Practical Generic Programming in OCaml

Jeremy Yallop
LFCS, University of Edinburgh
jeremy.yallop@ed.ac.uk

Abstract

We describe `deriving`, a system of extensible generic functions for OCaml implemented as a preprocessor and supporting library. We argue that generating code from type-definitions has significant advantages over a combinator approach, taking serialisation as an example application: our generate-your-boilerplate design results in a system that is easy to use, has comprehensive coverage of types and handles cyclic values without imposing a burden on the user. Users can extend generic functions with specialised implementations at particular types; we show how this can lead to dramatically improved performance in the serialisation task without the user writing a single line of serialisation code.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords OCaml, Deriving, Generic Programming

1. Introduction: generic functions

Generic programming offers a form of polymorphism which lies between the extremes of *parametric* and *ad-hoc*. The behaviour of a parametrically polymorphic function, such as *length*, is the same at every type. The behaviour of an ad-hoc polymorphic function, such as *sum*, is different at each type at which it is defined. The behaviour of a generic function, such as *eq*, also varies at each type, but in a way that is related to the type structure.

Put another way, parametric polymorphic functions are parameterised by *type*, ad-hoc polymorphic functions by *instance*, and generic functions by the *shape of types*. For example, *eq* is defined in the same way for every (immutable) record type: two values of a record type are equal if the values of corresponding fields are equal. The definition of *eq* varies with the shape of its argument type: in the case of records it is a function of the set of fields in the type.

The tool described in this paper, `deriving`, adds generic functions to OCaml by means of a Camlp4 syntax extension which operates on the syntax of types to generate functions. A user can request `deriving` to generate a function by adding the `deriving` keyword to the end of a type definition, followed by a list of the functions which should be generated. For example, we can define a pair of datatypes with an *eq* operation as follows:

```
type α tree =
    Empty
  | Node of α × (α forest)
and α forest =
    Nil
  | Cons of (α tree) × (α forest)
  deriving (Eq)
```

We can then compare values involving trees and forests using `deriving`'s `<>`-notation to apply `eq` at the appropriate type.

```
Eq.eq<bool tree list> [Node (true, Nil)] [Empty]
⟹ false
```

Finally, `deriving` also extends the syntax of signatures. A module that exports a type for which a function such as `Eq` has been derived can also export an interface to the generated function by adding `deriving (Eq)` after the type definition in the signature. The derived function may then be invoked in the usual way.

```
module type Trees = sig
  type α tree
       deriving (Eq)
  type α forest
       deriving (Eq)
  ...
end
 ...
Eq.eq<Trees.tree> t
```

The design of `deriving` is outlined in Sections 3-6.

1.1 Camlp4

The implementation of `deriving` relies on Camlp4, the Caml Pre-Processor-Pretty-Printer, to provide an interface to the syntax trees of Objective Caml programs. Camlp4 includes an OCaml parser which can be extended with new syntactic constructs or syntax-tree transformations. Camlp4 extensions are written in OCaml itself, augmented with a system of quotations and antiquotations which make it easy to switch between the object language and the metalanguage, while using OCaml syntax for both. Camlp4 may be viewed as a macro system with a level of abstraction somewhere between C macros [33] and Scheme macros [19]; unlike C's `#define` construct, Camlp4 operates on abstract syntax, not on token sequences; unlike macros written with Scheme's `syntax-rules`, Camlp4 extensions may capture variables (advertently or otherwise). Camlp4 is thus somewhat similar to Common Lisp's `defmacro` [34], but with complexity commensurate with that of OCaml's syntax.

Other Camlp4 extensions available include Carette and Kiselyov's *pa_monad* [2], which adds Haskell-like **do**-syntax to OCaml, Jambon's *pa_tryfinally* [16], which adds Java's **try** ... **finally**, and de Rauglaudre's *IoXML*, which generates XML parsing and printing functions for types.

The current version of `deriving` is available from

`http://code.google.com/p/deriving/`.

2. Introduction: serialisation

The development of `deriving` was motivated by the search for a robust solution to the *serialisation* problem. The need for serialisation — converting values to and from some external format — typically arises in a distributed setting, where values must be transferred between nodes.

We impose several requirements on acceptable serialisers:

1. Serialisation must be *efficient*. Space efficiency is of primary importance to keep network traffic low, and to avoid imposing too great a burden on clients.
2. Serialisation must be *type-safe*. Applying a serialisation function to invalid data should never cause an irrecoverable program crash.
3. Serialisation must be *extensible*. Most of the time an out-of-the-box serialiser will be sufficient, but there is sometimes a need for a user to supply a specialised implementation of serialisation at a particular type.
4. Serialisation must be *low-cost*. A solution which requires the user to write and maintain large amounts of code to perform basic serialisation is not acceptable.

The second portion of this paper (Sections 7-9) describes a serialiser developed for the web programming language Links [5], which is implemented in OCaml. During the execution of a Links program a computation may be suspended and transmitted over the network to a client to be retrieved and resumed at some future time. Transmitting the computation involves capturing its state as a sequence of bytes which can later be turned back into a value. Serialisation is thus a basic requirement of the Links implementation.

We first consider two existing approaches to serialisation: the standard OCaml module `Marshal`, and *pickler combinators*.

2.1 Marshal

The OCaml standard library includes a module, `Marshal`, with functions

```
val to_string : ∀α. α → extern_flags list → string
val from_string : ∀α. string → int → α
```

that allow almost any value to be serialised and reconstructed. While `Marshal` is certainly easy to use, its design is problematic when judged for flexibility and safety. The encoding of values is unspecified and fixed, leaving no way to specialise the encoding at a particular type. The type of the `to_string` function places no restrictions on its input, delaying detection of attempts to serialise unserialisable values until runtime.

```
# Marshal.to_string (lazy 0) [] ;;
Exception: Invalid_argument
      "output_value: abstract value (outside heap)".
```

Most seriously of all, it is easy to use Marshal to write programs that crash by interpreting a reconstructed value at the "wrong" type.

```
# (Marshal.from_string
     (Marshal.to_string 0 []) 0 : float) ;;
Segmentation fault
```

These flaws make Marshal most suitable for use as a basis for a safe implementation that includes some independent means of verifying the integrity and suitability of marshalled data.

2.2 Pickler combinators

Compositionality is perhaps the greatest benefit of functional programming [14], so it is natural to seek a combinator approach to the serialisation problem. There is a natural way to structure a combinator library for serialisation, as evinced by the similarity of proposed designs by Kennedy for Haskell [20] and Elsman for SML [7]. A parameterised type of serialisers

```
type α pu
```

together with serialisers for primitive types

```
val unit : unit pu
val bool : bool pu
```

and combinators for parameterised types that take serialisers to serialisers

```
val list : ∀α. α pu → α list pu
val pair : ∀α,β. α pu×β pu → (α × β) pu
```

and a function which takes conversions between types to conversions between serialisers

```
val wrap : ∀α,β. (α → β) → (β → α) → α pu → β pu
```

are sufficient to encode a wide variety of datatypes. To serialise user-defined algebraic types we need some way of discriminating between the constructors. The typical solution is to provide a combinator (called `data` in [7], `alt` in [20]) whose argument maps constructors to integers:

```
val alt : ∀α.(α → int) → α pu list → α pu
```

We might then write a `pu` for OCaml's `option` type as follows:

```
let option : ∀α. pu → α option pu =
  fun a ->
    alt
      (function None -> 0 | (Some _) -> 1)
      [wrap (fun () -> None)  (fun None -> ()) unit;
       wrap (fun v -> Some v) (fun (Some v) -> v) a];
```

The combinator approach has none of the problems seen with `Marshal`: the user can choose the encoding at each type, and there is no lack of type-safety. There are, however, serious drawbacks particular to this approach. It is tedious to recapitulate the structure of each user-defined type to obtain a serialiser: this is a prime example of the "boilerplate" which much recent research has sought ways to "scrap" [22]. The requirement for the user to supply a mapping from constructors to integers can lead to errors that are hard to track down. Finally, there are difficulties in handling both cyclic values, which can arise in ML through the use of references, and mutually recursive datatypes. The `refCyc` combinator described in [7] supports cyclic values, but requires the user to supply a "dummy" value to start off the cycle:

```
val refCyc : ∀α. α → α pu → α ref pu
```

Not only is this a rather unpleasant imposition, but it is not apparent whether it can be readily generalised to the OCaml situation, where the language primitive for mutability is not the `ref` type, but records with arbitrary numbers of mutable fields. Karvonen notes [18] that the need for a dummy value can be eliminated by using an additional generic function that constructs witnesses for types.

