Generic polymorphism on steroids
or
How to Solve the Expression Problem with Polymorphic Variants

Claudio Sacerdoti Coen

<claudio.sacerdoticoen@unibo.it>

Dipartimento di Informatica – Scienza e Ingegneria
Alma mater studiorum • Università di Bologna

28 March 2015

Content of the talk

The Expression Problem

- An archetipal recurring problem in programming.
- Widely used to justify object oriented programming.
- Very hard for most non object oriented functional languages

Polymorphic Variants

- A simple and natural programming language construct
- Elegantly solves the expression problem
- The solution is better than the OO one
- Requires and pushes to its limits generic polymorphism
- Available in OCaml only

Content of the talk

The Expression Problem

- An archetipal recurring problem in programming.
- Widely used to justify object oriented programming.
- Very hard for most non object oriented functional languages

Polymorphic Variants

- A simple and natural programming language construct
- Elegantly solves the expression problem
- The solution is better than the OO one
- Requires and pushes to its limits generic polymorphism
- Available in OCaml only

Outline

Outline

Principle of Uniformity

Every datum that can be

- passed to a function
- returned by a function
- stored in a variable or data structure

has the same constant size (tipically one or two words).

In OCaml:

- *primitive data are all stored in a word*
- *complex data are manipulated by reference (again a word)*

Principle of Uniformity

Every datum that can be

- passed to a function
- returned by a function
- stored in a variable or data structure

has the same constant size (tipically one or two words).

In OCaml:

- *primitive data are all stored in a word*
- *complex data are manipulated by reference (again a word)*

Generic (or parametric) polymorphism

Code that manipulates values without "using" them works identically on values belonging to different data types.

The types of generic parameters are universally quantified at the definition site, and instantiated at usage site.

```
let swap (x,y) = (y,x)         (* val swap: ∀α, β. α * β → β * α *)
                               (* val swap: 'a *'b → 'b *'a *)
let n,m = swap (2,3)           (* 'a ← int, 'b ← int *)
let n,s = swap ("Foo",0)       (* 'a ← string, 'b ← int *)
```

Sufficient condition: principle of uniformity

Generic (or parametric) polymorphism

Code that manipulates values without "using" them works identically on values belonging to different data types.

The types of generic parameters are universally quantified at the definition site, and instantiated at usage site.

```
let swap (x,y) = (y,x)           (* val swap: ∀α, β. α * β → β * α *)
                                 (* val swap: 'a *'b → 'b *'a *)
let n,m = swap (2,3)             (* 'a ← int, 'b ← int *)
let n,s = swap ("Foo",0)         (* 'a ← string, 'b ← int *)
```

Sufficient condition: principle of uniformity

Type Inference and ML

Type inference:

- the user writes untyped code
- the system infers the most general type

```
(* val f: 'a*int*('a*'b) → 'b*int *)
let f (x,y,z) = if x=fst z then (snd z,y) else (snd z,y+1)
```

General case: undecidable.

Hindley-Milner (ML):

- universal quantifiers only in prenex position
- type checking becomes decidable and efficient in practice

Type Inference and ML

Type inference:

- the user writes untyped code
- the system infers the most general type

```
(* val f: 'a*int*('a*'b) → 'b*int *)
let f (x,y,z) = if x=fst z then (snd z,y) else (snd z,y+1)
```

General case: undecidable.

Hindley-Milner (ML):

- universal quantifiers only in prenex position
- type checking becomes decidable and efficient in practice

Benefits

Parametric polymorphism + type inference:

+ Extremely concise code
+ Improved reusability
+ Resilience to changes in datatypes
- (Rarely) hard to understand typing errors

Types are assigned by globally analyzing the usage of data.

Errors are reported locally:

x has type S but it is expected to have type T

- S may be correct, and the error is elsewhere
- S and T may be extremely large expressions

Benefits

Parametric polymorphism + type inference:

+ Extremely concise code
+ Improved reusability
+ Resilience to changes in datatypes
- (Rarely) hard to understand typing errors

Types are assigned by globally analyzing the usage of data.

Errors are reported locally:

x has type S but it is expected to have type T

- S may be correct, and the error is elsewhere
- S and T may be extremely large expressions

Algebraic Data Types

ADT = recursive disjoint labelled union of products of types.

```
type 'a list = Nil | Cons of 'a * 'a list
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
type expr = Var of string | Const of int | Plus of expr * expr
let x = Cons (1, Cons (2, Nil))   (* The list [1;2] *)
let e = Plus (Var "x", Const 2)   (* The expression x+2 *)
```

Pattern matching:

```
let rec len x = match x with Nil → 0 | Cons (_,y) → 1 + len y
```

Coverage check: all cases must be considered exactly once.

