Towards a High-Productivity and High-Performance Marshaling Library for Compound Data

Albert Cohen
Alchemy Group, INRIA Futurs
PARC CLUB, 2-4 rue Jacques Monod
91893 Orsay Cedex, France
Albert.Cohen@inria.fr

Christoph A. Herrmann
University of Passau, FMI
Innstr. 33
94032 Passau, Germany
Christoph.Herrmann@uni-passau.de

ABSTRACT

Marshaling, also known as serialization, is a key software layer to support modern communication and storage systems. Implementations of marshaling and unmarshaling functions for such systems are heavily optimized to reduce memory management and control overhead. Simultaneously, traditional marshaling frameworks aim for polymorphism and generality, and few works addressed the customization to specific application needs and context. In particular, traditional frameworks do not enforce any compile-time safety properties on marshaled data and lack fine-grain accessing to parts of the marshaled data. This decreases software quality and leads many programmers to resort to custom implementations, hence reducing productivity. We experienced these problems during our ongoing research on generative and skeleton-based approaches to parallel computing, and this work presents our findings about how to design a safer, more customizable marshaling framework.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Metaprogramming

Keywords

High-Performance, MetaOCaml, Marshaling, Sized Types, Staging

1. INTRODUCTION

This paper presents an original library design tailored for high-productivity, low-overhead marshaling of typed compound data. It allows for some level of static type checking, elementwise access, in-place marshaling as well as in-place unmarshaling, and side-effects on array and tuple elements in marshaled data. Although not as polymorphic as the (type unsafe) standard OCaml marshaling library, it is compatible with it and can bring significant performance improvements, both in memory and computation overhead. This library is fully implemented in MetaOCaml, making heavy use of modules and functors to construct custom abstract types for marshalable and marshaled data. Type safety comes with some flexibility restrictions: all marshalable data is associated with a custom abstract type, derived from functor instantiations, with a statically enforced maximum memory footprint. As a consequence, the maximal size of arrays and unstructured marshaled data (handled by the standard OCaml marshaling library) must be known at module instantiation time.

Several optimizations reduce the run-time overhead of a MetaOCaml implementation. High-level optimizations include fast offset computations based on static size inference, multi-stage evaluation to implement specialization of custom and elementwise marshaling and unmarshaling functions, static coalescing of elementwise accesses on unboxed types, as well as in-place operation. Several low-level optimizations involve type cast tricks to operate on the internals of OCaml values. Performance improvements can be observed with respect to the OCaml standard marshaling library on some data types and operations, and native code generation allows to keep the performance degradation quite reasonable for more complex data.

We identified the following design alternatives for a type-safe marshaling library, listed in the order of increasing sophistication.

1. Direct Approach: each marshalable type is given a dedicated marshaling function. The disadvantage is obvious: as soon as the application programmer wants to marshal data of a new type, he must add the marshaling function himself. Including all the special features we mentioned above this is not a reasonable approach, even for experts. Thus, the marshaling functions in the following alternatives have to be polymorphic.

2. Algebraic data type approach: each data object of a marshalable type has to be given a tag to perform pattern matching on it. For both efficiency and programming comfort this approach is unacceptable for standard data types, but due to its simplicity it may be useful for types of a domain-specific language. Therefore, we discuss it in Section 2.

3. Object-oriented approach: instead of a tag, each marshalable type is made an instance of a marshalable class. With a clean construction, the entire data must be available to build the object hierarchy before the actual marshaling function can be generated, i.e., the first stage cannot be executed before run time.

4. Module-based approach: this provides us with the possibility to generate code expressions for parameterized types. The marshaling functions are specialized for each base type and each type constructor, inheriting the marshaling of the substructures. This is discussed in Section 3.

2. ALGEBRAIC DATA-TYPE APPROACH

Our first approach discussed the case that the data to be marshaled is embedded into an algebraic data type and the marshaling and unmarshaling functions work on that type. That provides us with a means to define marshaling and unmarshaling functions which operate on a single MetaOCaml type.

2.1 Structure of the Type Tree

The code for marshaling and unmarshaling is generated by interpretation of an object of type `minfo` which reflects the type tree of the data to be marshaled. This object can be calculated automatically from a tagged data representation. For simplicity, we discuss only three significant kinds: (1) `Int` denotes an int value which fits exactly in one cell of the marshaling buffer, (2) `Pair`(l,r) represents a pair of an element of type l and an element of type r and (3) `Arr`(n,a) an array of length n with each element being of type a. [1] Knowing the sizes of arrays is crucial for selective (un)marshaling parts of a marshaled buffer, our main motivation for this approach.

```
type minfo =
  Int
| Pair of minfo * minfo
| Arr of int * minfo
```

Function `size` calculates the number of entries in the marshaling buffer for the data whose shape is represented by the particular type tree. Abstracting the shape information of array elements by index would give us the possibility of dealing with array elements of different shapes. However, this would make the size calculation dependent on the array sizes and thus we abstained from it for performance reason.

```
let rec size x = match x with
  Int -> 1
| Pair (a,b) -> size a + size b
| Arr (n,a) -> n * size a
```

2.2 Navigation in the Type Tree

We provide an abstraction mechanism for the user to address a subobject of the data in the marshaled buffer, in order to read or replace it by an object of the same shape. An expression of type `mpos` describes the path from the root of the type tree to the node where the subobject of interest is rooted at. `Here` identifies the current node as the node of interest. In the case that the current node represents a pair, `Left` e or `Right` e continue the selection with the left or right entry of this pair, respectively. In the case of an array, `AIndex` (i, a) continues the selection with the type tree a for an element, notifying that it was element number i to be selected.

```
type mpos =
  Here
| Left of mpos
| Right of mpos
| AIndex of int * mpos
```

[1] Usually, the length of an array is not made part of its type, to enable a run-time length determination and compile-time decidability of type equality at the same time.

