Type-Specialized Serialization with Sharing

Martin Elsman
mael@itu.dk

IT University of Copenhagen, Denmark

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 generic serializers supported by traditional language implementations.

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, yet for other languages, programmers have relied on implementation support for serialization. The importance of efficient serialization techniques is partly due to its relation to remote method invocation (RMI) and distributed computing (marshalling). Other uses of serialization support include storing of program state on disk for future reinvocations of the program.

For most systems, serialization and deserialization procedures are provided by the system's runtime component. In this paper, we expand on Kennedy's type indexed approach to serialization [9], 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 written to the byte stream 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.

- 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.

- Type safety. A type specialized serializer may be applied only to values of the specialized type. 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. Moreover, the technique can be extended so that, before a value is deserialized, a type checksum in the serialized data is checked against the type checksum of the specialized deserializer.

- 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).

Contrary to Kennedy's approach, the approach we take here also leads to automatic compactness due to sharing. That is, serialization of two equivalent values leads to sharing in the serialized data. And more importantly, when the values are deserialized, the values share their representation in program memory, which may lead to drastic memory savings during program execution. Further, our 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 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 value). 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 pointing at identical values, which leads to serialization algorithms with a worst case quadratic time complexity.

We identify a partial solution to this problem, which requires the programmer to identify if a reference appears linearly (i.e., only once) in the serialized data. In this case, the programmer may use a particular combinator which avoids the recording that the value has been visited.

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 the performance benefits of using the linear reference combinator when serializing symbol table information in the ML Kit, a Standard ML compiler. Related work is described in Section 5. Finally, in Section 6, we conclude and describe possible future work.

2 THE SERIALIZATION LIBRARY

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

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

 (* type safe combinators *)
 val word   : word pu
 val int    : int pu
 val string : string pu
 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
 val list   : 'a pu -> 'a list pu
 val option : 'a pu -> 'a option pu
 val refCyc : 'a -> 'a pu -> 'a ref pu

 (* unsafe combinators *)
 val ref0   : 'a pu -> 'a ref pu
 val refLin : 'a pu -> 'a ref pu
 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

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

FIGURE 1. The `PICKLE` signature.

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, the function `unpickle` allows for deserializing a serialized value.

The interface provides a series of *base combinators*, for serializing values such as integers, words, and strings. The interface also provides a series of *constructive combinators*, for constructing serializers for pairs, triples, lists, and general datatypes. For example, it is possible to construct a serializer for lists of integer pairs:

```
val pu_ips:(int*int)list P.pu = P.list(P.pair(P.int,P.int))
val s:string = P.pickle pu_ips [(2,3),(1,2),(2,3)]
```

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.

The first part of the serialization combinators are truly type safe in the sense that, with this subset, deserialization results in a value equivalent to the value being serialized. The combinator `conv` makes it possible to construct serializers for Standard ML records, quadruples, and other datatypes that are easily converted into an already serializable type.

2.1 Datatypes

Given an enumeration datatype t with 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 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:

```
val pu_color : color P.pu =
  P.enum(fn R => 0 | G => 1 | B => 2, [R,G,B])
```

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 combinator `data2` makes it possible to construct serializers for two mutually recursive datatypes.

Given a datatype t with 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 constructed using C_i to the integer i and (2) a list of functions `[`$f_0, \cdots, f_{n-1}$`]`, where each function f_i is a serializer for the datatype for the constructor C_i, parameterized over a serializer to use for recursive instances of t. As an example, consider the following datatype:

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

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

```
val pu_T : T P.pu = P.data (fn L => 0 | N _ => 1,
  [fn pu=>P.con0 L pu,
   fn pu=>P.con1 N (fn N a=>a) (P.triple(pu,P.int,pu))])
```