Algebraic Data Types

ADT = recursive disjoint labelled union of products of types.

```
type 'a list = Nil | Cons of 'a * 'a list
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
type expr = Var of string | Const of int | Plus of expr * expr
let x = Cons (1, Cons (2, Nil))  (* The list [1;2] *)
let e = Plus (Var "x", Const 2)  (* The expression x+2 *)
```

Pattern matching:

```
let rec len x = match x with Nil → 0 | Cons (_,y) → 1 + len y
```

Coverage check: all cases must be considered exactly once.

Algebraic Data Types

ADT = recursive disjoint labelled union of products of types.

```
type 'a list = Nil | Cons of 'a * 'a list
type 'a tree = Leaf of 'a | Node of 'a tree * 'a * 'a tree
type expr = Var of string | Const of int | Plus of expr * expr
let x = Cons (1, Cons (2, Nil))  (* The list [1;2] *)
let e = Plus (Var "x", Const 2)  (* The expression x+2 *)
```

Pattern matching:

```
let rec len x = match x with Nil → 0 | Cons (_,y) → 1 + len y
```

Coverage check: all cases must be considered exactly once.

Nominal vs Structural typing

Structural typing:
two types are equal iff they expand to equal expressions

Nominal typing:
one type is only equal to itself
(the name/location of the declaration matters)

Mixed in ML:

```
(* type abbreviations: t1 is equal to t2 *)
type t1 = int * string
type t2 = int * string

(* type declarations: list1 is incompatible with list2 *)
type 'a list1 = Nil | Cons of 'a * 'a list1
type 'a list2 = Nil | Cons of 'a * 'a list2
```

Nominal vs Structural typing

Structural typing:
two types are equal iff they expand to equal expressions

Nominal typing:
one type is only equal to itself
(the name/location of the declaration matters)

Mixed in ML:

```
(* type abbreviations : t1 is equal to t2 *)
type t1 = int * string
type t2 = int * string

(* type declarations : list1 is incompatible with list2 *)
type 'a list1 = Nil | Cons of 'a * 'a list1
type 'a list2 = Nil | Cons of 'a * 'a list2
```

Outline

Why the Expression Problem?

- An archetipal recurring problem in programming.
- Widely used to justify object oriented programming.
- Very hard for most non object oriented functional languages

The solution with object oriented programming:

Uses:

- Inheritance
- Subtype polymorphism
- Dynamic dispatch
- Open recursion (via self)

Does not use:

- Incapsulation

We will do without all of the above!

Why the Expression Problem?

- An archetipal recurring problem in programming.
- Widely used to justify object oriented programming.
- Very hard for most non object oriented functional languages

The solution with object oriented programming:

Uses:

- Inheritance
- Subtype polymorphism
- Dynamic dispatch
- Open recursion (via self)

Does not use:

- Incapsulation

We will do without all of the above!

Why the Expression Problem?

- An archetipal recurring problem in programming.
- Widely used to justify object oriented programming.
- Very hard for most non object oriented functional languages

The solution with object oriented programming:

Uses:

- Inheritance
- Subtype polymorphism
- Dynamic dispatch
- Open recursion (via self)

Does not use:

- Incapsulation

We will do without all of the above!

The Expression Problem

Two problems to be solved reusing existent code
without in place modification or cut&paste

1. Given a data type for expressions, and functions on them
   - Add new functions
   - Add new constructors
2. Given two data types for expressions, and functions on them
   - Define the non disjoint union

```
type e = Const of int | Plus of e * e

let eval e =
 match e with
    Const n → n
  | Plus(e1,e2) → eval e1 + eval e2
```

```
type e = Mult of e * e

let eval e =
  match e with
   Mult(e1,e2) →
     eval e1 * eval e2
```

The Expression Problem in Java

```
interface E { int Eval(); }

class Const implements E {
    public int n;
    Const(int n) { this.n = n; }
    public int Eval() { return n; }
}

class Plus implements E {
    public E l,r;
    Plus(E l, E r) { this.l = l; this.r = r; }
    public int Eval() { return l.Eval() + r.Eval(); }
}
```

Too verbose and obfuscated.

Too open: what if I want to restrict to certain expressions?

The Expression Problem in Java

```
interface E { int Eval(); }

class Const implements E {
    public int n;
    Const(int n) { this.n = n; }
    public int Eval() { return n; }
}

class Plus implements E {
    public E l,r;
    Plus(E l, E r) { this.l = l; this.r = r; }
    public int Eval() { return l.Eval() + r.Eval(); }
}
```

Too verbose and obfuscated.
Too open: what if I want to restrict to certain expressions?

