Efficiently Scrapping Boilerplate Code in OCaml

Dmitri Boulytchev Sergey Mechtaev

St.Petersburg State University
Saint-Petersburg, Russia
dboulytchev@gmail.com mechtaev@gmail.com

Introduction

We present an adaptation of a well-known generic programming technique, often referred to as "Scrap Your Boilerplate", or SYB, to Objective Caml language. Since the original implementation for Haskell essentially relies on the presence of type classes, the first task in the course of adaptation is to express the main primitives of this approach in a way natural for Objective Caml. Besides that we make use of specialization and continuation-passing style (CPS) to improve the performance of our implementation. CPS is used as a common way to provide a tail-recursive implementation while specialization allows us to lift type-discriminated computations (natural to SYB) from the data level to the type level. As a result the presented version of SYB demonstrates good performance in comparison with non-generic hand-written code.

1. "Scrap Your Boilerplate"

"Scrap Your Boilerplate" is an attractive generic programming pattern originally developed for Haskell [Lämmel and Peyton Jones 2003, 2004, 2005]. We describe this approach in a nutshell by a simplified (and a bit artificial) example.

Consider the following definition for the type of simple arithmetic expression:

```
type expr =
  Add   of expr * expr
| Neg   of expr
| Const of int
| Var   of string
```

Suppose we need two transformations: the first increments each constant in the expression, the second appends some suffix to the name of each variable.

Hand-written versions of both transformations are given below:

```
let rec increment = function
  Add  (x, y) -> Add   (increment x, increment y)
| Neg   x     -> Neg   (increment x)
| Const x     -> Const (x + 1)
| Var   v     -> Var v

let rec suffixize = function
  Add  (x, y) -> Add (suffixize x, suffixize y)
| Neg   x     -> Neg (suffixize x)
| Const x     -> Const x
| Var   v     -> Var (v ^ "_suffix")
```

It is easy to see that both these functions share the same pattern: the traversal of a data structure. The specific actions performed by each of these functions are discriminated according to the type of some sub-value of traversed data structure.

SYB allows to encapsulate the common traversal code in a few separate per-type generic functions. Both transformations of aforementioned example can be expressed via the following function (indexed by the type t):

$$\text{gmapT}^t : (\forall\alpha.\alpha \to \alpha) \to t \to t$$

Informally speaking, `gmapT f x` performs a *shallow* traversal of the data structure specified by `x` and applies a function `f` to each of its immediate successors. Since the argument and the result of `f` both have the same type this allows to consider $\texttt{gmapT}^t$ as transformer of type `t` into itself. For out example

$$\text{gmapT}^{\text{expr}}\, f = \textbf{function}$$
```
    Add (x, y) -> Add (f x, f y)
  | Neg x -> Neg (f x)
  | Const i -> Const (f i)
  | Var n -> Var (f i)
```

The first argument of `gmapT` is a polymorphic function; the key feature of the approach in question is the way this function is constructed. In the SYB settings this function is acquired by the *lifting* a type-specific transformer. In abstract terms lifting can be considered as an operator with the following type[1]

$$\text{lift} : (t \to t) \to \forall\alpha.\alpha \to \alpha$$

and the following semantics:

$$\text{lift}\,(f : t \to t) = \lambda x : \alpha\,.\, \begin{cases} f\,x & , \quad \alpha = t \\ x & , \quad otherwise \end{cases}$$

Informally, lifted function performs type-analysis at the runtime: if the argument actually has the type t then f is applied; otherwise the argument is returned unchanged.

The final component of the solution is the function which turns the shallow traversal into the deep one:

$$\text{everywhere}\, f\, (x\, :\, t)\; =\; f\, (\text{gmapT}^t\, (\text{everywhere}\, f)\, x)$$

Under these assumptions, both functions from our example can be expressed in the following manner:

$$\text{everywhere}(\text{lift}(\lambda x : \text{int}\,.\, x + 1))$$
$$\text{everywhere}(\text{lift}(\lambda x : \text{string}\,.\, x + \text{"\_suffix"}))$$

In more general terms SYB allows to propagate some type-discriminated type-preserving transformations through arbitrary data structures. For example, given lifted version of integer increment one can easily increment all integers within a data structure regardless of its type (provided that that data structure implements some conventional interface).

