Generic Programming in OCAML

Florent Balestrieri, Michel Mauny

HAL Id: hal-01664286
https://inria.hal.science/hal-01664286

Preprint submitted on 14 Dec 2017

Generic Programming in OCaml

Florent Balestrieri

ENSTA-ParisTech, Université Paris-Saclay

`Florent.Balestrieri@ensta-paristech.fr`

Michel Mauny

Inria Paris

`Michel.Mauny@inria.fr`

We present a library for generic programming in OCaml, adapting some techniques borrowed from other functional languages. The library makes use of three recent additions to OCaml: generalised abstract datatypes are essential to reflect types, extensible variants allow this reflection to be open for new additions, and extension points provide syntactic sugar and generate boiler plate code that simplify the use of the library. The building blocks of the library can be used to support many approaches to generic programming through the concept of view. Generic traversals are implemented on top of the library and provide powerful combinators to write concise definitions of recursive functions over complex tree types. Our case study is a type-safe deserialization function that respects type abstraction.

1 Introduction

Typed functional programming languages come with rich type systems guaranteeing strong safety properties for the programs. However, the restrictions imposed by types, necessary to banish wrong programs, may prevent us to generalise over some particular programming patterns, thus leading to boilerplate code and duplicated logic. Generic programming allows us to recover the loss of flexibility by adding an extra expressive layer to the language.

The purpose of this article is to describe the user interface and explain the implementation of a generic programming library[1] for the language OCaml. We illustrate its usefulness with an implementation of a type-safe deserialization function.

1.1 A Motivating Example

Algebraic datatypes are very suitable for capturing structured data, in particular trees. However general tree operations need to be defined for each specific tree type, resulting in repetitive code.

Consider the height of a tree, which is the length of the longest path from the root to a leaf. We will define a different height function on lists, binary trees and rose trees.

For lists, the height corresponds to the length of the list.

```
let rec length = function
    | [] → 0
    | _ :: tail → 1 + length tail
```

For binary trees the definition is very similar, but in the inductive step we must now take the maximum of the heights of the children.

```
type α btree = Empty | Node of α btree × α × α btree
let rec bheight = function
    | Empty → 0
    | Node (l, _, r) → 1 + max (bheight l) (bheight r)
```

Rose trees have nodes of variable arity, we define them as records with two fields: attr is the data associated to a node, and children is the list of its immediate subtrees.

```
type α rtree = {attr: α; children: α rtree list}
let rec rheight rtree =
    match List.map rheight rtree.children with
    | [] → 0                                          (*   The height of a leaf is zero.   *)
    | h :: hs → 1 + List.fold_left max h hs
```

The reader can see how the definition of new types of trees would require the implementation of their own specialised height function. Yet we can see a common pattern emerging. Is it possible to factorise the common behaviour? Yes! thanks to parametric polymorphism and higher-order functions, we may abstract over the notion of children: the function gheight below takes an argument function children that computes the list of children of a node.

```
val gheight : (α → α list) → α → int
let rec gheight children tree =
    let subtrees = children tree in
    match List.map (gheight children) subtrees with
    | [] → 0
    | h :: hs → 1 + List.fold_left max h hs
```

Then each particular case above can be implemented using gheight by providing the appropriate implementation of children:

```
let length'  x = gheight (function [] → [] | _ :: tail → [tail]) x
let bheight' x = gheight (function Empty → [] | Node (l, _, r) → [l; r]) x
let rheight' x = gheight (fun x → x.children) x
```

Having factored the functionality of height, we're left with the task of implementing children for each datatype. This task follows systematically from the definition of a type and this time the pattern cannot be abstracted. This is when generic programming comes into play! With generic programming, we can write a single children function working over all types. It is indexed by the type representation of its tree argument: a value of type α ty is the value-level representation of the type α.

```
val children : α ty → α → α list
```

The type-indexed version of height is obtained by composing gheight and children:

```
val height : α ty → α → int
let height t = gheight (children t)
```

The implementation of children will be explained in Section 2.3.

Note: the type witness α ty is explicitly given to a generic function, for instance if x : α list we might call height list x where list is a suitable value of type α list ty. It is theorically possible to infer the type witness since there is a one to one correspondance between the witnesses and types. The work on *modular implicits* [34] promises to offer this functionality.

1.2   A Case for Generic Programming

Generic Traversals   Some common operations on structured data, eg. abstract syntax trees require a lot of boilerplate code to traverse the datastructure and modify it recursively, or extract some result. This boilerplate code needs to be adapted to each new datatype.

When writing specific traversals over an AST using pattern matching over the constructors, it often happens that only a few cases carry the meaningful computation, others being default cases. Boilerplate removal allows us to write such function by only giving the meaningful cases. In addition to conciseness, this has the benefit of making the code robust to changes in the AST type: since the same function would treat additional constructors using the default case. Generic traversals is the focus of section 3.

Ad-hoc Polymorphism   OCAML lacks an overloading mechanism such as Haskell type-classes. The generic library implements similar mechanisms, through explicit type representation and dynamic dispatch. This feature is illustrated in the generic traversals of section 3 in which we adapted some Haskell libraries that rely heavily on type-classes.

Safer Alternatives to the Built-in Generic Functions in OCAML   The OCAML standard library provides a few functions that perform black magic. Such functions are defined over the concrete memory model of the OCAML value runtime representation. In fact one of them is actually called `magic` : $\alpha \rightarrow \beta$ and does what its type suggests: casting a value to an arbitrary type, which is unsafe. Deserialization, as implemented by `Marshal.from_string` is also unsafe. Such operations can provoke segmentation faults if used unwisely. Other magical operations—such as polymorphic comparisons and the polymorphic hash function—break the abstraction provided by abstract types: such types are often defined as quotients over an equivalence relation, yet the structural comparisons work on their concrete implementation instead of the equivalence classes.

With a generic programming library, the user can define alternatives to the built-in functions that are well-behaved regarding both type safety and abstraction.

1.3   Overview of the article

Section 2 explains the three elements of a generic programming library: means to reflect types, to define type-dependent functions and to represent the structure of types. Section 3 shows how generic traversals can be defined on top of the library. Section 4 covers a complex generic program implementing safe deserialization. Section 5 gives some context to our approach, which is compared with other implementations. We also discuss genericity within other type systems. Section 6 sums up the main points of the article.

2   The Three Elements of Generic Programming

Following the *Generic Programming in 3D* approach [13], we identify three orthogonal dimensions in the design of generic programming libraries:

A reflexion of types at the value level over which our generic functions are defined.

A mechanism for overloading that enables us to define and call generic functions over different types.

A generic view of types that provides a uniform representation of types on top of which generic functions are recursively defined.

We describe each dimension in turn and finish the section with some examples of generic programs.

2.1   Type Reflexion

Generalised algebraic datatypes (GADT), introduced in OCAML version 4, are *type indexed* families of types. Using GADTs, we can define singleton types where each index of the family is associated with a single data constructor. The one to one correspondence between type indices and data constructors allows us to reflect types as values.

The syntax of GADT extends the syntax of variants by allowing the return type to be specified, where the indices may be instanciated to concrete types. Hence, we may reflect types as follows:

```
type _ ty =
    | Int : int ty
    | String : string ty
    | List : α ty → α list ty
    | Pair : α ty × β ty → (α × β) ty
    | Fun : α ty × β ty → (α → β) ty
```

Notice how we reflected type formers as value constructors of the same arity with type witnesses as arguments. A complex type is reflected straightforwardly:

```
Fun (List String, Fun (Int, (Pair (String, Int))))
    : (string list → int → string × int) ty
```

2.1.1   Open Types

Introduced in version 4.02, open types allow us to extend ty with new cases reflecting newly introduced user types. We declare an extensible type with:

```
type _ ty = ..
```

New cases are added with the syntax:

```
type _ ty += Float : float ty
type _ ty += Btree : α ty → α btree ty
```

Objects and Polymorphic Variants   Objects of anonymous classes and polymorphic variants are special amongst OCAML types in that they are not nominal types. Therefore, they do not fit nicely with the nominal type witnesses. One possibility to support them indirectly is to give them a name. Another possibility is to break the general scheme of type witnesses, and provide two special constructors for objects and polymorphic variants:

```
type _ ty += Object : α object_desc → α ty
type _ ty += PolyVariant : α polyvariant_desc → α ty
```

With suitable generic views object_desc and polyvariant_desc described in section 2.3.5.

2.2 Type-Indexed Functions

With type reflexion we can write type-indexed functions, for instance a pretty printer has the following type:

val show : α ty $\rightarrow$ α $\rightarrow$ string

Note how the reflected type is also used as a parameter of the function.

To implement show we need another extension to OCAML type system introduced in version 4.00: *locally abstract types*. This type annotation is necessary to help the type checker while pattern matching over a GADT: since the type indices of a GADT may be instantiated to different concrete types depending on the constructor case, which is not possible with the classical Hindley-Milner algorithm. In addition, our function uses polymorphic recursion: a call show (List a) recurses on the type of the list elements: show a, this requires the explicit polymorphic quantification of the locally abstract type a.

```
let rec show : type a . a ty → a → string
    = fun t x → match t with
        | Int → string_of_int x
        | String → "\"" ^ x ^ "\""
        | List a → "[" ^ String.concat "; " (List.map (show a) x) ^ "]"
        | Pair (a, b) → "(" ^ show a (fst x) ^ ", " ^ show b (snd x) ^ ")"
        | Fun (a, b) → "<fun>"
```

Such a definition by pattern matching is suitable for a closed type universe were the cases may be given exhaustively. However we want our universe to be extensible, so that we may add new types. Consequently the type indexed functions must also be extensible: as new type witnesses are added to ty, new cases must be added to the type indexed functions.

This problem problem of extending a datatype and a function on that datatype is known as Wadler's expression problem [32] and is indicative of the modularity of the language. Solutions in Haskell have been given involving type classes [30, 1] and cannot easily be adapted to OCAML. However, OCAML v. 4.02 introduced extensible variant types. The only missing ingredient is extensible functions. The rest of the chapter explains how they are implemented in the library.

2.2.1 Extensible Functions

To define extensible functions, we will use the imperative features of OCAML. The idea is to keep a reference to a function which will be updated when a new case is added. The reference is kept private while the public interface offers the means to add a new case:

val show_ext : ($\forall \alpha$. α ty $\rightarrow$ α $\rightarrow$ string) $\rightarrow$ unit

However this type is not correct in OCAML because a polymorphic function is not allowed as an argument. Fortunately, we are allowed polymorphic record fields, thus we define:

type show_fun = {apply : $\forall \alpha$. α ty $\rightarrow$ α $\rightarrow$ string}
val show_ext : show_fun $\rightarrow$ unit

We may already use this public interface to define the cases above. Once again the GATD forces us to provide type annotations. Compiler warnings also encourage us to explicitly raise an exception for the cases that do not concern us. Note that we use show for the recursive calls.

```
let () = begin
    show_ext { apply = fun (type a) (t : a ty) (x : a) → match t with
                            | Int → string_of_int x
                            | _ → raise Not_found };
    show_ext { apply = fun (type a) (t : a ty) (x : a) → match t with
                            | List a → "[" ^ String.concat "; " (List.map (show a) x) ^ "]"
                            | _ → raise Not_found };
    show_ext { apply = fun (type a) (t : a ty) (x : a) → match t , x with
                            | (Pair (a,b)) , (x, y) → "(" ^ show a x ^ ", " ^ show b y ^ ")"
                            | _ → raise Not_found };
end
```

All this syntactic noise could be avoided by the use of a PPX [20] providing the following syntactic sugar:

```
[%extend] show Int x = string_of_int x
[%extend] show (List a) x = "[" ^ String.concat "; " (List.map (show a) x) ^ "]"
[%extend] show (Pair (a, b)) (x,y) = "(" ^ show a x ^ ", " ^ show b y ^ ")"
```

2.2.2 A Simple Implementation of Extensible Functions

How can we implement show_ext? First we need to define a reference to a show_fun record.

```
val show_ref : show_fun ref
```

The reference is initialized to a function that always fails.

```
let show_ref = ref {apply = fun t x → failwith "show: type not supported yet"}
```

This reference is private, we define two public functions: show to call the function in the reference, and show_ext to update it.

```
let show t x = !show_ref.apply t x
```

To update the function, we simply try the new case, and resort to the previous version if it raises a Not_found exception.

```
let show_ext new_case =
    let old_show = !show_ref in
    show_ref := {apply = fun t x → try new_case.apply t x
                                    with Not_found → old_show.apply t x}
```

Semantics   The semantics of show depends on the order of the calls to show_ext, since the most recent extension is tried before the previous ones. When some patterns overlap between two extensions, it is the most recent extension that succeeds. This semantics is fragile since it depends on the order in which top level modules are linked.