Consider the value declaration

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

The value bound to t is commonly represented in memory 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' = (P.unpickle pu_T o P.pickle pu_T) t
```

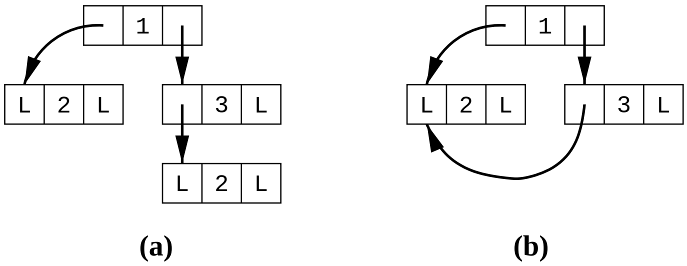

FIGURE 2. Representation of a tree value (a) without sharing and (b) with sharing.

2.2 References

In Standard ML, cyclic data can be constructed, only by use of references (not considering recursive closures). 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. As we shall see in Section 4, this combinator is important for efficiently serializing large values containing distinct references pointing at identical data (i.e., boolean references).

3 IMPLEMENTATION

Before we present the implementation of the serialization library, we first present a module Dyn for embedding values of arbitrary type into a type dyn (type dynamic) and a stream module for implementing input and output streams.

The signature DYN for the auxiliary structure Dyn is as follows:

```
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
```

Here is an implementation of this module, based on Filinski's implementation of type dynamic [6, page 106], but extended to provide a hash function and an equality function on values of type `dyn`:

```
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
```

The stream module `S` has the following signature:

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

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 is a function for writing words, `outw`, which compresses word values by assuming that smaller word values are written more often than larger ones. Dually, there is a function `getw` for reading compressed word values.

The final non-standard library used by the implementation is a hash table library. In the following, we assume a structure `H` matching a simplified version of the signature `POLYHASH` from the SML/NJ Library:

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

3.1 Representing Serializers

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

```
type pe = (Dyn.dyn, S.loc) H.ht
type upe = (S.loc, Dyn.dyn) H.ht
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 -> word*int -> word*int,
              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 for values of type τ, an unpickler 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`.

3.2 Serializers for Base Types

For constructing serializers, we shall make use of a small module `Hash` for constructing hash functions for serializable values:

```
structure Hash = struct
  val maxDepth = 50
  fun add w (a,d) = (w + a * 0w19, d - 1)
  fun maybeStop f (a,d) = if d <= 0 then (a,d) else f (a,d)
end
```

To ensure termination of hash functions in case of cycles and to avoid that values are traversed fully, the combinators count the number of hash operations performed by the hash functions.

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) =>
     case S.getw s of (w,s) => (w,(s,upe)),
   hasher = Hash.add,
   eq = op =}
```

3.3 Product Types

For constructing a pair serializer, the `pair` combinator takes as argument a serializer for each of the components of the pair:

```
fun pair (pu1 : 'a pu, pu2 : 'b pu) : ('a * 'b) pu =
  {pickler = fn (v1,v2) =>
     #pickler pu2 v2 o #pickler pu1 v1,
   unpickler = fn s =>
     case #unpickler pu1 s of (v1,s) =>
     case #unpickler pu2 s of (v2,s) => ((v1,v2),s),
   hasher = fn (v1,v2) => Hash.maybeStop
     (#hasher pu2 v2 o #hasher pu1 v1),
   eq = fn ((a1,a2),(b1,b2)) =>
     #eq pu1 (a1,b1) andalso #eq pu2 (a2,b2)}
```

Notice the use of the `Hash.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.

3.4 A Sharing Combinator

We shall now see how it is possible to make explicit use of stream locations and environment information to construct a combinator `share` that leads to sharing of serialized and deserialized data.

The `share` combinator, which is listed in Figure 3, takes any serializer as argument and generates a serializer of the same type as the argument.

For serializing a value, 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.

```
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,Hash.maxDepth)))
 in {pickler = fn v => fn (s,pe) =>
      let val d = toDyn v
      in case H.peek pe d of
          SOME loc => (S.outw(loc,S.outw(REF,s)),pe)
        | NONE => let val s = S.outw(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.getw 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 => raise Fail "impossible:share"
           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 => Hash.maybeStop (#hasher pu v),
     eq = #eq pu}
 end
```

FIGURE 3. The `share` combinator.

Dually, for deserializing a value, 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 and used for looking up the resulting value in the unpickler environment. If, on the other hand, 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 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 `Dyn` library, which supports a hash function on values of type `dyn` and an equality function on values of type `dyn`.[1]

[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

3.5 References and Cycles

To construct a serialization combinator 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 general serialization combinator for references is shown in Figure 4. 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 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 (i.e., `ref0` and `refLin`) are implemented as 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

---

`{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 REF = 0w0 and DEF = 0w1
      val (toDyn,fromDyn) = Dyn.new (op =)
        (fn ref v => #1 (#hasher pu v (0w0,Hash.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 => (S.outw(loc,S.outw(REF,s)),pe)
            | NONE => let val s = S.outw(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.getw 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 => raise Fail "impossible:ref"
             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 = fn ref v => #hasher pu v,
      eq = op =}
  end
```

FIGURE 4. Cycle supporting serializer for references.