In the original implementation of SYB for Haskell all needed functionality is encoded using Haskell-specific features like type classes and rank-2 types. Besides generic transformations (expressed via functions `gmapT`/`everywhere`) the original implementation introduces their monadic version and generic queries

[1] In the original paper lifting operators were denoted by `mkT`, `mkQ` etc.

gmapQ/everything which allow to collect or extract some data of interest from the data structure. In the rest of this paper we concentrate on simple non-monadic transformations since all other functionality of SYB can easily be reconstructed from their implementation.

2. Encoding SYB in OCaml

In this section we describe the direct implementation of SYB for OCaml. This implementation almost directly follows the Haskell version with the exception of *type markers* which encode type-safe cast in a different manner.

2.1 Type Equality and Type Markers

We start from the well-known (weak) type equality encoding [Baars and Swierstra 2002]. Such an encoding allows us to express the equality of types in terms of OCaml values. Namely, we utilize the following definitions:

```
type ('a, 'b) eq

val refl    : unit -> ('a, 'a) eq
val symm    : ('a, 'b) eq -> ('b, 'a) eq
val trans   : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
val coerce  : ('a, 'b) eq -> 'a -> 'b
```

Here the type `('a, 'b) eq` witnesses the equality of types `'a` and `'b`. The first three functions encode the axioms of the equivalence relation; the fourth one reifies a type equivalence into a coercion function. We omit the standard implementation due to space considerations.

Now, consider the following declarations:

```
type 'a marker

val make: unit -> 'a marker
val compare :
  'a marker -> 'b marker -> ('a, 'b) eq option
```

Here `'a marker` represents a *type marker*. We are going to introduce a separate type marker — some dedicated value of the type `t marker` — for each relevant type `t`:

```
let int_m    : int    marker = make ()
let string_m : string marker = make ()
let expr_m   : expr   marker = make ()
```

Now we can use the comparison function `compare` to decide whether two type markers correspond to the same type. As a result we either obtain type equality witness or nothing.

There are several possible design choices for the exact implementation of type markers and the comparison function; as for the simplified case described above the following variant may be sufficient:

```
type 'a marker = unit ref

let make () = ref ()
let compare x y =
  if x == y then Some (Obj.magic (refl ())) else None
```

Similarly to the original SYB implementation, which relies on `unsafeCoerce` we have to use `Obj.magic` once.

In the actual implementation we extended the simplified version in some way to support polymorphic type instantiations; for the same reason we also had to replace the weak type equality with Leibniz one. We omit the details for now.

2.2 Lifting

Next, we have to encode the lifted transformations, i.e. the transformations obtained from some function by extending its domain:

```
type lifted = {f : 'a . 'a marker -> 'a -> 'a}
```

This definition utilizes the OCaml encoding for rank-2 functions which are an essential prerequisite for SYB implementation. The lifted function takes an argument of arbitrary type together with marker of that type and performs type case analysis which we better illustrate by the implementation of lifting primitive:

```
let lift (m : 'a marker) (f' : 'a -> 'a) =  {
  f = fun n ->
        match compare m n with
        | None ->
            fun x -> x
        | Some e ->
            let forward = coerce (symm e) in
            let backward = coerce e in
            fun x -> backward (f' (forward x))
}
```

When we lift a function `f'` we supply as well the marker of its actual argument type. This marker is kept in the closure of lifted function and then used to compare with the type marker of argument passed on invocation. Consider the following example:

```
let f = lift int_m (fun x -> x+1)
let a = f.f int_m 1
let s = f.f string_m "abc"
```

Here we lift integer increment function; the lifted version then can be applied to the arguments of arbitrary types. For all types other than integer it operates as identity, so `a` equals "2" and `s` equals "abc".

Note that one cannot go wrong by specifying the invalid type information. Neither

```
f.f int_m "abc"
```

nor

```
f.f string_m 1
```

would pass the typechecker.

2.3 Type Information and Transformers

To complete SYB implementation we have to supply a (per-type) structure which encapsulates all functionality needed to perform generic transformations. This structure is represented by the polymorphic type `'a typeinfo`. To make it possible to perform transformations with some type `t` one needs to provide a value of type `t typeinfo`. The definition of `'a typeinfo` is as follows:

```
type 'a typeinfo = {
  marker : 'a marker;
  gmapT  : transform -> 'a -> 'a
}
and transform = {
  transform : 'a . 'a typeinfo -> 'a -> 'a
}
```

Type `'a typeinfo` contains the type marker for the type `'a` and the function `gmapT`, which is similar to the corresponding function from the original SYB. Since `gmapT` should as well transform all subvalues of its argument, which can have different types, an appropriate polymorphic traversal function should be provided. This function has the type `transform`; it in turn performs a traversal using appropriate `typeinfo`. We illustrate this construction by providing an examples of type information for some typical cases.

