Code reuse through polymorphic variants*

Jacques Garrigue

November 8, 2000

Abstract

Their support for code reuse has made object-oriented languages popular. However, they do not succeed equally in all areas, particularly when data has a complex structure, making hard to keep the parallel between data and code.

On the other hand, functional programming languages, which separate data from code, are better at handling complex structures, but they do not provide direct ways to reuse code for a different datatype.

We show here a way to achieve code reuse, even when data and code are separated. The method is illustrated by a detailed example.

1 Introduction

Design of complex programs sometimes bump into a difficult dilemma: which should be privileged of type safety and code reuse. Maintaining complete type safety forces to specialize the code, making it difficult to reuse it in similar situations. Decrease in reuse may mean more difficult maintenance and tracing of logical bugs. On the other hand, trading safety for more reuse also means weaker help from the type system during maintenance, and potential runtime type errors.

A typical instance of this dilemma is variant types. The way they are supported in typed functional languages, with strong typing and pattern matching, is a tremendous help when developping and maintaining a program. The type checker is able to detect all potential type errors, while the match compiler can warn about forgotten or redundant cases, which is particularly useful when modifying a variant type. This means a very good support for code reuse... but only through textual copying, and without subtyping. If you do not want to end-up duplicating code working on incompatible types, you have to weaken your discipline, and allow for potential match failures.

Object-oriented languages only support variant types through either `instanceof` statements or visitor patterns. The first case means that most of the type checking is deferred to runtime, while the second case is as rigid as the typed functional approach. Both cases lack the exhaustivity checking available in functional languages, and often force one to scatter code in various objects, which makes the program harder to understand than in the functional approach.

Objective Caml 3 [LDG$^+$00], which is a typed functional language with object-oriented extensions, provides a solution to this problem through polymorphic variants, a much more flexible form of variant types. We show here how they can be applied to a concrete example, the incremental construction of an evaluator for a lambda-calculus extended with arithmetic operations.

*Presented at FOSE 2000, Sasaguri, Japan

The idea for this example originally comes from a post by Phil Wadler on the Java-Genericity mailing list [W$^+$98], in which he proposed a solution to the Expression Problem, that is the problem of extending a variant type with new constructors without recompiling code for old ones. A similar problem and solution can be found in [KFF98], but untyped. It appeared later that Wadler's solution, which already supposed an extension of Generic Java, itself an extension of Java, could not be typed. Didier Rémy, Jérôme Vouillon and myself finally came up with a typable solution in an object-oriented extension of the 3rd order typed lambda-calculus, which was later checked correct by Haruo Hosoya in F-omega-sub-rec. On the other hand, I could provide a much shorter solution in about 15 lines of Objective Label using polymorphic variants, which were merged later in Objective Caml. The solution used only standard ML-polymorphism, which is a weakened 2nd order typed lambda-calculus, and can be inferred automatically.

Rather than presenting here the solution to the expression problem, which from the point of view of polymorphic variants would be only a toy example, we choose a full fledge problem, addressing issues like use of functionals as iterators, and multiple inheritance with a shared base class. Hopefully this should give a better idea of the expressive power of polymorphic variants.

2 Variants and functional programming

Algebraic datatypes, also known as sums or variants, are one of the essential features of strongly typed functional programming languages.

Usually such a type is defined once, and then many functions are defined to handle it throughout the program.

A good example of frequently used variant type is lists[1]:

```
type 'a list = Nil | Cons of 'a * 'a list
```

This defines two constructors, `Nil` and `Cons`, with different arities and argument types. The type itself is parametric in `'a`, the type of elements of the list. Here is a list of integers[2]:

```
let l = Cons (1, Cons (2, Nil))
```
val l : int list

One can then define functions on this datatype, for instance computing the length of a list.

```
let rec length = function
    Nil -> 0
  | Cons(hd, tl) -> 1 + length tl
```
val length : 'a list → int

Since the length does not depend on the type of elements, this function is naturally given a polymorphic type. That is, it can be applied indifferently to any instance of list.

Even more interesting, since functions are first class, it is very easy to define various kind of iterators, or *functionals* that will take a function as argument and apply it appropriately to the contents of the type. Such iterators, even when they actually build a new list, can still be polymorphic in both the type of the list and the type of the result.

[1] All examples in this paper are processed by Objective Caml 3.00 (`http://caml.inria.fr/ocaml/`).

[2] In examples, lines starting with *val* (printed in italic) are not part of the program, but are type information output by the compiler. They are mixed with the input to help understanding, since programs themselves contain very little type information.

```
let rec map f = function
    Nil -> Nil
  | Cons(hd, tl) -> Cons (f hd, map f tl)
```
val map : ('a → 'b) → 'a list → 'b list

Functionals play an important role in avoiding code duplication, by factoring most operations into a structural and a logical part.