Function `positionSize` computes the first position and the size of a subobject of interest in the marshaled buffer. It receives as arguments a base address, a navigation expression and the type tree. Mismatches between the type tree and the navigation expression can be detected when running the program the first time, without knowing the values of array sizes and indices the program might be specialized for later.

```
let rec positionSize acc p tt = match p,tt with
  Here,_  -> (acc,size tt)
| Left q, Pair(l,r)
          -> positionSize acc q l
| Right q, Pair(l,r)
          -> positionSize (acc + size l) q r
| AIndex (i,q),Arr(n,a)
          -> positionSize (acc + i * size a) q a
```

2.3 Data in the Application Domain

Without the module approach discussed in a later section of this paper, we have to tag all data objects we want to marshal, since the generated code can only produce values of one particular type. This type is given by `mystruct`. For the cases of ints and pairs this may lead to inefficiencies, especially when they occur as array elements, since the tags are repeated over and over again. Therefore, it is necessary to extend the three mentioned cases by all cases for which an efficient solution is desired, e.g., arrays of int pairs, matrices etc. We would like to mention that the three cases discussed serve mainly for a comparison with the module approach and that tags will be appropriate if used only for larger objects in an application domain.

```
type mystruct =
  MInt of int
| MPair of mystruct * mystruct
| MArr of mystruct array
```

The typetree of a tagged data object can be calculated automatically by function `typetree`. This provides us with two possibilities:

1. marshaling and unmarshaling code, once generated can be reused for data objects with the same type tree;
2. if data objects are to be send via a network, it is sufficient to send an encoding of the type tree in order to generate the appropriate unmarshaling code at the receiver's side.

```
let rec typetree x = match x with
  MInt _ -> Int
| MPair (l,r) -> Pair (typetree l, typetree r)
| MArr a -> let n = Array.length a
            in if n=0 then Arr (0,Int)
                      else Arr (n,typetree (a.(0)))
```

2.4 Generating Marshaling and Unmarshaling

Functions which generate the marshaling and unmarshaling code are analogous to each other: they take the code name of the buffer (bufv), the code name of the current buffer index (posv) and the type tree as arguments. While the marshaling code does a pattern match on the application data and overwrites the contents of the marshaling buffer, the unmarshaling code use the buffer content to fill the entries of the application object whose structure is given by the type tree.

```
let gen_marshal bufv posv typetree =
  let rec gm tt = match tt with
    Int  -> .< fun (MInt x) ->
               begin
                (.~bufv).(!(.~posv)) <- x;
                (.~posv) := !(.~posv)+1
               end >.
  | Pair(l,r) -> .< fun (MPair (x,y)) ->
                    .~(gm l) x;
                    .~(gm r) y  >.
  | Arr(n,a) ->
      let n1=n-1
      in .< fun (MArr xs) ->
            for i=0 to n1 do
               .~(gm a) (xs.(i))
            done >.
  in gm typetree

let gen_marshal_at start tt =
 .< fun buf ->
     let pos = ref start
     in .~(gen_marshal .<buf>. .<pos>. tt) >.


let gen_unmarshal bufv posv typetree =
  let rec gu tt = match tt with
    Int -> .< let r = (.~bufv).(!(.~posv))
              in (.~posv) := !(.~posv)+1;
                MInt r >.
  | Pair (l,r) -> .<let x = .~(gu l) in
                    let y = .~(gu r) in
                    MPair (x,y) >.
  | Arr (n,a) ->
    .< MArr (Array.init n (fun i -> .~(gu a))) >.
  in gu typetree

let gen_unmarshal_at start tt =
  .< fun buf ->
     let pos = ref start
     in .~(gen_unmarshal .<buf>. .<pos>. tt) >.
```

2.5 Example Expression

Our example expression is an array of length 3, whose elements are pairs of an int an array of 4 ints. The int values are of no interest here.

```
MArr (Array.init 3 (fun i ->
     MPair (MInt i,
            MArr (Array.init 4 (fun j ->
                 MInt (10*i+j) )))))
```

Function typetree computes the following type tree:

```
Arr (3, Pair (Int, Arr (4, Int)))
```

For this type tree, the following marshaling code is generated, where the first argument (buf_1) is the memory area the data structure in the second argument (MArr (xs_3)) is marshaled to.

```
- : ('a, int array -> mystruct -> unit) code =
.<fun buf_1 ->
   let pos_2 = (ref 0) in
   fun MArr (xs_3) ->
    for i_4 = 0 to 2 do
     ((fun MPair (x_5, y_6) ->
        ((fun MInt (x_10) ->
           buf_1.(! pos_2) <- x_10;
           (pos_2 := ((! pos_2) + 1))) x_5);
        ((fun MArr (xs_7) ->
           for i_8 = 0 to 3 do
            ((fun MInt (x_9) ->
               buf_1.(! pos_2) <- x_9;
               (pos_2 := ((! pos_2) + 1)))
              xs_7.(i_8))
           done) y_6)) xs_3.(i_4))
    done>.
```

Although the code appears to be quite concise, it is not competitive with the standard marshaling routine. Tag elimination is one of the necessary steps to improve performance, as well as further optimizations to take advantage of flat data layout of unboxed values. These optimizations are implemented in the next section, leading to a more advanced type-safe implementation combining the module system with multi-stage evaluation.