For the shallow types the definitions are as follows:

```
let int = {
  marker = int_m;
  gmapT = fun _ x -> x
}

let string = {
  marker = string_m;
  gmapT = fun _ x -> x
}
```

Indeed, for these types no deep traversal is possible, so `gmapT` operates trivially.

For non-recursive algebraic data types we should couple their type markers with the function which performs pattern matching and applies transformation to the matched subvalues, providing appropriate type information. The following example illustrates this case:

```
type t = A of int | B of string

let t_m : t marker = make ()
let t = {
  marker = t_m;
  gmapT = fun t ->
    let ti = t.transform i in
    let ts = t.transform string in
    function
    | A i -> A (ti i)
    | B s -> B (ts s)
}
```

Finally, type information for the recursive types is provided by a recursive function:

```
let expr =
  let rec inner () = {
    marker = expr_m;
    gmapT  = (
      fun t ->
        let te = t.transform (inner ()) in
        let ti = t.transform int in
        let ts = t.transform string in
        function
        | Add (x, y) -> Add (te x, te y)
        | Neg x      -> Neg (te x)
        | Const i    -> Const (ti i)
        | Var s      -> Var (ts s)
    )
  }
  in inner ()
```

Note that these definitions are fully type-driven; it is possible to generate them directly from the type descriptions. We implemented the syntax extension in the form

$$\begin{array}{lcl} \textbf{datatype} & typeid_1 & = & type\_expr_1 \\ \textbf{and} & typeid_2 & = & type\_expr_2 \\ \ldots & & & \end{array}$$

which generates type declarations as well as type information and markers.

With these definitions it is possible to express the function `everywhere` from SYB which combines generic traversal with the "interesting function":

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti
```

Now we can rewrite the example from the beginning of the paper:

```
let increment = everywhere expr
  (lift int_m (fun i -> i+1))

let suffixize = everywhere expr
  (lift string_m (fun s -> s ^ "_suffix"))
```

The difference between this implementation and the original one is that here one needs to specify the type information explicitly while in Haskell the compiler infers it automatically. Nevertheless, as we pointed out, invalid type specification would not pass the typechecker. So from the user point of view the difference in implementations is just as the difference between the type checking and type inference.

3. Optimizations

The naïve implementation described in the previous section performs poorly by mean of both space and time consumption. This deficiency is typical to the original SYB implementation as well — according to some reports the slowdown of SYB-based vs. hand-coded implementation can reach two orders of magnitude [Chakravarty et al. 2009]. We used two speed-up techniques: CPS and specialization. Since conversion into the tail-recursive form via CPS is rather a standard transformation we will not consider it here.

The main reason for the slowdown is that in the course of a generic traversal the lifted transformation function is applied to the each element of the data structure. At the moment of each application type markers are compared and a type-specific function is selected. Moreover it is always possible to construct an artificial example where the hand-coded implementation would outperform the naïve one with the arbitrarily chosen factor. To achieve this result it would be enough to choose the data structure in such a way that the "interesting" values would be distributed sufficiently sparse. In a hand-coded version it is easy (and quite natural) to avoid traversing datatypes which lack the interesting values while in the naïve implementation it is generally impossible.

In this section we present two specialization techniques which overcome the both mentioned above deficiencies.

3.1 Specialization on Data Type

The idea of specialization of this kind explores the observation that it is sufficient to pre-calculate all type-discriminated comparisons and selections by inspecting the type of a data structure instead of the data structure itself. So generally we reduce the number of required type-discriminated decisions from the size of a data structure to the size of its type.

Recall the definition of function `everywhere` from the previous section:

```
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti x -> f.f ti.marker (ti.gmapT {transform} x)
  in
  transform ti
```

Clearly, this version performs the traversal of the data structure (note the subexpression "`ti.gmapT {transform} x`") and then applies lifted function to the result. The idea of optimization is to separate traversal function construction from the traversal itself. The first attempt naturally would be as follows:

```
let compose f g x = f (g x)
let everywhere ti f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
```

