neg (self#apply t))
  end
end

module FDbleTest = struct
  open FBase
  open FBasePlusNeg
  open FDblePlusNeg
  let dble =
    let apply = ref (fun _ → assert false) in
    ignore (new dble apply);
    !apply
  open FShowTest
  (* We reuse the expression from the previous test, multiplying nums by 2 *)
  let e = dble e
  let () = Printf.printf "%s␣=␣%d\n" (show e) (eval e)
end
```

We create a stub for the #*apply* method, and extract it as a side-effect of object creation

Module Exp2

This is another approach to the functional decompsition of the expression problem.

Rather than using coercions later, we hide the #*apply* method from the beginning. All visitors inherit from *accumulators*, and are called via *extract*

```
module FBase = struct
  class type ['v] exp = object
    method accept : 'v → unit
  end
  class ['v] num value = object (_ : 'v #exp)
    method accept v = v#visitNum value
  end
  (* f is an out parameter *)
  class virtual ['a, 'e] accumulator f = object (self : 's)
    val mutable result = None
    method private return x =
      result ← Some x
    method private apply (t : 's #exp as 'e) =
      t#accept self;
      match result with Some x → x | None → assert false
    initializer f := self#apply
  end
  class ['e] eval f = object (self)
    inherit [int,'e] accumulator f
    method visitNum value =
      self#return value
  end
  let extract cons =
    let f = ref (fun _ → assert false) in
    cons f; !f
  (* We could use this as let eval = extract (new eval) *)
end

module FBasePlus = struct
  class ['e] eval f = object (self)
    inherit ['e] FBase.eval f
    method visitPlus l r =
      self#return (self#apply l + self#apply r)
  end
  (* Since we have hidden #apply, #eval is the same as #visitor *)
  class ['v] plus l r = object (_ : ('e #eval as 'v) #FBase.exp as 'e)
```

```
    method accept v =
      v#visitPlus l r
  end
end

module FBaseNeg = struct
  class ['e] eval f = object (self)
    inherit ['e] FBase.eval f
    method visitNeg t =
      self#return(− (self#apply t))
  end
  class ['v] neg t = object (_ : ('e #eval as 'v) #FBase.exp as 'e)
    method accept v =
      v#visitNeg t
  end
end

module FBasePlusNeg = struct
  class ['e] eval f = object (self)
    inherit ['e] FBasePlus.eval f
    inherit ['e] FBaseNeg.eval f
  end
end

module FShowPlusNeg = struct
  open FBasePlusNeg
  class ['e] show f = object (self)
    inherit [string,'e] FBase.accumulator f
    method visitNum v =
      self#return (string_of_int v)
    method visitPlus l r =
      self#return ("("^ self#apply l ^"+"^ self#apply r ^")")
    method visitNeg t =
      self#return ("(-" ^ self#apply t ^")")
  end
end

module FShowTest = struct
  open FBase
  open FBasePlusNeg
  open FShowPlusNeg
  let eval = extract (new eval)
  let show = extract (new show)
```

```
  open FBasePlus
  open FBaseNeg
  let e = new plus (new neg (new plus (new num 1) (new num 2))) (new num 3)
  let () = Printf.printf "%s = %d\n" (show e) (eval e)
end

module FDblePlusNeg = struct
  class ['e] dble f = object (self)
    inherit ['e, 'e] FBase.accumulator f
    method visitNum v =
      self#return (new FBase.num (v × 2))
    method visitPlus l r =
      self#return (new FBasePlus.plus (self#apply l) (self#apply r))
    method visitNeg t =
      self#return (new FBaseNeg.neg (self#apply t))
  end
end

module FDbleTest = struct
  open FBase
  open FBasePlusNeg
  open FDblePlusNeg
  let dble = extract (new dble)

  open FShowTest
  let e = dble e
  let () = Printf.printf "%s = %d\n" (show e) (eval e)
end
```