3. MODULE-BASED APPROACH

From a methodical point of view, using functors is the appropriate way to specialize functions for compound types statically. Therefore, OCaml provides the constructs module and functor.

In contrast to the variety of marshalable types, the marshaled data itself will in the following have the concrete type `int * int array`, where the array member stores the actual encoded data and the int member keeps tracks of offset calculations for elementwise accesses. This concrete type is used by internal functions of the library, but all user-visible functions use an abstract type encapsulation that eventually allows to enforce type safety.

3.1 Building Abstract Types for Marshaled Data

The module system of OCaml permits us to define overloaded functions whose types are expressed in terms of type parameters, which are instantiated individually at each point where the respective function is applied. In contrast to a polymorphic function, a function overloaded by a module can be specialized for particular instances, e.g., for ints, pairs of ints or int arrays. Because we want pairs and arrays be nested in arbitrary depth and order, (un)marshaling functions for composite types (pairs and arrays) will be enclosed in type-parameterized modules, whereas functions operating on ints and floats will be enclosed in monomorphic modules. This defines an inductive module hierarchy with a generic implementation for each marshalable data type.

We first distinguish types which can be marshaled directly (module type CAM: core atomic marshaling). Ints and floats are made instances of this module. All types of data to be marshaled have to be made instance of module type CCM (core composite marshaling). Ints and floats are transformed from CAM to CCM automatically by the functor CCM_build, see Figure 1. Pairs and arrays have to be provided a dedicated implementation.

Core Atomic Marshaling Module

The atomic marshaling module contains a generic type name (data) for the data object and an integer value (size) carrying the number

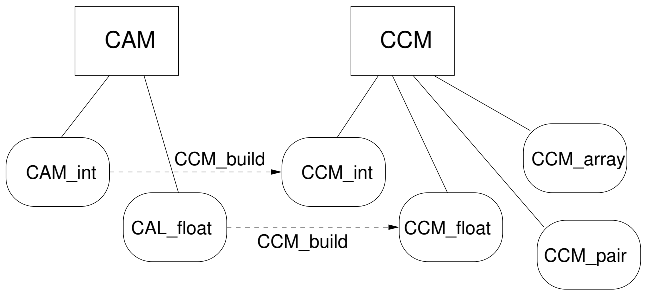

Figure 1: Core marshaling modules

of cells the value will consume in the marshaling buffer.

```
module type CAM =
sig
  type data
  val size : int
end
```

Ints are made an instance of `CAM` by the following definition. They consume one cell because the internal representation is an int array. Floats are handled analogously with a size value of 2.

```
module CAM_int =
struct
  type data = int
  let size = 1
end
```

Core Composite Marshaling Module

The module `CCM` includes the type and the size from `CAM`. Additionally, it specifies whether the value to be marshaled has a boxed representation in OCaml (used for optimization purposes), and it contains the core marshaling and unmarshaling functions. These functions will be described in detail later.

```
module type CCM =
sig
  include CAM
  val boxed : bool
  val marshal_at_static :
      ('a, int) code -> ('a, data) code
   -> ('a, int array) code -> ('a, unit) code
  val unmarshal_at_static :
      ('a, int) code -> ('a, int array) code
   -> ('a, data) code
  ...
  val marshal_at :
    int -> data -> int array -> unit
  ...
end
```

Build a CCM from a CAM

Each instance of module `CAM` is converted into an instance of module `CCM` automatically by functor `CCM_build`. The type and size is included and the boxed attribute set to false. Since knowledge of neither substructures nor memory layout of `CAM` types is assumed but just knowledge of the base address and the size, low-level memory copying functions (e.g., `blit_static`) are called for the (un)marshaling.

```
module CCM_build (A : CAM) : CCM
  with type data = A.data =
struct
  include A
  let boxed = false
  let marshal_at_static off_code x_code buf_code =
    blit_static .< ref .~x_code >. .< 0 >. buf_code
                off_code size
```

Implementation for Pairs

Pairs are made an instance of `CCM` parameterized in the types of the first (`A`) and second (`B`) component. All layout information is given by the type so the size in the marshaled buffer is simply the sum of the sizes of both components.

```
module CCM_pair (A : CCM) (B : CCM) :
       CCM with type data = A.data * B.data =
struct
  type data = A.data * B.data
  let size = A.size + B.size
  let boxed = true
  ...
```

Implementation for Arrays

Arrays are made an instance of `CCM` parameterized in the element type (`A`) and a size-carrying type for the array length (`N`), defined in module `INT`. In contrast to a sized (dependent) type, the size must be instantiated in the program text to a concrete value.

```
module CCM_array (A : CCM) (N : INT) :
                 CCM with type data = A.data array =
struct
  type data = A.data array
  let size = N.value * A.size
  let boxed = true
  ...
```