2.2.3 A Generic, Efficient and Robust Implementation of Extensible Functions

The previous implementation of extensible functions has a couple of issues: (1) it is not general, the same boilerplate we wrote for show would need to be written for new functions, (2) it is not very efficient

because a list of cases is tried until one match succeeds and (3) it is fragile as the semantics depends on the order in which the cases are added at runtime.

The implementation we give now solves those problems by (1) using an encoding of higher-kinded polymorphism, (2) using a hashtable indexed by the constructors, and (3) using a partial order on patterns.

Higher Kinded Polymorphism   To state the obvious, the type of a type indexed function may depend on its type index. For instance

```
show : α ty → α → string
read : α ty → string → α
enumerate : α ty → α list
equal : α ty → α → α → bool
```

How can we define a general type for type-indexed functions? They are all instances of the same scheme, which would be expressed like this if OCAML allowed it:

```
type φ ty_fun = ∀α . α ty → α φ
```

For instance show : show_t ty_fun where **type** α show_t $= \alpha \to$ string. Unfortunately OCAML does not allow the use of type parameters of higher kinds (partially applied types like show_t above). However, we may use a defunctionalisation method to emulate them [37]. The idea is that (α, ϕ) app represents the type application of ϕ to the type α. We make app an extensible variant so that new constructors may be added as we need them.

```
type (α,φ) app = ..
```

Concretely, each type abstraction $\Lambda \alpha$. $\phi(\alpha)$ is represented by an empty type ϕ' and we add a new constructor App_ϕ such that (α, ϕ') app is isomorphic to $\phi(\alpha)$

```
type (_,_) app += App_φ : φ(α) → (α, φ') app
```

For instance, the list type former is represented by a type list' (with no parameters), and its semantics is given by:

```
type (_,_) app += App_list : α list → (α, list') app
```

Type Indexed Functions   Using app we can give a general type for type-indexed functions:

```
type φ ty_fun = { f : ∀α . α ty → (α,φ) app }
```

For instance, show may be defined using the type abstraction $\Lambda \alpha$. $\alpha \to$ string, represented with the abstract type show' and the app constructor

```
type (_,_) app += App_show : (α → string) → (α, show') app
```

So that show' ty_fun is isomorphic with $\forall \alpha$. α ty $\to \alpha \to$ string.

An extensible function is a collection of ty_funs. A collection is created with a function create returning a record ϕ closure whose field f is the extensible function, and field ext allows us to extend it with a new case by providing a type pattern and a ty_fun matching that type pattern.

```
val create : string → φ closure
type φ closure = {
    f   : ∀α. α ty → (α, φ) app;
```

```
        ext : ∀α. α pat → φ ty_fun → unit;
}
```

Type patterns are inductively defined using the type constructors of α ty plus a universal pattern Any that acts as a wildcard, matching any type. For simplicity, type patterns are defined as synonyms of type witnesses.

```
    type α pat = α ty
```

The constructor Any : $\forall\alpha$. α ty, may only be used in a context where a pattern is expected. For instance, we may extend our show function to lists with the statement:

```
    show.ext (List Any) {f = show_list}
```

show_list is a type indexed function that expects a type witness of the form List a.

```
    let show_list : type a . a ty → (a, show') app = function
        | List a → App_show (show_list_of a)
        | _ → invalid_arg "show_list: expected a list"
    val show_list_of : α ty → α → string
```

Extensibility   For fast application, we store the ty_fun's in a hashtable indexed by a type pattern where all the parameters of a type constructor are set to Any. For instance, if an indexed function has independent cases for Pair (Int, Bool), Pair (List String, Bool), Pair (Any, Bool), Pair (Float, Any), they will all be associated to the same entry in the hashtable, with key Pair (Any, Any).

For each constructor pattern we store a list of functions ordered by their type pattern so that when applying the extensible function to some given type, the more general patterns are tried after all the more specific ones have failed to match the type. This mechanism ensures that the behaviour of an extensible function does not depend on the order in which the cases are given. In the example above, Pair (Int, Bool) and Pair (List String, Bool) will be tried before Pair (Any, Bool) and Pair (Float, Any) will be tried before Pair (Any, Any).

The order on patterns is lexicographic: so that Pair (Int, Any) matches before Pair (Any, Int). Pair (Int,Int) matches before both of them, and Pair (Any, Any) matches after all of them. Any is the most general pattern and matches when all the other patterns fail to match.

This approach combines both *efficient* application in the most frequent case in which there will be only one definition per constructor, and *flexibility* as it allows nested pattern matching and the order in which the function is extended does not matter.

2.2.4   Type Equality and Safe Coercion

GADTs allows us to define type indexed types, which can also be seen as type predicates (or relations), and their values can be seen as the proof of the predicates. Hence we may define a binary type predicate for type equality, with a single constructor for the proof of reflexivity.

```
    type (_, _) equal = Refl : (α,α) equal
```

Pattern matching on the Refl constructor forces the type-checker to unify the type parameters of equal. This allows us to define a safe coercion function.

```
    let coerce_from_equal : type a b . (a, b) equal → a → b
        = function Refl → fun x → x
```

That definition is possible because by the time we match Refl, the type *b* is unified with *a* hence, the variable *x* may be used both with type *a* and *b*.

We define an extensible type-indexed function ty_equal.

val ty_equal : α ty $\rightarrow$ β ty $\rightarrow$ (α, β) equal option

Adding new cases to ty_equal is very systematic. They may be automatically derived by a PPX extension described in section 2.4.

```
ty_equal_ext Char { f = fun (type a) (type b) (a : a ty) (b : b ty) →
    match a , b with
    | Char, Char → Some (Refl : (a, b) equal)
    | _ , _ → None };
```

In the case of parametric types, all type parameters must be recursively checked for equality.

```
ty_equal_ext (List Any) { f = fun (type a) (type b) (a : a ty) (b : b ty) →
    match a , b with
    | List x, List y → (match ty_equal x y with
        | Some Refl → Some (Refl : (a, b) equal)
        | None → None)
    | _ , _ → None };
```

By composing the previous functions we may derive a type-indexed safe coercion function:

```
let coerce : type a b . a ty → b ty → a → b option
    = fun a b → Option.map coerce_from_equal (ty_equal a b)
```

Where **val** Option.map : (α $\rightarrow$ β) $\rightarrow$ α option $\rightarrow$ β option.

coerce is used as the basis for implementing the function children, as well as the Uniplate and Multiplate scrap functions (sections 3.1 and 3.2).

2.3   Generic Views

A generic view is a uniform representation of the structure of types. Most libraries are built around a single view. In our design, we allow the user to choose among many views and even to define his own.

Depending on the task, some views might be more appropriate than others. For instance, to implement safe deserialization in Section 4 we need a low level view reflecting the specificities of OCAML types. When such details can be ignored, a higher level view is more adequate and easier to work with. For instance, the *Multiplate* library is written on top of the list-of-constructors view.

The low level view desc is special in the library in that it is the primitive view and is automatically derived from type definitions using the PPX attribute reify described in section 2.4. All the high level views are defined as a generic function by using the low level view. We show some examples in Section 2.3.

2.3.1   What is a View?

A generic view is given by a datatype α view together with a type indexed function view : α ty $\rightarrow$ α view. It maps a type witness to a value giving the structure of the type.

The rest of the chapter describes some common views that are available in the library. For each view, we show how binary trees are represented and we give an implementation of the function children : α ty $\to \alpha \to \alpha$ list seen in the introduction.

Note that abstract types may have a public representation associated to them: their generic view gives the structure of that public representation.

2.3.2   Sum of Products

The sum of products view represents algebraic datatypes using finite products and finite sums. The implementation follows closely that of the Haskell LIGD library [4].

We must define an empty type and a binary sum type from which all finite sums may be constructed:

```
type empty
type (α, β) sum = Left of α | Right of β
```

The sum of products representation is given by an indexed type α sp whose index α is the type being represented. We first give a representation of finite sums and products by reifying the index:

```
type α sp =
   | Empty : empty sp
   | Sum   : α sp × β sp → (α, β) sum sp
   | Unit  : unit sp
   | Prod  : α sp × β sp → (α × β) sp
```

With only those constructors, α sp would only allow us to represent types α built out of empty, sum, unit and ($\times$). We extend it to user defined types (variants, records, etc) by providing an isomorphism between the user type and a representation as a sum of products.

```
   | Iso   : α sp × (α, β) Fun.iso → β sp
```

Meta information may be attached to the representation, for instance we may provide the name of variant constructors and record fields with:

```
   | Con   : string × α sp → α sp
   | Field : string × α sp → α sp
```

We provide a type witness for the types that cannot be represented as a sum of products, those are the base cases that require a specific behaviour: int, float, char, string, array, etc.

```
   | Base  : α ty → α sp
```

Finally, we need a constructor to delay the computation of a view, similar in its role to the lazy keyword in that it prevents an infinite term to be computed.

```
   | Delay  : α ty → α sp
```

Intuitively, the meaning of Delay t is lazy (view t). Delaying the recursive computation of the view is very useful. Without it, it wouldn't be possible to define the function children for instance (see below).

Sum of Products View for Binary Trees   The generic function sumprod : α ty $\to \alpha$ sp computes the sum of products representation of any type, it is derived from the low level desc view. In the case of binary trees, the view is equivalent to:

```
sumprod (Btree a) ≡
   Iso (Sum ( Con ("Empty", Unit)
            , Con ("Node", Prod ( Delay (Btree a)
                                , Prod (Delay a, Delay (Btree a)))))
       , { fwd = (function Left () → Empty | Right (l, (x, r)) → Node l x r)
         ; bck = (function Empty → Left () | Node l x r → Right (l, (x, r)))   })
```

Generic Equality A common pattern when defining a generic function is to define two mutually recursive functions, one working on the type witness and the other one working on a generic view. The first one may implement ad-hoc cases by pattern matching on the type witness and a generic case covering the other cases.

