Type-Specialized Serialization with Sharing

Martin Elsman
mael@itu.dk
IT University of Copenhagen
February 24, 2004

ABSTRACT

In this paper we present an implementation of a Standard ML combinator library for serializing and deserializing data structures. The combinator library supports serialization of cyclic data structures and sharing. It generates compact serialized values, both due to sharing, but also due to type specialization. The library is type safe in the sense that a type specialized serializer can be applied only to values of the specialized type. In the paper, we demonstrate how programmer control provided by the combinator library can lead to efficient serializers compared to how values are serialized with generic serializers supported by traditional language implementations.

Keywords

Serialization, Pickling, Type specialization, Dynamic types, Standard ML

1. INTRODUCTION

Most practical programming language systems provide means for serializing values to byte streams. In some cases, for instance for Java and C#, serialization is part of the language specification. The importance of efficient serialization techniques is partly due to its relation to remote method invocation (RMI) and distributed computing. Another use of efficient serialization techniques is for storing program state on disk for future reinvocations of the program.

For most systems, serialization and deserialization procedures are provided by the systems runtime component. In this paper, we describe a type indexed approach to serialization and deserialization, which provides the programmer with a combinator library for constructing pairs of a serializer and a deserializer for a given datatype. The approach has the following key advantages:

- Compactness due to specialization. No type information (tagging) is exported by the serializer, which leads to compact serialized data. All necessary type information for deserializing the serialized value is present in the type specialized deserializer.
- Compactness due to sharing. Serialization of two equivalent values leads to sharing in the serialized data. When the values are deserialized, the values share their representation.
- Type safety. A type specialized serializer may be applied only to values of the specialized type. Moreover, when a value is deserialized, a type checksum in the serialized data can be checked against the type checksum of the specialized deserializer. A subset of the library is truly type safe in the sense that with this subset it is not possible to construct serializers that do not behave as expected.
- Programmer control. The programmer may exploit knowledge about data invariants to obtain efficient serializers in cases where hash-consing does not perform well (e.g., for serializing many values of type `bool ref` in cases where each value is used linearly; that is, with only one pointer to it).
- No need for runtime tags. The combinator library imposes no restrictions on the representation of values. In particular, the technique supports a tag-free representation of values, as the library is written entirely in the language itself.

As traditional serialization techniques, the approach also provides support for serializing mutable and cyclic data structures.

We have already mentioned that, in general, there is a problem with serializing Standard ML references. In general, in order for deserialized values to be indistinguishable from non-serialized values, the serializer must preserve distinctness and sharing of references. Also notice that it is not possible in Standard ML to access the pointer value of a reference (indeed, a garbage collection could change the pointer). Thus, the best possible solution for computing a hash function for a reference is to compute the hash value of the content of the reference (and in the process avoid cycles). But this solution does not give distinct hash values to two distinct references to identical values, which leads to serialization algorithms with a worst case quadratic time complexity.

A partial solution to this problem requires the programmer to identify if a reference appears linearly (i.e., only once)

This paper is published as a technical report ITU-TR-43 at the IT University of Copenhagen, Glentevej 67, DK-2400 Copenhagen NV, Denmark. ISBN-87-7949-065-4.

in the serialized data. In this case, the programmer may use a particular combinator which avoids adding the value to the hash table.

1.1 Outline

In Section 2, we present the serialization library interface and show some example uses of the library combinators. In Section 3, we describe the implementation of the combinator library. In particular, we describe the use of hashing and an implementation of type dynamic in Standard ML to support sharing and cycles in deserialized values, efficiently.

In Section 4, we describe how the serialization library is used in the ML Kit compiler to serialize symbol tables containing type information and information about calling conventions, and so on. In particular, we describe the performance benefits of using the linear reference combinator in cases where it is known that there is only one pointer to the reference value.

Related work is described in Section 5. Finally, in Section 6, we conclude and describe possible future work.

2. THE SERIALIZATION LIBRARY

The programming interface to the serialization library is given in Standard ML as a structure `Pickle` with the signature `PICKLE` presented in Figure 1.

The serialization interface is based on an abstract type `'a pu`. Given a value of type τ `pu`, for some type τ, it is possible to serialize values of type τ into a stream of characters, using the function `pickle`. Similarly, it is possible to deserialize serialized values of type τ using the function `unpickle`.

The interface provides a series of *base combinators*, for serializing values of base types, such as integers, word values, characters, strings, reals, and so on. The interface also provides a series of *constructive combinators*, for constructing serializers for constructed types, such as pairs, triples, lists, and general datatypes. For example, by combining the `int`, `pair`, and `list` combinators, it is possible to construct a serializer for integer-pair lists:

```
val pu_pairs : (int * int) list pu =
  let open Pickle in list (pair (int,int))
  end
val s : string =
  Pickle.pickle pu_pairs [(2,3),(1,2),(2,3)]
```

As we shall see later, although the pair `(2,3)` appears twice in the serialized list, sharing is introduced by the serializer, which means that when the list is deserialized, the pairs `(2,3)` in the list share the same representation.