Can we combine the flexibility and safety of the combinator approach with the ease of use of `Marshal`? The `deriving` system aims to do precisely that by building on the combinator approach with a tool that derives functions for serialisation and other generic functionality automatically from type declarations.

The rest of this paper is divided into two parts. Part I (Sections 3-6) introduces generic functions as provided by `deriving`. Section 3 gives an example of the use of `deriving`. The translation strategy from generic functions into standard OCaml is outlined in Section 4, followed by the specification of a generic function for

```
(*pp deriving *)
type point = { mutable x : int; mutable y : int }
    deriving (Eq)
type α seq =
    Nil
  | Cons of α * (α seq)
  deriving (Eq)
let eq =
  Eq.eq<point seq>
    (Cons ({ x = 10; y = 20 }, Nil))
    (Cons ({ x = 10; y = 20 }, Nil))
```

Figure 1. Using generic equality

equality in Section 5 and a brief summary of related work in Section 6. Part II (Sections 7-9) gives the design of a generic serialiser. Section 7 describes the use of `deriving` for value-oriented serialisation. Section 8 explores the design and spacewise performance of a generic structure-sharing serialiser. Section 9 lists work related to generic serialisation. Section 10 concludes.

Part I: Generic Functions

3. Generic functions: an example

We now introduce generic functions in `deriving` using generic equality as an illustrative example.

The simple program in Figure 1 illustrates the use of `deriving`. The clause `deriving (Eq)` that appears after type definitions of the types `point` and `seq` indicates that the `deriving` preprocessor should generate an equality predicate for each type. The term `Eq.eq<point seq>` is a use of the generic equality function at the type `point seq`: that is, it denotes a function of type

```
point seq → point seq → bool
```

Since OCaml already boasts two polymorphic equality predicates — = and ==, which test for structural and physical equality respectively — generic equality may seem like a relatively useless addition. However, there are situations in which neither of the built-in functions is suitable. The structure-sharing serialiser described in Section 8 requires a predicate that equates structurally-equal immutable values (unlike ==) and distinguishes physically equal mutable values (unlike =). Without the first it will share too little; without the second it would share too much, since conflating distinct mutable values changes the semantics of programs. (In OCaml sharing of immutable values is detectable as well, using ==, but the precise behaviour is not specified; we are therefore unconcerned about changing the semantics of programs that depend on its use.) In short, we need an equality predicate corresponding to = in Standard ML.

Generating instances of `Eq` is straightforward: for records with mutable fields equality is defined as physical equality (==); for other values equality is defined inductively, i.e. in terms of the definition of `Eq` on component types. As for most other classes, `deriving` will signal an error on encountering an attempt to generate an instance for `Eq` at a function type.

The commented phrase `(*pp deriving *)` at the top of Figure 1 is a convenient way to indicate to the `OCamlMakefile` build system that the program should be preprocessed with `deriving` before compilation.

4. Generic functions: compilation

Figure 2 shows the result of processing the program in Figure 1 with `deriving`. (We have taken the liberty of increasing legibility a little by shortening generated names and removing trivial bindings, but it is operationally identical to the actual code generated.)

```
type point = { mutable x : int; mutable y : int }
module rec Eq_point
  : Eq.Eq with type a = point =
Eq.Defaults(struct
              type a = point
              let eq = Pervasives.(==)
            end)
type α seq =
    Nil
  | Cons of α * (α seq)
module Eqs (A : Eq.Eq) =
struct
  module rec Eq_seq
      : Eq.Eq with type a = A.a seq =
  Eq.Defaults
    (struct
       type a = A.a seq
       let eq l r = match l, r with
       | Nil, Nil -> true
       | Cons (x1, x2), Cons (y1, y2) ->
           (let module M =
              struct
                type a = A.a * A.a seq
                let eq (x2, x1) (y2, y1) =
                  A.eq x2 y2 && Eq_seq.eq x1 y1 && true
              end
            in M.eq) (x1, x2) (y1, y2)
       | _ -> false
     end)
end
module Eq_seq (A : Eq.Eq) =
struct
    module P = Eqs(A)
    include P.Eq_seq
end
let eq =
   (let module Eq =
      struct
        module rec Eq_inline
            : Eq.Eq with type a = point seq =
          Eq.Defaults(Eq_seq(Eq_point))
        include Eq_inline
      end
    in Eq.eq)
     (Cons ( x = 10; y = 20; , Nil))
     (Cons ( x = 10; y = 20; , Nil))
```

Figure 2. Figure 1, processed with `deriving`

In place of each `deriving (Eq)` clause, `deriving` inserts a module definition: `Eq_point` following the definition of `point`, and `Eq_seq` following the definition of `seq`. Each module implements a signature with two components:

```
type a
val eq : a → a → bool
```

The definition of `Eq_point.eq` is simply `Pervasives.(==)`, the physical equality predicate, since `point` values have mutable fields. The definition of `eq` for `seq` is dependent on the functor argument: another structure implementing equality. This matches our intuition that to compare values of `t seq` we must first know how to compare values of type `t`.

The call to `Eq.eq<point seq>` is expanded into a local module definition: yet another structure implementing the `Eq` signature. The definition is constructed by application of the functor `Eq_seq` to the structure `Eq_point`, mirroring the application of the type constructor `seq` to the type `point`.

Having given a particular example, we move on to describe the general principles behind the translation.

(a) Classes

```
class Eq a where              module type Eq = sig
  eq :: a → a → Bool            type a
                                  val eq : a → a → bool
                                end
```

(b) Instances

```
instance Eq Int where         module EqInt
  eq = eqInt                    : Eq with type a = int =
                                struct
                                  type a = int
                                  let eq = eqInt
                                end
```

(c) Instances for parameterised types

```
instance (Eq a) => Eq [a]     module EqList (A : Eq)
  where eq l r =                : Eq with type a = A.a list =
   and (zipWith eq l r)         struct
                                  type a = A.a list
                                  let eq = and (zipWith A.eq l r)
                                end
```

(d) Default method implementations

```
class Eq a where               module EqDefaults
  eq, neq :: a → a → Bool        (A : sig
  neq l r = not (eq l r)                type a
                                        val eq : a → a → bool
                                      end) =
                                 struct
                                   include A
                                   let neq l r = not (eq l r)
                                 end
```

(e) Superclasses

```
class (Show a,Eq a) => Num a   module type Num = sig
where (+), (-) :: a → a → a      type a
                                   module Show
                                       : Show with type a = a
                                   module Eq
                                       : Eq with type a = a
                                   val (+) : a → a → a
                                   val (-) : a → a → a
                                 end
```

Figure 3. Correspondence between type classes and modules

4.1 Modules and type classes

The basic design of `deriving` is inspired by the construct of the same name in Haskell [25]. Haskell's *deriving* is tied to type classes, which have no direct parallel in OCaml. However, there is a well-known correspondence [6, 21, 30] between ML's modules and Haskell's type classes, which we use to guide our design. The relevant facets of the correspondence are illustrated in Figure 3:

(a) A class in Haskell maps to a signature in OCaml with a type component to specify the overloaded type and a set of value components to specify the methods.

