Generic Programming with Combinators and Objects

Dmitry Kosarev
Dmitrii.Kosarev@protonmail.ch

Dmitry Boulytchev
dboulytchev@math.spbu.ru

Saint Petersburg State University
Saint Petersburg, Russia

1 Introduction

Generic programming in the context of OCaml lately gains additional attention [1, 2, 3]. We present a generic programming library GT[1] (Generic Transformers), which has been in an active development and use since 2014. This library is an inheritor of our earlier work [13] on implementation of "Scrap Your Boilerplate" approach [9, 10, 11]. However, our experience has shown, that the extensibility of SYB is insufficient; in addition the uniform transformations, based solely on type discrimination, turned out to be inconvenient to use (for example, they can allow one to break through the encapsulation barrier). Our idea initially was to combine combinator and object-oriented approaches — the former would provide means for parameterization, while the latter — for extensibility via late binding utilization. This idea in the form of a certain design pattern was successfully evaluated [12] and then reified in a library and a syntax extension [14]. Our follow-up experience with the library [15] has (once again) shown some flaws in the implementation. The version we present here is almost a complete re-implementation with these flaws fixed.

From an end user perspective, our library is comprised of four layers:

1. On the top level it provides a syntax extension (in terms of both `camlp5` and `ppxlib`) with a number of plugins (`map`, `fold`, `show`, etc.) The interface of generated features is combinatorial, so their utilization is rather straightforward.

2. On the middle level it turns out, that all these features are implemented via object-encoded transformations with some reasonable default behavior. This behavior can be modified/overridden using inheritance. Thus, customized transformations can be acquired using the default ones.

3. On the low level it turns out, that all features are in fact instantiations of some very general transformation scheme; thus, transformations, which do not fit in any pre-supplied plugin can still be implemented manually.

4. In the basement, the users can implement their own plugins; note, since all plugins are just instantiations of some generic scheme, the implementation requires only a limited amount of work. In particular, all plugins use the same single traversal function, which need not to be generated.

In fact, there is also an underground layer — all generic features are combined into an object, which can be passed as a parameter or modified. While currently the library does not contain any conventional interface to deal with the object, it can be provided in the future (which opens a potentially interesting opportunities for integration with existing proposals for *ad-hoc* polymorphism [16]).

[1] `https://github.com/kakadu/GT/tree/ppx-new`

2 Design

The design of the library is based on the idea to describe transformations (e.g. catamorphisms [6]) in terms of transformations, described by attribute grammars [7, 8]. In short, we consider only the transformations of the following type

$$\iota \to t \to \sigma$$

where t is the type of value to transform, ι and σ — types for *inherited* and *synthesized* values. We do not use attribute grammars as a mean to describe the algorithmic part of transformations; we only utilize their terminology to describe the types of transformations.

When the type under consideration is parameterized, the transformation becomes parameterized as well:

$$\begin{array}{l}(\iota_1 \to \alpha_1 \to \sigma_1) \to \\ \quad \ldots \\ (\iota_k \to \alpha_k \to \sigma_k) \to \quad \iota \to (\alpha_1, \ldots, \alpha_k)\, t \to \sigma\end{array}$$

In general the argument-transforming functions operate on inherited values of different types and return synthesized values of different types.

The second idea is to encode a transformation for an algebraic data type as an object with per-constructor transformation methods (the similar idea is used in [2]). For example, for a type

```
type α t = A of α | B of α t * α t
```

a transformation object would have the following structure

```
object
  method c_A : ι → α → σ
  method c_B : ι → α t → α t → σ
end
```

To automatically mass-produce transformation objects, a number of classes is generated: first, the common base virtual class for all transformations for given type, and then one customized class per feature, requested by mean of plugins. All these classes are concrete, inherit from the base one and are additionally parameterized by type parameters-transforming functions, including the function for transforming the type itself (thus using open recursion pattern).