2.1 Datatypes

The combinator `enum` can be used for constructing a serializer for a datatype consisting of only nullary value constructors. Given such a datatype t with n nullary value constructors $C_0 \cdots C_{n-1}$, a serializer (of type t `pu`) may be constructed by passing to the `enum` combinator, (1) a function mapping each constructor C_i to the integer i, where $0 \leq i < n$, and (2) the list $[C_0, \cdots, C_{n-1}]$. Thus, for constructing a serializer for the datatype

```
datatype color = R | G | B
```

we can write the following:

```
signature PICKLE = sig
 (* abstract pickler/unpickler type *)
 type 'a pu
 val pickler   : 'a pu -> 'a -> string
 val unpickler : 'a pu -> string -> 'a

 (* base picklers *)
 val word   : word pu
 val int    : int pu
 val bool   : bool pu
 val string : string pu
 val char   : char pu
 val real   : real pu

 (* pickle constructors *)
 val pair   : 'a pu * 'b pu -> ('a * 'b) pu
 val triple : 'a pu * 'b pu * 'c pu
              -> ('a * 'b * 'c) pu
 val vector : 'a pu -> 'a Vector.vector pu

 (* reference picklers *)
 val refCyc : 'a -> 'a pu -> 'a ref pu
 val ref0   : 'a pu -> 'a ref pu
 val refLin : 'a pu -> 'a ref pu

 (* datatype picklers *)
 val enum   : ('a->int) * 'a list -> 'a pu
 val data   : ('a->int) * ('a pu->'a pu) list
              -> 'a pu
 val data2  : ('a->int) * ('a pu*'b pu->'a pu) list
              * ('b->int) * ('a pu*'b pu->'b pu) list
              -> 'a pu * 'b pu

 val con0   : 'a -> 'b -> 'a pu
 val con1   : ('a->'b) -> ('b->'a) -> 'a pu -> 'b pu

 (* useful predefined picklers *)
 val list   : 'a pu -> 'a list pu
 val option : 'a pu -> 'a option pu

 (* other useful combinators *)
 val conv   : ('a->'b) * ('b->'a) -> 'a pu -> 'b pu
 val share  : 'a pu -> 'a pu
 val reg    : 'a list -> 'a pu -> 'a pu
end
```

Figure 1: The `PICKLE` signature.

```
val pu_color : color Pickle.pu =
  let open Pickle
  in enum(fn R => 0 | G => 1 | B => 2, [R,G,B])
  end
```

In general, for constructing serializers for datatypes, the combinator `data` may be used, but only for datatypes that are not mutually recursive with other datatypes. The combinators `data2` and `data3` make it possible to construct serializers for mutually recursive datatypes for two and three datatypes, respectively.

Given a datatype t with n value constructors $C_0 \cdots C_{n-1}$, a serializer (of type t `pu`) may be constructed by passing to the `data` combinator, (1) a function mapping a value con-

structed using C_i to the integer i, where $0 \leq i < n$, and (2) a list of functions `[`$f_0, \cdots, f_{n-1}$`]`, where each function f_i, $0 \leq i < n$, is a serializer for the datatype for the constructor C_i, parameterized over a serializer to use for recursive instances of t. So for instance, consider the datatype `tree`, defined as follows:

```
datatype tree = L | N of tree * int * tree
```

To construct a serializer for the datatype `tree`, the `data` combinator can be applied, together with the utility functions `con0` and `con1`:

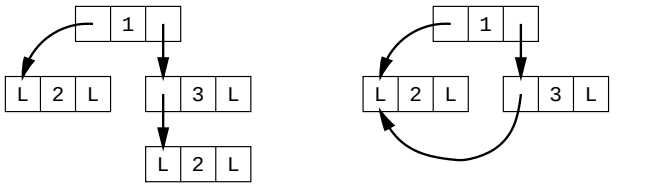
```
val pu_tree : tree Pickle.pu =
  let open Pickle
      fun pu_L pu = con0 L pu
      fun pu_N pu = con1 N (fn N a => a)
        (triple(pu,int,pu))
  in data (fn L => 0 | N _ => 1, [pu_L, pu_N])
  end
```

Consider the value

```
val t = N(N(L,2,L),1,N(N(L,2,L),3,L))
```

This value is commonly represented in memory by your favorite Standard ML compiler as shown in Figure 2(a). Serializing the value and deserializing it again results in a value that shares the common value `N(L,2,L)`, as pictured in Figure 2(b):

```
val t' =
  let open Pickle
  in (unpickle pu_tree o pickle pu_tree) t
  end
```



(a) (b)

Figure 2: Representation of a tree value in memory (a) without sharing and (b) with sharing.