(b) An instance of the class maps to a structure implementing the signature, with sharing constraints to expose the representation of the overloaded type.

(c) An instance for a parameterised type maps to a functor which takes for each type parameter a structure satisfying the class signature and yields another such structure.

(d) The set of default methods for each class maps to a functor which takes a structure containing the user-defined methods and returns a structure implementing the full signature.

(e) A superclass maps to a signature which contains the superclass module type as an element.

We will use the language of type classes to describe our design in the remainder of this paper, referring to the signatures used by `deriving` as "classes", to generated structures as "instances", and so on.

d	::=	**type** $(\alpha_1,\ldots\alpha_n)$ $\mathrm{t}_1 = r_1$	declarations
		$\ldots$	
		and $(\alpha_1,\ldots,\alpha_n)$ $\mathrm{t}_m = r_m$	
r	::=		representations
		$\{ f_1 \langle ; \ldots ; f_n \rangle \}$	record
		$c_1 \mid \ldots \mid c_n$	sum
		$[\, ts_1 \mid \ldots \mid ts_n \,]$	polymorphic variant
		e	type expression
e	::=		type expressions
		α	type variable
		$e_1 \times \ldots \times e_n$	tuple
		e'	
e'		$(e_1,\ldots, e_n)$ c	constructor applications
ts	::=		tag specs
		‘T ⟨ **of** e ⟩	tag
		e'	constructor application
f	::=	⟨**mutable**⟩ l : e	field specs
cs	::=	C ⟨ **of** $e_1 \ldots e_n$ ⟩	constructor specs

Figure 4. Type language of OCaml (restricted)

4.2 Summary of OCaml's type system

Figure 4 shows a subset of the OCaml grammar for type definitions that is accepted by `deriving`. Some of the constructs not listed here are accepted by `deriving` in certain circumstances: for instance, private types (which expose constructors that can appear in patterns, but not expressions) are permitted for classes such as `Eq` which do not construct values.

In the full OCaml language, polymorphic variant type expressions can occur anywhere within type expressions, and admit an optional binder `as 'a` to enable recursion. Neither of these features brings any additional expressive power, so `deriving` uses a normalised form internally which eliminates both by introducing additional top-level definitions, leading to a simpler implementation and a reduction in generated code size.

There is one additional restriction not captured by the grammar: we require types to be regular for reasons described in the following section.

4.3 Recursive functors

Here we encounter our first hurdle. Under the correspondence described in Section 4.1, the natural encoding of a parameterised mutually-recursive type

```
type α tree =
    Empty
  | Node of α × (α forest)
and α forest =
    Nil
  | Cons of (α tree) × (α forest)
```

is a group of mutually-recursive functors

```
module rec Eq_tree(A : Eq)
  : Eq.Eq with type a = A.a tree =
struct
  ...
end
and Eq_forest(A : Eq)
  : Eq.Eq with type a = A.a forest =
struct
  ...
end
```

```
module IntSet : sig
  type t
  val empty : t deriving (Eq)
  val add : int → t → t
  val mem : int → t → bool
end =
struct
  type t = int list
  let empty = []
  let add i l = i :: l
  let mem = List.mem
  module Eq_t : Eq with type a = t =
  struct
    type a = t
    let eq l r = Eq.eq<int list>
                   (sort l) (sort r)
  end
end
```

Figure 5. Integer sets using integer lists

OCaml does not permit recursive functors, so an alternative encoding is needed. While recursive functors are prohibited, there is no problem with functors whose bodies contain recursive modules as components; we can therefore encode the recursive group as a single functor whose body is a group of recursive modules, then project out the modules separately.

```
module Eq_tree_forest(A : Eq) =
struct
  module rec Eq_tree
    : Eq.Eq with type a = A.a tree =
  struct
    ...
  end
  and Eq_forest
    : Eq.Eq with type a = A.a forest =
  struct
    ...
  end
end
module Eq_tree(A : Eq)
   = Eq_tree_forest(A).Eq_tree
module Eq_forest(A : Eq)
   = Eq_tree_forest(A).Eq_forest
```

This solution requires that all types in the group have the same parameters and that every occurrence of the type constructors on the right hand side of the definition is applied to exactly those parameters in the same order. We therefore adopt this restriction for types to be processed by `deriving`.

4.4 Specialisation of generated functions

One of our complaints about OCaml's `Marshal` was the lack of a facility for *specialisation*: providing behaviour at a particular type that differs from the default. This is accomplished in `deriving` by writing a module definition with a particular name and type in place of adding the `deriving` annotation to the type. For example, consider the implementation of integer sets given in Figure 5, in which unordered lists of integers are used as the representation type. Using `deriving` we can define equality on IntSet to consider two sets equal if the result of sorting their representing lists is equal. The builtin equality operator regards such sets as unequal, exposing the fact that there are various representations for values that the abstraction considers equivalent:

```
let onetwo = IntSet.add 1 (IntSet.add 2 (IntSet.empty))
let twoone = IntSet.add 2 (IntSet.add 1 (IntSet.empty))
Eq.eq<IntSet.t> onetwo twoone  ⟹ true
onetwo = twoone ⟹ false
```

(Precisely the same problem arises with the set implementation provided in the OCaml standard library: sets considered equal under the set equality predicate do not always satisfy =.) Using `deriving`'s `Eq`, our abstraction-respecting definition of equality will be used wherever values of `IntSet.t` are compared with `Eq`; in particular, it will be used where they occur in some larger data structure:

```
Eq.eq<IntSet.t list> [onetwo; twoone] [twoone; onetwo]
⟹ true
```

5. Generic functions: implementation

Figures 6 and 7 give a specification for `Eq`. The meta-functions *eq*, *eq-rep* and *eq-def* operate on the syntax of types to construct terms. The notation is reminiscent of that used in the actual implementation, but more concise: in the specification we use ellipses where the implementation uses folds, for example. The *eq* function takes the representation of a type `t` to a term of type `t → t → bool`. The *eq-def* function constructs a module `Eq_t` satisfying the `Eq` signature (Figure 3) for each type `t` in a set of mutually-recursive type definitions. The `<< >>` notation introduces a quoted OCaml term; there is also an anti-quotation `$ $` for inserting computed terms into quoted terms.

Figure 7 defines the translation of angle bracket syntax (`<>`) for the class `Eq`.

6. Generic functions: related work

There is a significant body of literature on generic programming. A common approach [3, 13, 31] is to reflect types as values. Generic functions can then be written by case analysis on the structure of these values. An alternative view [10, 17] treats datatypes as fixed points of regular functors, exposing the recursive structure in a way that allows a variety of traversals to be expressed generically. A third approach [22], particularly suited to generic transformations, uses a type-safe cast operator to locate nodes of particular types within larger values. Hinze et al give a useful comparison of designs based on these and other approaches [11].

The preprocessor-based approach to deriving instances of generic functions is used in the Haskell tools DrIfT [32] and Derive [24]; several Camlp4 extensions involve the generation of functions from type declarations, including Martin Jambon's *json-static* [15], Martin Sandin's *Tywith* [27], and Daniel de Rauglaudre's IoXML [26].