Module `INT` contains an inductive definition that builds size-carrying types for (a bounded number of) integers, to be used as length parameters for arrays. `INT_inc` is a functor which provides us with the successor function on this type. We also provide a functor to double the value of such a type, and a geometric series of predefined types (for the most frequent array sizes).

```
module type INT =
sig
  val value : int
end

module INT_inc =
  functor (I : INT) ->
struct
  let value = I.value + 1
end

module Int1 : INT = struct let value = 1 end

module Int2 : INT = INT_inc (Int1)
```

3.2 Operating on Marshaled Data

Each module instance of the `CCM` module type implements a couple of functions for (un)marshaling data to/from a given location.

```
val marshal_at :
```

```
  int -> data -> int array -> unit

val unmarshal_at :
  int -> int array -> data
```

Staging is possible thanks to static type and size information about all pieces of data to be marshaled.

```
(** Code combinator for marshaling
    data to a given location. *)
val marshal_at_static :
  ('a, int) code -> ('a, data) code
  -> ('a, int array) code -> ('a, unit) code

(** Code combinator for unmarshaling
    data from a given location. *)
val unmarshal_at_static :
  ('a, int) code -> ('a, int array) code
  -> ('a, data) code
```

These (un)marshaling functions are built for all inductively defined types on int, pairs and arrays, and are not accessible to the end-user.

For example, here are the respective marshaling and unmarshaling functions for the pair functor. A and B are the module parameters of the pair, of module type `CCM`.

```
let marshal_at off p buf =
  if A.boxed || B.boxed then
    begin
      A.marshal_at off (fst p) buf;
      B.marshal_at (off + A.size) (snd p) buf
    end
  else
    blit p 0 buf off size

let marshal_at_static off_code p_code buf_code =
  if A.boxed || B.boxed then
    let off2_code = .< .~off_code + A.size >.
    and x_code = .< fst (.~p_code) >.
    and y_code= .< snd (.~p_code) >. in
    .<
      begin
       .~(A.marshal_at_static off_code x_code buf_code);
       .~(B.marshal_at_static off2_code y_code buf_code)
      end
    >.
    else
      blit_static p_code .< 0 >. buf_code off_code size
```

The respective functions for arrays are more complex: they provide specialized implementation for very short (fully unrolled), medium (partially unrolled) and long (call to the blit function of the standard library) arrays, taking advantage of static typing and size inference to specialize offset computations and memory allocation.

3.3 In-Place Marshaling

All marshaling functions defined in the core modules operate *in place*. This is critical for performance and also because they are mostly used as support functions for elementwise accesses in compound marshaled data. More classical marshaling semantics — associated with the allocation of a fresh buffer for the marshaled data — are supported at the higher level of abstraction provided as end-user programming interface.

3.4 In-Place Unmarshaling and Record Types

Another benefit from static typing is the ability to define *in-place unmarshaling* functions for mutable types, and primarily for arrays. Currently, our implementation defines in-place functions for all types, but immutable ones (e.g., unboxed or pair types) trigger exceptions when used, and generator-stage exceptions for staged versions. In-place operation is currently limited to the first level, yet deeper in-place unmarshaling would not be difficult to define.

Beyond arrays, in-place operation applies to record types with mutable fields as well. We did not implement marshaling functions for such types yet because it is unclear whether the module system offers a flexible enough framework to define specific module instantiations for each custom record type. This is left for future work.

4. MARSHALING PROGRAMMER INTERFACE

Let us now present the upper layer of the library. This layer provides the elementwise accessing functions and abstracts the marshaled data type to enforce type-safety.

4.1 Selection of the Staging Level

For each data type, it is possible to customize the staging level of all (un)marshaling functions. For example, considering an array of pairs of ints, one may choose to specialize operations on each individual pair within arrays, specialize operations for the whole structure, or prefer a fully dynamic implementation without staging. This selection of the staging level makes use of the following functor, to map unstaged functions to the dynamic dispatch of once-and-for-all generated code expressions.

```
module CCM_stage (A : CCM) =
struct
  include A

  let marshal_at =
    .!
      .<
        fun off x buf ->
          .~(A.marshal_at_static
                .< off >. .< x >. .< buf >.)
      >.
  let unmarshal_at =
    .!
      .<
        fun off buf ->
          .~(A.unmarshal_at_static .< off >. .< buf >.)
      >.
  let unmarshal_in_place_at =
    .!
      .<
        fun off buf d ->
          .~(A.unmarshal_in_place_at_static
                .< off >. .< buf >. .< d >.)
      >.
end
```

When using this functor, the user is guaranteed that all code generation takes place at module instantiation time, incurring no run-time overhead, but only a kind of just-in-time overhead.