The Expression Problem in ML

```
type e = Const of int | Plus of e * e

let eval e =
 match e with
    Const n -> n
  | Plus(e1,e2) -> eval e1 + eval e2
```

```
type e = Mult of e * e

let eval e =
  match e with
   Mult(e1,e2) ->
     eval e1 * eval e2
```

```
type e = Const of int | Plus of e * e | Mult of e * e

let eval e =
 match e with
    Const n -> n
  | Plus(e1,e2) -> eval e1 + eval e2
  | Mult(e1,e2) -> eval e1 * eval e2
```

Cut & paste
no code reused

The Expression Problem in ML

```
type e = Const of int | Plus of e * e

let eval e =
 match e with
    Const n -> n
  | Plus(e1,e2) -> eval e1 + eval e2
```

```
type e = Mult of e * e

let eval e =
  match e with
   Mult(e1,e2) ->
     eval e1 * eval e2
```

```
type e = Const of int | Plus of e * e | Mult of e * e

let eval e =
 match e with
    Const n -> n
  | Plus(e1,e2) -> eval e1 + eval e2
  | Mult(e1,e2) -> eval e1 * eval e2
```

Cut & paste
no code reused

The Expression Problem in ML

First problem: closed recursion

Opening the recursion:

```
type 'a e = Cons of int | Plus of 'a * 'a

(* val eval: ('a → int) → 'a e → int *)
let eval f x =
 match x with
    Cons n → n
  | Plus(e1,e2) → f e1 + f e2
```

Coverage still checked

Open recursion can be closed at any time:

```
type e1 = e1 e

(* val eval1: e1 → int *)
let rec eval1 x = eval eval1 x
```

No unwanted elements

The Expression Problem in ML

First problem: closed recursion

Opening the recursion:

```
type 'a e = Cons of int | Plus of 'a * 'a

(* val eval: ('a → int) → 'a e → int *)
let eval f x =
 match x with
     Cons n → n
   | Plus(e1,e2) → f e1 + f e2
```

Coverage still checked

Open recursion can be closed at any time:

```
type e1 = e1 e

(* val eval1: e1 → int *)
let rec eval1 x = eval eval1 x
```

No unwanted elements

The Expression Problem in ML

First problem: closed recursion

Opening the recursion:

```
type 'a e = Cons of int | Plus of 'a * 'a

(* val eval: ('a → int) → 'a e → int *)
let eval f x =
 match x with
     Cons n → n
   | Plus(e1,e2) → f e1 + f e2
```

Coverage still checked

Open recursion can be closed at any time:

```
type e1 = e1 e

(* val eval1: e1 → int *)
let rec eval1 x = eval eval1 x
```

No unwanted elements

The Expression Problem in ML

Problems:

- Closed recursion: fixed
- Lack of extensibility:
  an ADT fixes once and for all the constructors

How to add a constructor to

type 'a e = Cons **of int** | Plus **of** 'a * 'a

without cut&paste?

No solution in ML. Approximated solution, only for first formulation:

type ('a,'b) e = Cons **of int** | Plus **of** 'a * 'a | More **of** 'b

Needs an empty type to close the recursion (not in ML). Symmetrically merging two datatypes is impossible.

The Expression Problem in ML

Problems:

- Closed recursion: fixed
- Lack of extensibility:
  an ADT fixes once and for all the constructors

How to add a constructor to

type 'a e = Cons **of int** | Plus **of** 'a * 'a

without cut&paste?

No solution in ML. Approximated solution, only for first formulation:

type ('a,'b) e = Cons **of int** | Plus **of** 'a * 'a | More **of** 'b

Needs an empty type to close the recursion (not in ML).
Symmetrically merging two datatypes is impossible.

Outline

Polymorphic variants

Decompose ADT into two constructions:

1. Labelling of products of types
2. Non disjoint union of types

ADTs up to structural equality can be recovered.

```
(* val x : [> `A of string * int ] *)
let x = `A ("foo",2)

(* val m : [> `A of string * int | `B of int | `C] list *)
let m = [ `A ("foo",2) ; `B 4 ; `C ]
```

Polymorphic variants

Decompose ADT into two constructions:

1. Labelling of products of types
2. Non disjoint union of types

ADTs up to structural equality can be recovered.

```
(* val x : [> `A of string * int ] *)
let x = `A ("foo",2)

(* val m : [> `A of string * int | `B of int | `C] list *)
let m = [ `A ("foo",2) ; `B 4 ; `C ]
```

Typing Polymorphic Variants

```
(* val x : [> `A of string * int ] *)
let x = `A ("foo",2)
```

[> ‘A **of** string $*$ **int**] is the most general type for ‘A ("foo",2)
How to read it?