2.2 References

Included in the set of constructive combinators are several combinators for constructing serializers for references. Recall that, in Standard ML, cyclic data can be constructed, only by use of references. The combinator `ref0` assumes that the reference—when serialized—does not contribute to a cycle in the value. On the other hand, the combinator `RefCyc` takes as its first argument a dummy value for the type of the reference content, which allows the deserializer to reintroduce cycles appearing in the original value. The final combinator for constructing serializers for references is the `refLin` combinator, which assumes that for each of the reference values, there is only ever one pointer to the reference. This combinator is important for efficiently serializing large values containing distinct references to identical data (i.e., boolean references). In Section 4, we shall see the importance of this final combinator for serializing references.

2.3 Other Combinators

The combinator `conv` makes it possible to construct serializers for Standard ML records, quadruples, and other datatypes that are easily converted into a datatype that is already serializable. Here is how a serializer for the type `person = {name:string,age:int}` is constructed:

```
val pu_person : {name:string,age:int} Pickle.pu =
  let open Pickle
  in conv (fn {name,age} => (name,age),
           fn (name,age) => {name=name,age=age})
          (pair(string,int))
  end
```

The combinator `reg` (read: register) takes as its first argument a list l of values of some type τ as argument and as its second argument a serializer for values of type τ. The combinator returns a new serializer for values of type τ, which avoids serialization of a value v in the list l by using the value v upon deserialization. In effect, the combinator can be used to register values that are readily available at deserialization time.

2.4 The Truly Type Safe Combinator Subset

A subset of the serialization combinators are truly type safe in the sense that it is not possible to construct serializers using these combinators for which deserialization does not result in a value equivalent to the serialized value. The truly type safe subset is listed in Figure 3.

```
signature PICKLE_TYPE_SAFE = sig
 (* abstract pickler/unpickler type *)
 type 'a pu
 val pickler   : 'a pu -> 'a -> string
 val unpickler : 'a pu -> string -> 'a

 (* base picklers *)
 val word   : word pu
 val int    : int pu
 val bool   : bool pu
 val string : string pu
 val char   : char pu
 val real   : real pu

 (* pickle constructors *)
 val pair   : 'a pu * 'b pu -> ('a * 'b) pu
 val triple : 'a pu * 'b pu * 'c pu
              -> ('a * 'b * 'c) pu
 val vector : 'a pu -> 'a Vector.vector pu

 (* reference picklers *)
 val refCyc : 'a -> 'a pu -> 'a ref pu

 (* useful predefined picklers *)
 val list   : 'a pu -> 'a list pu
 val option : 'a pu -> 'a option pu

 (* other useful combinators *)
 val share  : 'a pu -> 'a pu
 val reg    : 'a list -> 'a pu -> 'a pu
end
```

Figure 3: The truly type safe combinator subset.

3. IMPLEMENTATION

The serialization library builds on two auxiliary modules, the first of which is a module for embedding values of any type into a type `dyn` (type dynamic). This auxiliary module is presented in Section 3.1. The second auxiliary module, which we present in Section 3.2, implements input and output streams. The auxiliary modules are described in the next two sections.

3.1 Type Dynamic in Standard ML

The signature `DYN` for the auxiliary structure `Dyn` used by the implementation of the serialization library, is presented in Figure 4. An obvious implementation that matches the signature `DYN` uses exceptions to implement the type `dyn`. Less obvious is it that there exists an implementation that does not use exceptions. This implementation is presented in Figure 5. The implementation extends Filinski's basic implementation of type dynamic [10, page 106] with the addition that is provides a hash function and an equality function on values of type `dyn`.

```
signature DYN = sig
  type dyn
  val new  : ('a*'a->bool) -> ('a->word)
          -> ('a->dyn) * (dyn->'a)
  val eq   : dyn * dyn -> bool
  val hash : dyn -> word
end
```

Figure 4: Type dynamic signature in Standard ML.

```
structure Dyn :> DYN = struct
 datatype method = RESET | EQ | SET | HASH
 type dyn = method -> word
 fun new eq h =
  let val r = ref NONE
  in  ( fn x =>
         fn HASH => h x
          | RESET => (r := NONE; 0w0)
          | SET => (r := SOME x; 0w0)
          | EQ =>
           case !r of NONE => 0w0
                    | SOME y =>
             if eq(x,y) then 0w1
             else 0w0
      , fn f => ( r := NONE
                ; f SET
                ; valOf(!r)
                )
      )
  end
 fun eq (f1,f2) =
   ( f2 RESET ; f1 SET ; f2 EQ = 0w1 )
 fun hash f = f HASH
end
```

Figure 5: Exception-free implementation of type dynamic in Standard ML with equality and hash function support.