4.2 Enforcing Type Safety

From the core modules above, we provide an abstract interface for safe user-level operations. Module type CM hides the set of marshal_at internal functions and makes the marshaled type abstract; it also defines functions that provides the standard marshaling, unmarshaling and in-place unmarshaling operations on this abstract type, as well as elementwise set, get and in-place get functions.

```
module type CM =
sig
  include CCM

  type marshaled

  val marshal : data -> marshaled
  val unmarshal : marshaled -> data
  val unmarshal_in_place : marshaled -> data -> unit

  val set : data -> marshaled -> unit
  val get : marshaled -> data
  val get_in_place : marshaled -> data -> unit

  (* ... *)
end
```

This signature is used to encapsulate the core modules above through the following functor:

```
module CM_build (C : CCM) : CM
  with type data = C.data
  and type marshaled = int * int array =
struct
  include C

  type marshaled = int * int array

  let marshal d = let buf = create size
                  in marshal_at 0 d buf; (0, buf)
  let unmarshal (_, buf) = unmarshal_at 0 buf
  let unmarshal_in_place (_, buf) d
      = unmarshal_in_place_at 0 buf d

  let set d (off, buf) = marshal_at off d buf
  let get (off, buf) = unmarshal_at off buf
  let get_in_place (off, buf) d
      = unmarshal_in_place_at off buf d
end
```

This functor is typically used to define the interface for (un)marshaling ints, called CM_int, in a straightforward way:

```
module CM_int : CM with type data = int
              = CM_build (CCM_int)
```

4.3 Elementwise Access

The interface for pairs and arrays are a bit more complex, since they must support offset computations to access their individual elements (through the get and set functions).

```
module CM_pair (A : CM) (B : CM) :
sig
  include CM
  (* Make the marshaled type abstract but export
     fst and snd despite the CM type constraint   *)
  val fst : marshaled -> A.marshaled
  val snd : marshaled -> B.marshaled
end
  with type data = A.data * B.data =
struct
  include CM_build (CCM_pair (A) (B))
          (* Build a concrete marshaled type *)

  let fst ((off, buf) : marshaled) : A.marshaled =
    Obj.magic (off, buf)
  let snd ((off, buf) : marshaled) : B.marshaled =
    Obj.magic (off + A.size, buf)
end

module CM_array (A : CM) (N : INT) :
sig
  include CM

  val idx : marshaled -> int -> A.marshaled
end
  with type data = A.data array =
struct
  include CM_build (CCM_array (A) (N))

  let idx ((off, buf) : marshaled) i : A.marshaled =
    if i < 0 || i >= N.value then
      failwith ("CM_array.idx: out of bounds index "
                ^ string_of_int i)
    else
      Obj.magic (off + i*A.size, buf)
end
```

Since the marshaled type is abstract in CM, we have a technical problem defining these offset computation functions: they need to access the int component of the concrete type. For the sake of simplicity, we use a type cast for that (Obj.magic); it would be cleaner to use a get_offset function to retrieve the offset, but that would only solve half of the problem since the concrete offsets and int arrays need to be converted back to the abstract type. We are still looking for a better solution.

Eventually, using the CM_stage functor we defined staged versions of the CM_pair and CM_array functors as well, e.g.:

```
module CM_staged_pair (A : CM) (B : CM) :
sig
  include CM

  val fst : marshaled -> A.marshaled
  val snd : marshaled -> B.marshaled
end
  with type data = A.data * B.data =
struct
  include CM_build (CCM_stage (CCM_pair (A) (B)))

  let fst ((off, buf) : marshaled) : A.marshaled =
    Obj.magic (off, buf)
  let snd ((off, buf) : marshaled) : B.marshaled =
    Obj.magic (off + A.size, buf)
end
```

4.4 Simple Examples

In the following example, we define some modules to operate on arrays of pairs of ints, listing all possible ways to stage the computation.

```
module N = Int256;;
```

```
module CM_pi = CM_pair (CM_int) (CM_int);;
module CM_spi = CM_staged_pair (CM_int) (CM_int);;
module CM_api = CM_array (CM_pi) (N);;
module CM_aspi = CM_array (CM_spi) (N);;
module CM_sapi = CM_staged_array (CM_pi) (N);;
```

The "s" in the above module names stands for "staged" and marks the boundary between staged and unstaged computations. E.g., `CM_aspi` is the module to (un)marshal arrays of pairs of ints where all functions operating on pairs have been specialized.

In the next example, marshaling and unmarshaling are successively applied to the same array of pairs of ints `api`. The intermediate results `m_api` and variants with an "s" are all identical but of some different abstract type. This is a case where making the marshaled data type abstract is over-conservative and rules out legitimate exchanges of values between staged and unstaged variants; this will be discussed further on.

```
let m_api = CM_api.marshal api;;
let m_aspi = CM_aspi.marshal api;;
let m_sapi = CM_sapi.marshal api;;

CM_api.unmarshal m_api;;
CM_aspi.unmarshal m_aspi;;
CM_sapi.unmarshal m_sapi;;

for j = 0 to (N.value - 1) do
  CM_int.set 42 (CM_pi.fst (CM_api.idx m_api j));
  CM_int.set 17 (CM_pi.snd (CM_api.idx m_api j))
done;;

for j = 0 to (N.value - 1) do
  Printf.printf "%d " (CM_pi.get (CM_api.idx m_api j))
done;;
```