Finally, a single traversal function is generated. It takes a transformation object, an inherited attribute, and a value to traverse, performs pattern-matching and calls appropriate methods of the object. For the example in question the traversal function may look like

```
let transform obj ι = function
| A x       → obj # c_A ι x
| B (l, r)  → obj # c_B ι l r
```

Note, the traversal function is non-recursive; the recursion (if any) is indirectly handled in object's methods.

Within this infrastructure is turned out to be possible to implement such features as `show`, `fmap`, `fold`, as well as `eq` and `compare`, which usually are expressed in an *ad-hoc* manner in other frameworks. All these features are implemented as plugins, which instantiate the generic components. Plugins also generate the top-level functions, tying the recursive knot, and combine this functions into a data structure with the same name as the type of interest. All plugins supply a (universal) access function, which takes this data structure as its first parameter. Under these conventions, `show(int)` designates a `show` function for `int`s, while `fmap(list)` — `fmap` for lists.

Beyond this simplified scheme some other things have to be done; for example, a special care has to be taken to support polymorphic variants, which we consider an important feature of our library.

3 Examples

In this section we demonstrate some examples, written with the aid of our library. In this examples we will use `camlp5` syntax extension, although `ppxlib` plugin can be used equally.

First, we consider a simple type to represent arithmetic expressions:

```
@type expr  = Var   of string
            | Add   of expr * expr
            | Mul   of expr * expr
            | Div   of expr * expr
            | Const of int with fmap
```

Here we requested a feature `fmap`, which implements the conventional functor semantics. Since the type is not polymorphic, the function `fmap(expr)` just copies its argument. Although the copying can be considered useful on its own, this result a bit disappointing. However, with the aid of our framework we actually can acquire a number of useful transformations, taking the copying as the starting point. For example, given a state `st` we can substitute the values of all variables in this state in an expression:

```
let substitute st e = fix
  (fun f →
     transform(expr)
       (object inherit [_] @expr[fmap] f
          method c_Var _ x = Const (st x)
        end)
        ()
  ) e
```

Indeed, all we need is to redefine the copy behavior for constructor `Var`. In order to do this we inherit from the class `fmap` for the type `expr` (denoted by `@expr[gmap]` in the snippet), and rewrite the method `c_Var` (note the use of generic function `transform(expr)` and fix point combinator). As it can be seen from this example, we needed to implement only "the interesting" part of the transformation. All other functionality (recursive propagation through the whole data structure) is handled by a framework-generated code.

For another example we consider an expression simplifier, which performs all possible calculations with constants and utilizes some simple arithmetic equalities like $0 * x = 0$ or $0 + x = x$:

```
class simplifier f =
object inherit [_] @expr[fmap] f
  method c_Div _ x y =
    match f x, f y with
    | Const x, Const y  → Const (x / y)
    | x      , Const 1  → x
    | x      , y        → Div (x, y)
  method c_Mul _ x y =
    match f x, f y with
    | Const x, Const y          → Const (x * y)
    | Const 0, _ | _, Const 0   → Const 0
    | Const 1, y                → y
    | x, Const 1                → x
    | x, y                      → Mul (x, y)
  method c_Add _ x y =
    match f x, f y with
    | Const x, Const y  → Const (x + y)
    | Const 0, y        → y
    | x, Const 0        → x
    | x, y              → Add (x, y)
end
```

Since the interesting part is concentrated in the class definition, we omitted the top-level function, which looks exactly like the previous one, since we are still dealing with the same feature `fmap`. The class definition is much longer, than the previous one, but this is inevitable — the interesting part is that long, indeed.

Note, the simplifier we implemented is strict — it evaluates both operands of a multiplication even if the first is equal 0. We can implement a non-strict simplifier on top of the strict one:

```
class ns_simplifier f =
object inherit simplifier f
  method c_Mul _ x y =
```

```
    match f x with
    | Const 0 → Const 0
    | Const 1 → f y
    | Const x → (match f y with
                 | Const y → Const (x * y)
                 | y       → Mul   (Const x, y)
                )
    | x       → (match f y with
                 | Const 0 → Const 0
                 | Const 1 → x
                 | y       → Mul (x, y)
                )
end
```

Again, this definition consists of only interesting part.

