JFP **14** (6): 727–739, 2004. © 2004 Cambridge University Press
DOI: 10.1017/S0956796804005209 Printed in the United Kingdom

FUNCTIONAL PEARL

Pickler combinators

ANDREW J. KENNEDY
Microsoft Research, 7 J J Thomson Avenue, Cambridge CB3 0FB, UK
(*e-mail:* `akenn@microsoft.com`)

Abstract

The tedium of writing pickling and unpickling functions by hand is relieved using a combinator library similar in spirit to the well-known parser combinators. Picklers for primitive types are combined to support tupling, alternation, recursion, and structure sharing. Code is presented in Haskell; an alternative implementation in ML is discussed.

1 Introduction

Programs frequently need to convert data from an internal representation (such as a Haskell or ML datatype) into a persistent, portable format (typically a stream of bytes) suitable for storing in a file system or transmitting across a network. This process is called *pickling* (or *marshalling*, or *serializing*) and the corresponding process of transforming back into the internal representation is called *unpickling*.

Writing picklers by hand is a tedious and error-prone business. It's easy to make mistakes, such as mapping different values to the same pickled representation, or covering only part of the domain of values, or unpickling components of a data structure in the wrong order with respect to the pickled format.

One way of avoiding these problems is to build pickling support into the programming language's run-time system (Sun Microsystems, 2002; Leroy, 2003). The main drawback of this approach is the lack of programmer control over how values get pickled. Pickling may be version-brittle, dependent on a particular version of the compiler or library and on the concrete implementation of abstract data types such as sets. Opportunities for compact pickling of shared structure will be missed: run-time pickling of heap values will pick up on "accidental" sharing evident in the heap, but will ignore sharing implied by a programmer-specified equivalence of values.

In this paper we use functional programming techniques to build picklers and unpicklers by composing primitives (for base types such as `Int` and `String`) using combinators (for constructed types such as pairs and lists). The consistency of pickling with unpickling is ensured by tying the two together in a pickler/unpickler pair. The resulting picklers are mostly correct by construction, with additional proof obligations generated by some combinators.

```
pickle       :: PU a -> a -> String
unpickle     :: PU a -> String -> a

unit         :: PU ()
bool         :: PU Bool
char         :: PU Char
string       :: PU String
nat          :: PU Int
zeroTo       :: Int -> PU Int

pair         :: PU a -> PU b -> PU (a,b)
triple       :: PU a -> PU b -> PU c -> PU (a,b,c)
quad         :: PU a -> PU b -> PU c -> PU d -> PU (a,b,c,d)
pMaybe       :: PU a -> PU (Maybe a)
pEither      :: PU a -> PU b -> PU (Either a b)
list         :: PU a -> PU [a]
wrap         :: (a->b, b->a) -> PU a -> PU b
alt          :: (a -> Int) -> [PU a] -> PU a
```

Fig. 1. The pickler interface.

Custom pickling of abstract data types is easily performed through wrapper combinators, and we extend the core library with support for representation of shared structure. Building particular picklers from a core set of primitives and combinators also makes it easy to change the implementation, if, for example, better compression is required at some performance cost.

The pickler library is implemented in Haskell using just its core features of parameterized datatypes and polymorphic, higher-order functions. Porting the code to ML raises some issues which are discussed in section 5.

This pearl was practically motivated: an SML version of the pickler library is used inside the SML.NET compiler (Benton *et al.*, 2004). A variety of data types are pickled, including source dependency information, Standard ML type environments, and types and terms for the typed intermediate language which serves as object code for the compiler. The combinatory approach has proved to be very effective.

2 Using picklers

Figure 1 presents Haskell type signatures for the pickler interface. The type `PU a` encapsulates both pickling and unpickling actions in a single value: we refer to values of type `PU a` as "picklers for `a`". The functions `pickle` and `unpickle` respectively use a pickler of type `PU a` to pickle or unpickle a value of type `a`, using strings (lists of characters) as the pickled format.

First we specify picklers for built-in types: unit, booleans, characters, strings, non-negative integers (`nat`), and integers between 0 and *n* inclusive (`zeroTo` *n*).

Next we have pickler constructors for tuple types (`pair`, `triple` and `quad`), optional values (`pMaybe`)[1], binary alternation (`pEither`), and lists (`list`).