One may get the wrong idea that, since variant types are defined only once, and do not admit any form of inheritance, they are difficult to extend. In fact, thanks to the power of type checking and match analysis, this is just the contrary: you can easily add a case to an existing variant type, and the compiler will help you step-by-step to find all the places that must be modified.

For instance, suppose that we want to extend our lists to handle concatenation in $O(1)$. We can add a new constructor `Conc`:

```
type 'a list = Nil | Cons of 'a * 'a list | Conc of 'a list * 'a list
```

Now if we try to recompile our functions, we get the following warning.

```
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
Conc (_, _)
```

Following these warnings, one just has to add missing cases were they are needed, so that the scattering of these definitions around the program is not a problem.

```
let rec length = function
    Nil -> 0
  | Cons(hd, tl) -> 1 + length tl
  | Conc(l1, l2) -> length l1 + length l2
```
val length : 'a list → int

3 Polymorphic variants

While variants have proved to be a very powerful tool for program development and maintenance, their strength is also their weakness. Their staticness, which allows for thorough checking, also means that the only possible way to reuse code is textual copy. Experience shows that this is practical enough in many cases, but from the point of view of software engineering, one would prefer to be actually able to share the code between different extensions of the same variant type.

Polymorphic variants [Gar98] were introduced to permit a more flexible use of variants. The main conceptual difference is that constructors are no longer tied to a single type, but do exist independently of type definitions. For instance one can use the value `` `Var "x" `` without any previous definition.

```
let v = `Var "x"
```
val v : `` [> `Var of string] ``

In Objective Caml [LDG+00], polymorphic variants constructors are only distinguished from usual variants by a backquote "`" in front of their name. Types are written in a similar way, but they are wrapped into square brackets, and type descriptions may appear directly

Table 1: Variables, common to all languages

```
type var = [‘Var of string]

let eval_var sub (‘Var s as v : var) =
  try List.assoc s sub with Not_found -> v
```
*val eval_var : (string * (*`[> ‘Var of string]` *as 'a)) list → var → 'a*

inside normal type expressions, like for function types or tuples. The ">" at the left of the above type means that this type is not fully specified: it may actually contain other constructors. Of course, one may also define fully specified types, like appears in table 1, and later use them to constrain such partially specified values.

Symmetrically to values, pattern matchings are also given partially specified types, to allow polymorphism.

```
let f = function ‘Var s -> String.length s | ‘Num n -> n
val f : [< ‘Var of string | ‘Num of int] → int
```

The above type means that `f` accepts both `‘Var` and `‘Num` as input, and in a polymorphic way.

Finally, pattern matching is extended in order to allow dispatching according to sets of constructors, rather than on an individual basis. This is actually the combination of two mechanisms, one which uses `as`-patterns to refine types, and another which allows to use directly type names in patterns.

```
let f g h = function #var as x -> g x | ‘Num _ as y -> h y
val f :
  ([> ‘Var of string] → 'a) → ([> ‘Num of 'b] → 'a) →
  [< ‘Var of string | ‘Num of 'b] → 'a
```

Here `#var` is derived from the type definition of `var`, and stands as a shorthand for `‘Var(_ : string)`. This explains why `string` appears in the type, while `‘Num` has no specified argument type. The magic part is that we have split the `‘Var` and `‘Num` cases into two independent types, and passed them to the functions `g` and `h`.

This is all we will use in our main example.

4 Application: a modular evaluator of extended lambda-calculus

This example is intended to illustrate the expressive power of polymorphic variants with respect to traditional variants, and typed object-oriented languages. It demonstrates most of the capacity of the system, but had to be kept simple to fit into this paper: there is only one operation, `eval`, and the expression is not decorated. In the real-size example, multiple operations with dependencies are cleanly encoded through the Objective Caml class system, and variant dispatch still works on a decorated tree.

The code is a complete: tables 1 to 4 include all the code of the program, which can be directly compiled with Objective Caml 3.00. You just have to remove phrases starting by *val*, which are output from the compiler.