3.2 Input and Output Streams

The signature for a simple stream module used by the serializer library is presented in Figure 6.

```
signature STREAM = sig
 type IN and OUT   (* kinds *)
 type 'k stream
 val getLoc : 'k stream -> word
 val out    : char * OUT stream -> OUT stream
 val get    : IN stream -> char * IN stream
 val outw   : word * OUT stream -> OUT stream
 val getw   : IN stream -> word * IN stream
 val outcw  : word * OUT stream -> OUT stream
 val getcw  : IN stream -> word * IN stream
 val toString : OUT stream -> string
 val openOut  : unit -> OUT stream
 val openIn   : string -> IN stream
end
```

Figure 6: A library for input and output streams.

A stream is either an input stream of kind `IN` or an output stream of kind `OUT`. The function `getLoc` makes it possible to extract the location of a stream as a word. For output streams there are functions for writing characters and words and there is a function `outcw`, which compresses word values by assuming that smaller word values are written more often than larger word values. Similarly, there are functions for reading characters and words, and, dual to the `outcw` function, there is a function `getcw` for reading compressed word values. In the following, we shall assume that a stream module matching the signature in Figure 6 is bound to a structure identifier `S`.

3.3 Hash Tables

The final library used by the serialization implementation is a hash table library. A simplified version of the `POLYHASH` signature from the SML/NJ Library is presented in Figure 7. In the following code, we assume that the structure identifier `H` is bound to a hash table implementation matching the signature `POLYHASH` in Figure 7.

```
signature POLYHASH = sig
 type ('key, 'data) hash_table
 val mkTable  : ('key->int) * ('key*'key->bool)
             -> int*exn -> ('key,'data) hash_table
 val insert   : ('key,'data) hash_table
             -> 'key*'data -> unit
 val peek     : ('key,'data) hash_table
             -> 'key -> 'data option
end
```

Figure 7: Simplified `POLYHASH` signature.

3.4 Representing Serializers

The abstract type `'a pu` is defined by the following type declarations:

```
type pe = (Dyn.dyn, S.loc) H.hash_table
type upe = (S.loc, Dyn.dyn) H.hash_table
type instream = S.IN S.stream * upe
```

```
type outstream = S.OUT S.stream * pe
type 'a pu =
  {pickler   : 'a -> outstream -> outstream,
   unpickler : instream -> 'a*instream,
   hasher    : 'a -> int*word -> int*word,
   eq        : 'a*'a -> bool}
```

A *pickler environment* (of type `pe`) is a hash table mapping values of type `Dyn.dyn` to stream locations. Moreover, an *unpickler environment* (of type `upe`) is a hash table mapping stream locations to values of type `Dyn.dyn`. A value of type `outstream` is a pair of an output stream and a pickler environment. Similarly, a value of type `instream` is a pair of an input stream and an unpickler environment.

Given a type τ, a value of type τ `pu` is a record containing a pickler (a serializer) for values of type τ, an unpickler (a deserializer) for values of type τ, a hash function for values of type τ, and an equality function for values of type τ.

From a value `pu` of type τ `pu`, for some type τ, it is straightforward to implement the functions `pickle` and `unpickle` as specified in the `PICKLE` signature, by composing functionality in the stream structure `S` with the `pickler` and `unpickler` fields in the value `pu`.

3.5 Serializers for Base Types

For constructing serializers, we shall use a small set of primitive hash combinators for constructing hash functions for serializable values. To avoid that values are traversed fully by the constructed hash functions and to ensure termination of hash functions in case of cycles, the combinators count the number of hash operations performed by the hash function. The hash combinators that we shall make use of are shown in Figure 8.

```
local val Alpha = 0w65599
      val Beta = 0w19
      val maxDepth = 50
      val maxLength = 500
      val hashTag = ref 1
in fun hashAdd w (a,d) = (w + a*Alpha, d-1)
   fun hashAddSmall w (a,d) = (w + a*Beta, d-1)
   fun maybestop f ((a,d) as s) =
     if d <= 0 then s else f s
   fun newHashTag() =
     !hashTag before hashTag := !hashTag + 1
end
```

Figure 8: Hash function combinators.

We can now show how serializers are constructed for base types, exemplified by a serializer for word values:

```
val word : word pu =
  {pickler= fn w => fn (s,pe) => (S.outw(w,s),pe),
   unpickler = fn (s,upe) =>
     let val (w,s) = S.getw s
     in (w,(s,upe))
     end,
   hasher = hashAdd,
   eq = op =}
```

3.6 Product Types

We shall now see how a serializer is constructed for pairs. The `pair` combinator takes as argument a serializer for each of the components of the pair:

```
fun pair (pu1,pu2) : ('a * 'b) pu =
  let val hashTag = newHashTag()
  in {pickler = fn (v1,v2) => fn s =>
       let val s = #pickler pu1 v1 s
       in #pickler pu2 v2 s
       end,
      unpickler = fn s =>
       let val (v1,s) = #unpickler pu1 s
           val (v2,s) = #unpickler pu2 s
       in ((v1,v2),s)
       end,
      hasher = fn (v1,v2) =>
        maybestop (fn s => #hasher pu2 v2
                    (#hasher pu1 v1
                     (hashAddSmall hashTag s))),
      eq = fn ((a1,a2),(b1,b2)) =>
       #eq pu1 (a1,b1) andalso
       #eq pu2 (a2,b2)}
  end
```

Notice that the hash function is guided by the `maybestop` combinator, which returns the hash result when the hash counter has reached zero.