[1] We use `pMaybe` and `pEither` because functions `maybe` and `either` already exist in the Standard Prelude.

Finally there are a couple of general combinators: `wrap` for pre- and post-composing functions with a pickler, and `alt` for using different picklers on disjoint subsets of a type.

Let's look at some examples. First, consider a browser application incorporating bookmarks that pair descriptions with URL's. A URL consists of a protocol, a host, an optional port number, and a file name. Here are some suitable type definitions:

```
type URL = (String, String, Maybe Int, String)
type Bookmark = (String, URL)
type Bookmarks = [Bookmark]
```

Picklers for these simply follow the structure of the types:

```
url :: PU URL
url = quad string string (pMaybe nat) string

bookmark :: PU Bookmark
bookmark = pair string url

bookmarks :: PU Bookmarks
bookmarks = list bookmark
```

In a real program we're more likely to use a record datatype for URLs. Then we can apply the `wrap` combinator to map back and forth between values of this datatype and quadruples:

```
data URL = URL {protocol::String, host::String, port::Maybe Int, file::String}
url = wrap (\ (pr,h,po,f) -> URL {protocol=pr, host=h, port=po, file=f},
            \ URL {protocol=pr,host=h,port=po,file=f} -> (pr,h,po,f))
           (quad string string (pMaybe nat) string)
```

We might prefer a hierarchical folder structure for bookmarks:

```
data Bookmark = Link (String, URL) | Folder (String, Bookmarks)
```

Here we must address two aspects of datatypes: alternation and recursion. Recursion is handled implicitly – we simply use a pickler inside its own definition. (For call-by-value languages such as ML we instead use an explicit `fix` operator; see later). Alternation is handled using `alt`, as shown below:

```
bookmark :: PU Bookmark
bookmark = alt tag [wrap (Link,   \(Link   a) -> a) (pair string url),
                    wrap (Folder, \(Folder a) -> a) (pair string bookmarks)]
           where tag (Link _) = 0; tag (Folder _) = 1
```

The `alt` combinator takes two arguments: a tagging function that partitions the type to be pickled into n disjoint subsets, and a list of n picklers, one for each subset. For datatypes, as here, the tagging function simply identifies the constructor.

Here is another example: a pickler for terms in the untyped lambda calculus.

```
data Lambda = Var String | Lam (String, Lambda) | App (Lambda, Lambda)

lambda = alt tag [ wrap (Var, \(Var x) -> x) string,
                   wrap (Lam, \(Lam x) -> x) (pair string lambda),
                   wrap (App, \(App x) -> x) (pair lambda lambda) ]
         where tag (Var _) = 0; tag (Lam _) = 1; tag (App _) = 2
```

An alternative to `alt` is to define maps to and from a "sum-of-products" type built from tuples and the `Either` type, and then to define a pickler that follows the structure of this type using tuple picklers and `pEither`. The picklerfor lambda terms

```
module CorePickle ( PU, pickle, unpickle, lift, sequ, base, belowBase ) where

type St = [Char]
data PU a = PU { appP :: (a,St) -> St,
                 appU :: St -> (a,St) }

pickle :: PU a -> a -> String
pickle p value = appP p (value, [])

unpickle :: PU a -> String -> a
unpickle p stream = fst (appU p stream)

base :: Int
base = 256

belowBase :: PU Int
belowBase = PU (\ (n,s) -> toEnum n : s)
               (\ (c:s) -> (fromEnum c, s))

lift :: a -> PU a
lift x = PU snd (\s -> (x,s))

sequ :: (b->a) -> PU a -> (a -> PU b) -> PU b
sequ f pa k = PU (\ (b,s) -> let a = f b
                                 pb = k a
                             in appP pa (a, appP pb (b,s)))
                 (\ s     -> let (a,s') = appU pa s
                                 pb = k a
                             in appU pb s')
```

Fig. 2. The core pickler implementation.

then becomes

```
lambda :: PU Lambda
lambda = wrap (sumlam,lamsum)
           (pEither string (pEither (pair string lambda) (pair lambda lambda)))
         where
           lamsum (Var x) = Left x
           lamsum (Lam x) = Right (Left x)
           lamsum (App x) = Right (Right x)
           sumlam (Left x)          = Var x
           sumlam (Right (Left  x)) = Lam x
           sumlam (Right (Right x)) = App x
```

3 Implementing picklers

Figure 2 presents the core of a pickler implementation, defining types and a very small number of functions from which all other picklers can be derived. The pickler type `PU a` is declared to be a pair consisting of a pickling action (labelled `appP`) and unpickling action (labelled `appU`). The pickling action is a function which transforms state and consumes a value of type `a`; conversely, an unpickling action is a function which transforms state and produces a value of type `a`. In both cases the accumulated state `St` is a list of bytes, generated so far (during pickling) or yet to be processed (during unpickling).