- $\simeq$ It is of the form $\forall\alpha.$[‘A **of** string $*$ **int** $|$ α] for any union of tagged types α
- $+$ It is of the form $\forall\alpha.\alpha$ such that
  1. α is a union of tagged types
  2. α includes at least [‘A **of** string $*$**int**]

  (cfr. bounded polymorphism)
- \- It is of the form $\forall\alpha.\alpha$ for α being a subtype of [‘A **of** string $*$**int**]
  (no: implicit subtyping violates principal typing)

Typing Polymorphic Variants

It is of the form $\forall\alpha.$ [`A **of** string * **int** | α]
for any union of tagged types α

```
(* val x : [> `A of string * int ]              Generic type *)
let x = `A of ("foo",2)
(* val y : [`A of string * int | `B]             α ←[`B] *)
let y = (x : [`A of string * int | `B])
```

WARNING: in OCaml (t : T) is not a cast.
It is a type instantiation operation:

- It checks if T is an instance of the generic type of t
- It behaves as t
- It is typed as T

Typing Polymorphic Variants

It is of the form $\forall\alpha.\alpha$ such that

1. α is a union of tagged types
2. α includes at least [‘A **of** string ∗**int**]

(cfr. bounded polymorphism)

```
(* val x : [> ‘A of string * int ]          Generic type *)
let x = ‘A of ("foo",2)
(* val y : [‘A of string * int | ‘B]        α ←[‘A of string * int | ‘B] *)
let y = (x : [‘A of string * int | ‘B])
```

Generic Polymorphism on Steroids

```
(* val f :
   [< `A of int * 'a | `B of 'b | `C ] ->
   [> `Ko | `Ok of int ] *)
let f x =
 match x with
     `A (n,m) -> `Ok (n+1)
   | `B n     -> `Ok 0
   | `C       -> `Ko
```

Inferred type: $\forall \alpha, \beta.\ \alpha \to \beta$ such that

- α has at most `A, `B, `C (of the given types)
- β has at least `Ok, `Ko (of the given types)

Coverage check $\Rightarrow$ inference of most generic type

Generic Polymorphism on Steroids

```
(* val f :
   [< `A of int * 'a | `B of 'b | `C ] ->
   [> `Ko | `Ok of int ] *)
let f x =
 match x with
     `A (n,m) -> `Ok (n+1)
   | `B n     -> `Ok 0
   | `C       -> `Ko
```

Inferred type: $\forall \alpha, \beta.\ \alpha \to \beta$ such that

- α has at most ‘A, ‘B, ‘C (of the given types)
- β has at least ‘Ok, ‘Ko (of the given types)

Coverage check $\Rightarrow$ inference of most generic type

Generic Polymorphism on Steroids

```
(* val f : ([> `Chihuaua | `Dog | `Maltese ] as 'a) → 'a *)
let rec f x =
 match x with
     `Chihuaua → `Dog
   | `Maltese → `Dog
   | y → y
```

Inferred type: $\forall\alpha.\alpha \to \alpha$ such that
α has at least ‘Chihuaua, ‘Dog, ‘Maltese.

Generic Polymorphism on Steroids

```
(* val f: [> `A of int | `B ] → int *)
(* val g: [> `A of int | `C ] → bool *)

(* val h: [> `A of int | `B | `C ] → int * bool *)
let h x = (f x, g x)
```

$[>$ ‘A **of int** $|$ ‘B $|$ ‘C$] \simeq \forall\alpha.[$‘A **of int** $|$ ‘B $|$ ‘C $| \alpha]$
is the most general instance of both
$[>$ ‘A **of int** $|$ ‘B $] \simeq \forall\beta.[$‘A **of int** $|$ ‘B $| \beta]$ (by $\beta \leftarrow [$‘C $| \alpha]$)
$[>$ ‘A **of int** $|$ ‘C $] \simeq \forall\gamma.[$‘A **of int** $|$ ‘C $| \gamma]$ (by $\gamma \leftarrow [$‘B $| \alpha]$)

The “non disjoint union” possible because the type of ‘A was the same.