The function equal calls equal_*sp* on the sumprod view:

```
let rec equal : type a . a ty → a → a → bool
   = fun t → equal_sp (sumprod t)
and equal_sp : type a . a sp → a → a → bool
   = fun s x y → match s with                    (*   ...   *)
```

and equal_*sp* is by induction on the type structure. Equality on unit is trivial:

```
      | Unit → true
```

Equality for products is component wise.

```
      | Prod (a, b) → (match x , y with
        | (xa, xb) , (ya, yb)
           → equal_sp a xa ya && equal_sp b xb yb)
```

Equality on sums is also straightforward:

```
      | Sum (a, b) → (match x , y with
        | Left xa   , Left ya   → equal_sp a xa ya
        | Right xb , Right yb → equal_sp b xb yb
        | _ , _ → false)
```

Meta information is ignored:

```
      | Con  (_, a) → equal_sp a x y
      | Field (_, a) → equal_sp a x y
```

User types are delt with by recursion over the representation:

```
      | Iso (s, fb) → equal_sp s (fb.bck x) (fb.bck y)
```

The constructor Delay is used to delay the computation of a view to avoid infinite representations. Note the mutual recursion with equal since *t* is a type witness rather than a sum of product representation.

```
      | Delay t → equal t x y
```

Equality over the empty type is the empty function. We use empty_elim : $\forall \alpha$. Empty → α.

```
      | Empty → Sum.empty_elim x
```

For the basic types (char, float, etc) we resort to the primitive equality:

```
      | Base t → x = y
```

Children   The function children makes crucial use of the Delay constructor to check whether the delayed type is the same as the type of the term.

```
let rec children t = children_sp t (sumprod t)
and children_sp : type b . α ty → b sp → b → α list
    = fun t s x → function
    | Delay t' → child t t' x
```

The children lists of both components of a product are concatenated:

```
    | Prod (a, b) → (match x with (xa, xb)
        → children_sp t a xa @ children_sp t b xb)
```

All the remaining cases simply recurse, following the type structure.

```
    | Sum (a, b) → (match x with
        | Left xa → children_sp t a xa
        | Right xb → children_sp t b xb)
    | Con (_, s')
    | Field (_, s') → children_sp t s' x
    | Iso (s', fb) → children_sp t s' (fb.bck x)
```

Base and Empty have no children.

```
    | Base _ | Empty → []
```

child builds a singleton list only if the two type witnesses are equal.

```
    val child : α ty → β ty → β → α list
```

To implement child we use a function to coerce a value from a type to another if they are equal, coerce is a primitive of the library discussed in section 2.2.4.

```
    val coerce : α ty → β ty → α → β option
    let child a b x = list_of_opt (coerce b a x)
```

list_of_opt does what its name suggests:

```
    let list_of_opt = function None → [] | Some x → [x]
```

2.3.3   Spine

The spine view underlies the Scrap Your Boilerplate library for Haskell [12]. It allows to write generic functions very concisely compared to other views like the sum of product view. However it has some limitations, for instance it is not possible to define generators (such as a generic parser for instance). It is only really useful to define consumers (such as a generic print function).

The spine view is unusual in that it gives a view on a typed value rather than on the type alone:

```
    val spine : α ty → α → α spine
```

The spine representation shows the applicative structure of a value as a constructor applied to its arguments:

```
    type α spine =
    | Con : α → α spine
    | App : (α → β) spine × α ty × α → β spine
```

For instance, the value Node (Empty, 1, Node(Empty, 2, Empty)) is represented as

App (App (App (Con node, Btree Int, Empty), Int, 1), Btree Int, Node(Empty, 2, Empty))

Where node is a curried constructor function.

let node l x r = Node (l,x,r)

Spine View for Binary Trees   Let us look at the spine view of binary trees. We define a function spine_btree of type:

val spine_btree : α ty $\rightarrow$ α btree $\rightarrow$ α btree spine

such that spine (Btree a) = spine_btree a. Note that spine—like all the high level views—is in fact generically defined in terms of the low level view.

```
let spine_btree a = function
    | Empty → Con Empty
    | Node (left, x, right) → App (App (App (Con node, Btree a, left), a, x), Btree a, right)
```

Equality   Generic equality can be implemented using the spine view with a small modification to our spine type.

```
let rec equal : type a . a ty → a → a → bool
    = fun t → equal_het t t
and equal_het : type a b . a ty → b ty → a → b → bool
    = fun a b x y → equal_spine (view a x, view b y)
and equal_spine : type a b . a Spine.t × b Spine.t → bool
    = function
        | Con x , Con y → raise Undefined
        | App (f, a, x) , App (g, b, y) →
            equal_het a b x y
            && equal_spine (f, g)
        | _ , _ → false
```

First, notice that we need to generalise the type of equality to arguments of different types (heterogeneous equality). This is because the spine view for each argument gives rise to independent existentially quantified variables. As a result, when comparing two constructors for equality, we are left with two values of different types, since Con : $\alpha \rightarrow \alpha$ spine. And we are stuck, since not even primitive equality can work on different types. To fix our problem, we need to extend the spine datatype to carry more information about constructors, such as its name, arity, module, file location, and so on. If we make sure that the meta information associated to a constructor uniquely identifies it then we can finish our implementation of equal by changing the erroneous line with:

| Con (x, meta_x), Con (y, meta_y) → meta_x $\equiv$ meta_y

Using primitive equality to check that the meta information is indeed the same.

Children   Let us implement the children function from the introduction using the spine view.

> **let** children a x = children_spine a (spine a x)

children_spine takes the type witness of the tree—this is also the type of the children—and a spine whose type is different. This is because when we recursively go through the spine, the type of the spine changes.

> **val** children_spine : α ty → β spine → α list

If the spine is a constructor, it contains no child and we return the empty list. If the spine is an application f x, we collect the children of both f and x. A type annotation is necessary because the recursive calls change the type of b spine.

> **let rec** children_spine : **type** b . α ty → b spine → α list
> = **fun** t → **function**
> | Con _ → []
> | App (f, a, x) → children_spine t f @ child t a x

See how much simpler that definition of children is in comparison to the one using the sum of products view.

2.3.4   Low Level View

Whereas the high level views give a uniform structural representation of types, the low level view desc captures the particularities.

OCAML types are grouped in categories. Each of them is identified by a constructor of the desc view:

> **type** α desc =
> | Array : β ty × (**module** Array_intf **with type** t = α **and type** elt = β) → α desc
> | Product : β product × (β , α) iso → α desc
> | Record : (β,α) record → α desc
> | Variant : α variant → α desc
> | Extensible : α ext → α desc
> | Custom : α custom → α desc
> | Class : α class_t → α desc
> | Synonym : α synonym → α desc
> | Abstract : α abstract → α desc
> | NoDesc : α desc

Array   OCAML has a few array-like types: array, string and bytes. They can be handled generically using a common interface.

> **module type** Array_intf = **sig**
> **type** t
> **type** elt
> **val** length : t → int
> **val** get : t → int → elt
> **val** set : t → int → elt → unit
> **val** init : int → (int → elt) → t
> **val** max_length : int
> **end**

The rest of the array operations can be derived from this minimal interface.

The desc view for arrays consists of a witness for the type of the array elements, and a first class module of type Array_intf.

Product   Tuple types are a family of builtin types. The desc view for n-ary products consists of an isomorphism between the product and n nested binary products, *i.e.* $\alpha \times \beta \times \gamma \cong \alpha \times (\beta \times (\gamma \times \text{unit}))$.

Right-nested binary products are fully captured by the following indexed-type:

```
type α product =
  | Nil : unit product
  | Cons : α ty × β product → (α × β) product
```

The isomorphism is given by two functions which are required to be each other's inverse.

```
type (α,β) iso = {fwd : α → β; bck : β → α}
```

Record   A record type is described as a set of fields, each field is described by its name, type and a procedure to update its value if it is mutable.

To complete the description of a record type, it is associated to a product type and an isomorphism to convert between the record and the product.

```
type (β, α) record =
  { name : string
  ; module_path : string list
  ; fields : (β, α) fields
  ; iso : (β, α) iso
  }
type (β, α) fields =
  | Nil : (unit, α) fields
  | Cons : (β, α) field × (γ, α) fields → (β × γ, α) fields
type (α, ρ) field =
  { name : string
  ; ty : α ty
  ; set : (ρ → α → unit) option
  }
```

Variant   A variant is described as a set of constructors.

```
type α variant =
  { name : string
  ; module_path : string list
  ; cons : α cons
  }
```

The set of constructors is an abstract type cons. The primitive way to build a description of the set of constructors is through the function cons that turns a list of single constructor description into the abstract set cons.

```
val cons : α con list → α cons
```

Through the public functions, non-constant constructors and constant constructors are accessed separately and ordered by their internal tag. This information is crucial for checking the compatibility of runtime values with a given variant and is needed in the implementation of safe-deserialisation, see section 4.

The interface consists of functions to get the number of constant constructors, access a constant constructor of a given tag and construct the list of all constant constructors ordered by their tags.

```
val cst_len : α cons → int
val cst_get : α cons → int → α con
val cst : α cons → α con list
```

The same set of functions is provided for non-constant constructors.

```
val ncst_len : α cons → int
val ncst_get : α cons → int → α con
val ncst : α cons → α con list
```

Finally the list of all constructors (constant and non-constant) may be computed with con_list.

```
val con_list : α cons → α con list
```

Each constructor is described by its name, the types of its arguments given as a nested product, a function to embed a value of the nested product to the variant type, and a partial projection function that only succeeds when its argument is built with precisely this constructor.

```
type (β, α) con =
  { name : string           (*   name of the constructor    *)
  ; args : (β, α) fields     (*   arguments of the constructor    *)
  ; embed : β → α           (*   applies the constructor to the arguments.    *)
  ; proj : α → β option     (*   tries to deconstruct that constructor    *)
  }
```

Finally, the function conap : α cons → α → α conap deconstructs—in constant time—a value into a pair of a constructor and its arguments.

```
type α conap = Conap : (β, α) con × β → α conap
```

The function conap enjoys the following property: conap cs x = Conap (c, y) $\implies$ c.embed $y = x$.

Note that GADTs may be described as variants. The set of constructors may vary depending on the concrete type index of the GADT.

Extensible   Extensible variants allow new constructors to be added to a type after it has been defined. The generic view for extensible variants must also be extensible so that the description of the new constructors may be added to the description of the extensible variant.

```
type α extensible =
  { name : string
  ; module_path : string list
  ; ty : α ty
  ; cons : ext_cons
  }
```

The extensible set of constructors for the type α is given by the field `cons : ext_cons` where `ext_cons` is an abstract type. An initially empty set is created with:

```
val create : unit → cons
```

A public interface allows us to modify and query the set of constructors:

```
val add_con : α extensible → con → unit    (*   add the description of a new constructor   *)
val con_list :  α extensible → α con list   (*   return the list of existing constructors   *)
val con : α extensible → α → α con          (*   find the constructor of a given value   *)
val conap : α extensible → α → α conap      (*   deconstruct a value as a constructor application   *)
```

Finally, the function fix is of particular interest for deserialization:

```
val fix : α extensible → α → α
```

Serialising then deserializing an extensible value (including exceptions) does not preserve structural equality: `umarshall (marshall x)` $\neq x$. In particular, pattern matching the deserialized value does not work as expected. The function fix fixes that: when given a deserialized value, it returns a value structurally equal to the one that was serialized: `fix (unmarshall (marshall x))` $= x$

In details, when an extensible value has been deserialized, its memory representation will be different from that of the value before it was serialized. This is because constructors of extensible variants are implemented as object blocks (same as OCaml objects), and they get assigned a unique identifier when created. `fix ext x` replaces the constructor of x with the original constructor object that is stored in the α extensible datastructure ext.