Combinators for serializing triples and quadruples are easily constructed using the `conv` and `pair` combinators, as in the following declaration of the triple combinator:

```
fun triple (pu1,pu2,pu3) =
  conv (fn ((pu1,pu2),pu3) => (pu1,pu2,pu3),
        fn (pu1,pu2,pu3) => ((pu1,pu2),pu3))
  (pair(pair(pu1,pu2),pu3))
```

3.7 A Sharing Combinator

Until now we have not made explicit use of stream locations and environment information when serializing and deserializing data. We shall now see how it is possible to construct a combinator `share` that leads to sharing of serialized and deserialized data. The `share` combinator is listed in Figure 9.

Notice that the `share` combinator takes any serializer as argument and generates a serializer of the same type as the argument.

When a value is serialized, it is first checked if some identical value is associated with a location l in the pickle environment. In this case, a REF-tag is written to the outstream together with a reference to the location l. If there is no value in the pickle environment identical to the value to be serialized, a DEF-tag is written to the output stream, the current location l of the output stream is recorded, the value is serialized, and an entry is added to the pickle environment mapping the value into the location l. In this way, future serialized values identical to the serialized value can share representation with the serialized value in the outstream.

Dually, when a value is deserialized by the `share` combinator, first the tag (i.e., REF or DEF) is read from the input stream. If the tag is a REF-tag, a location l is read from the input stream, and the location l is used for looking up a resulting value for the deserializer in the unpickler environment. On the other hand, if the tag is a DEF-tag, the location l of the input stream is recorded, a value v is deserialized with the argument deserializer, and finally, an entry is added to the unpickler environment mapping the

```
fun share (pu:'a pu) : 'a pu =
 let val REF = 0w0 and DEF = 0w1
     val (toDyn,fromDyn) = Dyn.new (#eq pu)
       (fn v => #1 (#hasher pu v (0w0,maxDepth)))
 in
  {pickler = fn v => fn (s,pe) =>
    let val d = toDyn v
    in case H.peek pe d of
        SOME loc => let val s = S.outcw(REF,s)
                        val s = S.outw(loc,s)
                    in (s,pe)
                    end
      | NONE =>
        let val s = S.outcw(DEF,s)
            val loc = S.getLoc s
            val res = #pickler pu v (s,pe)
        in case H.peek pe d of
            SOME _ => res
          | NONE => (H.insert pe (d,loc); res)
        end
    end,
   unpickler = fn (s,upe) =>
    let val (tag,s) = S.getcw s
    in if tag = REF then
         let val (loc,s) = S.getw s
         in case H.peek upe loc of
             SOME d => (fromDyn d, (s,upe))
           | NONE => fail "share.error"
         end
       else (* tag = DEF *)
         let val loc = S.getLoc s
             val (v,(s,upe)) = #unpickler pu (s,upe)
         in H.insert upe (loc,toDyn v); (v,(s,upe))
         end
    end,
   hasher = fn v => maybestop (#hasher pu v),
   eq = #eq pu}
 end
```

Figure 9: The `share` combinator.

location l into the value v, which is also the result of the deserialization.

One important point to notice here is that efficient inhomogeneous environments, mapping values of different types into locations, are possible only through the use of the particular `Dyn` library, which supports a hash function on values of type `dyn` and an equality function on values of type `dyn`.[1]

3.8 References and Cycles

To construct a serialization combinator in Standard ML for references, a number of challenges must be overcome. First, for any two reference values contained in some value, it can be observed (either by equality or by trivial assignment) whether or not the two reference values denote the same reference value. It is crucial that such reference invariants are not violated by serialization and deserialization. Second, for data structures that do not contain recursive closures, all cycles go through a `ref` constructor. Thus in general, to ensure termination of constructed serializers, it is necessary (and sufficient) to recognize cycles that go through `ref` constructors. The pickle environment introduced earlier is used for this purpose. Third, once a cyclic value has been serialized, it is crucial that when the value is deserialized again, the cycle in the new constructed value is reestablished (the knot must be tied).

The general serialization combinator for references is shown in Figure 10. The dummy value given as argument to the `refCyc` combinator is used for the purpose of "tying the knot" when a serialized value is deserialized. The first time a reference value is serialized, a DEF-tag is written to the current location l of the outstream. Thereafter, the pickle environment is extended to associate the reference value with the location l. Then the argument to the reference constructor is serialized. On the other hand, if it is recognized that the reference value has been serialized earlier (i.e., by finding an entry in the pickle environment mapping the reference value to a stream location l), a REF-tag is written to the outstream, followed by the location l.