Table 2: The lambda calculus

```
type 'a lambda = [`Var of string | `Abs of string * 'a | `App of 'a * 'a]

let gensym = let n = ref 0 in fun () -> incr n; "_" ^ string_of_int !n

let eval_lambda eval_rec subst : 'a lambda -> 'a = function
    #var as v -> eval_var subst v
  | `App(l1, l2) ->
      let l2' = eval_rec subst l2 in
      begin match eval_rec subst l1 with
        `Abs(s, body) ->
          eval_rec [s,l2'] body
      | l1' ->
          `App (l1', l2')
      end
  | `Abs(s, l1) ->
      let s' = gensym () in
      `Abs(s', eval_rec ((s,`Var s')::subst) l1)
val eval_lambda :
  ((string *
    ([> `Var of string | `Abs of string * 'a | `App of 'a * 'a] as 'a))
   list -> 'a -> 'a) ->
  (string * 'a) list -> 'a lambda -> 'a

let rec eval1 subst = eval_lambda eval1 subst
val eval1 : (string * ('a lambda as 'a)) list -> 'a -> 'a
```

First we define a basic language containing only variables (table 1). Evaluation for this language means looking in an environment for a binding associated to the variable name, keeping it as is if there is none. Notice that since we have used dispatch (on a single case), the type of `v` in `eval_var` is `[> `Var of string]` and not `var`. The form `... as 'a` denotes that this type is shared between the environment list, and the result of `eval_var`.

Then we define lambda-calculus by extending this basic language (table 2). Notice that we are careful to define `lambda` as an open recursive type (subterms are of type `'a`), and to use open recursion in `eval_lambda`, to be able to extend this definition later. The first case of `eval_lambda` is just delegating to `eval_var`. Since the return type of `eval_var` is polymorphic, it is compatible with other branches, which may return `` `Abs `` or `` `App ``. Again you can see the same pattern as in `eval_var`, the input type of `eval_lambda` is `lambda`, but its output type is `` [> `Var of string | `Abs of string * 'a | `App of 'a * 'a] as 'a ``, which is the extensible version of `lambda`. Finally we build a specific evaluator `eval1` for `lambda` by closing the recursion. Notice that this also closes the recursion at the type level: both input and output types are now `'a lambda as 'a`.

```
let e1 = eval1 [] (`App(`Abs("x",`Var"x"), `Var"y"))
val e1 : 'a lambda as 'a = `Var "y"
```

Along the same lines we define the `expr` language, which adds numbers, addition and multiplication to the basic language (table 3). As for traditional variants, it comes handy to define a map function, that uniformly applies a function to the subterms of an expression.

Table 3: The expr language of arithmetic expressions

```
type 'a expr =
  [`Var of string | `Num of int | `Add of 'a * 'a | `Mult of 'a * 'a]

let map_expr (f : _ -> 'a) : 'a expr -> 'a = function
    #var as v -> v
  | `Num _ as n -> n
  | `Add(e1, e2) -> `Add (f e1, f e2)
  | `Mult(e1, e2) -> `Mult (f e1, f e2)
val map_expr :
  (([> `Var of string
     | `Num of int
     | `Add of 'a * 'a
     | `Mult of 'a * 'a] as 'a) → 'a) → 'a expr → 'a

let eval_expr eval_rec subst (e : 'a expr) : 'a =
  match map_expr (eval_rec subst) e with
    #var as v -> eval_var subst v
  | `Add(`Num m, `Num n) -> `Num (m+n)
  | `Mult(`Num m, `Num n) -> `Num (m*n)
  | e -> e
val eval_expr :
  ((string *
    ([> `Var of string
      | `Num of int
      | `Add of 'a * 'a
      | `Mult of 'a * 'a] as 'a)) list → 'a → 'a) →
  (string * 'a) list → 'a expr → 'a

let rec eval2 subst = eval_expr eval2 subst
val eval2 : (string * ('a expr as 'a)) list → 'a → 'a
```

You can also see that evaluation is quite different of what it is on `lambda`: we first evaluate all subterms to make redexes apparent, and then pattern match. Again, we define an evaluator `eval2` for the closed language `'a expr as 'a`.

```
let e2 = eval2 [] (`Add(`Mult(`Num 3, `Num 2), `Var"x"))
val e2 : 'a expr as 'a = `Add (`Num 6, `Var "x")
```

Finally we take the union of `lambda` and `expr`, to create a full evaluator (table 4). Since types are inferred by the compiler, all it takes is actually the 3 lines of `eval_lexpr`: dispatch constructors from `lambda` to `eval_lambda` and constructors from `expr` to `eval_expr`. This is so simple that one may overlook that what we are doing here is multiple inheritance, with a shared common ancestor (the `var` language). Since all languages must share the same semantics for variables, this causes no complication. And this language is still extensible. An evaluator for our language is `eval3`.

```
let e3 = eval3 []
  (`Add(`App(`Abs("x",`Mult(`Var"x",`Var"x")),`Num 2), `Num 5))
val e3 : 'a lexpr as 'a = `Num 9
```