Part II: Generic Serialisation

7. Serialisation: a value-oriented serialiser

We now return to the topic of serialisation introduced in Section 2. We begin, in this section, with a description of a simple serialiser, `Dump`. The following sections present the design of `Pickle`, a more realistic serialiser based on `Dump`, which supports features such as compression, structure-sharing and serialisation of cyclic values.

At the heart of the problems with the combinator approach to serialisation is the fact that ML-style types are constructed from a variadic algebra. We can construct tuples of arbitrary size, but the type of n-tuples is unrelated to the type of $n + 1$ tuples; similar properties hold for records and variants. Since OCaml provides no means to parameterise functions by the size of a tuple, any combinator library that processes tuples must be either limited

1. Records with mutable fields

```
eq << { ... ; mutable l_j : $_$ ; ... } >> =
<< Pervasives.(==) >>
```

2. Records with no mutable fields

```
eq << { $l_1$ : $t_1$ ; ... ; $l_n$ : $t_n$ } >> =
<< fun { $l_1$ = x_1 ; ... ; $l_n$ = x_n }
       { $l_1$ = y_1 ; ... ; $l_n$ = y_n } ->
    Eq.eq<t_1> x_1 y_1 && ... && Eq.eq<t_n> x_n y_n
      && true >>
```

3. Sums

```
eq <<$C_1$ ⟨of $t^1_1,...,t^1_m$⟩ |...| $C_n$ ⟨of $t^n_1,...,t^n_m$⟩>> =
<< fun l r -> match l, r with
   | $C_1$ ⟨(x_1,...,x_m)⟩, $C_1$ ⟨(y_1,...,,y_m)⟩ ->
          ⟨Eq.eq<$t^1_1×...×t^1_m$> (x_1,...,x_m)(y_1,...,y_m) &&⟩
          true
     ...
   | $C_n$ ⟨(x_1,...,x_m)⟩, $C_n$ ⟨(y_1,...,y_m)⟩ ->
          ⟨Eq.eq<$t^1_n×...×t^1_n$> (x_1,...,x_m) (y_1,...,y_m) &&⟩
          true
   | _ -> false >>
```

4. Polymorphic variants

```
eq << [ $`T_1$ ⟨of $t_1$⟩ |...| $`T_n$ ⟨of $t_n$⟩
      | ($t^1_1, ..., t^1_m$) $c_1$ | ...
      | ($t^n_1, ..., t^n_m$) $c_n$ ] >> =
<< fun l r -> match l, r with
   | $`T_1$ ⟨x⟩, $`T_1$ ⟨y⟩ -> Eq.eq<$t_1$> x y
     ...
   | $`T_n$ ⟨x⟩, $`T_n$ ⟨y⟩ -> Eq.eq<$t_n$> x y
   | (#$c_1$ as x), (#$c_1$ as y) ->
         Eq.eq<($t^1_1, ..., t^1_m$) $c_1$> x y
     ...
   | (#$c_n$ as x), (#$c_n$ as y) ->
         Eq.eq<($t^n_1, ..., t^n_m$) $c_n$> x y
   | _ -> false >>
```

5. Tuples

```
eq << $t_1$ * ... * $t_n$ >> =
<< fun (x_1,...,x_n) (y_1,...,y_n) ->
   Eq.eq<$t_1$> x_1 y_1 &&...&& Eq.eq<$t_n$> x_n y_n && true >>
```

6. Type constructors

```
eq << ($t_1, ..., t_n$) c >> =
<< let module E = Eq_$c$($eq t_1$) ... ($eq t_n$) in
      E.eq >>
```

7. Type definitions

```
eq-def << type ($α_1,...α_n$) $t_1$ = $r_1$
              ...
          and ($α_1,...α_n$) $t_1$ = $r_m$ >> =
<<
module Eqs (A_1 : Eq) ... (A_n : Eq) =
struct
  module rec Eq_r_1 : Eq with type a = $r_1$[A_i/α_i]
   = $eq-rep r_1[A_i/'a_i]$
  ...
  and Eq_r_n : Eq with type a = $r_n$[A_i/α_i]
   = $eq-rep r_n[A_i/α_i]$
end
module Eq_r_1(A_1 : Eq)...(A_n : Eq)
    = Eqs(A_1)...(A_n).Eq_r_1
module Eq_r_m(A_1 : Eq)...(A_n : Eq)
    = Eqs(A_1)...(A_n).Eq_r_m >>
eq-rep t =
<<  Eq.Defaults(struct
      type a = $t$
      let eq = $eq t$
    end
>>
```

Figure 6. Specification of Eq

```
Eq.eq<t> = let module Eq =
             struct
               type t' = $t$ deriving (Eq)
               include Eq_t'
             end in Eq.eq
```

Figure 7. Translation of <> notation (at method Eq.eq)

to accepting a finite subset of tuple types or dependent on user-supplied functions to convert between isomorphic types. The need for the user to supply a function to map constructors of a sum type to integers seen in Section 2.2 has the same problem at root. If n-ary sums were defined in terms of binary sums there would be no need for such functions.

The Dump class provides value-oriented serialisation for OCaml values. Serialising a value involves serialising its subvalues in turn; at primitive types serialisation operates by ad-hoc conversion to a sequence of bytes. There are similarities to the combinator approach in the sense that a serialiser at a compound type is defined in terms of serialisers at its component types. However, by using an external meta-language (i.e. the defining language of the preprocessor) rather than a set of combinators we escape the arity tarpit and achieve full generality.

For reasons of efficiency the functions generated by `deriving` for Dump instances operate on extensible buffers (for the output) and character streams (for the input). The `Defaults` functor applied to each instance produces more convenient operations `to_string` and `from_string`, which produce and consume strings.

```
type α bush = Leaf of α
            | Fork of (α bush) × (α bush)
                deriving (Dump)
Dump.to_string<char bush>
  (Fork (Fork (Leaf 'a', Leaf 'b'), Leaf 'c'))
⟹ "\001\001\000a\000b\000c"
Dump.from_string<char bush> "\001\001\000a\000b\000c"
⟹ Fork (Fork (Leaf 'a', Leaf 'b'), Leaf 'c')
```

While the coverage of Dump is superior to that of the combinator approach, the general design is somewhat naive. There is no attempt to increase or even preserve sharing between values: a subvalue that appears twice in the input tree has a serialised representation that appears twice in the output. However, primitive values are serialised fairly efficiently: the size of the representation of integers increases with their value; small integers (by far the most common, since we use sequentially numbered integers to serialise constructors) use only a single byte.

7.1 Specialisation

As noted in 4.4, `deriving` operations can be customised at particular types. We make use of this facility in the Links implementation to shrink the representation of continuations sent between server and client, essentially serialising a code pointer rather than a representation of code. Deserialisation then involves resolving the pointer to the full expression. Only the Dump instance at the expression type need be customised; when instances for types that contain expressions can be derived, the custom instance will be used automatically. Further examples of specialisation in serialisation are given in Section 8.5.