Custom   Custom data are defined outside of OCAML, typically in the language C. They are considered as abstract from an OCAML perspective. The only information available is their identifier, given by the homonymous field of the C-struct `custom_operations` defined in `<caml/custom.h>`.

```
type α custom =
    { name : string
    ; module_path : string list
    ; identifier : string
    }
```

Generic support for custom types comes from the use of a public representation, just like with abstract types, see section 2.3.7.

Class   A class is given as a list of methods.

```
type α class_t =
    { name : string
    ; module_path : string list
    ; methods : α method list
    }
```

A method has a name, a type, and a function that is executed when the corresponding method is called on an object of the class.

```
type α method = Method : (β, α) method_desc → α method
type (β, α) method_desc =
```

```
{ name : string
; send : β → α
; ty : β ty
}
```

For example, the point class

```
class point init = object
    val mutable x = init
    method get_x = x
    method move d = x ← x + d
end
```

is described by

```
let get_x = { name = "get_x"; send = (fun c → c #get_x); ty = Int }
and move = { name = "move"; send = (fun c → c #move); ty = Fun (Int, Unit) }
in Class { name = "Point"; methods = [ Method get_x; Method move ] }
```

Synonym   The view for a type synonym **type** $s = t$ consists of a type witness for t and a proof that the two types are equal. The equality type corresponds to the equivalence of type synonyms in OCAML; it was introduced in section 2.2.4.

```
type α synonym =
    { name : string
    ; module_path : string list
    ; ty : β ty
    ; eq : (β,α) equal
    }
```

Abstract   No information is associated with an abstract type, except for its name, thus respecting the desire of the programmer to hide the concrete implementation of the type. Still, we may run generic functions over abstract types if they have a public representation. This is explained in section 2.3.7.

```
type α abstract =
    { name : string
    ; module_path : string list
    }
```

NoDesc   The constructor NoDesc is used to signify that a view is not yet, or cannot be, associated with a type. For instance a function type does not have a meaningful generic view.

2.3.5   Objects and Polymorphic Variants

Objects types and polymorphic variants types are structural types rather than nominal types: an object type is given by the set of his methods signatures, and a polymorphic variant type is given by the set of its constructors signatures. Since there is no type name to be reflected in a type witness, one must instead provide a view as the type witness.

```
type _ ty += Object : α object_desc → α ty
type _ ty += PolyVariant : α polyvariant_desc → α ty
```

To describe objects, the method datatype of the previous section is reused:

```
type α object_desc = α method list
```

Values of polymorphic variant types have a different memory representation than values of normal variants, as such they need a distinct generic description. Each data constructor of a polymorphic variant is associated to a hash value, thus we provide operations to compute that hash value and operations to compute the data constructor corresponding to a given hash.

We define an abstract type α poly_variant representing the set of constructors of a polymorphic variant type α, together with functions to create a poly_variant and extract a constructor and compute its hash value.

```
type α poly_variant
val poly_variant : α Con.t list → α poly_variant
val hash : α con → int
val find : α poly_variant → int → α con
```

conap deconstructs a polymorphic variant value into its data constructor and its arguments (See the paragraph on variants, in section 2.3.4).

```
val conap : α poly_variant → α → α conap
```

Important note: the support for objects and polymorphic variant types is very fragile and should be considered experimental. It is not obvious how they may be compared for equality. As a result we may not yet extend a type-indexed function with a case for an object or polymorphic variant type. It is however possible to define generic functions that work on them (through a generic view). An example is the deserialization function presented in section 4.

2.3.6   List of Constructors

The list of constructor view is similar to the underlying view of the RepLib Haskell library [33]. In a nutshell, the view sees all types as variants. Products and records are viewed as variants of a single constructor. The other categories of types do not fit well under that description, and are left as base cases with no constructors.

The view is similar to the sum-of-products view in the sense that each constructor is associated to a product and the type is the sum of those products.

We reuse the type of constructor descriptions con defined for the low level desc view. For instance, given the witness of the type parameter a : α ty, the list-of-constructors view for α btree is:

```
let cons_btree a =
    let empty = Con
        { name = "Empty"
        ; args = Nil
        ; embed = (function () → Empty)
        ; proj = (function Empty → Some () | _ → None)
        }
    and node = Con
        { name = "None"
```

```
            ; args = f3 (Btree a) a (Btree a)
            ; embed = (function (l,(x,(r,()))) → None (l, x, r))
            ; proj = (function Node (l,x,r) → Some (l,(x,(r,()))) | _ → None)
            }
    in [ empty; node ]
```

f3 computes a lists of three fields with empty labels.

```
    val f3 : α ty → β ty → γ ty → (α × (β × (γ × unit)), δ) fields
```

Equality   Typically, functions using the list-of-constructors view are defined by three mutually recursive functions: the first works with type witnesses and delegates the work on the view, the second works on the view and iterates through the list of constructors to find the matching constructor, the third works on the product of arguments of a single constructor.

```
    let rec equal : type a . a ty → a → a → bool
      = fun t → match conlist t with
          | [] → ( = )                              (*   Base case (core types)   *)
          | cs → equal_conlist cs                   (*   Generic case   *)


    and equal_conlist : type a . a con list → a → a → bool
      = fun cs x y → match conap cs x with
          | Conap (c,x') → match c.proj y with
              | None → false                        (*   Not the same constructor   *)
              | Some y' → equal_prod (product c) x' y'


    and equal_prod : type p . p product → p → p → bool
      = function
          | Nil → fun _ _ → true
          | Cons (t, ts) → fun (x,xs) (y,ys) → equal t x y && equal_prod ts xs ys
```

A particularly useful function is conap which deconstructs a value into a constructor and its arguments, it has the same semantics as the homonymous function on variants given earlier, but this one has a linear complexity since it must walk through its list argument in order to find a matching constructor.

```
    val conap : α con list → α → α conap
```

Children   The list-of-constructor view makes our job really easy here: conap computes the list of all children, whatever their types, and we only need to keep those that have the same type as the parent. The function child was defined p. 12.

```
    let rec filter_child t = function
        | Nil , () → []
        | Cons (t', ts) , (x, xs) →
            child t t' x @ filter_child t (ts, xs)


    let children t x = match conap (conlist t) x with
        | Conap (c, y) → filter_child t (product c, y)
```

2.3.7 Abstract Types

Abstract types are an essential element of modular programming. Separating the public interface from the concrete implementation allows us to change the implementation without consequences for the users of the module. Generic functions should respect the abstraction, therefore the concrete type structure of an abstract type should not be available through the generic views. This is why the low level view provides a constructor Abstract for abstract types that only exports their names. However, in order to compute anything useful, one needs a generic view to convert back and forth between the abstract type and a public representation on which the generic functions may act.

The view is given by a type α repr that specifies a representation for an abstract type α, and a type-indexed function repr that returns the representation associated to a type witness:

val repr : α ty $\rightarrow$ α repr

The type α repr is existentially quantified over the representation type β:

type α repr = Repr : (α,β) repr_by $\rightarrow$ α repr

type (α,β) repr_by specifies how the abstract type α is represented by the type β. It is a record type whose fields we explain below:

repr_ty	: β ty	Witness of the representation type.
to_repr	: $\alpha \rightarrow \beta$	Conversion from the abstract type to the representation.
from_repr	: $\beta \rightarrow \alpha$ option	Partial conversion from representation to the abstract type. It may fail with None if the representation is not valid.
default	: α	A default value.
update	: $\alpha \rightarrow \beta \rightarrow$ unit	When possible, update x y should modify in place the abstract value α to match the representation β.

The reason for the last two fields will become clear when we explain the implementation of type safe deserialization in section 4, which was the primary reason for including them.

Example An abstract type for natural numbers implemented as int.

The module signature hides the implementation of nat. The type witness Nat must be exported as well if we want to support generic programming. However, the views desc and repr are extended as side effects and are not visible in the signature.

```
module Nat : sig
    type nat
    type _ ty += Nat : nat ty
end = struct
    type nat = int
    type _ ty += Nat : nat ty
```

We define Nat as Abstract in the low level view.

```
Desc_fun.ext Nat {f = fun (type a) (t : a ty) → (match t with
    | Nat → Abstract { name = "nat"; module_path = ["Test"; "Nat"] }
    | _ → assert false : a desc) };;
```

We define the representation using int and making sure that negative integers are not converted to nat:

```
let nat_repr = Repr
    { repr_ty = Int
    ; to_repr = (fun x → x)
    ; from_repr = (fun x → if x ≥ 0 then Some x else None)
    ; default = 0
    ; update = (fun _ _ → ()) }
```

The abstract view repr must be extended manually.

```
Repr.ext Nat { f = fun (type a) (t : a ty) → (match t with
    | Nat → nat_repr
    | _ → assert false : a repr) };;
  end                                              (*  end of Nat module  *)
```

2.4   Syntax Extensions

The library is compatible with OCAML version 4.04. In order to provide support for a user type τ, one should add a corresponding type witness τ ty, and add a corresponding case to the low level view τ desc, as well as τ ty desc, and also extend the type equality function ty_equal (section 2.2.4). This involves a lot of boilerplate which can be fully automated by using extension points (PPX).

When the structure item attribute [@@reify] is associated with a type declaration, a type witness obtained by capitalizing the type name is defined and the low level view is extended. The generated code is placed right after the type declaration.

Alternatively a global (floating) attribute [@@@reify−all], placed at the top of the file, ensures that every type declaration is reified, unless the type declaration is marked with [@@dont_reify].

The attribute [@@abstract] ensures that the low level view for that type is Abstract and hides the concrete structure of the type. However, the repr view must still be extended manually.

The attribute [@@no_desc] maybe used in case the user wants to provide his own implementation of the low level view for the type but still wants the type witness to be generated and a new case for the type equality function.

Potential Compile-Time Errors   Name conflicts may arise from the generated type witnesses, which are new data constructors extending the type ty: constructor names are obtained from type names by capitalizing them.

The user should make sure that all the necessary types and type witnesses are in scope. The PPX does not open any module. In particular, it is usually necessary to open Generic.Core.Ty.T which exports the witnesses for the builtin types (bool, char, int, int32, int64, nativeint, float, bytes, string, array, exn, ref, option, list, ty, unit, and tuples up to decuples).

When reifying a type, the witnesses for all the types that are mentionned in the definition should be in scope (fields of records, constructors of variants, definitions of synonyms).

Reifying GADTs   Currently the low level view for GADTs must be written by hand. The view that is derived by default for variant datatypes doesn't work with GADTs, one must use the attribute [@@no_desc] to prevent the generation of the view, or [@abstract] if one wants to make the type abstract.

Reifying Classes   Currently, classes are reified as abstract datatypes, the class representation (low level view) must be written by hand. In that case, one must use the attribute [@@no_desc].

Extensible Type-Indexed Functions   There is currently no syntax support for creating and extending type-indexed functions. Extending by hand the low level view desc with a case for btree requires the following boiler plate:

```
ext (Btree Any) { f = fun (type a) (ty : a ty) → (match ty with
    | Btree a → Variant {name = "btree"; module_path = ["Example"]; cons = cons_btree a}
    | _ → assert false : a Desc.t)
};
```

cons_btree was defined above, p. 19.

3   Boilerplateless Generic Traversals

The examples in this chapters are adapted from Mitchell [25]. Consider a simple expression language with constants, negation, addition, substraction, variables, and bindings.

```
type expr = Cst of int | Neg of expr | Add of expr × expr| Sub of expr × expr
            | Var of string | Let of string × expr × expr [@@reify]
```

Let us compute the list of all constants occurring in an expression:

```
let rec constants = function
    | Cst x → [x]
    | Neg x → constants x
    | Add (x, y) → constants x @ constants y
    | Sub (x, y) → constants x @ constants y
    | Var n → []
    | Let (n, x, y) → constants x @ constants y
```

This definition has the three characteristics of a boilerplate problem: (1) adding a constructor to the type would require adding a new case to the function. (2) most cases are repetitive and systematic, only one case here—Cst—is really specific. (3) the code is tied to a particular operation and cannot be shared.