The `pickle` function simply applies a pickler to a value, with the empty list as the initial state. The `unpickle` function does the converse, feeding a list of bytes to the unpickler, and returning the resulting data. Any dangling bytes are ignored; a more robust implementation could signal `error`.

Now to the picklers themselves. The basic pickler `belowBase` pickles an integer i in the range $0 \leqslant i <$ `base`. We have chosen bytes as our unit of pickling, so we define `base` to be 256. It is easy to change the implementation to use bit-streams instead of byte-streams and thereby achieve better compression.

The `lift` combinator produces a no-op pickler for a particular value. It has a trivial definition, leaving the state unchanged, producing the fixed value when unpickling, and ignoring its input when pickling. (A more robust implementation would compare the input against the expected value and assert on failure).

Finally we define the `sequ` combinator, used for sequential composition of picklers. It is more general than `pair` in that it supports sequential dependencies in the pickled format: the encoding of a value of type `a` pickled using `pa` precedes the encoding of a value of type `b` whose encoding depends on the first value, as given by the parameterized pickler `k`. When pickling, the value of type `a` is obtained by applying the projection function `f`. Notice how `pb` is applied before `pa`: this ensures that bytes end up in the list in the correct order for unpickling. If instead `pa` was applied first when pickling then the `pickle` function would need to reverse the list after applying `appP`.

Readers familiar with monadic programming in Haskell will have noticed that `lift` and `sequ` bear a striking resemblance to the `return` and `>>=` combinators in the `Monad` class (also known as *unit* and *bind*). This is no coincidence: considering just their unpickling behaviour, `PU`, `lift` and `sequ` do make a monad.

Figure 3 completes the implementation, building all remaining combinators from Figure 1 using the primitives just described.

The combinators `pair`, `triple` and `quad` use `sequ` and `lift` to encode the components of a tuple in sequence. The `wrap` combinator pre- and post-composes a pickler for `a` with functions of type `a->b` and `b->a` in order to obtain a pickler for `b`. It is defined very concisely using `sequ` and `lift`.

The `zeroTo` n pickler encodes a value between 0 and n in as few bytes as possible, as determined by n. For example, `zeroTo 65535` encodes using two bytes, most-significant first. Picklers for `bool` and `char` are built from `zeroTo` using `wrap`.

In contrast with the `zeroTo` combinator, the `nat` pickler assumes that small integers are the common case, encoding $n < 128 =$ `base/2` as a single byte n, and encoding $n \geqslant 128$ as the byte $128 + n \bmod 128$ followed by $\lfloor n/128 \rfloor - 1$ encoded through a recursive use of `nat`. Signed integers can be encoded in a similar fashion.

Lists of known length are pickled by `fixedList` as a simple sequence of values. The general `list` pickler first pickles the length using `nat` and then pickles the values themselves using `fixedList`. As the Haskell `String` type is just a synonym for `[Char]`, its pickler is just `list char`.