Generic Polymorphism on Steroids

```
(* val f: [> `A of int | `B ] -> int *)
(* val g: [> `A of int | `C ] -> bool *)

(* val h: [> `A of int | `B | `C ] -> int * bool *)
let h x = (f x, g x)
```

$[>$ ‘A **of int** $|$ ‘B $|$ ‘C$] \simeq \forall\alpha.[$‘A **of int** $|$ ‘B $|$ ‘C $| \alpha]$
is the most general instance of both
$[>$ ‘A **of int** $|$ ‘B $] \simeq \forall\beta.[$‘A **of int** $|$ ‘B $| \beta]$ (by $\beta \leftarrow [$‘C $| \alpha]$)
$[>$ ‘A **of int** $|$ ‘C $] \simeq \forall\gamma.[$‘A **of int** $|$ ‘C $| \gamma]$ (by $\gamma \leftarrow [$‘B $| \alpha]$)

The “non disjoint union” possible because the type of ‘A was the same.

Outline

The Expression Problem Solved

Problems:

- Closed recursion: fixed (close later)
- Lack of extensibility: fixed (polymorphic variants)

The Expression Problem Solved

```
type 'a e =
 [ `Cons of int
 | `Plus of 'a * 'a ]

(* val eval_e:
   ('a -> int) -> [< 'a e] -> int *)
let eval_e h x =
 match x with
    `Cons n -> n
  | `Plus(e1,e2) -> h e1 + h e2
```

```
type 'a f =
  [ `Mult of 'a * 'a ]

(* val eval_f:
   ('a -> int) -> [< 'a f] -> int *)
let eval_f h x =
 match x with
  | `Mult(e1,e2) -> h e1 + h e2
```

```
(* The non disjoint union of e and f *)
type 'a ef = [ 'a e | 'a f ]
```

#t matches all constructors of t

```
(* val eval_ef:
   ('a -> int) -> [< 'a ef ] -> int *)
let eval_ef f x =
 match x with
    #e as y -> eval_e f y
  | #f as y -> eval_f f y
```

The Expression Problem Solved

```
(* The non disjoint union of e and f *)
type 'a ef = [ `a e | `a f ]
```

#t matches all constructors of t

```
(* val eval_ef:
   ('a -> int) -> [< 'a ef ] -> int *)
let eval_ef f x =
 match x with
    #e as y -> eval_e f y
  | #f as y -> eval_f f y
```

Open recursion can be closed at any time:

```
(* The only inhabitants are `Con, `Plus, `Mult *)
type expr = expr ef

(* val eval: [< expr] -> int *)
let rec eval x = eval_ef eval x
```

The Expression Problem Solved

```
(* The non disjoint union of e and f *)
type 'a ef = [ `a e | `a f ]
```

#t matches all constructors of t

```
(* val eval_ef:
   ('a -> int) -> [< 'a ef ] -> int *)
let eval_ef f x =
 match x with
    #e as y -> eval_e f y
  | #f as y -> eval_f f y
```

Open recursion can be closed at any time:

```
(* The only inhabitants are `Con, `Plus, `Mult *)
type expr = expr ef

(* val eval: [< expr] -> int *)
let rec eval x = eval_ef eval x
```

Outline

Conclusions

- Polymorphic variants decompose ADT naturally:
  1. Tagged values
  2. Non disjoint unions
- Polymorphic Variants + Generic Polymorphism + Type Inference + Nominal Typing rocks!
- Who needs Inheritance, Subtype Polymorphism, Dynamic Dispatch and Open Recursion?

  No inheritance/subtyping $\Rightarrow$ most general types, type inference
  Static dispatch $\Rightarrow$ more efficiency
  Closed recursion $\Rightarrow$ static checks, coverage
  Much simpler semantics, easier to implement

What about duality?

Polymorphic variants = non disjoint union of tagged values

- + structural typing
- + generic polymorphism and type inference

Extensible records = duplicate-merging products of tagged values (fields)

- + structural typing
- + generic polymorphism and type inference

```
(* val stats :
  {> height : int & width : int} →
  {< perimeter : int & area : int} *)
let stats { height=h ; width=w } =
  { perimeter=2*(h+w) ; area=h*w }
```

Similar to objects but much simpler

What about duality?

Polymorphic variants = non disjoint union of tagged values

- + structural typing
- + generic polymorphism and type inference

Extensible records = duplicate-merging products of tagged values (fields)

- + structural typing
- + generic polymorphism and type inference

```
(* val stats :
  {> height : int & width : int} →
  {< perimeter : int & area : int} *)
let stats { height=h ; width=w } =
  { perimeter=2*(h+w) ; area=h*w }
```

Similar to objects but much simpler