In real world compilers, AST have many more constructors and those issues become all the more frustrating. Generic traversals are the answer.

With a generic function family : α ty → α → α list that returns the list of all the subexpressions of an expression, the previous example may be written:

```
let is_cst = function
    | Cst k → [k]
    | _ → []
let constants e = List.concat (List.map is_cst (family Expr e))
```

Notice that (1) *is*_cst only mentions the constructor Cst, therefore adding new constructors to the type wouldn't break the behaviour of constants, (2) the repetitive cases have disappeared, (3) the traversal code is shared in the library function family.

A few libraries for Haskell offer a similar functionality, of which Uniplate and Multiplate were our main inspiration.

3.1   Uniplate

The whole Uniplate library[2] relies on a single generic function scrap which may easily be implemented using the spine view, or the list-of-constructors view.

> **val** scrap : α ty $\rightarrow$ β $\rightarrow$ β list $\times$ (β list $\rightarrow$ α)

scrap a b returns the list of children of a value of type a and a function to replace the children. By children, we mean the maximal substructures of the same type. For instance, the tail of a list is the only child of a non-empty list.

3.1.1   Children, Descendents, Family

From scrap we can of course derive children and replace_children which are simply the first and second components of the result:

> **val** children : α ty $\rightarrow$ β $\rightarrow$ β list
> **val** replace_children : α ty $\rightarrow$ α $\rightarrow$ α list $\rightarrow$ α

Note that replace_children expects a list of the same size as the one returned by children, that property is only checked at runtime.

Let us define a *descendent* of a value as either the value itself or a descendent of one of its children. The *family* is the set of all the descendents of a value.

> **let rec** family a x = x :: List.concat (List.map (family a) (children a x))

Most applications of family consist in filtering the descendents and extracting some information.

3.1.2   Transformation and Queries

A transformation is modifying a value and has some type $\alpha \rightarrow \alpha$, whereas a query is a extracting some information: its type is $\alpha \rightarrow \beta$. The generic traversals in Uniplate are higher-order functions that take a transformation or a query to compute a more complex transformation or query. For instance, one can define a non-recursive transformation to rename a variable and use the combinator map_family to apply it recursively on an AST, with the effect of changing every variable of an expression.

3.1.3   Paramorphisms

A paramorphism is a bottom-up recursive function whose inductive step may also depend on the initial value in addition to the recursive results [24]. Accordingly, we express the inductive step of our para operator as a function with type $\alpha \rightarrow \rho$ list $\rightarrow \rho$, that takes the initial value and the list of the children's results. The para combinator then recursively applies the inductive step to compute a result for the whole expression.

> **val** para : α ty $\rightarrow$ ($\alpha \rightarrow \rho$ list $\rightarrow \rho$) $\rightarrow$ $\alpha \rightarrow \rho$
> **let rec** para a f x = f x (List.map (para a f) (children a x))

As an example, the family function could be expressed as a paramorphism:

> **let** family a = para a @@ **fun** x xs $\rightarrow$ x :: List.concat xs

---

[2]The combinators have been renamed for consistency with the Multiplate library, a correspondence is given in section 5.

The height function from the introduction may also be computed directly as a paramorphism:

```
let height a = para a @@ fun _ children_heights →
    match children_heights with
        | [] → 0
        | h :: hs → 1 + List.fold_left max h hs
```

3.1.4   Top-Down Transformations

map_children rewrites each child of the root using a given transformation.

```
let map_children : α ty → (α → α) → (α → α)
    = fun a f x → let (children, replace) = scrap a x
                     in replace (List.map f children)
```

Let us define substitution for our expressions. First we define a finite map for our environments:

```
module Env = Map.Make (struct type t = string;; let compare = Pervasives.compare end)
type env = expr Env.t
```

Substitution is only really concerned with two cases, Let and Var, the rest of the cases involve recursing on the children (and rebuilding the term).

```
let rec subst : env → expr → expr
    = fun env → let open Env in function
        | Let (n, x, y) → let env' = filter (fun n' _ → n ≠ n') env
                             in Let (n, subst env x, subst env' y)
        | Var n when mem n env → find n env
        | x → map_children Expr (subst env) x
```

3.1.5   Bottom-Up Transformations

map_family recursively applies a transformation in a bottom-up traversal.

```
val map_family : α ty → (α → α) → (α → α)
```

For instance, on a list $[x; y; z]$ the transformation is applied along the spine of the list.

```
map_family (List a) f [x; y; z] ≡ f (x :: f (y :: f (z :: f [])))
```

The families of the children are transformed before the value itself:

```
let rec map_family a f x = f (map_children a (map_family a f) x)
```

For instance, we may remove double negations. The one-step transformation is applied bottom-up, ensuring that all double negations are removed.

```
let simplify = map_family Expr @@ function
    | Neg (Neg x) → x
    | x → x
```

We may implement constant folding, i.e. evaluate the subexpressions involving only constants:

```
let const_fold = map_family Expr @@ function
    | Add (Cst x, Cst y) → Cst (x + y)
    | Sub (Cst x, Cst y) → Cst (x − y)
    | Neg (Cst x) → Cst (−x)
    | x → x
```

3.1.6   Normal Forms

In some cases, we want to apply a rewriting rule exhaustively until a normal form is reached. The rewriting rule is given as a function of type $\alpha \to \alpha$ option which returns None when its argument is in normal form and otherwise performs one reduction step.

val reduce_family : α ty $\to$ ($\alpha \to \alpha$ option) $\to \alpha \to \alpha$

reduce_family applies the rewriting rule until it returns None for all the descendents of the result.

```
let rec reduce_family a f x =
    let rec g x = match f x with None → x | Some y → map_family a g y
    in map_family a g x
```

We may extend our previous example with another rewrite rule to remove the use of substraction from our expressions:

```
let simplify_more = reduce_family Expr @@ function
    | Neg (Neg x) → Some x
    | Sub (x, y) → Some (Add (x, Neg y))
    | _ → None
```

The rewrite rule for Sub introduces a Neg constructor, which is itself on the left hand side of a rewrite rule, this may create a new rewriting opportunity: for instance Sub x (Neg y) rewrites to Add (x, Neg (Neg y)) which in turns rewrites to Add (x,y). Using reduce_family ensures that no rewriting opportunity is missed.

3.1.7   Effectful Transformations

Finally, all the traversals combinators have an effectful counterpart that threads the effects of an effectful transformation. We used the encoding of higher-kinded polymorphism explained earlier p. 7 to parameterise the functions over applicative functors and monads.

val traverse_children : ϕ applicative $\to \alpha$ ty $\to$ ($\alpha \to (\alpha, \phi)$ app) $\to$ ($\alpha \to (\alpha, \phi)$ app)
val traverse_family : ϕ monad $\to \alpha$ ty $\to$ ($\alpha \to (\alpha, \phi)$ app) $\to$ ($\alpha \to (\alpha, \phi)$ app)
val mreduce_family : ϕ monad $\to \alpha$ ty $\to$ ($\alpha \to (\alpha$ option, $\phi)$ app) $\to$ ($\alpha \to (\alpha, \phi)$ app)

We will show later how one could use traverse_family with a state monad to rename each variable to be unique. Beforehand, we must introduce some definitions.

Functors, Applicative, Monad, Monoid   Using our encoding of higher-kinded polymorphism, we define the operations of functorial, applicative and monadic types. We assume the reader knows about those operations, they have been extensively discussed in the literature [31, 23]

```
type φ functorial =
    { fmap : α β . (α → β) → (α, φ) app → (β, φ) app
    }
type φ applicative =
    { pure : α . α → (α,φ) app
    ; apply : α β . (α → β, φ) app → (α, φ) app → (β, φ) app
    }
type φ monad =
    { return : α . α → (α,φ) app
    ; bind : α β . (α , φ) app → (α → (β, φ) app) → (β, φ) app
    }
```

Applicative functors and monads are functors:

```
val fun_of_app : α applicative → α functorial
val fun_of_mon : α monad → α functorial
let fun_of_app {pure; apply} =
    {fmap = fun f → apply (pure f)}
let fun_of_mon {return; bind} =
    {fmap = fun f kx → bind kx (fun x → return (f x))}
```

Monads are applicative functors:

```
val app_of_mon : α monad → α applicative
let app_of_mon ({return; bind} as m) =
    { pure = return
    ; apply = fun kf kx → bind kf (fun f → (fun_of_mon m).fmap f kx)
    }
```

Pure functions may be lifted to an applicative functor or a monad, with functions liftA,..liftA4, and liftM variants. For instance, liftA2 lifts a binary function:

```
val liftA2 : φ applicative → (α → β → γ) →
    (α, φ) app → (β, φ) app → (γ, φ) app
```

We may traverse a list, executing an effectful function on each element.

```
val traverse : φ applicative → (α → (β, φ) app) → α list → (β list, φ) app
let rec traverse a f = function
    | [] → a.pure []
    | h :: t → liftA2 a cons (f h) (traverse a f t)
```

A specific case of traversing is when the list contains effectful elements. We may sequence the effects of the element and obtain an effectful list of pure elements. The list functor and the applicative functor commute.

```
val sequence : φ applicative → (α, φ) app list → (α list, φ) app
let sequence a = traverse a (fun x → x)
```

We may derive monadic versions of traverse and sequence:

```
let traverseM m = traverse (app_of_mon m)
let sequenceM m = sequence (app_of_mon m)
```

Reader Monad   The reader monad is parameterised by the type of an environment β which may be read as a side effect of a monadic computation. A value in the reader monad is a function $\beta \rightarrow \alpha$ from the environment to a result.

```
type β reader = READER
type (_,_) app += Reader : (β → α) → (α, β reader) app
let run_reader = function
    | Reader f → f
    | _ → assert false
```

return brings a pure value into the reader monad, the environment is ignored. bind $x\ f$ passes the environment to both x and f.

```
let reader =
    { return = (fun x → Reader (fun env → x))
    ; bind = (fun x f →
                Reader (fun env →
                    let y = run_reader x env
                    in run_reader (f y) env))
    }
```

In addition to the monad primitives, the reader monad has a primitive ask to access the environment and local to run a reader action in a modified environment.

```
val ask : (α, α reader) app
val local : (α → β) → (γ, β reader) app → (γ, α reader) app
let ask = Reader (fun x → x)
let local modify r = Reader (fun env → run_reader r (modify env))
```

State Monad   The state monad is parameterised by the type of a state β and allows the threading of a state as a side effect of a monadic computation. A value in the state monad is a function $\beta \rightarrow \alpha \times \beta$ from an initial state to a result and a new state.

```
type β state = STATE
type (_,_) app += State : (β → α × β) → (α, β state) app
let runState = function
    | State f → f
    | _ → assert false
```

return brings a pure value into the state monad. The state is left untouched. bind $x\ f$ runs the stateful x with a state s obtaining a result y and a new state s'; then f is applied to y yielding a stateful computation which is run in the new state s'.

```
let state =
    { return = (fun x → State (fun s → (x, s)))
    ; bind = (fun x f →
                State (fun s →
                    let (y, s') = runState x s
                    in runState (f y) s'))
    }
```