One may also produce the specialized code to operate on ints or pairs of ints individually, using the `_code` variants of the get and set functions:

```
CM_int.get_code (CM_pi.fst (CM_api.idx m_api 42));;
- : ('a, CM_int.marshaled -> CM_int.data) code =
.<fun (_, buf_1) -> ([|buf_1.(84)|]).(0)>.

CM_pi.get_code (CM_api.idx m_api 42);;
- : ('a, CM_pi.marshaled -> CM_pi.data) code =
.<fun (_, buf_1) ->
   let s_2 = (Array.create 2 0) in
   begin s_2.(0 + 0) <- buf_1.(84 + 0);
         s_2.(0 + 1) <- buf_1.(84 + 1) end;
   s_2>.

CM_api.get_code m_api;;
- : ('a, CM_api.marshaled -> CM_api.data) code =
.<fun (_, buf_1) ->
   (Array.init 256
     (fun i_2 ->
       let s_3 = (Array.create 2 0) in
       begin
        s_3.(0 + 0) <- buf_1.((0 + (i_2 * 2)) + 0);
        s_3.(0 + 1) <- buf_1.((0 + (i_2 * 2)) + 1)
       end;
       s_3))>.

CM_int.set_code (CM_pi.fst (CM_api.idx m_api 42));;
- : ('a, CM_int.data -> CM_int.marshaled -> unit) code =
.<fun d_1 -> fun (_, buf_2) ->
    buf_2.(84 + 0) <- (ref d_1).(0 + 0)>.

CM_pi.set_code (CM_api.idx m_api 42);;
- : ('a, CM_pi.data -> CM_pi.marshaled -> unit) code =
.<fun d_1 ->
   fun (_, buf_2) ->
    buf_2.(84 + 0) <- d_1.(0 + 0);
    buf_2.(84 + 1) <- d_1.(0 + 1)>.

CM_api.set_code m_api;;
- : ('a, CM_api.data -> CM_api.marshaled -> unit) code =
.<fun d_1 ->
   fun (_, buf_2) ->
    for i_3 = 0 to (256 - 1) do
     buf_2.((0 + (i_3 * 2)) + 0) <- (d_1.(i_3)).(0 + 0);
     buf_2.((0 + (i_3 * 2)) + 1) <- (d_1.(i_3)).(0 + 1)
    done>.
```

5. PERFORMANCE TUNING

This section gathers OCaml hacks dependent on the current implementations of the compiler, mostly associated with the use of `Obj.magic`. These hacks are justified by performance, software engineering... and paradoxically, even type safety reasons.

5.1 Friendly Abstract Types

Abstracting the `marshaled` type in the `CM` module is critical for type safety. Yet, `CM` is the main interface for the user to build new marshalable/marshaled types, and functions implementing the offset calculations must operate directly on the concrete type (which holds the current offset tight with the marshaled data buffer itself). This seems to be contradictory: how would the `idx`, `fst` and `snd` functions (in `CM_array` and `CM_pair` respectively) operate on the concrete type while arguments of their scoping functor are of module type `CM`?

The current solution is unfortunately to use `Obj.magic` to cast the abstract type into the concrete one. A satisfactory solution would for example allow to define "friend" relationships within modules. There are several ways to simulate priviledged access to abstract types, e.g., (1) making the `marshaled` type concrete and enclosing all functor compositions with a final "abstraction" functor, asking the user to be kind enough to follow the rule (otherwise safety is lost), or (2), providing read/write access to the offset through a pair of `get`/`set` functions. The first solution is fragile and tedious, especially since it is often needed to name every partial functor composition (to enable elementwise access). The second solution is much better yet exposes implementation details to the user, which may tempt him into unsafe hacks.

Due to a bug in MetaOCaml (conflicts for types with identical names across modules, which is apparently a feature indeed, since MetaOCaml does not intensionally support the module system), we could not yet get the second solution working, hence the temporary concretization cast with `Obj.magic`.

5.2 Compatibility with the Standard Marshaling Library

Marshaling arbitrary unstructured data is not possible with our approach, yet it is possible to operate on arbitrary values as long as they have a *statically bounded size*. This is still a strong restriction because the maximal size must be known at module instantiation time, i.e., at compilation time before any MetaOCaml execution stage takes place.

Practically, given a bound on the size of marshaled data, static size inference an offset computations are possible; it is then straightforward to embed calls to the standard marshaling library into the inductive (un)marshaling scheme. Of course, all elementwise accessing capabilities are lost beyond these calls.

The functor for marshaling any data of bounded size is called `CCM_any`. It is parameterized by a module defining the type of the data to be marshaled (of signature `sig type data end`) and by a static integer module (of signature `INT`). Interestingly, type safety is only partially lost: only data of the correct type will be marshaled through this module interface, yet the size bound will not be checked statically (it is easy to check it at run-time but it was not yet implemented).

5.3 Lowering Abstraction Penalty of a Pure MetaOCaml Implementation

A fast marshaling library requires fast array allocation and copy operations. OCaml provides reasonably fast functions on strings, and less efficient functions for arrays (due to initialization overhead and safety tests). Therefore, our first choice was to use strings, which also favored compression of marshaled data thanks to byte-addressability.

However, it quickly appeared that memory operations on small heap objects were as important as optimized functions for large memory blocks, e.g., when marshaling an array of tuples. In extreme cases, more than 80% of the time was spent in function call overhead. Copying small heap objects require short inlined and unrolled loops, and memory accesses must be word-sized and word-aligned. For the sake of simplicity, we thus moved to 32-bit integer arrays, and implemented type-unsafe array allocation, array copy (a.k.a. blit) and sub-array extraction functions.