The alternation combinator `alt` takes a tagging function and a list of picklers, an element of which is determined by the result of applying the tagging function (when pickling) or by the encoded tag (when unpickling). Picklers for `Maybe` and `Either` type constructors follow easily.

```
module Pickle (PU, pickle, unpickle, unit, char, bool, string, nat, zeroTo,
               wrap, alt, pair, triple, quad, pMaybe, pEither, list) where
import CorePickle; import Maybe

pair :: PU a -> PU b -> PU (a,b)
pair pa pb = sequ fst pa (\ a ->
             sequ snd pb (\ b ->
             lift (a,b)))

triple :: PU a -> PU b -> PU c -> PU (a,b,c)
triple pa pb pc = sequ (\ (x,y,z) -> x) pa (\a ->
                  sequ (\ (x,y,z) -> y) pb (\b ->
                  sequ (\ (x,y,z) -> z) pc (\c ->
                  lift (a,b,c))))

quad :: PU a -> PU b -> PU c -> PU d -> PU (a,b,c,d)
quad pa pb pc pd = sequ (\ (w,x,y,z) -> w) pa (\a ->
                   sequ (\ (w,x,y,z) -> x) pb (\b ->
                   sequ (\ (w,x,y,z) -> y) pc (\c ->
                   sequ (\ (w,x,y,z) -> z) pd (\d ->
                   lift (a,b,c,d)))))

wrap :: (a->b, b->a) -> PU a -> PU b
wrap (i,j) pa = sequ j pa (lift . i)

zeroTo :: Int -> PU Int
zeroTo 0 = lift 0
zeroTo n = wrap (\ (hi,lo) -> hi * base + lo, (`divMod` base))
                (pair (zeroTo (n `div` base)) belowBase)

unit :: PU ()
unit = lift ()

char :: PU Char
char = wrap (toEnum, fromEnum) (zeroTo 255)

bool :: PU Bool
bool = wrap (toEnum, fromEnum) (zeroTo 1)

nat :: PU Int
nat = sequ (\x -> if x < half then x else half + x `mod` half)
           belowBase
           (\lo -> if lo < half then lift lo
                                else wrap (\hi->hi*half+lo, \n->n `div` half - 1) nat)
      where half = base `div` 2

fixedList :: PU a -> Int -> PU [a]
fixedList pa 0 = lift []
fixedList pa n = wrap (\(a,b) -> a:b, \(a:b) -> (a,b)) (pair pa (fixedList pa (n-1)))

list :: PU a -> PU [a]
list = sequ length nat . fixedList

string :: PU String
string = list char

alt :: (a -> Int) -> [PU a] -> PU a
alt tag ps = sequ tag (zeroTo (length ps-1)) (ps !!)

pMaybe :: PU a -> PU (Maybe a)
pMaybe pa = alt tag [lift Nothing, wrap (Just, fromJust) pa]
            where tag Nothing = 0; tag (Just x) = 1

pEither :: PU a -> PU b -> PU (Either a b)
pEither pa pb = alt tag [wrap (Left, fromLeft) pa, wrap (Right, fromRight) pb]
                where tag (Left _) = 0; tag (Right _) = 1
                      fromLeft (Left a) = a; fromRight (Right b) = b
```

Fig. 3. Completed pickler implementation.

```
01                                       [
00                                        Link(
06 41 6e 64 72 65 77                      "Andrew",
04 68 74 74 70                            URL { protocol = "http",
16 72 65 73 65 61 72 63 68 2e
6d 69 63 72 6f 73 6f 66 74 2e 63 6f 6d          host = "research.microsoft.com",
00                                              port = Nothing,
0b 75 73 65 72 73 2f 61 6b 65 6e 6e             file = "users/akenn" })]
```

Fig. 4. Example of pickling for bookmark lists.

Figure 4 presents a value of type `Bookmarks`, pickled using the above implementation.

4 Structure sharing

The picklers constructed so far use space proportional to the size of the input when expressed as a *tree*. They take no account of sharing of structure in the data, either implicit but non-observable (because the runtime heap representation is a graph), or, implicit but observable (because there is a programmer-defined equality between values), or, in the case of an impure language like ML, explicit and directly observable (using `ref`). At the very least, pickled formats usually have some kind of symbol table mechanism to ensure that strings occur once only.

We would like to share arbitrary structures, for example encoding the definition of `k` just once when pickling the value `kki` of type `Lambda` shown below:

```
x = Var "x"
i = Lam("x", x)
k = Lam("x", Lam("y", x))
kki = App(k, App(k, i))
```

We can encode sharing in the following way. Suppose that some data D pickles to a byte sequence P. The first occurrence of D is pickled as $\mathit{def}(P)$ and subsequent occurrences are pickled as $\mathit{ref}(i)$ if D was the i'th definition of that type to be pickled in some specified order. For $\mathit{def}(P)$ we use a zero value followed by P, and for $\mathit{ref}(i)$ we use our existing `zeroTo` n pickler on $1 \leqslant i \leqslant n$, where n is the number of def occurrences encoded so far. The technique is reminiscent of Lempel-Ziv text compression (Bell *et al.*, 1990), which utilises the same 'on-the-fly' dictionary construction.