In addition to the monad primitives, the state monad has a primitive get to access the state and set to replace the state with a new one.

```
val get : (α, α state) app
val put : α → (unit, α state) app
let get = State (fun s → (s, s))
let put s = State (fun _ → ((), s))
```

Abstracting over Constants   Consider the task of replacing all constants in an expression with unique variables. We will use the state monad to hold a counter. Let us write a function that increments the counter and returns the last value:

```
val incr : (int, int state) app
let incr = let (≫=) = state.bind and return = state.return in
    get          ≫= fun i →
    put (i+1)   ≫= fun () →
    return i
```

The core of the program uses the effectful traversal combinator traverse_family to recursively apply the transformation in a bottom-up traversal of the expression.

```
let abstract_state = traverse_family state Expr @@ function
    | Cst _   → liftM state (fun i → Var ("x" ^ string_of_int i)) incr
    | x → state.return x
```

The main function runs the stateful action with an initial counter value.

```
val abstract : expr → expr
let abstract x = run_state (abstract_state x) 0
```

Free Variables[3]   To collect the free variables of an expression, we will use the reader monad to keep track of the variables in scope. The reader environment is the list of variables in scope.

```
type scoped = string list reader
```

We may define the function in_scope that checks if a variable is in the environment:

```
let in_scope n = Reader (List.mem n)
```

We also need to extend the scope with a new bound variable. extend_scope n c runs the scoped computation c in the scope extended with n.

```
val extend_scope : string → (α, scoped) app → (α, scoped) app
let extend_scope n = local (fun ns → n :: ns)
```

The function free_vars is run in an initially empty scope:

```
val free_vars : expr → string list
let free_vars x = run_reader (free_vars_scoped x) []
```

---

[3]This example is adapted from Sebastian Fischer's blogpost `http://www-ps.informatik.uni-kiel.de/˜sebf/` `projects/traversal.html`

The core of the algorithm uses the fold combinator to recursively apply an inductive step. The step takes a list of scoped lists of free variables *rs* which are sequenced and concatenated into a scoped list of free variables *r*. Only two cases are significant: when the expression is a variable, we check if it is bound or free before returning either the empty list or a singleton; when an expression is a let binding, we run the scoped result in an extended scope including the new bound variable.

```
val free_vars_scoped : expr → (string list, scoped) app
let free_vars_scoped = para Expr @@ fun expr rs →
    let r = liftM reader List.concat (sequenceM reader rs)
    in match expr with
        | Var n → reader.bind (in_scope n) (fun is_in_scope →
                    reader.return (if is_in_scope then [] else [n]))
        | Let (n, _, _) → extend_scope n r
        | _ → r
```

3.2   Multiplate

One of the design goal of the Uniplate library was the simplicity of its types [25]. However, this came at the cost of a loss of generality: the traversals are only expressed in terms of a single recursive type. The library had a slight generalisation to two mutually recursive types called *biplate*. However, yet another generalisation called *Multiplate* made it possible to deal with any number of mutually defined types [27].

Multiplate is very similar to Uniplate, it has the same combinators that take a simple transformation and apply it to the children or recursively to the whole family of descendents. However that transformation, rather than being on a single type $\alpha \to \alpha$, is a type indexed transformation that can transform any type $\forall \alpha \, . \, \alpha \; \text{ty} \to \alpha \to \alpha$. Accordingly, the notion of children in Multiplate is more general: now children can have any type. The children of a variant value are all the arguments of its constructor. The children of a record are all of its fields.

3.2.1   Deconstructing a value

Multiplate generalises the function scrap. Since the children have different types, we cannot use lists anymore. The concrete type for the tuple of children is captured by the product GADT already seen in p. 15.

```
type α scrapped = Scrapped : β product × β × (β → α) → α scrapped
```

The function scrap : $\alpha \; \text{ty} \to \alpha \to \alpha$ scrapped is quite simple to define using the list-of-constructors view:

```
let scrap_conlist : α Conlist.t → α → α scrapped
    = fun cs x → match Conlist.conap cs x with
        | Conap (c, y) → Scrapped (product c, y, c.embed)
let scrap : α ty → α → α scrapped
    = fun t x → match Conlist.view t with
        | [] → Scrapped (Nil, (), const x)
        | cs → scrap_conlist cs x
```

where product : (β, α) con → β product.

3.2.2 Plates

In Multiplate terminology, an effectful transformation is called a *plate*

> **type** ϕ plate = {plate : $\forall \alpha$. α ty → α → (α,ϕ) app}   (*   for some applicative functor ϕ *)

We also specialise plate for the identity and the constant functors, obtaining the types of pure transformations and queries:

> **type** id_plate = {id_plate : $\forall \alpha$. α ty → α → α}
> **type** β const_plate = {const_plate : $\forall \alpha$. α ty → α → β}

3.2.3 Applying an Effectful Transformation to the Children

All of Multiplate's combinators may be derived from a single combinator, called *Multiplate* in the original article [27] but that we renamed traverse_children for consistency with our presentation of Uniplate.

> **val** traverse_children_p : ϕ applicative → ϕ plate → ϕ plate

Thinking about the corresponding Uniplate function, traverse_children_p modifies the children of a value using a given effectful transformation. We also provide a version where the plate is inlined:

> **val** traverse_children : ϕ applicative → ϕ plate → α ty → α → (α, ϕ) app

The implementation is strikingly similar to the Uniplate version.

```
let traverse_children_p a f = {plate = fun t x →
    let Scrapped (p, cs, rep) = scrap t x
    in (fun_of_app a).fmap rep (traverse a f p cs)}
let traverse_children a f = (traverse_children_p a f).plate
```

The function traverse used above generalises the homonymous function on lists (p. 27) to tuples. It applies an effectful transformation to each component of a tuple from left to right, returning the modified tuple in an effectful context.

```
val traverse : φ applicative → φ plate → α product → α → (α, φ) app
let traverse a {plate} p x
    = let rec go : type p . p product × p → (p, φ) app
        = let open Product in function
            | Nil , () → a.pure ()
            | Cons (t, ts) , (x, xs) →
                let pair a b = (a, b) in
                liftA2 a pair (plate t x) (go (ts, xs))
    in go (p,x)
```

3.2.4 Module Interface

All the Uniplate functions can be generalised. Their type signature in Multiplate are:

> **val** children : α ty → α → dyn list
> **val** family : α ty → α → dyn list

val traverse_children_p : ϕ applicative $\rightarrow$ ϕ plate $\rightarrow$ ϕ plate
val map_children_p : id_plate $\rightarrow$ id_plate
val fold_children_p : τ monoid $\rightarrow$ τ const_plate $\rightarrow$ τ const_plate

val traverse_family_p : ϕ monad $\rightarrow$ ϕ plate $\rightarrow$ ϕ plate
val map_family_p : id_plate $\rightarrow$ id_plate
val pre_fold_p : τ monoid $\rightarrow$ τ const_plate $\rightarrow$ τ const_plate
val post_fold_p : τ monoid $\rightarrow$ τ const_plate $\rightarrow$ τ const_plate

val para_p : $(\rho$ list $\rightarrow \rho)$ const_plate $\rightarrow$ ρ const_plate

They all have variants with the result plate inlined. that map_ functions, are specialisations of the corresponding traverse_ functions to the identity functor. fold_children is a specialisation of traverse_children to the constant functor.

3.2.5   Open Recursion

In the OCAML compiler libraries, one can find the modules Ast_mapper and Ast_iterator whose purpose is to ease the definition of traversals over the Parsetree mutually defined types. Both Ast_mapper and Ast_iterator implement open recursion which takes the form of a large record mapper (*resp.* iterator) where each field corresponds to one of the mutually defined type and is a function that takes a mapper (*resp.* iterator) a value of the corresponding type, and outputs a value of the same type (*resp.* unit). Defining mappers (*resp.* iterators) is usually done by modifying a default record implementing the identity (*resp.* a traversal of the tree without side effect). Only specific fields of interests need to be modified, the rest being taken care of by the default behaviour.

The behaviour of Ast_mapper and Ast_iterator can be implemented using Multiplate by defining a recursive type:

type ϕ openrec = { run : ϕ openrec $\rightarrow$ ϕ plate }

The default records of Ast_mapper and Ast_iterator correspond to the openrec function default which uses the openrec parameter to continue the recursion below the immediate children of a value.

let default a = { run = **fun** r $\rightarrow$ traverse_children_p a (r.run r) }

Ast_mapper corresponds to the specialisation of openrec to the identity functor, while Ast_mapper corresponds to a use of openrec with the IO monad to embed OCAML effectful computations:

type io
val io : io monad

Effectful computations may be embedded in the IO monad with embed_io. They are functions from unit to some result type ['a] which may carry side effects when evaluated.

val embed_io : (unit $\rightarrow$ α) $\rightarrow$ (α, io) app

IO computations may be executed with [run_io].

val run_io : (α, io) app $\rightarrow$ α

Discussion   `Ast_mapper` and `Ast_iterator` are long pieces of boilerplate that cannot be reused for other AST types, and do not allow easy extension or addition of new operations. In contrast, Multiplate implements the same functionality (and more) in a concise implementation that exploits the theorical properties of applicative functors and monads. Built on top of the generic library, Multiplate may be used with any type with no boilerplate.

4   A Case Study: Safe Deserialization

Serialization and deserialization in OCAML are provided by the standard library module Marshal. They are generic functions defined in C that rely on the concrete structure of the runtime values of OCAML programs. There is a serious safety issue with the use of the deserialization primitive: since it must be able to reconstruct values of any type from an input channel, it is polymorphic in its return type:

> **val** `from_channel : in_channel` → α

This may easily cause a segmentation fault if the deserialized value is used with the wrong type. This problem may be solved using the generic library: the return type may be constrained by its witness:

> **val** `from_channel :` α `ty` → `in_channel` → α

The low level generic view gives us the type structure to guide the deserialization process.

4.1   Outline of the Algorithm

The algorithm follows the same approach as the work of Henry [7] to check that the result of the standard deserialization function is compatible with a given type. A substantial difference is that our implementation also deals with abstract types. This involves converting the abstract values to a public representation before serialization, and converting back the representation to the abstract type after deserialization. Therefore, the heart of the program is a function `convert` that not only deals with such conversions but also does the compatibility check. `convert` takes a `direction` argument (*to* or *from* the public representation), a type witness and converts directly the runtime values using the `Obj` module, which gives an API to the memory representation of OCAML values.

> **type** `direction` = `To | From`
> **val** `convert : direction` → α `ty` → `obj` → `obj`

`convert` is private to the module, the type obj is not shown to the user of the library. The module exports type safe functions that call `convert` internally.

> `to_string :` α `ty` → α → `string`
> `from_string :` α `ty` → `string` → α

`from_string` can safely cast the result of `convert` from obj to α, because the compatibility check ensures that the value of type obj is also a valid value of type α.

4.2   Type Compatibility

OCAML runtime values are either immediate values taking a word of memory minus one bit, or a pointer to a block of memory allocated in the heap, which has a header containing the size of the block and a tag indicating how the block is structured. Our goal is to check that such a runtime value is compatible with

a certain type. The structure of runtime values is accessible throught the standard module Obj and the structure of the type is described by the low-level generic view desc presented in section 2.3.4.

The algorithm is recursive. The base cases are immediate values: to check that a value is compatible with an int value for instance, we simply check that it is an immediate value. Checking a record involves checking that the value is a block of tag 0, that it has a number of fields corresponding to that of the record type, and recursively that each field is compatible with its corresponding type. To check that a value is compatible with a variant type, we must check that: either it is an immediate value and corresponds to one of the constant constructors, or it is a block whose tag corresponds to one of the non-constant constructors and the fields must be recursively checked with the types of the constructor arguments.