```
  fun ti ->
    compose
      (f.f ti.marker)
      (ti.gmapT {transform})
in
transform ti
```

Here we first construct the traversal function using the traversal functions for subvalues, lifted transformation and composition operator and then return this function as a result. Unfortunately this version loops on a recursive types so we need to act more accurate. Namely, we utilize the properties of depth-first traversal to memoize the transformations for those nodes of type graph which has already been involved into the traversal. To implement this technique we introduce a mapping from type markers into traversal functions. Note that in this mapping the type of the mapped value depends on the type of the key. The mapping itself can easily be implemented using type markers; we omit the details here. Under these assumptions the specializing implementation of function `everywhere` looks like the following:

```
let everywhere ti f =
  let context = T.create () in
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti ->
      let m = ti.marker in
      let f = f.f m in
      try
        let tr = T.find context m in
        compose f (fun x -> !tr x)
      with Not_found ->
        let tr = T.stub () in
        T.add context m tr;
        tr := ti.gmapT {transform};
        T.remove context m;
        compose f !tr
in
transform ti
```

Here we create an empty mapping between type markers and transformation functions and then start to traverse the graph of the type in a depth-first order.

For each encountered type we first try to retrieve the associated transformation from the context. If there is some mapping for the given type marker then we just compose the acquired transformation with the lifted function. This situation corresponds to the event of processing a backward edge in the graph of the type.

If there is no mapping for the given type marker yet then we associate that type marker with the *stub* which is just a temporary value of appropriate type. This value is then modified in-place by the result of calculation of transformation function. Finally we compose that transformation with the lifted function and remove the association from the context.

The technique used here is in fact a version of *witnessing fixpoint* calculations [Karvonen 2007].

3.2 Specialization on Interesting Function

The next optimization allows to completely avoid the traversal of a whole region of data structure if it can be proven that no values of interest can occur within that region. To distinguish regions of interest we again can use type information. Namely, we avoid traversal of a data region if the type of interest does not have occurrences in the type of that region. For example, if we have function `lift float_m (fun x -> x +. 1.0)` then we can skip traversal of the regions of type `expr` since no floats can appear within data of that type.

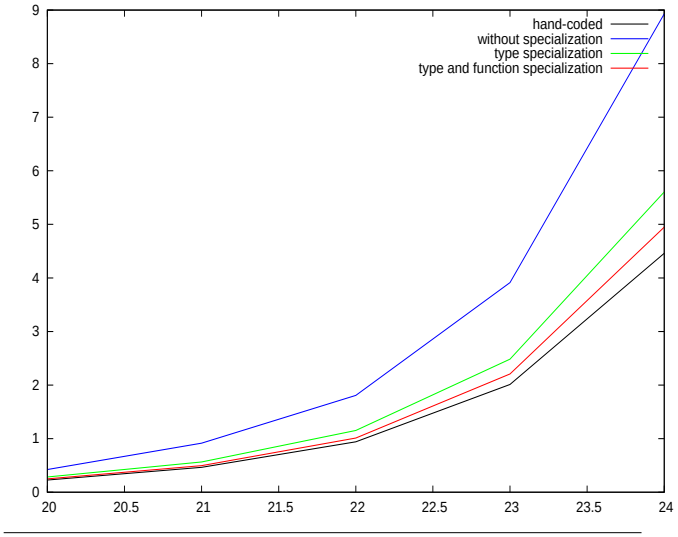

Figure 1. `suffixize` benchmark

The implementation of this optimization is a bit cumbersome since actually we do not have explicit representation of types in the course of specialization. Nevertheless the definition of `everywhere` can be rewritten to incorporate this optimization as well; we do not present it here due to space considerations.

4. Results

We evaluated the performance of our implementation on the following benchmarks:

- `suffixize` — appends a given suffix to all variable names in an arithmetic expression;
- `substitute` — replaces all occurences of a given variable with some constant value;
- `increase` — an implementation of canonical example of salary increment from the original SYB paper.

Four implementations were compared for all three benchmarks:

- `hand-coded` — an ad-hoc handcoded implementation;
- `without-specialization` — the naïve implementation of SYB for OCaml;
- `type-specialization` — implementation with specialization on type;
- `type and function specialization` — implementation with specializations on both type and function.

All these variants were implemented in a continuation-passing style.

The results of the evaluation are shown on the Figures. 1-3. On each graph horizontal direction corresponds to the size of transformed data structure (in a logarithmic scale); vertical axis corresponds to the transformation time in seconds.

The evaluation shows that our implementation of SYB with both aforementioned optimizations performs almost as good as handcoded.

5. Limitations and Drawbacks

The implementation we discussed in the previous sections suffers from the following limitations.

First, it does not allow to process cyclic or shared data structures properly. For example, an attempt to traverse an infinite list will result in infinite loop. Similar problem will arise during processing

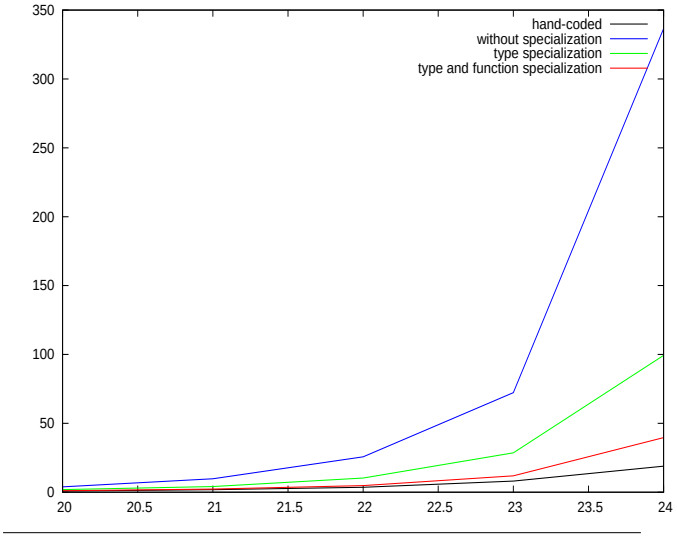

Figure 2. `increase` benchmark

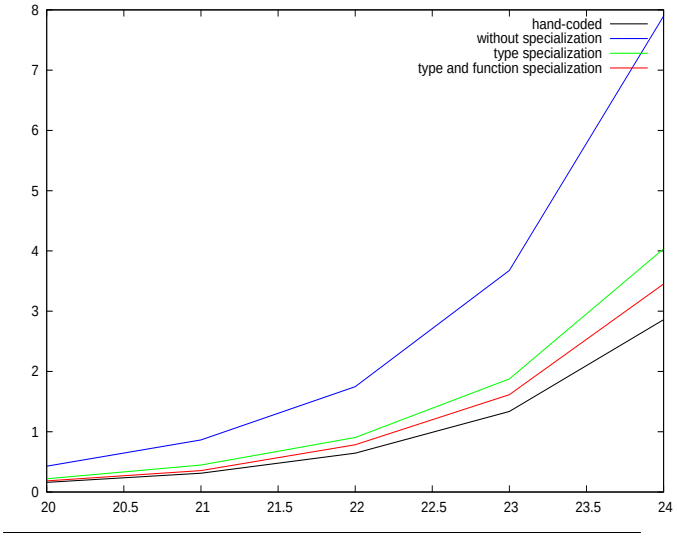

Figure 3. `substitute` benchmark

of shared data — each shared region will be processed multiple times. Nevertheless, we believe that an *ad-hoc* solution, which we consider as a future improvement, can be found.

Another problem arises from the fact that for testing the type equivalence we use type marker equality that is stronger then the type equivalence itself. This means that equal types may have different type markers. Even if all markers are provided transparently to the end-used by the syntax extension, there are still some cases when this drawback can affect the behavior. For example, our implementation is functor-sensitive. Consider the following code:

```
module F (X : sig type t end) =
  struct
    datatype t = A of X.t | B
  end
module A = F (struct type t = int end)
module B = F (struct type t = int end)
```

Despite the types `A.t` and `B.t` are equivalent, their type markers are not equal, which can lead to a silent (and hard to detect) misbehavior of a generic traversals. We consider this drawback as unavoidable for the current implementation; as a weak workaround an accurate case analysis can be provided for users to properly describe the use cases when the genericity can be used with no harm.

Acknowledgements

We thank Ilya Sergey for his comments and suggestions on the draft version of this abstract.

References

Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 157–166, 2002.

Manuel M. T. Chakravarty, Gabriel C. Ditu, and Roman Leshchinskiy. Instant generics: Fast and easy. 2009.

Vesa A.J. Karvonen. Generics for the working ml'er. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 71–82, 2007.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003)*, pages 26–37, 2003.

Ralf Lämmel and Simon Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 244–255, 2004.

Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 204–215, 2005.