The following function implements this encoding for a fixed dictionary `dict`, transforming a dictionary-unaware pickler into a dictionary-aware pickler:

```
tokenize :: Eq a => [a] -> PU a -> PU a
tokenize dict p = sequ (\x -> case List.elemIndex x dict of
                                Just i -> n-i ; Nothing -> 0)
                       (zeroTo n)
                       (\i -> if i==0 then p else lift (dict !! (n-i)))
                  where n = length dict
```

```
module SCorePickle (PU, pickle, unpickle, lift, sequ, base, belowBase, useState) where

type St s = ([Char], s)
data PU a p = PU { appP :: (a,St p) -> St p,
                   appU :: St p -> (a,St p) }

pickle :: PU a s -> s -> a -> String
pickle p s value = fst (appP p (value, ([],s)))

unpickle :: PU a s -> s -> String -> a
unpickle p s cs = fst (appU p (cs, s))

base :: Int
base = 256

belowBase :: PU Int p
belowBase = PU (\ (n,(cs,s)) -> (toEnum n : cs,s))
               (\ (c:cs,s) -> (fromEnum c, (cs,s)))

lift :: a -> PU a s
lift x = PU snd (\s -> (x,s))

sequ :: (b->a) -> PU a s -> (a -> PU b s) -> PU b s
sequ f pa k = PU (\ (b,(cs,s)) -> let a = f b
                                      pb = k a
                                      (cs'',s'') = appP pb (b, (cs,s'))
                                      (cs',s') = appP pa (a, (cs'',s))
                                  in (cs',s''))
                 (\ s     -> let (a,s') = appU pa s
                             in appU (k a) s')

useState :: (a -> s -> s) -> (s -> PU a s) -> PU a s
useState update spa =
      PU (\ (x,(cs,s)) -> let (cs',s') = appP (spa s) (x,(cs,s)) in (cs',update x s'))
         (\ (cs,s) -> let (x,(cs',s')) = appU (spa s) (cs,s) in (x,(cs',update x s'))
```

Fig. 5. Core pickler implementation with structure sharing.

For homogeneous lists of values, the dictionary can be constructed on-the-fly simply by threading it through a recursive call:

```
add :: Eq a => a -> [a] -> [a]
add x d = if elem x d then d else x:d

memoFixedList :: Eq a => [a] -> PU a -> Int -> PU [a]
memoFixedList dict pa 0 = lift []
memoFixedList dict pa n = sequ head (tokenize dict pa) (\x ->
                          sequ tail (memoFixedList (add x dict) pa (n-1))
                          (\xs -> lift (x:xs)))

memoList :: [a] -> PU a -> PU [a]
memoList dict = sequ length nat . memoFixedList dict
```

Here the function `memoList` takes an initial value for the dictionary state (typically `[]`) and a pickler for a, and returns a pickler for `[a]` that extends the dictionary as it pickles or unpickles.

With heterogeneous structures such as the `Lambda` type defined above, it becomes much harder to thread the state explicitly. Instead, we can adapt the core pickler combinators to thread the state implicitly (Figure 5). Picklers are now parameterized

```
module SPickle (PU, pickle, unpickle, unit, char, bool, string, nat, zeroTo,
                wrap, sequ, pair, triple, quad, pMaybe, pEither, list, share) where

import SCorePickle; import Maybe; import List
...

add :: Eq a => a -> [a] -> [a]
add x d = if elem x d then d else x:d

tokenize :: Eq a => [a] -> PU a s -> PU a s
tokenize dict p = sequ (\x -> case List.elemIndex x dict of
                                Just i -> n-i; Nothing -> 0)
                       (zeroTo n)
                       (\i -> if i==0 then p else lift (dict !! (n-i)))
                  where n = length dict

share :: Eq a => PU a [a] -> PU a [a]
share p = useState add (\dict -> tokenize dict p)
```

Fig. 6. Completed pickler implementation with structure sharing.

on the type `a` of values being pickled and the type `s` of state used for the dictionary. The `pickle` and `unpickle` functions take an additional parameter for the initial state, discarding the final state on completion. The `belowBase` and `lift` combinators simply plumb the state through unchanged. The plumbing in `sequ` is more subtle: during pickling the dictionary state must be threaded according to the sequencing required by `sequ`, passing it through pickler `pa` and then through pickler `pb`, but the list of bytes must be threaded in the opposite direction, because bytes produced by `pa` are prepended to a list which already contains bytes produced by `pb`. Fortunately laziness supports this style of *circular programming* (Bird, 1984).