Finally, with substitutions an simplifications we can define an evaluation (first substitute, then simplify). Thus, the object layer of our framework provides us with the powerful tool to create and modify transformations.

For another example we take the support for polymorphic variants [4, 5], which we consider an important feature since it complements the opportunity to provide composable data structures with the opportunity to create composable transformations. For the concrete problem we take the transformation from named to nameless representations for lambda terms.

First, we define the generic part of the terms:

```
@type ('name, 'lam) lam = [
| `App of 'lam * 'lam
| `Var of 'name
] with eval
```

The `eval` plugin here generates a transformation `eval(lam)`, which is analogous to `fmap`, but additionally uses some environment, which by default is propagated unchanged. We here follow [5] and use an open non-recursive definition of the type; our `eval` corresponds to `map` in terms of [2].

Then, we define a binding construct — abstraction:

```
@type ('name, 'term) abs = [
| `Abs of 'name * 'term
] with eval

class ['term, 'term2] de_bruijn ft =
object
  inherit [string, unit, 'term, 'term2,
           string list, 'term2] @abs[eval]
    (fun _ → assert false)
    (fun _ _ → ())
    ft
  method c_Abs env name term =
    `Abs ((), ft (name :: env) term)
end
```

This time we have to define a conversion transformation since for the abstraction the default behavior of `eval` is not enough. We introduce the subclass for `@abs[eval]`, in which we specify the type of the environment (`string list`), the representations for names in the input and output values (`string` and `unit` respectively), and representations for subterms in the input and output values (abstract for now). The last, sixth type parameter for `@abs[eval]` is needed for open recursion. The semantics of the single method of this class reflects the normal behavior of the abstraction during the conversion into the nameless representation: it adds the variable to the environment and uses this environment to convert the subterm. The parameter `ft` corresponds to the subterm conversion transformation. Since we do not know it yet, we have to abstract over it.

Now we can combine two types into the single type for lambda terms:

```
@type ('n, 'b) term = [
| ('n, ('n, 'b) term) lam
| ('b, ('n, 'b) term) abs
] with eval

@type named    = (string, string) term
@type nameless = (int, unit) term
```

Here we distinguish names in binder positions (`'b`) and bound positions (`'n`) since their behavior during the transformation essentially different: names in binder positions are erased, while in bound positions are substituted with corresponding DeBruijn index. We also define a shortcuts for the terms in named and nameless representations.

Similarly to the types, the transformations can be combined as well:

```
class de_bruijn fself =
object
  inherit [string, int, string, unit,
           string list, nameless] @term[eval]
    fself
    ith
    (fun _ _ → ())
  inherit [named, nameless] Abs.de_bruijn fself
end
```

For the generic part of the terms we reused the `eval` transformation, while for abstractions we took the customized one (`de_bruijn`); in any case the final transformation is build via inheritance with no other glue; here `ith` is a function, which finds a names in an environment and returns their indices.

It is interesting, that with polymorphic variants is becomes possible to define a transformation with an output type, different from the input beyond parameterization:

```
class ['term, 'term2] de_bruijn' ft =
object
  inherit [string, string list, unit,
           'term, string list, 'term2,
           string list, 'term2, 'term] @abs
  method c_Abs env name term =
    `Abs (ft (name :: env) term)
end

@type named = [
| (string, named) lam
| (string, named) abs
```

```
] with eval

@type nameless = [
| (int, nameless) lam
| `Abs of nameless
] with eval

class de_bruijn fself =
object
  inherit [string, int,
           named, nameless,
           string list,
           nameless] @lam[eval] fself ith fself
   inherit [named, nameless] Abs .de_bruijn' fself
end
```

Please note the implementation of method `c_Abs` — now it returns a constructor `` `Add `` with *one* argument. In short, we defined a transformation into a nameless representation, which completely removes the names in binder positions.