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.

3.6 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 particular number of mutually recursive datatypes. The implementation of the `data` serialization combinator is shown in Figure 5.

```
fun data (toInt:'a->int, fs:('a pu->'a pu)list):'a pu =
let 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.outw (Word.fromInt i, s)
     in #pickler(getPUPI i) v (s,pe)
     end
    and up (s,upe) =
     case S.getw s of (w,s) =>
       #unpickler(getPUPI (Word.toInt w)) (s,upe)
    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 =
     Hash.maybeStop (fn p =>
       let val i = toInt v
       in Hash.add (Word.fromInt i) (#hasher (getPUPI i) v p)
       end)
in getPUP()
end
```

FIGURE 5. Single datatype serialization combinator.

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 3.4. 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

TABLE 1. Serialization time (S-time in seconds), deserialization time (D-time in seconds), and file sizes (in kilobytes) for serializing 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 (Mb)
Full sharing	14.2	4.0	1.88
No linear references	302	3.7	1.96
No sharing	297	3.4	4.10

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.

4 EXPERIMENTS WITH THE ML KIT

In this section, we present experiments with serializing symbol table information in the ML Kit [12], a Standard ML compiler, which allows arbitrary symbol table information to migrate across module boundaries at compile time [5].

Many of the compilation phases in the ML Kit make use of the possibility of passing compilation information across compilation boundaries, thus symbol tables tend to be large. For instance, the region inference analysis in the ML Kit [13] is a type-based analysis, which associates function identifiers with so called region type schemes, which provide information about in which regions function arguments and results are stored.

Table 4 presents measurements for serializing ML Kit symbol tables for 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 2.80 GHz Intel Pentium 4 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 3.4.) 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.

5 RELATED WORK

There is a series of related work concerned with dynamic typing issues for distributed programming where values of dynamic type are transmitted over a network [1, 3, 4, 11]. Recently, Leifer et al. have worked on ensuring that invariants on distributed abstract data types are not violated by checking the identity of operations on abstract datatypes [10].

The Zephyr Abstract Syntax Description Language (ASDL) project [15] aims at providing a language independent data exchange format by generating 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. The approach is in this respect similar to ours and to the Packed Encoding Rules (PER) of ASN.1 [14].

Independently of the present work, Kennedy has developed a similar combinator library for serializing data structures [9]. His combinator library is used in the SML.NET compiler [8] for serializing type information to disk so as to support separate compilation. Contrary to our approach, Kennedy's `share` combinator requires the programmer to provide functionality for mapping values to integers, which in principle violates abstraction principles. Moreover, Kennedy's `fix` combinators for constructing serializers for datatypes do not support sharing of subparts of datatypes, as our datatype combinators.

Also related to this work is work on garbage collection algorithms for introducing sharing to save space by the use of hash-consing [2].

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented a Standard ML combinator library for serialization and deserialization. The combinator library may introduce sharing in deserialized values even in cases where sharing was not present in the value that was serialized. The approach works with mutable and cyclic data, and is made possible through the use of an implementation of type dynamic. We further identify how a linear combinator for references may lead to efficient serializers that cannot cleanly be made available using generic approaches to serialization.

A possibility for future work is to investigate if it is possible to use a variant of multiset discrimination [7] for eliminating the need for the linear reference combinator of Section 3.5. Another possibility 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.

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 and Marcelo J. R. Gonçalves. Hash-consing garbage collection. Technical Report CS-TR-412-93, Princeton University, February 1993.

[3] 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.

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

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

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

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

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

[9] Andrew J. Kennedy. Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, November 2004. Functional Pearl.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.