The `useState` combinator provides access to the state. It takes two parameters: `update`, which provides a means of updating the state, and `spa`, which is a state-parameterized pickler for `a`. The combinator returns a new pickler in which `spa` is first applied to the internal state value, the pickling or unpickling action is then applied, and finally the state is updated with the pickled or unpickled value.

Figure 6 presents the remainder of the implementation. We omit the definitions for most of the combinators as they are identical to those of Figure 3 except that every use of `PU` in type signatures takes an additional state type parameter. The new combinator `share` makes use of `useState` to provide an implementation of sharing using a list for the dictionary and the `tokenize` function that we saw earlier. A more efficient implementation would, for instance, use some kind of balanced tree data structure.

We can then apply the `share` combinator to pickling of lambda terms:

```
slambda = share (alt tag [ wrap (Var, \(Var x) -> x) string,
                           wrap (Lam, \(Lam x) -> x) (pair string slambda),
                           wrap (App, \(App x) -> x) (pair slambda slambda) ] )
```

Figure 7 presents an application of it to `kki`. The superscripted figures represent the indices that the pickler generates for subterms, allocated in depth-first order. Notice how the two occurrences of terms `k` and `x` have been shared; also note that

```
                                 02            $^{6}$App(
                                 01 01 78        $^{3}$Lam("x",
x = Var "x"                      01 01 79          $^{2}$Lam("y",
i = Lam("x", x)                  00 01 78            $^{1}$Var "x")),
k = Lam("x", Lam("y", x))        00 02           $^{5}$App(
kki = App(k, App(k, i))          03                k,
                                 00 01 01 78       $^{4}$Lam("x",
                                 01                  x)))
```

Fig. 7. Sharing example (superscripts represent dictionary indices).

terms pickled under an empty dictionary have no preceding zero byte because the pickler `zeroTo 0` used to encode the zero is a no-op.

It is interesting to note that pickling followed by unpickling – for example, `unpickle slambda [] . pickle slambda []` – acts as a compressor on values, maximizing sharing in the heap representation. Of course, this sharing is not observable to the programmer.

Sometimes it is useful to maintain separate symbol tables for separately-shared structure. This can be done using tuples of lists for `p` in `PU a p`. For example, we can write variants of `share` that use the first or second component of a pair of states:

```
shareFst :: Eq a => PU a ([a],s) -> PU a ([a],s)
shareFst p = useState (\x -> \(s1,s2) -> (add x s1, s2))
                      (\(s1,s2) -> tokenize s1 p)
shareSnd :: Eq a => PU a (s,[a]) -> PU a (s,[a])
shareSnd p = useState (\x -> \(s1,s2) -> (s1, add x s2))
                      (\(s1,s2) -> tokenize s2 p)
```

These combinators can then be used to share both variable names and lambda terms in the type `Lambda`:

```
lambda = shareFst (
         alt tag [ wrap (Var, \(Var x) -> x) var,
                   wrap (Lam, \(Lam x) -> x) (pair var lambda),
                   wrap (App, \(App x) -> x) (pair lambda lambda) ])
         where tag (Var _) = 0; tag (Lam _) = 1; tag (App _) = 2
               var = shareSnd string
```

5 Discussion

Pickler combinators were inspired very much by parser combinators (Wadler, 1985; Hutton & Meijer, 1998), which encapsulate parsers as functions from streams to values and provide combinators similar in spirit to those discussed here. The essential new ingredient of pickler combinators is the tying together of the pickling and unpickling actions in a single value.

Parser combinators also work well in call-by-value functional languages such as ML (Paulson, 1996). However, they suffer from a couple of wrinkles which re-occur in the ML implementation of picklers.