8. Serialisation: a structure-sharing serialiser

The `Pickle` class of `deriving` offers serialisation for cyclic values and structure sharing. The essential idea is to build a graph representing a value which is written out (using Dump) when complete. Representations are reused where possible by starting serialisation

1. Pickle at record types

```
pickle << { $l_1$ : $t_1$; ...; $l_n$ : $t_n$ } >> =
  << fun ({ $l_1$ = $l_1$; ...; $l_n$ = $l_n$ } as obj) ->
       W.allocate obj
        (fun id ->
           Pickle.pickle<$t_1$> v_1 >>= fun v_1 ->
           ...
           Pickle.pickle<$t_1$> v_n >>= fun v_n ->
           W.store_repr id (Repr.make [v_1; ...; v_n])) >>
```

2. Unpickle at record types

```
  unpickle << { $l_1$ : $t_1$; ...; $l_n$ : $t_n$ } >> =
<< let module Mutable =
     struct
       type t = { mutable $l_1$ : $t_1$;
                  ...;
                  mutable $l_n$ : $t_n$ }
     end in
       R.record $n$
         (fun self -> function
           | [ v_1; ...; v_n ] ->
               let mself : Mutable.t = Obj.magic self in
                 Pickle.unpickle<$t_1$> v_1 >>= fun v_1 ->
                 ...
                 Pickle.unpickle<$t_n$> v_n >>= fun v_n ->
                 mself.Mutable.l_1 <- v_1;
                 ...
                 mself.Mutable.l_1 <- v_n;
                 return self
           | _ -> raise UnpicklingError) >>
```

Figure 8. Specification of Pickle at record types

1. Pickle at tags

```
pickle-pcase << `$T$ >> =
<< `$T$ ->
    W.allocate obj
    (fun id ->
     W.store_repr id (Repr.make ~ctor:$hash T$ [])) >>
pickle-pcase << `$T$ of $t$ >> n =
<< `$T$ x -> Pickle.pickle<$t$> p >>= fun id_2 ->
     W.allocate obj
     (fun id ->
      W.store_repr id (Repr.make ~ctor:$hash n$ [id_2])) >>
```

2. Pickle at constructor application tag specifications

```
pickle-pcase << ($t_1, ..., t_n$) $c$ >> =
 << #$c$ as obj ->
      Pickle.pickle<$(t_1 , ..., t_n) c$> obj >>
```

3. Pickle at polymorphic variant types

```
pickle << [ $t_1$ | ... | $t_n$ ] >> =
 << function
      $pickle-pcase t_1$
      ...
    | $pickle-pcase t_n$ >>
```

4. Unpickle at tags

```
unpickle-tag_1 << `$T$ >> =
<< ($hash T$, []) -> return `$T$ >>
unpickle-tag_1 << `$T$ of $t$ >> n =
<< ($hash T$, [obj]) ->
    Pickle.unpickle<$t$> obj >>= fun obj ->
    return (`$T$ obj) >>
```

5. Unpickle at constructor application tag specifications

```
unpickle-tag_2 << $t_1$ | ... | $t_n$ >> =
<< (n,_) -> try ... try (raise UnknownTag)
            with UnknownTag ->
               (Pickle.unpickle<$t_1$> :> a Read.m)
            ...
            with UnknownTag ->
               (Pickle.unpickle<$t_n$> :> a Read.m) >>
```

6. Unpickle at polymorphic variant types

```
unpickle << [`$T_1$ ⟨ of $t_1$ ⟩ | ... |`$T_n$ ⟨ of $t_n$ ⟩
             $t_{n+1}$ | ... | $t_{m+1}$ ] >> =
<< R.sum (function
            $unpickle-tag_1 <<`$T_1$ ⟨ of $t_1$ ⟩>>$
            ...
          | $unpickle-tag_1 <<`$T_n$ ⟨ of $t_n$ ⟩>>$
          | $unpickle-tag_2 <<$t_{n+1}$ | ... | $t_{m+1}$ >>$ >>
```

Figure 9. Specification of Pickle at polymorphic variant types

of each value with a comparison to values of the same type already serialised.

A specification for `Pickle` is given in Figure 8 for record types and in Figure 9 for polymorphic variants. Other cases are similar in the essential details and so omitted. (For the definition of <> see, *mutatis mutandis*, the translation given for `Eq` in Figure 7.) The definition at each type consists mostly of straightforward uses of the *allocate* and *store_repr* functions described in the next section. One difference from the definition of `Dump` deserves comment: rather than representing polymorphic variant tags as sequentially numbered integers we use a hash function which guarantees a consistent global mapping from tag names to numbers. This allows us to safely equate polymorphic variant tags which are compatible, but declared differently (see Section 8.2). Tags are also represented by OCaml as a hash of their names [9]. The OCaml type checker rejects types containing tags, such as `` `squigglier`` and `` `premiums``, whose hash numbers collide; using the same hash function guarantees that we will not be troubled by collisions. The type compatibility problem comes from the need to share structure between values of compatible types, so it did not arise with `Dump`.

8.1 The pickling algorithm

The pickling algorithm requires three pieces of state, which we represent as a record and thread through the computation using a state monad [29]. The use of a monad to organise computation in an impure functional language is unusual, but the benefits are familiar: essential properties such as the order of accesses to state encoded in types rather than in terms; convenient access to computation-scope data without explicit state-passing; compositionality at the level of computations, making it easy to piece together an algorithm from smaller components.

```
type s = {
  nextid : Id.t;
  objmap : Id.t DynMap.t;
  id2rep : Repr.t IdMap.t;
}
```

The *nextid* field is a source of fresh identifiers, *objmap* maps values to identifiers, and *id2rep* maps identifiers to their serialised representations. The work is performed by two functions: *allocate*, which allocates identifiers for values, and *store_repr*, which stores the association between an identifier and its serialised representation. (The *allocate* function depends on auxiliary definitions for equality as described in Sections 3-5 and dynamic typing as described in the following section.)

```
let allocate o f =
  let obj = T.make_dynamic o in
  get >>= fun ({nextid=n;objmap=objs} as s) ->
    match DynMap.find obj objs with
    | Some id -> return id
    | None ->
     let id, n = n, Id.next n in
        put {s with
             objmap = DynMap.add obj id comparator objs;
             nextid = n} >>
        f id >>
        return id
let store_repr id repr =
  get >>= fun state ->
  put state with idmap = IdMap.add id repr state.idmap
```

After converting the input to type `dynamic`, *allocate* searches the map *objmap* to determine whether the value has been previously serialised. If it finds a match it returns the identifier already allocated; if there is no match then a fresh identifier is allocated, the association between identifier and value is recorded and the argument function *f* is called to serialise the value. This strategy of allocating identifiers before serialising subvalues is sufficient for serialisation of cyclic values: the next time a reference to this node is encountered there will be an identifier in the map, so serialisation will not loop.

8.2 Dynamic typing

The *objmap* map used in the algorithm in Section 8.1 is used to store values of arbitrary type. To achieve this, `deriving` includes support for a form of dynamic typing. The *Typeable* class provides an upcast operation to a universal type, `dynamic`, and safe downcast operations from `dynamic` to each instance type.

```
module type Typeable =
sig
  type a
  val mk : a → dynamic
  val cast : dynamic → a option
  val throwing_cast : dynamic → a
end

Typeable.cast<int>
   (List.hd [Typeable.mk<int> 3; Typeable.mk<unit> ()])
⟹ Some 3
Typeable.cast<int>
   (List.hd [Typeable.mk<unit> (); Typeable.mk<int> 3])
⟹ None
```

There are well-known techniques for implementing dynamic typing in pure SML (see p105-106 of [8], for example), but these deal only with generative types. Following [22], we use instead an implementation based on pairs of values and representations of their types together with an unsafe cast that is performed only if type representations match, which allows us to test types for structural equality where appropriate. Our implementation divides types into nominal (or “generative”) types (sums, records), which are represented as a string for the type constructor and a sequence of argument types, and structural types — i.e. polymorphic variants— which are represented as lazy infinite trees. Tuples are treated as nominal, using the arity as the type constructor.

```
module TagMap = Map.Make(Interned)
type t =
    [‘Variant of (delayed option TagMap.t)
    |‘Gen of Interned.t×delayed list ]
    ×int
and delayed = unit → t
```

Nominal types are considered equal if they have the same constructors and equal arguments. Polymorphic variant types are considered equal if they have the same set of labels and if the arguments to corresponding labels are equal. Recursive polymorphic variant types have infinite representations; termination of the equality test is assured by associating an identifier with each node — the `int` component of the type representation — which is used to record which nodes have been visited, i.e. to detect cycles. Since polymorphic variant types are always regular all recursion eventually passes through an already visited node.