These functions call standard OCaml functions when the data size goes over a threshold and short loops otherwise. Specializing these functions for statically known sizes is the dominant source of speedup from staging in our library. These specialized functions alleviate the need to resort to C versions, at the cost of using `Obj.magic`; all uses of these functions are embedded into internals of the implementation and have no impact on the overall type-safety of the library. In addition, great care is taken to avoid generating code expressions with cross-stage persistent values, and to avoid function calls in short loops (including `Obj.magic`). As an illustrative example, the function to copy array blocks is shown in Figure 2.

6. EXPERIMENTAL RESULTS

We conducted preliminary experiments to compare the performance of the functor-based implementation with the standard marshaling library. Remember the goal is to achieve at least the same performance while providing elementwise access to composite structures. As an indirect benefit from type safety, it is possible to enable in-place unmarshaling, removing an often undue overhead when dealing with arrays. All experiments use the native MetaOCaml compiler since the library was optimized for it.[2] The target platform is a 2.4GHz Pentium 4 (Northwood) Processor with 512MB of DDR memory. Results are presented as speedups with respect to the standard marshaling library. Some abbreviations are used: *s* and *u* means *staged* and *unstaged*, respectively; *ip* means *in place*.

[2]Using short loops harms on the bytecode version, compared to calling C function wrappers in the OCaml library.

```
let do_blit_static source_buf source_off target_buf
                   target_off size =
  if size>64 then
    .<
      (Obj.magic String.unsafe_blit)
        .~source_buf .~source_off .~target_buf
        .~target_off (4*size)
    >.
  else if size>16 then
    .<
      for i = 0 to size-1 do
          (.~target_buf).(.~target_off+i)
        <- (.~source_buf).(.~source_off+i)
      done
    >.
  else
    let emit_leaf i =
      .<    (.~target_buf).(.~target_off+i)
         <- (.~source_buf).(.~source_off+i) >.
    in let rec emit_rec i =
        if i < size-1 then
          .<
            begin
              .~(emit_leaf i);
              .~(emit_rec (i+1))
            end
          >.
        else emit_leaf (size-1)
    in emit_rec 0

let blit_static = Obj.magic do_blit_static
```

Figure 2: Specialized function to copy array blocks

Comparison with Standard Library

Figure 3 shows performance results for round-trip marshal and unmarshal of an array of integers, with 4 array sizes; performance is measured as *speedup with respect to the standard marshaling library* of our custom implementation, staged or unstaged, in place or not. Figure 4 reproduces these measurements for an array of pairs of integers. On the latter figure, notice staged (*s*) means both array and pair marshaling call staged functions; staging only at the pair level results in performance in between the unstaged and the fully staged versions.

Size	16	64	256	1024
u	1.39	**1.41**	**0.67**	**0.66**
s	**1.50**	1.11	0.66	0.65
u ip	2.60	**4.74**	**1.46**	**1.43**
s ip	**2.88**	2.09	1.45	1.42

Figure 3: Relative performance on `int array`

Size	16	64	256	1024
u	0.40	0.26	0.069	0.071
s	**0.60**	**0.40**	**0.072**	**0.075**
u ip	1.12	0.79	0.30	0.33
s ip	**1.92**	**1.32**	**0.36**	**0.37**

Figure 4: Relative performance on `(int * int) array`

These results are quite promising: our library can perform similarly to and sometimes better than the highly tuned C-implemented standard marshaling functions. Yet on larger array sizes with non-flat data structures (array of pairs), our implementation suffers dramatically from the lack of fast routine to allocate of pool of heap objects in one chunk of memory, and to a lesser extent, from the overhead of array initialization. These overheads can certainly be eliminated with additional loop optimizations (e.g., partial loop unrolling and software pipelining) for mid-size structures, and custom C functions for very large arrays.

Notice the standard functions suffer only little overhead due to run-time data introspection, compared to Java Serialization routines. This explains the limited speedups achieved through partial evaluation of offset computations and staging, compared to run-time code generation strategies for Java [1].

Evaluation of Elementwise Access Performance

The first motivation for our library is elementwise marshaling and unmarshaling from composite data structures. Our experiments on arrays of pairs of integers are fully satisfactory for this matter:

- traversing the whole array, marshaling and unmarshaling each integer individually (in each pair) degrades performance by an average factor of 2.6, almost independent on the array size (compared to the global round-trip evaluation with our implementation);
- marshaling and unmarshaling each pair individually is much faster as expected, degrading performance only by 10% to 20% when using the staged implementation, and 30% to 40% with the unstaged one.

7. THE CONTEXT OF OUR WORK

Dynamic type safety for (un)marshaling has been proposed and implemented in multiple environments. The Java serialization library encodes dynamic type information in the serialized data, as well as the AliceML pickling mechanism [7]. In the latter case, the authors of AliceML a define a clean semantical definition of marshaling for all kinds of functional or not, stateful or not, resource-bound or not data, compatible with the concurrent execution extensions of the language.