For deserializing a reference value, first the location l of the input stream is obtained. Second, a reference value r is created with the argument being the dummy value that was given as argument to the `refCyc` combinator. Then the unpickle environment is extended to map the location l to the reference value r. Thereafter, a value is deserialized, which is then assigned to the reference value r. This assignment establishes the cycle and the dummy value no longer appears in the deserialized value.

As mentioned in the introduction, it is difficult to find a better hash function for references than that of using the hash function for the reference argument. Equality on references reduces to pointer equality.

The two other serialization combinators for references, `ref0` and `refLin`, are special cases of the general reference combinator `refCyc`. The `ref0` combinator assumes that no cycles appear through reference values serialized using this combinator.

The `refLin` combinator assumes that the entire value being serialized contains only one pointer to each value being serialized using this combinator (which also does not allow cycles) and that the `share` combinator is used at a higher level in the type structure, but lower than a point where there can be multiple pointers to the value. With these assumptions, the `refLin` combinator avoids the problem mentioned earlier of filling up hash table buckets in the pickle environment with distinct values having the same hash value. In general, however, it is an unpleasant task for a programmer to establish the requirements of the `refLin` combinator.

3.9 Datatypes

It turns out to be difficult in Standard ML to construct a general serialization combinator that works for any number of mutually recursive datatypes. In this section, we describe the implementation of the serialization combinator `data` from Section 2.1, which can be used for constructing a serializer and a deserializer for a single recursive datatype. It is straightforward to extend this implementation to any par-

[1] The straightforward implementation in Standard ML of type dynamic using exceptions can also be extended with a hash function and an equality function, by defining the type `dyn` to have type `{v:exn, eq:exn*exn->bool, h:exn->word}`, where `v` is the actual value packed in a locally generated exception, `eq` is an equality function returning `true` only for identical values applied to the same exception constructor, and `h` is a hash function for the packed value.

```
fun refCyc (dummy:'a) (pu:'a pu) : 'a ref pu =
  let val hashTag = newHashTag()
      val REF = 0w0 and DEF = 0w1
      fun hasher (ref v) =
        maybestop (fn p => hashAddSmall hashTag
                    (#hasher pu v p))
      val (toDyn,fromDyn) = Dyn.new (op =)
        (fn v => #1 (hasher v (0w0,maxDepth)))
  in
   {pickler = fn r as ref v => fn (s,pe) =>
      let val d = toDyn r
      in case H.peek pe d of
           SOME loc => let val s = S.outcw(REF,s)
                           val s = S.outw(loc,s)
                       in (s,pe)
                       end
         | NONE => let val s = S.outcw(DEF,s)
                       val loc = S.getLoc s
                   in H.insert pe (d,loc)
                    ; #pickler pu v (s, pe)
                   end
      end,
    unpickler = fn (s,upe) =>
      let val (tag,s) = S.getcw s
      in if tag = REF then
          let val (loc,s) = S.getw s
          in case H.peek upe loc of
               SOME d => (fromDyn d, (s, upe))
             | NONE => fail "ref.error"
          end
         else (* tag = DEF *)
          let val loc = S.getLoc s
              val r = ref dummy
              val _ = H.insert upe (loc,toDyn r)
              val (v,(s,upe)) = #unpickler pu (s,upe)
          in r := v ; (r, (s,upe))
          end
      end,
    hasher = hasher,
    eq = op =}
  end
```

Figure 10: Cycle supporting serializer for references.

ticular number of mutually recursive datatypes.[2] The implementation of the `data` serialization combinator is shown in Figure 11.

To allow for arbitrary sharing between parts of a data structure (of some datatype) and perhaps parts of another data structure (of the same datatype), the combinator makes use of the `share` combinator from Section 3.7. It is essential that the `share` combinator is not only applied to the resulting serialization combinator for the datatype, but that this sharing version of the combinator is the one that is used for recursive occurrences of the type being defined. Otherwise, it would not, for instance, be possible to obtain sharing between the tail of a list and some other list appearing in the value being serialized. Also, it would not be possible to support the sharing obtained with the `tree` value in Figure 2(b).

Thus, in the implementation, the four functions (the pickler, unpickler, equality function, and hash function) that make up the serializer are mutually recursive and a caching mechanism (the function `getPUP`) makes sure that the `share` combinator is applied only once.