Table 4: The lexpr language, reunion of lambda and expr

```
type 'a lexpr =
  [ `Var of string | `Abs of string * 'a | `App of 'a * 'a
  | `Num of int | `Add of 'a * 'a | `Mult of 'a * 'a ]

let eval_lexpr eval_rec subst : 'a lexpr -> 'a = function
    #lambda as x -> eval_lambda eval_rec subst x
  | #expr as x -> eval_expr eval_rec subst x
val eval_lexpr :
  ((string *
    ([> `Num of int
      | `Abs of string * 'a
      | `Add of 'a * 'a
      | `App of 'a * 'a
      | `Mult of 'a * 'a
      | `Var of string] as 'a)) list → 'a → 'a) →
  (string * 'a) list → 'a lexpr → 'a

let rec eval3 subst = eval_lexpr eval3 subst
val eval3 : (string * ('a lexpr as 'a)) list → 'a → 'a
```

5 Comparison with other approaches

It is interesting to see what makes hard to encode the above example into traditional variants. There the standard way to produce the union of two types is to embed them in a type containing a constructor for each of them. In table 5 we give types for such an encoding, and some sample of the code.

One first problem, that you can see immediately by looking at the type definitions, is that this approach does not support multiple inheritance properly: `lexpr` contains two type of variables, `Lambda(VarL "x")` and `Expr(VarE "x")`. This is due to the fact traditional variants provide only disjoint sum, while union of polymorphic variants provides coalesced sum, required to handle the sharing of a common ancestor in multiple inheritance.

Another problem is that you need to do a lot of wrapping/unwrapping to have types working in your pyramid of languages: `var` is not directly compatible with `expr` or `lambda`, which in turn are incompatible with `lexpr`. Basically, all functions should have two extra arguments `wrap` and `unwrap`, doing this administrative work. Contrary to the problem of multiple inheritance, which cannot be solved uniformly without coalesced sums, we have here a workaround, but only at the cost of much verbosity. For instance `eval_lexpr` (omitted since it would not help understanding) requires 12 lines of code where polymorphic variants do with just 3 lines.

Comparing with object-oriented languages, there is not very much to say. It seems already difficult to typecheck a much simpler version of the expression problem in them (a 3rd order type system is required, as stated in the introduction), and this example adds a large number of difficulties. Independently of the problem of types, any translation of the above example in object-oriented style, using the visitor pattern, is going to be much more verbose, to such an extent that only giving snippets of the code would already be difficult.

Table 5: Encoding using normal variants

```
type var = string
type 'a lambda = VarL of var | Abs of string * 'a | App of 'a * 'a
type 'a expr =
  VarE of var | Num of int | Add of 'a * 'a | Mult of 'a * 'a
type 'a lexpr = Lambda of 'a lambda | Expr of 'a expr

let eval_var wrap sub (s : var) =
  try List.assoc s sub with Not_found -> wrap s
```
*val eval_var : (var → 'a) → (var * 'a) list → var → 'a*

```
let eval_expr eval_rec wrap unwrap subst e =
  let e' = map_expr (eval_rec subst) e in
  match map_expr unwrap e' with
    VarE v -> eval_var (fun x -> wrap (VarE x)) subst v
  | Add(Some(Num m), Some(Num n)) -> wrap (Num (m+n))
  | Mult(Some(Num m), Some(Num n)) -> wrap (Num (m*n))
  | _ -> wrap e'
```
val eval_expr :
*((var * 'a) list → 'b → 'c) → ('c expr → 'a) →*
*('c → 'd expr option) → (var * 'a) list → 'b expr → 'a*

6 Conclusion

We have demonstrated here how polymorphic variants allow one to combine the benefits of algebraic datatypes —namely clean separation of code and data, pattern-matching, and refined type checking— with code reuse at a level of simplicity comparable with object-oriented languages.

The added expressive power is significant: encoding in either traditional variants or object-oriented style would incur problems ranging from loss of clarity by a major increase in verbosity, to type theoretical limits such as the absence of coalesced sums for traditional variants or the need for higher-order typing with object-oriented style.

References

[Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, September 1998.

[KFF98] Shriram Krishnamurti, Mathias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. Technical Report 98-299, Department of Computer Science, Rice University, April 1998. preliminary version in European Conference on Object-Oriented Programming, 1998.

[LDG+00] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.00, Documentation and user's manual.* Projet Cristal, INRIA, April 2000.

[W+98] Philip Wadler, et al. The expression problem. Discussion on the Java-Genericity mailing list, December 1998.