4.3   Sharing and Cycles

The structure of runtime values is a directed graph where the vertices are memory blocks and the edges are the pointers that may be stored in the fields of a block. Sharing and cycles in the graph raise two questions: (1) could we avoid checking again a value that has already been checked? (2) how do we ensure that the algorithm terminates? In a monomorphic setting, both questions may be answered by storing the addresses of blocks together with the witness of their expected type when they are visited for the first time, and upon subsequent visits, simply check that the new witness is equal to the stored one. However both sharing and cycles can be polymorphic. Here is a contrived example of an infinite tree with polymorphic recursion, whose representation is a (finite) cyclic graph:

```
type α t = Leaf of int | Node of α t × (α × α) t
let poly_cycle = let rec go : ∀α . α t = Node (Leaf 0, go) in go
```

The type parameter of the tree is not ever used, which makes it a bit pointless but this illustrates perfectly the sort of complex situations that our compatibility checker must deal with. In this context, the previous solution does't work anymore. Non-termination becomes an issue because the number of types to check is infinite.

If an OCAML value admits many types, then they must all be instances of a more general type scheme. In the previous example, the value go admits the types

$$\forall \alpha . \alpha\ t$$
$$\forall \alpha . (\alpha \times \alpha)\ t$$
$$\forall \alpha . ((\alpha \times \alpha) \times (\alpha \times \alpha))\ t$$
$$\dots$$

and so on, of which the first is the most general.

The *anti-unifier* of a set of types is the type that is minimally most general than any of them. For instance, the anti-unifier of int and bool is $\forall \alpha . \alpha$, and the anti-unifier of int list and bool list is $\forall \alpha . \alpha$ list.

Now, we may keep track of the anti-unifier so far associated with a block. The sequence of updated anti-unifiers is sure to reach a fixed-point in which case we no longer need to visit that block. When checking the poly_cycle example, the first visit would already be checking go with its most general type $\forall \alpha . \alpha$, hence no other visit would be performed.

Now the algorithm terminates, but it may be improved: when there is no cycle, it is faster to traverse the graph in topological order. During the traversal, all the expected types of a block may be collected before that block is visited. The anti-unifier of a block's set of expected types may then be computed and the block needs only be visited once. In the presence of cycles, one may still apply that strategy on the

strongly connected components of a graph and compute a lower bound for the anti-unifier of the roots of each strongly connected component before traversing it.

The interested reader is referred to Henry [7] for the theoretical background and justifications.

4.4 Abstract types

The standard (de)-serialization functions from the module `Marshal` break type abstraction since it becomes possible to inspect a value that has been serialized and to cast a value to an abstract type. In order to resolve that issue and respect the abstraction, we must serialize a public representation of the abstract value. This is the purpose of the datatype repr given in section 2.3.7. `convert To` uses the field `to_repr` to serialize the representation, while `convert From` uses the field `from_repr` on the deserialized value.

Abstract types introduce some complexity. Instead of `check : ` α ` ty → obj → bool` that returns a boolean when a value is compatible with a type, we define `convert : direction → ` α ` ty → obj → obj` that computes a new value where all the blocks of abstract types have been converted (in one direction or the other). `convert` may fail with an exception is the input value is not compatible with the type witness.

Note that when we serialize a value, we first convert it by recursively computing the representations of the abstract values are in its memory graph. The result of that conversion usually does not have a corresponding OCAML type in the program. However when converting back after deserializing a value, we rebuild a value of a valid type by recursively computing the abstract values.

The main challenge is that the conversion should preserve the graph structure yet recursively transform its subgraphs.

Sharing   Sharing in an acyclic graph does not cause too much trouble. In addition to keeping track of visited blocks and their most general type so far, we also memoize the function `convert`. The result of converting a block are stored, so that when the same block is visited again and the expected type is not more general, the previous result may be retrieved directly.

Cycles   Cycles in memory graphs on the other hand require a lot of care. When visiting a block, the presence of cycles means that we will visit the same block again even before the first visit is completed, therefore before we have been able to store the converted block.

One direction is easy: when serializing a value we must first convert abstract values using `to_repr`, obtaining the public representation wich may then be recursively converted. The solution is to introduce an indirection, an `obj option ref` which is set to `None` upon visiting a block for the first time, and updated with the result when the function returns. If during the visit, the same block is tried, the reference is already known even if the content is not. In a final traversal of the graph, we may remove all the indirections.

We might be tempted to do the same thing during a deserialization, however that approach fails because the order of the transformation is reversed: now we must first recursively convert the serialized value to obtain the public representation which may then be converted to the abstract type using `from_repr`. During the recursion, other occurences of `from_repr` will be called, but the cyclic input that we must give them is not fully computed yet. In order to fix that, we use the `default` abstract value provided by the `repr` view (section 2.3.7). When visiting a block for the first time, its `default` value is stored as a temporary result. Therefore the recursive calls visiting the same block will use that default value. The cycle has been broken and must be restored. This involves keeping track of all the pointers to the block so that they may be updated to point to the new block. And finally, since the public representations have changed, we use the `update` function from the `repr` view to update the corresponding abstract values.

5   Other Works

Generic programming is a very rich topic that we have barely touched in this article, the reader may consult the following tutorials for a deeper understanding [16, 10, 9]. Generic libraries have blossomed in the past twenty years, and the many different approaches have been compared extensively [29, 11, 13].

Our design with separate type witnesses and generic view was directly influenced by Hinze and Löh [13].

5.1   Views

The open design of the library enables the user to define his own views. In addition to the low level view, we have included the sum of product view underlying the LIGD library [4] and Instant-Generics [3, 22], the spine view underlying the SYB library [14] and the list-of-constructors view underlying RepLib [33].

Adapting other libraries is possible when their underlying type representation is first order—where closed types are reflected.

5.2   Type Representation

The SYB library relies on a type reflection using a non-parameterised type `TypRep` and an unsafe coercion operation [19]. In contrast, our type witness GADT reflecting its type parameter makes it possible to define a safe coerce (section 2.2.4).

`TypRep` is similar to our α ty in that it captures an open universe of types. However, α ty does so with an extensible variant ensuring strong type guarantees, whereas `TypRep` does so with a unique integer tag.

LIGD [4] and Replib [33] use a GADT for type representation in the same way as we did with α ty, however their representation is closed and is fused with the view. Note that open variants are not natively supported in Haskell.

With TypeCase [28], a GADT is made implicit through by using typeclasses to implement catamorphisms over the GADT. That technique may be used to make a Haskell-98 compatible library, since GADT are not valid Haskell-98. LIGD and PolyP have been adapted using TypeCase.

In Instant-Generics [3], the representation is given by a type family—a GHC extension which allows to define type functions.

5.3   Higher-Order Kinded Types

The choice of a type representation determines the universe of types that can be represented. Our library represents types with first-order kind.

In a first order representation, the list type constructor is represented as a data constructor of arity one:

$$\mathsf{List} : \alpha \text{ ty} \rightarrow \alpha \text{ list ty}$$

Whereas in a second order type representation, it would be represented as a constant data constructor:

$$\mathsf{List} : \mathsf{list'} \text{ ty'}$$

Where «list'» corresponds to the unapplied list constructor in our encoding of higher-order kinded type variables.

The latter approach allows us to define generic functions that work on type constructors, like a generic map:

val `gmap :` ϕ `ty'` $\rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha,\ \phi)$ `app` $\rightarrow (\beta,\ \phi)$ `app`

Ours was a choice of simplicity since OCAML does not have a native support for higher order kind type variables. A higher order kind generic library is possible in OCAML using the encoding presented in this article. More flexibility is obtained at the cost of more complexity.

In Haskell, representing higher kinded types is possible [13]. For instance, PolyP [15] and its library implementation [26] represent parameterised datatypes as fixed-points of functors. Generic Deriving [21], another Haskell library and GHC extensions, allows users to define generic instances of type-classes. It has two type representations: one for closed types and one for parametric types (of one parameter). More representations could be defined in the same way, and would allow users to derive class-instances for types of the corresponding kind.

5.4   Type Indexed Functions

The first version of SYB relied on operations `mkT`, `extT`, `mkQ`, `extQ`, `mkM`, `extM` to define extensible type-indexed functions. Their implemention suffered from the same shortcomings as the simple implementation given in section 2.2.2, and furthermore once a generic function was defined, no more ad-hoc cases could be added. A latter version of SYB [18] resolved this issue with a clever use of type-classes, requiring some extensions to the class system. With our explicit use of a type witness to define type-indexed functions, we face no such difficulties.

Most Haskell generic libraries rely on the powerful type-class system to implement type-indexed functions. Usually, they have a `Rep` class that builds the representation. A generic function is usually implemented by a class with a default instance depending on a `Rep` constraint to implement the generic behaviour. Ad-hoc behaviour may be defined by implementing instances of the class for specific types.

5.5   Language Extensions

Library implementations of genericity must fight with the limits set by the programming language. Extending the language with direct support for generic programming through a dedicated syntax and semantics gives much more freedom to the designer.

PolyP [15] represent parameterised datatypes as fixed-points of functors, which makes it possible to define a generic map. That extension initially implemented as a preprocessor was subsequently implemented as a library using extensions to the typeclass system [26].

Generic Haskell [10] is the most expressive of all generic systems so far. In Generic Haskell the type contains types of any kinds. Functions defined by induction on the structure of types have a type that is defined by induction on the structure of kinds. This allows a truly generic map that works on types of any kinds.

The level of expressivity of generic haskell may be achieved by a library in OCAML, but at such a cost in readability that one may wonder if that would be useful.

5.6   Generic Traversals

Our implementation of Uniplate and Multiplate follows directly from the work on the homonymous Haskell libraries [25, 27], but also on Fischer's implementation of Uniplate for his naming convention. The use of applicative functors for generic traversals has a small history [23, 6, 2, 27].

Other approaches to generic traversals include Compos [2] and SYB [19] which are equivalent to Multiplate in expressivity. The fundamental mechanism underlying Compos and Multiplate is the same. They differ in their Haskell embodiment by the way that type-classes are used. In our OCAML implementation, the type-classes are replaced with explicit type-indexed functions. In fact, the missing link between SYB, Compos and Multiplate, is yet another variation called Traverse-with-class [5]. It is the closest to what we have implemented: in essence, our single type-indexed function `traverse_children` corresponds to his single type-class `Gtraverse`:

```
class Gtraverse where
    gtraverse :: Applicative c => (forall d . GTraversable d => d → c d) → a → c a
```

The SYB primitive gfold corresponds to a catamorphism over the spine view [14].

Compos, Uniplate, Multiplate, and Traverse-with-class are independent on a particular type representation, their necessary class instances may be either written manually or derived using Template Haskell or using a Generic Deriving mechanism, or using the SYB Typeable or even Data class.

5.7   Generic Libraries and Extensions in ML

Generics for the Working ML'er [17] is a library for SML implementing Generics for the Mass [8] which is a variation of LIGD that uses a type-class instead of a GADT for the type representation. In the SML implementation, a module is used instead.

```
module type Rep = sig
    type α ty
    val int : int ty
    val list : α ty → α list ty
    ...
end
```

Generic functions are all modules of the same signature, whose functions correspond to the constructors of the GADT, and in the context of the library, they correspond to the type representation constructors. For instance, let us write a generic show function:

```
module Show : Rep = struct
    type α ty = α → string
    let int = string_of_int
    let list = fun show_x xs → "[" ^ String.concat "; " (List.map show_x xs) ^ "]"
    ...
end
```

There is an inherent problem with this approach: the type representations are not unique as they must be instanciated with the module of the generic function that is called.