Metaphor is a very promising variant of C# which combines multi-staged evaluation, static type-checking of future stages, and a limited form of introspection [5]. This apparently contradictory combination of static type safety and reflexion is solved by extending C# types with more general generic first-order types. The authors of Metaphor demonstrate, with a simple and elegant example, that staged interpretation and type introspection can be combined to implement a type-safe code generator for (un)marshaling [5]. It seems to be the first attempt to go beyond the fully-static, module or macro-based generation of type-specific (un)marshaling functions, and to provide more safety than the dynamically typed approaches a la Java (or fully untyped as in OCaml). Whether good performance can be achieved with such a framework remains unclear, but we believe it is a strong motivation to extend MetaOCaml with limited support for reification and introspection. Whether a similar behavior can be simulated with tag elimination is not clear either, and will be investigated in further studies.

Run-time program generation for optimizing marshaling is an active research topic. Kamin et al. presented an implementation for Java [1] using the Jumbo framework [4], advocating for run-time code generation to specialize (un)marshaling functions to a given data type. In contrast to their work, our approach is to perform as much analysis as possible beforehand, generating specialized functions once and for all and benefitting from static size inference; conversely, our choice of using the module system forbids any run-time code generation and specialization, which incurs a code size overhead and requires the static knowledge of all data sizes. To overcome these limitations, we will continue investigating polytypic solutions la Metaphor [5], allowing to map data types to new marshaled types automatically without resorting to the module system; then OCamlJIT [6] would be a very natural option to reach performance levels competitive with the standard marshaling library.

We did not discuss closure and continuation marshaling. Indeed, OCaml has a limited support for that, and other languages like SISC-Scheme and Gambit-C (a Scheme compiler with C back-end) propose closure marshaling as a first class operation. Yet robust support of closures has a high overhead, due to the tracking of all live references, and due to the frequent polution with unwanted references (e.g., global variables) which may end-up require to store the whole heap of the process. When dealing with one application with persistent storage (e.g., a web server, the typical application of SISC-Scheme and Gambit-C), closure and continuations are very useful to propagate state information across the whole life time of the application (i.e., across HTTP requests). Yet when considering distributed computing and emphasizing predictability and efficiency, we believe sending marshaled pieces of code, i.e., marshaling open staged computations, is a better approach. We did not yet study the existing and ongoing support for code marshaling in MetaOCaml.

Finally, our work was inspired by successful methodological results in architecture-aware optimization of loop programs [2], as well as parallelization controlled by size inference [3].

8. OPEN QUESTIONS

We introduced a functor named `CCM_stage` to serve as a frontier between dynamic and static marshaling functions. Although this brings a lot of flexibility in defining custom specialized operations for particular data types, it introduces a typing inconsistency in the design. Indeed, composing with the `CCM_stage` functor does not impact the input and output data of (un)marshaling functions; and the intuitive behavior would be that the abstract mashalled data types for the staged and unstaged implementations would be identical. This is unfortunately not the case, due to the implementation of abstract types in OCaml: `CCM_int.marshaled` and `CCM_stage (CCM_int).marshaled` are not the same, although the use interface would be more intuitive if it was. It is unlikely that this behavior can be implemented with OCaml's module system, since the type constraint mechanism (in functor instantiations) impacts local types only.

Another open question is the following: is it possible to reproduce this work with classes and objects? We believe it is not possible, due to the lack of member types in class definitions. In general, it does not seem possible to simulate all features of modules and functors with parameterized classes.

9. CONCLUSIONS

We presented a library design for high-productivity, low-overhead marshaling of typed compound data. It allows for some level of static type checking, elementwise access, in-place marshaling as well as in-place unmarshaling, and side-effects on array and tuple elements in marshaled data. This library is fully implemented in MetaOCaml. Several optimizations allow to reduce the run-time overhead of a MetaOCaml implementation. High-level optimizations include fast offset computations based on static size inference, multi-stage evaluation to implement specialization of custom and elementwise marshaling and unmarshaling functions, static coalescing of elementwise accesses on unboxed types, as well as in-place operation. Several low-level optimizations involve type cast tricks to operate on the internals of OCaml values. Performance improvements can be observed with respect to the OCaml standard marshaling library on some data types and operations, and native code generation allows to keep the performance degradation quite reasonable for more complex data.

10. REFERENCES

[1] B. Aktemur, J. Jones, S. Kamin, and L. Clausen. Optimizing marshalling by run-time program generation. In *Generative Programming and Component Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 2005.

[2] A. Cohen, S. Donadio, M.-J. Garzaran, C. Herrmann, and D. Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 2005. In review process.

[3] C. A. Herrmann. Generating message-passing programs from abstract specifications by partial evaluation. *Parallel Processing Letters*, 15(3), Sept. 2005.

[4] S. Kamin, L. Clausen, and A. Jarvis. Jumbo: run-time code generation for java and its applications. In *ACM Conf. on Code Generation and Optimization (CGO'03)*, pages 48–56, 2003.

[5] G. Neverov and P. Roe. Metaphor: a multi-staged, object-oriented programming language. In *ACM Intl. Conf. on Generative Programming and Component Engineering*, oct 2004.

[6] B. Starynkevitch. OCamlJIT: A faster just-in-time OCaml implementation. In *1st MetaOCaml Workshop (associated with GPCE)*, Vancouver, British Columbia, Oct. 2004.

[7] G. Tack, L. Kornstaedt, and G. Smolka. Generic pickling and minimization. In *ACM Workshop on ML*, 2005.