[2]By exposing more of the internal workings of the combinator library in the library interface, the library could be made to support the construction of serialization combinators for arbitrary mutually recursive datatypes.

```
fun data (toInt: 'a -> int,
          fs : ('a pu -> 'a pu) list) : 'a pu =
let
 val hashTag = newHashTag()
 val res : 'a pu option ref = ref NONE
 val ps : 'a pu vector option ref = ref NONE
 fun p v (s,pe) =
  let val i = toInt v
      val s = S.outcw (Word.fromInt i, s)
  in #pickler(getPUPI i) v (s,pe)
  end
 and up (s,upe) =
  let val (w,s) = S.getcw s
  in #unpickler(getPUPI (Word.toInt w)) (s,upe)
  end
 and eq(a1:'a,a2:'a) : bool =
  let val n = toInt a1
  in n = toInt a2 andalso #eq (getPUPI n) (a1,a2)
  end
 and getPUP() =
  case !res of
    NONE =>
     let val pup = share {pickler=p,hasher=h,
                          unpickler=up,eq=eq}
     in res := SOME pup; pup
     end
  | SOME pup => pup
 and getPUPI (i:int) =
   case !ps of
    NONE =>
     let val ps0 = map (fn f => f (getPUP())) fs
         val psv = Vector.fromList ps0
     in ps := SOME psv; Vector.sub(psv,i)
     end
  | SOME psv => Vector.sub(psv,i)
 and h v = maybestop (fn p =>
   let val i = toInt v
   in hashAddSmall (Word.fromInt i)
       (hashAddSmall hashTag
        (#hasher (getPUPI i) v p))
   end)
in getPUP()
end
```

Figure 11: Single datatype serialization combinator.

4. EXPERIMENTS WITH THE ML KIT

The ML Kit [16] is a Standard ML compiler, which allows for type and compilation information to migrate across module boundaries at compile time [8, 9]. In this section, we present some experiments with serialization of type and compilation information in the ML Kit.

In the ML Kit, compilation is built around a series of *in-*

termediate languages $L_1, \cdots, L_n$, where L_1 is the Standard ML source language and L_n is Intel X86 assembler language. In the following we write P_i to denote an *intermediate program*, written in the intermediate language L_i, $1 \leq i \leq n$.

Compilation in the ML Kit is then a composition of $n-1$ translation phases, where the i'th translation phase translates program representations in the intermediate language L_i into the intermediate language L_{i+1}, $1 \leq i < n$. The translation of some intermediate program P_i into an intermediate program P_{i+1} is performed with respect to a *translation environment* E_i, which maps free identifiers in P_i into translation environment objects for the particular translation phase. Each translation phase can be specified using judgments of the form $E_i \vdash P_i \Rightarrow (N)(P_{i+1}, E_i')$, which are read "in the translation environment E_i, the intermediate program P_i is translated into the intermediate program P_{i+1} and result translation environment E_i'. The *name set* N is a set of new names (i.e., identifiers) generated during translation.

The composition of translation phases leads to a notion of *compiler bases*, ranged over by B, which are n-tuples of translation environments $(E_1, \cdots, E_n)$.

For the incremental cut-off recompilation scheme implemented in the ML Kit, it is important that compiler bases can be written to disk so as to avoid unnecessary recompilation upon change of source code.

Examples of translation phases in the ML Kit include type inference, various optimizing translations [2], elimination of polymorphic equality [7], region inference [15, 17], various region representation analyses [4], closure conversion, instruction selection, and register allocation. Many of these translation phases make use of the possibility of passing value representation information across compilation boundaries. For instance, the region inference analysis is a type-based analysis, which associates function identifiers with so called region type schemes, which provides information about in which regions arguments to the function should be stored and in which regions the result of the function is stored. Because the region inference analysis tracks the effects (represented as graphs) of calling the function, region type schemes can become large compared to the underlying ML type schemes, which also leads to large compiler bases.

4.1 Constructing the Serializers

The type structure for compiler bases, as they are implemented in the ML Kit, has a spanning tree with a high depth (higher than 8). For each concrete type τ in the type structure, a serializer is constructed with type τ `pu`. At the bottom of the type structure, serializers for various identifiers and names are constructed, whereas closer to the root of the type structure, serializers for region type schemes and region type scheme environments are constructed.

4.2 Measurements

Table 4.2 presents measurements for serializing a compiler basis in the ML Kit for parts of the Standard ML Basis Library. The table shows serialization times, deserialization times, and file sizes for three different serialization configurations. The measurements were run on a 750Mhz Pentium III Linux box with 512Mb of RAM. The first configuration implements full sharing of values (i.e., with consistent use of the `share` combinator from Section 3.7.) The second configuration disables the special treatment of programmer specified linear references by using the more general `ref0` combinator instead of the `refLin` combinator. Finally, the third configuration supports sharing only for references (which also avoids problems with cycles). The third configuration entails unsoundness of the special treatment of programmer specified linear references, which is therefore also disabled in this configuration.

Table 1: Serialization time (S-time in seconds), deserialization time (D-time in seconds), and file sizes (in kilobytes) for serializing parts of the compiler basis for the Standard ML Basis Library. Different rows in the table show measurements for different configurations of the serializer.

	S-time (s)	D-time (s)	Size (Kb)
Full sharing	17.5	3.1	406
No linear references	138	2.8	433
No sharing	125	2.3	550

When sharing is enabled, serializing the entire compiler basis takes 130 seconds and deserialization takes 19 seconds. The size of the serialized compiler basis for the Standard ML Basis Library is 2.24 megabytes.