Polymorphic variant types which are considered equivalent in the OCaml type system are treated as equivalent by `deriving` regardless of how they are declared, so the following cast will succeed:

```
type α seq = [‘Nil | ‘Cons of α×α seq]
 deriving (Typeable)
type nil = [‘Nil]
  deriving (Typeable)
type intlist = ([nil| ‘Cons of int×α ] as α)
  deriving (Typeable)
Typeable.cast<intlist>
  (Typeable.mk<int seq>
    (‘Cons (1, ‘Cons (2, ‘Cons (3, ‘Nil)))))
⟹ Some (‘Cons (1, ‘Cons (2, ‘Cons (3, ‘Nil))))
```

Similarly, the type representation used by Typeable does not respect module abstraction boundaries, so the following cast will also succeed:

```
module M : sig
   type t
      deriving (Typeable)
   val v : t
end  =
struct
   type t = int
      deriving (Typeable)
   let v  = 0
end
Typeable.cast<int> (Typeable.mk<t> M.v)
```

This is a deliberate design decision: `Typeable`’s raison-d’être is to maximise sharing, which is best achieved by identifying types which can easily be determined to have the same representation. However, casts between nominal types never succeed, even if they are likely to have the same memory layout; the following will fail:

```
type cartesian = { x : float ; y : float }
type polar     = { r : float ; t : float }
Typeable.cast<polar>
    (Typeable.mk<cartesian> { x = 1.0; y = 3.0 })
```

This difference can be accounted for by the fact that a programmer can cause a single value to have different types using module abstraction, but there is no way for a value to receive more than one sum or record type in a program that does not use unsafe operations.

There are two further operations in `Typeable`, for testing whether a dynamic value belongs to a particular type, and for retrieving the type representation.

```
val has_type : dynamic → bool
val type_rep : unit → TypeRep.t
```

Extension of equality on the type representation to a total ordering makes `TypeRep` values suitable for use as keys in finite maps.

```
module TypeRep :
sig
  type t
  val compare : t → t → int
  val eq : t → t → bool
  ...
end
```

To distinguish nominal types at runtime, `deriving` constructs a unique string for each type constructor:

```
type α r = R of α deriving (Typeable)
⟹
module Typeable_r (V_a : Typeable.Typeable)
   : Typeable.Typeable with type a = V_a.a r
 Typeable.Defaults
   (struct
      type a = V_a.a r
      let type_rep =
        TypeRep.mkFresh "t.ml_2_1381210804.870562_r"
           [V_a.type_rep]
    end)
  end
```

The string is constructed from the source file name, the type name and the time. A significant shortcoming in the current implementation is the lack of any guarantee that the same string will be emitted by different runs of the program (in fact, there is a guarantee that this will *not* happen!). Appending the time is intended to compensate for the fact that the file name and type name are not alone sufficient to guarantee global uniqueness; there is only limited information about the context available to `deriving` when the definition is seen. A solution to this problem is given in [1] in the context of a modification to the OCaml compiler, but a solution based on local syntactic transformations remains elusive.

8.3 Storing the map

We now resume the description of the pickling algorithm.

Section 8.1 outlined the construction of the graph *id2rep*. Once this graph has been fully constructed it is serialised using `Dump`, discarding the other components of the state record. The implementation performs two operations to reduce the size of the output:

(i) Stripping constructors from the representation type.

During serialisation, each OCaml value is represented as either an opaque byte string or as a pair of an optional constructor and a sequence of subvalue identifiers.

```
module Repr : sig
  type t = Bytes of string
         | CApp of (int option×Id.t list)
  ...
  val make : ?ctor:int → Id.t list → t
  val of_string : string → t
end
```

To avoid storing extra bytes alongside every value to indicate whether it should be deserialised as a `Byte`, a `CApp` with an `int` tag or an untagged `CApp` the unpickling code splits the map into three separate maps, one for each alternative.

```
t IdMap.t ⟹
  string IdMap.t
× (int×Id.t list) IdMap.t
× (Id.t list) IdMap.t
```

(ii) Removing explicit identifiers.

Finally, before `Dump` is applied, identifiers are removed by encoding the information implicitly using the ordering of serialised values. For example, given the graph

```
{4 => "k"; 3 => "p"}, {1 => (2,[4;3])}, {2 => [1;4]}
```

listing the nodes in sequence gives

```
(4,"k"), (3,"p"), (1,(2,[4;3])), (2,[1;4])
```

and then systematically replacing each identifier with its position in the sequence gives

```
(1,"k"), (2,"p"), (3,(2,[1;2])), (4, [3;1])
```

Since the identifiers associated with each node are now ordered they contribute no information and we are free to dispense with them:

```
Dump.to_string<string list
              ×(int×(int list))
              ×int list>
  (["k","p"], [(2,[1;2])], [[3;1]])
```

There remain many opportunities for shrinking the output value. For example, the smallest unit that can be written in the current implementation is a full byte; we might instead use a single bit to indicate which constructor of a binary sum is present.

8.4 Unpickling

Detecting cycles is straightforward; reconstructing cycles is trickier. Values in OCaml are constructed from subvalues: the subvalues must exist first, but this breaks down if the value is among its own subvalues (either directly or as a more distant descendent). As we saw in Section 2.2, Elsman solves this problem by requiring the user to supply an initial value to start the recursion going. Our solution is to use low-level primitives to allocate an uninitialised value whose address can be used in constructing subvalues. In generated code the risk of a mistake in memory manipulation is low, whilst the advantages — increased abstraction and ease of use — are great.

The unpickling algorithm is based on three combinators, corresponding to the three possible configurations for serialised values seen in Section 8.3:

```
val sum    : (int×id list → T.a m) → id → T.a m
val tuple  : (id list → T.a m) → id → T.a m
val record : (T.a → id list → T.a m) → int → id
             → T.a m
```

The *sum* operator takes a function and an identifier and returns a monadic computation that returns the deserialised value associated with the identifier when run. Running *sum* applies the function to a constructor number and a list of subvalue identifiers, enabling the function to construct the value. The *tuple* operator is identical to *sum* except for the omission of the constructor argument. The type of the *record* operator reflects the fact that records with mutable fields allow cycles to arise. The second argument to *record* lists the number of fields, which is sufficient information to allocate a blank object. The object is passed to the argument function, which initialises it by supplying a value for each field.

The specification in Figure 8 gives more details. The generated instance for a record type r includes a type structurally similar to r, but with every field declared mutable. The uninitialised value passed in to the argument to *record* is cast to the mutable type, allowing fields to be assigned values after all subvalues have been reconstructed. This is sufficient to handle all cycles based on mutability; there is no attempt to handle the immutable cycles which arise from OCaml's value recursion extension. As a side effect of the implementation, immutable cycles which pass through records successfully reconstruct objects; other immutable cycles cause the unpickling algorithm to loop.