4 Related Works

Among existing generic programming frameworks for OCaml we can name two, which resemble ours: `ppx_deriving` and `Visitors` [2].

As `ppx_deriving` implements a purely combinatorial approach, based on higher-order functions, we suppose, that our library subsumes it in terms of expressivity (at least potentially).

`Visitors`, on the other hand, explore a similar to ours object-oriented approach, in which many decisions, rejected by us, were taken (and vice versa). Here we summarize the main differences:

- `Visitors` are excessively object-oriented — in order to use them one needs to instantiate some object and call a proper method. In our case as long as only predefined features are required a user can use a more native combinatorial interface.

- `Visitors` implement a number of useful transformations in an *ad-hoc* manner; in our case all transformations are instances of the same generic scheme. It is possible to combine different transformations via inheritance as long as the types of underlying scheme unify. We also argue, that in our framework the implementation of user-defined plugins is much easier.

- Following SYB, `Visitors` take a type-discriminating route: for each type of interest (including the built-in ones) there is a dedicated transformation method in each object, representing a transformation. While this solution indeed adds some flexibility, we firmly oppose it, since it breaks the abstraction: inspecting the methods of a transformation (which cannot be hidden in a module signature) one can retrieve some information about the implementation of encapsulated types. Even worse, the data structures of abstract types can be manipulated in an unprescribed manner using the public type-transforming interface.

- In our case the type parameters for transformation classes have to be specified by an end user. With `Visitors` this burden is offloaded to the compiler with the aid of some neat trick. However, this trick makes it impossible to use `Visitors` syntax extension in module signatures. There is no such problem in our case — our framework can be equally used in both implementation and interface files.

- `Visitors` in their current state do not support polymorphic variants.

5 Conclusion

In this paper we presented an improved version of Generic Transformers, extended by support of PPX rewriters and type abbrevations. Although it uses the similar idea as in some related works, we claim that it allows to solve some problems in a more convenient manner.

References

[1] Jeremy Jallop. Staged Generic Programming // ICFP-2017.

[2] François Pottier. Visitors Unchained // ICFP-2017.

[3] Florent Balestrieri, Michel Mauny. Generic Programming in OCaml // OCAML-2016.

[4] Jacques Garrigue. Programming with Polymorphic Variants // ML-1998.

[5] Jacques Garrigue. Code Reuse Through Polymorphic Variants // FOSE-2000.

[6] Erik Meijer, Maarten Fokkinga, Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire // 5th ACM Conference on Functional Programming Languages and Computer Architecture, 1991.

[7] Donald E. Knuth. Semantics of Context-Free Languages // Mathematical Systems Theory, Vol. 2, No. 2, 1967.

[8] Marcos Viera, S. Doaitse Swierstra, Wouter Swierstra. Attribute Grammars Fly First-Class: How to do Aspect Oriented Programming in Haskell // ICFP-2009.

[9] Ralf Lämmel, Simon Peyton Jones. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming // Workshop on Types in Language Design and Implementation, 2003.

[10] Ralf Lämmel, Simon Peyton Jones. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts // ICFP-2004.

[11] Ralf Lämmel, Simon Peyton Jones. Scrap Your Boilerplate with Class: Extensible Generic Functions // ICFP-2005.

[12] Dmitry Boulytchev. Combinators and Type-driven Transformers in Objective Caml // Science of Computer Programming, Vol. 114, December 2015.

[13] Dmitry Boulytchev, Sergey Mechtaev. Efficiently Scrapping Boilerplate Code in OCaml // ML-2011.

[14] Dmitry Boulytchev. Code Reuse with Transformation Objects // CoRR abs/1802.01930 (2018).

[15] Dmitry Kosarev, Dmitry Boulytchev. Typed Embedding of a Relational Language in OCaml // ML-2016.

[16] Leo White, Frédéric Bour, Jeremy Yallop. Modular Implicits // EPTCS 198, 2015.