5. RELATED WORK

There is a series of related work about serialization, in particular in the context of marshalling, the task of communicating values between distributed processes over byte streams. Many languages provide support for serializing values, including Modula-3, Java, and C#. For other languages, such as Standard ML, programmers have relied on implementation support for serialization (SML/NJ 0.93 has built-in support for serialization, for instance.)

There is a series of work concerned with dynamic typing issues for distributed programming where values of dynamic type are transmitted over a network [1, 5, 6, 14]. Filinski recognizes a direct implementation of type dynamic in ML [10, page 106], which does not make use of the folklore implementation technique that uses exceptions to implement an extensible datatype.

Recently, Leifer et al. have worked on ensuring that invariants on distributed abstract data types are not violated by checking the identity of modules implementing the operations on the abstract datatype [13].

Also related to this work is work on garbage collection algorithms for introducing sharing to save space by the use of hash-consing [3].

The Zephyr Abstract Syntax Description Language (ASDL) project [19] aims at providing a language independent data exchange format by, for a series of languages, generating type declarations and serialization code from generic datatype specifications. Whereas generated ASDL serialization code does not maintain sharing, it does avoid storing of redundant type information by employing a type specialized prefix encoding of tree values. This approach is similar to the Packed Encoding Rules (PER) of ASN.1 [18] (in contrast to the Basic Encoding Rules (BER) of ASN.1).

Independently of the present work, Kennedy has developed a very similar combinator library for serializing data structures in Standard ML. His combinator library is used in

the SML.NET compiler [12] for serializing type information to disk so as to support separate compilation. Although the approaches share many of the same ideas, Kennedy's `share` combinator is more complicated to use than ours in that it requires the programmer to provide functionality for mapping values to integers, which in principle violates abstraction principles. In our approach, hashing functionality and equality predicates are constructed inductively on the type of the serializer, which leads to efficient and easy to use combinators. Moreover, Kennedy's `fix` combinators for constructing serializers for datatypes do not support sharing of subparts of datatypes, as our datatype combinators, as illustrated in Figure 2.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented a Standard ML combinator library for serializing values to character streams and deserializing character streams into Standard ML values. The combinator library may introduce sharing in deserialized values even in cases where sharing was not present in the value that was serialized.

An obvious candidate for future work is to investigate if it is possible to use multiset discrimination [11] to distinguish references more efficiently. Although multiset discrimination of references requires runtime system support in that it requires language implementors to provide a generalization of equality on references (in terms of a discriminator), better support for distinguishing references could eliminate the need for the linear reference combinator, described in Section 3.8.

Another obvious candidate for future work is to implement a tool for generating serializers and deserializers for a given datatype, using the combinator library.

Acknowledgments

I would like to thank Henning Niss and Ken Friis Larsen for many interesting discussions about this work and Claudio Russo for letting me know (at POPL'2004) that Andrew Kennedy independently has written an almost identical serialization combinator library for Standard ML, which is used in the SML.NET compiler.

7. REFERENCES

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Andrew W. Appel and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, February 1993.

[4] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Procedings of ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183. ACM Press, January 1996.

[5] Dominic Duggan. A type-based semantics for user-defined marshalling in polymorphic languages. In *Second International Workshop on Types in Compilation (TIC'98)*, March 1998.

[6] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *Transactions on Programming Languages and Systems*, 21(1):11–45, January 1999.

[7] Martin Elsman. Polymorphic equality—no tags required. In *Second International Workshop on Types in Compilation (TIC'98)*, March 1998.

[8] Martin Elsman. *Program Modules, Separate Compilation, and Intermodule Optimisation*. PhD thesis, Department of Computer Science, University of Copenhagen, January 1999.

[9] Martin Elsman. Static interpretation of modules. In *Procedings of Fourth International Conference on Functional Programming (ICFP'99)*, pages 208–219. ACM Press, September 1999.

[10] Andrzej Filinski. *Controlling Effects*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, May 1996.

[11] Fritz Henglein. Multiset discrimination. In preparation, September 2003. Available from `http://www.plan-x.org/msd/`.

[12] Andrew Kennedy, Claudio Russo, and Nick Benton. *SML.NET 1.1 User Guide*, November 2003. Microsoft Research Ltd. Cambridge, UK.

[13] James Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *International Conference on Functional Programming (ICFP'03)*, August 2003.

[14] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4), 1993.

[15] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):734–767, July 1998. (plus 24 pages of electronic appendix).

[16] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, and Peter Sestoft. Programming with regions in the ML Kit (for version 4). Technical Report TR-2001-07, IT University of Copenhagen, October 2001.

[17] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[18] International Telecommunication Union. ASN.1 encoding rules: Specification of Packed Encoding Rules (PER). Information Technology. SERIES-X: Data Networks and Open Systems Communications. X.691, July 2002.

[19] Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra. The Zephyr abstract syntax description language. In *USENIX Conference on Domain-Specific Languages*, October 1997.