8.5 Specialisation example: alpha-equivalence

The equality predicate specified in Section 5 enables the `Pickle` serialiser to introduce sharing between immutable values that have identical structure, potentially shrinking the serialised representation. Since equality can, like all operations provided by `deriving`, be customised at a particular type, there is room for even greater compression if we can specialise the definition of equality to a more inclusive predicate. For example, if we have a representation

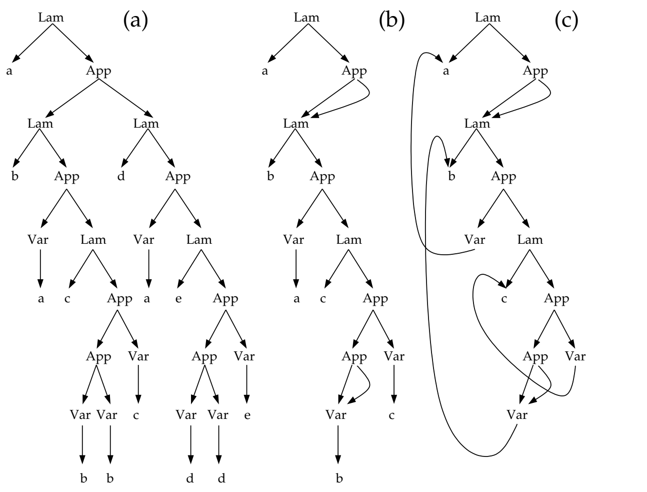

Figure 10. Sharing lambda terms

of λ terms we might reasonably consider all α-equivalent terms as equal. If we introduce an instance of `Eq` which is visible to the `Pickle` instance for λ terms then when terms are serialised only one member of each α-equivalence class will be written out.

Here we encounter a small technical difficulty with the current design of `deriving`. The preprocessor inserts generated instance definitions immediately following the definition of the associated type. Further, since `Eq` is a superclass of `Pickle`, the `Eq` instance for a type must be visible within its `Pickle` instance. Our definition of `Eq` must follow the type definition for the constructors to be visible, but the preprocessor will insert the `Pickle` definition between the definition of the type and the definition of `Eq`. We can dodge the difficulty using a recursive module, which allows us to bring a definition of `Eq` into scope within the preceding `Pickle` definition, but this is a rather distasteful necessity. We intend to address the difficulty in future versions of `deriving`.

The code for making the α-equivalence test visible to `Pickle` appears in Figure 11. Parts (a) and (b) of Figure 10 show a call-by-value fixpoint combinator represented as a value of the `exp` datatype and the sharing introduced by `Pickle`.

There is one more opportunity to increase the potential for sharing. OCaml strings are mutable, so the default definition of equality compares their physical addresses rather than their contents. A value-oriented instance of `Eq` for strings (taking OCaml's `=` for `eq`) leads to further sharing under `Pickle`, as seen in Figure 11 (c).

There is one further point of interest here. Our notation for overloaded operations is generally more cumbersome than in a language such as Haskell where there is no necessity to specify the type at each call. In this case the burden becomes an advantage: `deriving` selects instances according to the declared name of a type, so we can use different instances for a given type at various points in the program by introducing a type alias for each instance. For example, we can supply and use our new instance of `Eq` for strings as follows:

```
type istring = string
module Eq_istring = struct
  type a = istring
  let eq = (=)
end
...
type exp = Var of istring
         | App of exp×exp
         | Lam of istring×exp
           deriving (Eq)
```

```
module Env = Env.Make(String)
module rec Exp :
sig
  type exp = Var of string
           | App of exp * exp
           | Lam of string * exp
             deriving (Dump, Eq, Pickle, Typeable)
end =
struct
  module Eq_exp = struct
   open Exp
   type a = exp
   let eq (l : a) (r : a)
   = let rec alpha_eq (lenv, renv as envs) n =
     function
     | Var l, Var r ->
        (match Env.mem l lenv, Env.mem r renv with
         | true, true ->
             Env.find l lenv = Env.find r renv
         | false, false -> l = r
         | _ -> false)
     | App (fl,pl), App (fr,pr) ->
         alpha_eq envs n (fl, fr)
      && alpha_eq envs n (pl, pr)
     | Lam (vl,bl), Lam (vr,br) ->
         alpha_eq
           (Env.add vl n lenv, Env.add vr n renv)
           (n+1)
           (bl, br)
     | _ -> false
   in alpha_eq (Env.empty, Env.empty) 0 (l,r)
  end
  type exp = Var of string
           | App of exp * exp
           | Lam of string * exp
             deriving (Dump, Typeable, Pickle)
end
```

Figure 11. Using α-equivalence as equality to increase sharing

This definition allows `Eq.eq<exp>` to use structural equality for string comparisons, but has no effect on the rest of the program. (A similar point is made in [6], which couples Haskell-style overloading with an operation for explicitly making a particular instance "canonical" in a certain scope.)

8.6 Relative compactness of serialised data

We measured the spacewise performance of the serialisers described in Sections 7 and 8 in two applications. First, we used Camlp4 to parse a selection of variously-sized OCaml files (`path.ml` (1.8k), `predef.ml` (7.6k) and `includemod.ml` (15k)) from the OCaml distribution, passed the resulting syntax trees to the `Marshal`, `Dump` and `Pickle` serialisers, and compared the size of the output. Next, we compared the sizes of the representations generated for lambda terms by each serialiser on lambda terms, using the α-equivalence-aware serialiser developed in Section 8.5 and three collections of randomly-generated open lambda terms published with [23]. The results of both tests are shown in Table 1.

Since `Marshal` has access to the low-level representations of values we might expect that its output will be the most compact. Indeed, the results of the "OCaml" test bear this out: both `Dump` and `Pickle` performed significantly (although not disastrously) worse. Surprisingly, on the "Lambda" test, the naive `Dump` serialiser consistently outperformed `Marshal`, albeit only slightly. Most gratifyingly, however, the α-equivalence-enhanced `Pickle` surpassed both, yielding output between two and four times smaller (and up to eleven times smaller than the uncustomised `Pickle` serialiser). These results suggest that it is worthwhile exploring

	Marshal	**Dump**	(× Marshal)	**Pickle**	(× Marshal)	**Pickle (specialised)**	(× Marshal)
OCaml (path.ml)	5752	9904	**1.72**	9178	**1.59**	-	-
OCaml (predef.ml)	27890	48918	**1.75**	39596	**1.42**	-	-
OCaml (includemod.ml)	53292	108016	**2.03**	75839	**1.42**	-	-
Lambda (reg.)	8732	8447	**0.97**	21948	**2.51**	5233	**0.60**
Lambda (med.)	34993	33906	**0.97**	89990	**2.57**	10045	**0.29**
Lambda (big.)	63513	61557	**0.97**	176811	**2.78**	15259	**0.24**

Table 1. Comparative size in bytes of the output of Dump, Pickle and Marshal serialisers on various inputs.

domain-specific solutions to serialisation problems, and that extensibility to incorporate domain knowledge is a valuable facet of the design of a serialiser.

8.7 Safety

We saw in Section 2.1 the disastrous effects of passing invalid data to the `Marshal` functions. Since `Pickle` and `Dump` are type-aware, no such problems arise. If the string passed to `Pickle.from_string` is a valid representation of the declared return type then it will be interpreted at that type; if it is not, an exception will be raised. The `Dump` serialiser behaves similarly.

```
Pickle.from_string<float> (Pickle.to_string<int> 0)
⟹ 0.0
Pickle.from_string<Exp.Exp.exp>
     (Pickle.to_string<int> 0)
⟹ Exception: Pickle.Pickle.UnpicklingError
       "Error unpickling constructor".
```

Although this behaviour is "safe" in the sense that it will not cause a program crash, it may be that it is not quite what is desired. For some applications it might be more appropriate to ignore representation compatibility, and raise an exception whenever there is a mismatch between serialisation and deserialisation types. It would be straightforward to add this behaviour by serialising a representation of the type along with the value, but until the difficulty with mapping type constructors to unique names described in Section 8.2 is resolved it would only be possible to check equality of serialised types within the same compiled instance of the program which produced them.