Deriving [35] is an extension to OCAML implemented using the preprocessor camlp4. Generic functions are defined over the structure of types definitions, using a special syntax. The extension is used to implement a safe deserialization function which supports sharing and cycles, and allows the user to override the default behaviour for specific types.

SYB was implemented in MetaOcaml extended with modular implicits and was shown to perform on par with manually written traversals [36]. The implementation uses an extensible GADT for type witness, like in our library. There is no support for extensible type indexed functions. The spine view

is implicit much like in the Haskell implementation. The Haskell `Typeable` and `Data` type classes are directly translated as a modules, using the correspondance explained in modular implicit [34].

The addition of GADTs to OCAML made it possible to reflect types as a basis for generic programming.

6 Conclusion

We have presented our library for generic programming in OCAML, it is built modularly around three main ingredients: (1) an extensible GADT that reflects the names of types; (2) an implementation of extensible type-indexed functions, suitable to define ad-hoc polymorphic functions; (3) generic views which reflect the structure of types. Views are implemented as type-indexed functions, and new views can be added by the user. The built-in view is automatically derived by a PPX for the types marked with an attribute `reify`. Abstract types are supported by means of a public representation.

On top of the library, we implemented a library for generic traversal that removes a lot of the boilerplate normally associated with the functions on mutually defined recursive types with a large number of constructors. The library was seamlessly adapted from a Haskell library.

Finally we presented a complex generic function that fixed some of the shortcomings of the built-in deserialization: not only is our function type-safe, it also respects abstract types by serializing their public representation.

References

[1] Patrick Bahr & Tom Hvitved (2011): *Compositional Data Types*. In: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/2036918.2036930. Available at `http://doi.acm.org/10.1145/2036918.2036930`.

[2] Björn Bringert & Aarne Ranta (2006): *A Pattern for Almost Compositional Functions*. SIGPLAN Not. 41(9), pp. 216–226, doi:10.1145/1160074.1159834. Available at `http://doi.acm.org/10.1145/1160074.1159834`.

[3] Manuel M. T. Chakravarty, Gabriel C. Ditu & Roman Leshchinskiy (2009): *Instant Generics: Fast and Easy*.

[4] James Cheney & Ralf Hinze (2002): *A Lightweight Implementation of Generics and Dynamics*. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, ACM, New York, NY, USA, pp. 90–104, doi:10.1145/581690.581698. Available at `http://doi.acm.org/10.1145/581690.581698`.

[5] Roman Cheplyaka (2013): *Generalizing generic fold*. `https://ro-che.info/articles/2013-03-11-generalizing-gfoldl` `http://hackage.haskell.org/package/traverse-with-class`.

[6] Jeremy Gibbons & Bruno César dos Santos Oliveira (2009): *The Essence of the Iterator Pattern*. Journal of Functional Programming 19(34), pp. 377–402, doi:10.1017/S0956796809007291. Available at `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf`. Revised version of [6].

[7] Grégoire Henry, Michel Mauny, Emmanuel Chailloux & Pascal Manoury (2012): *Typing Unmarshalling Without Marshalling Types*. In: *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, ACM, New York, NY, USA, pp. 287–298, doi:10.1145/2364527.2364569. Available at `http://doi.acm.org/10.1145/2364527.2364569`.

[8] Ralf Hinze (2004): *Generics for the masses*. In Kathleen Fisher, editor: *Proceedings of the ninth ACM SIGPLAN international conference on Functional Programming (ICFP '04)*, New York, NY, USA, pp. 236–243, doi:10.1145/1016850.1016882.

 [9] Ralf Hinze & Johan Jeuring (2003): *Generic Haskell: Applications*. In Roland Backhouse & Jeremy Gibbons, editors: *Generic Programming: Advanced Lectures*, 2793, Springer Berlin / Heidelberg, pp. 57–96, doi:10.1007/978-3-540-45191-4_2.

[10] Ralf Hinze & Johan Jeuring (2003): *Generic Haskell: Practice and Theory*. In: *In Generic Programming, Advanced Lectures, volume 2793 of LNCS*, Springer-Verlag, pp. 1–56.

[11] Ralf Hinze, Johan Jeuring & Andres Löh (2007): *Comparing Approaches to Generic Programming in Haskell*. In: *Proceedings of the 2006 International Conference on Datatype-generic Programming*, SSDGP'06, Springer-Verlag, Berlin, Heidelberg, pp. 72–149. Available at `http://dl.acm.org/citation.cfm?id=1782894.1782896`.

[12] Ralf Hinze & Andres Löh (2006): *"Scrap Your Boilerplate" Revolutions*. In: *Proceedings of the 8th International Conference on Mathematics of Program Construction*, MPC'06, Springer-Verlag, Berlin, Heidelberg, pp. 180–208, doi:10.1007/11783596_13. Available at `http://dx.doi.org/10.1007/11783596_13`.

[13] Ralf Hinze & Andres Löh (2009): *Generic Programming in 3D*. *Science of Computer Programming* 74(8), pp. 590–628, doi:10.1016/j.scico.2007.10.006.

[14] Ralf Hinze, Andres Löh & Bruno C.d.S. Oliveira (2006): *"Scrap Your Boilerplate" Reloaded*. In Masami Hagiya & Philip Wadler, editors: *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, 3945, Springer Berlin / Heidelberg, pp. 13–29, doi:10.1007/11737414_-3.

[15] Patrik Jansson & Johan Jeuring (1997): *PolyP&Mdash;a Polytypic Programming Language Extension*. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, ACM, New York, NY, USA, pp. 470–482, doi:10.1145/263699.263763. Available at `http://doi.acm.org/10.1145/263699.263763`.

[16] Patrik Jansson, Johan Jeuring & Lambert Meertens (1999): *Generic programming: An introduction*. In: *3rd International Summer School on Advanced Functional Programming*, Springer-Verlag, pp. 28–115.

[17] Vesa A.J. Karvonen (2007): *Generics for the Working ML'Er*. In: *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, ACM, New York, NY, USA, pp. 71–82, doi:10.1145/1292535.1292547. Available at `http://doi.acm.org/10.1145/1292535.1292547`.

[18] Ralf Lämmel & Simon Peyton Jones (2005): *Scrap Your Boilerplate with Class: Extensible Generic Functions*. *SIGPLAN Not.* 40(9), pp. 204–215, doi:10.1145/1090189.1086391. Available at `http://doi.acm.org/10.1145/1090189.1086391`.

[19] Ralf Lämmel & Simon Peyton Jones (2003): *Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming*. In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, ACM, New York, NY, USA, pp. 26–37, doi:10.1145/604174.604179. Available at `http://doi.acm.org/10.1145/604174.604179`.

[20] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy & Jérôme Vouillon (2016): *The OCaml system, release 4.04, Documentation and user's manual, section 7.19: Extension nodes*. INRIA. Available at `http://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec248`.

[21] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring & Andres Löh (2010): *A Generic Deriving Mechanism for Haskell*. In: *Proceedings of the Third ACM Haskell Symposium on Haskell*, Haskell '10, ACM, New York, NY, USA, pp. 37–48, doi:10.1145/1863523.1863529. Available at `http://doi.acm.org/10.1145/1863523.1863529`.

[22] José Pedro Magalhães & Johan Jeuring (2011): *Generic Programming for Indexed Datatypes*. In: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*, WGP '11, ACM, New York, NY, USA, pp. 37–46, doi:10.1145/2036918.2036924. Available at `http://doi.acm.org/10.1145/2036918.2036924`.

[23] Conor Mcbride & Ross Paterson (2008): *Applicative Programming with Effects*. *J. Funct. Program.* 18(1), pp. 1–13, doi:10.1017/S0956796807006326. Available at `http://dx.doi.org/10.1017/S0956796807006326`.

[24] Lambert Meertens (1992): *Paramorphisms*. *Formal Aspects of Computing* 4(5), pp. 413–424, doi:10.1007/BF01211391. Available at `http://dx.doi.org/10.1007/BF01211391`.

[25] Neil Mitchell & Colin Runciman (2007): *Uniform Boilerplate and List Processing*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop*, Haskell '07, ACM, New York, NY, USA, pp. 49–60, doi:10.1145/1291201.1291208. Available at `http://doi.acm.org/10.1145/1291201.1291208`.

[26] Ulf Norell & Patrik Jansson (2004): *Polytypic Programming in Haskell*. In: *Proceedings of the 15th International Conference on Implementation of Functional Languages*, IFL'03, Springer-Verlag, Berlin, Heidelberg, pp. 168–184, doi:10.1007/978-3-540-27861-0_11. Available at `http://dx.doi.org/10.1007/978-3-540-27861-0_11`.

[27] Russell O'Connor (2011): *Functor is to Lens as Applicative is to Biplate: Introducing Multiplate*. *CoRR* abs/1103.2841. Available at `http://arxiv.org/abs/1103.2841`.

[28] Bruno C. d. S. Oliveira & Jeremy Gibbons (2005): *TypeCase: A Design Pattern for Type-indexed Functions*. In: *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*, Haskell '05, ACM, New York, NY, USA, pp. 98–109, doi:10.1145/1088348.1088358. Available at `http://doi.acm.org/10.1145/1088348.1088358`.

[29] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyov & Bruno C. d. S. Oliveira (2008): *Comparing Libraries for Generic Programming in Haskell*. *SIGPLAN Not.* 44(2), pp. 111–122, doi:10.1145/1543134.1411301. Available at `http://doi.acm.org/10.1145/1543134.1411301`.

[30] Wouter Swierstra (2008): *Data types à la carte*. *J. Funct. Program.* 18(4), pp. 423–436. Available at `http://dx.doi.org/10.1017/S0956796808006758`.

[31] Philip Wadler (1992): *The Essence of Functional Programming*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, ACM, New York, NY, USA, pp. 1–14, doi:10.1145/143165.143169. Available at `http://doi.acm.org/10.1145/143165.143169`.

[32] Philip Wadler (1998): *The expression problem*. Posted on the Java Genericity mailing list. Available at `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`.

[33] Stephanie Weirich (2006): *RepLib: A Library for Derivable Type Classes*. In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/1159842.1159844. Available at `http://doi.acm.org/10.1145/1159842.1159844`.

[34] Leo White, Frédéric Bour & Jeremy Yallop (2014): *Modular implicits*. In: *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, pp. 22–63, doi:10.4204/EPTCS.198.2. Available at `http://dx.doi.org/10.4204/EPTCS.198.2`.

[35] Jeremy Yallop (2007): *Practical Generic Programming in OCaml*. In: *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, ACM, New York, NY, USA, pp. 83–94, doi:10.1145/1292535.1292548. Available at `http://doi.acm.org/10.1145/1292535.1292548`.

[36] Jeremy Yallop (2016): *Staging Generic Programming*. In: *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, ACM, New York, NY, USA, pp. 85–96, doi:10.1145/2847538.2847546. Available at `http://doi.acm.org/10.1145/2847538.2847546`.

[37] Jeremy Yallop & Leo White (2014): *Functional and Logic Programming: 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, chapter Lightweight Higher-Kinded Polymorphism, pp. 119–135. Springer International Publishing, Cham, doi:10.1007/978-3-319-07151-0_8. Available at `http://dx.doi.org/10.1007/978-3-319-07151-0_8`.