9. Serialisation: related work

The serialisation problem has received considerable attention in the ML community. The HashCaml [1] project extends the OCaml bytecode compiler with type-passing to make the standard `Marshal` operation type-safe. Cohen and Herrmann [4] discuss the implementation of a staged serialiser in MetaOCaml which bears some similarity to our preprocessor-based approach, with even more reliance on unsafe low-level operations, used by the serialiser to break through abstraction barriers. Tack, Kornstaedt and Smolka [28] describe the addition of serialisation as a primitive service to Alice ML, giving an elegant account of their serialisation algorithm in terms of a domain-specific instruction set.

We discussed two libraries of serialisation combinators [7, 20] in Section 2.2. Karvonen [18] discusses ways to address some of the deficiencies in Elsman's library. Finally, serialisation has been a common example program in the generic programming community: see for example [12]. One disadvantage suffered by our approach in comparison to almost all alternatives is a certain lack of type-safety: it is easy to use Camlp4 to generate code that subsequently fails to compile. Although it would be preferable to detect the error at an earlier stage, we do not view this as a serious drawback, since it does not lead to the execution of ill-typed programs.

10. Conclusions and future work

We have described the design and implementation of `deriving`, a system for generating generic functions from type definitions, and its particular application to the serialisation problem. In contrast to similar systems for OCaml, `deriving` requires no support from tools outside the standard OCaml distribution, which makes it easy to incorporate into existing projects. The `deriving` encoding of serialisation places a much lower burden on the user and much greater coverage than the combinator approach; it offers greater safety and flexibility than standard OCaml marshalling. Perhaps most enticingly, the specialisation property, which allows a user to supply an alternative implementation for a derived function at a particular type, makes it possible to improve serialisation using domain knowledge without writing any serialisation code, leading to a considerable reduction in the size of serialised data. Generated instances produce acceptably compact output compared to `Marshal`, even without specialisation.

We have shown how an understanding of the correspondence between modules and type-classes can guide software design and described techniques to make it viable in the absence of a full system of recursive modules. We have outlined various other techniques discovered during the development of `deriving` that should be transferable to similar projects, such as the simulation of regular recursive functors by functors and recursive modules.

We have several future directions in mind for `deriving`. The simplest is the addition of new classes for various tasks: parsing, traversals and random value generation, for instance. The performance of the `Pickle` serialiser could be greatly improved by basing sharing on hashing rather than equality; the current implementation requires a linear search to detect sharing. A convenient facility for writing functions which are parameterised over the instances of a class would significantly improve the usefulness of `deriving`: it is presently possible to write such functions using functors but the syntax is cumbersome. Finally, `deriving` currently provides more support for *using* the generic functions supplied with the implementation than for *writing* new generic functions. It is obviously desirable to eliminate this bias. The `deriving` implementation provides a fairly convenient framework for expressing generic functions in a Camlp4 extension, but it would be preferable to write generic functions directly in OCaml. One possible approach is to use `deriving` to generate general value-level representations of datatypes which can then be used by generic functions to traverse values of those types.

Acknowledgments

Thanks to Philip Wadler, Sam Lindley, Ezra Cooper, Vesa Karvonen and the three anonymous reviewers for their helpful comments which greatly improved the presentation.

References

[1] John Billings and Peter Sewell and Mark Shinwell and Rok Strniša. Type-safe distributed programming for OCaml *MCM SIGPLAN Workshop on ML*, 2006.

[2] Jacques Carette, Lydia E. van Dijk and Oleg Kiselyov. Syntactic extension for Monads in Ocaml. `http://www.cas.mcmaster.ca/~carette/pa_monad/`.

[3] James Cheney and Ralf Hinze. A lightweight implementation of generics and dynamics. *Haskell Workshop*, 2002.

[4] Albert Cohen and C. Herrmann. Towards a High-Productivity and High-Performance Marshaling Library for Compound Data. *2nd MetaOCaml Workshop*, 2005.

[5] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. *FMCO*, 2006.

[6] Derek Dreyer, Robert Harper, Manuel Chakravarti, and Gabriele Keller. Modular Type Classes. *POPL*, 2007.

[7] Martin Elsman. Type-Specialized Serialization with Sharing. *Trends in Functional Programming*, 2005.

[8] Andrzej Filinski. Controlling Effects. PhD Thesis, School of Computer Science, Carnegie Mellon University, May 1996.

[9] Jacques Garrigue. Programming with Polymorphic Variants. *ML Workshop*, 1998.

[10] Jeremy Gibbons. Design patterns as higher-order datatype-generic programs. *Workshop on Generic Programming*, 2006.

[11] Ralf Hinze, Johan Jeuring, and Andres Löh. Comparing approaches to generic programming in Haskell. *Spring School on Datatype-Generic Programming*, 2006.

[12] Ralf Hinze. Generics for the masses. *ICFP*, 2004.

[13] Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. "Scrap Your Boilerplate" Reloaded *FLOPS*, 2006.

[14] John Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2), 1989.

[15] Martin Jambon. json-static. `http://martin.jambon.free.fr/json-static.html`.

[16] Martin Jambon. pa_tryfinally. `http://martin.jambon.free.fr/pa_tryfinally.ml`.

[17] Patrik Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. *POPL*, 1997.

[18] Vesa Karvonen. Generics for the Working ML'er. *ML Workshop*, 2007.

[19] R. Kelsey, W. Clinger, J. Rees (eds.). Revised^5 Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation*, 11(1), 1998.

[20] Andrew J. Kennedy. Functional Pearl: Pickler Combinators. *JFP*, 14(6), 2004.

[21] Oleg Kiselyov. Post to Haskell list, August 2004. `http://haskell.org/pipermail/haskell/2004-August/014463.html`

[22] Ralf Lämmel and Simon Peyton-Jones. Scrap your boilerplate: a practical design pattern for generic programming. *TLDI*, 2003.

[23] Chuck Liang and Gopalan Nadathur. Tradeoffs in the Intensional Representation of Lambda Terms. *RTA*, 2002.

[24] Neil Mitchell and Stefan O'Rear. Derive. `http://www-users.cs.york.ac.uk/~ndm/derive/`.

[25] Simon Peyton Jones and John Hughes (editors). Haskell 98: A Non-strict, Purely Functional Language. February, 1999.

[26] Daniel de Rauglaudre. IoXML. `http://cristal.inria.fr/~ddr/IoXML/`.

[27] Martin Sandin. Tywith. `http://tools.assembla.com/tywith/wiki`.

[28] Guido Tack, Leif Kornstaedt and Gert Smolka. Generic Pickling and Minimization. *ML Workshop*, 2005.

[29] Philip Wadler The essence of functional programming. *POPL*, 1992.

[30] Stefan Wehr. ML Modules and Haskell Type Classes: A Constructive Comparison. Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany, November 2005.

[31] Stephanie Weirich. RepLib: A library for derivable type classes. *Haskell Workshop*, 2006.

[32] Noel Winstanley. DrIFT. `http://repetae.net/~john/computer/haskell/DrIFT/`.

[33] Programming languages - C. ISO/IEC 9899:1999.

[34] American National Standard for Programming Language Common LISP. ANSI X3.226:1994. J13/SC22/WG16 Common LISP