First, it is not possible to define values recursively in ML, e.g. the following is illegal:

```
val rec bookmark =
    alt tag
    [ wrap (Link,   fn Link   x => x) (pair string url),
      wrap (Folder, fn Folder x => x) (pair string (list bookmark))]
```

The problem is that recursive definition is only valid for syntactic functions. Here we have a value with abstract type `'a PU`. This problem is overcome in ML implementations of parser combinators (Paulson, 1996) by exposing the concrete function type of parsers, and then abstracting on arguments. So instead of writing a parser for integers sequences as

```
val rec intseq = int || int -- $"," -- intseq
```

one writes

```
fun intseq s = (int || int -- $"," -- intseq) s
```

We can't apply this trick because the concrete type for `'a PU` is a *pair* of functions. Instead, it is necessary to be explicit about recursion, using a fixpoint operator whose type is `('a PU -> 'a PU) -> 'a PU`. This is somewhat cumbersome, especially with mutual recursion, for a family of fixpoint combinators `fix_`n are required, where n is the number of functions defined by mutual recursion.

The second problem is ML's "polymorphic value restriction", which restricts polymorphic typing to syntactic values. This is particularly troublesome in the implementation of state-parameterized picklers (Figure 5), in which every combinator or primitive pickler is polymorphic in the state. For example, the ML version of `char` might be written

```
val char = wrap (Char.chr, Char.ord) (zeroTo 255)
```

but `char` cannot be assigned a polymorphic type because its right-hand-side is not a syntactic value.

In the implementation of structure sharing we made essential use of laziness in order to thread the dictionary state in the opposite direction to the accumulated list of bytes (function `sequ`). An ML version cannot do this: instead, both dictionary and bytes are threaded in the same direction, with bytes produced by pickler `pa` prepended first, then bytes produced by pickler `pb` prepended. The `pickle` function must then reverse the list in order to produce a format ready for unpickling.

The representation we use for picklers of type `PU a` can be characterized as an *embedding-projection* pair (p, u) where p is an embedding from `a` into a 'universal' type `String`, and u is a projection out of the universal type into `a`. To be a true projection-embedding it would satisfy the following 'round-trip' properties:

$$u \circ p = \mathtt{id} \qquad (1)$$

$$p \circ u \sqsubseteq \mathtt{id} \qquad (2)$$

where `id` is the identity function on the appropriate domain, and $\sqsubseteq$ denotes the usual definedness ordering on partial functions. (Strictly speaking, given a pickler `pa`, it is p = `pickle pa` and u = `unpickle pa` which have this property). More concretely, (1) says "pickling followed by unpickling generates the original value", and (2) says "successful (i.e. exhaustive and terminating) unpickling followed by pickling produces the same list of bytes". Note that (1) is valid only if values pickled by combinators such as `lift`, `zeroTo` and `nat` are in the intended domain; also observe that (2) is broken by structure sharing, as the pickler could produce a string that has different sharing from the one that was unpickled.

Combinators over embedding-projection pairs have been studied in the context of embedding interpreters for little languages into statically-typed functional languages (Benton, 2004; Ramsey, 2003); indeed some of the combinators are the same as those defined here.

Pickling has been studied as an application of *generic programming* (Morrisett & Harper, 1995; Jansson & Jeuring, 2002; Hinze, 2002), in which pickling and unpickling functions are defined by induction on the structure of types. Using a language such as as Generic Haskell (Clarke *et al.*, 2001), we can extend our combinator library to provide default picklers for all types, but leaving the programmer the option of custom pickling where more control is required.

Following submission of the final version of this article, Martin Elsman brought to the author's attention a combinator library (Elsman, 2004) somewhat similar to the one described here. Elsman's library is for Standard ML, and takes a slightly different approach to structure sharing, maintaining a single dictionary for all shared values. Values of different types are stored in the dictionary using an encoding of dynamic types. The library also supports the pickling of values containing ML references, possibly containing cycles.

Acknowledgements

Thanks are due to Nick Benton, Simon Peyton Jones, and Claudio Russo for their helpful comments on this paper and its previous incarnation as an SML implementation. I would also like to thank Ralf Hinze and the anonymous referees for their constructive comments.

References

Bell, T. C., Cleary, J. G. and Witten, I. H. (1990) *Text Compression*. Prentice Hall.

Benton, P. N. (2004) Embedded interpreters. *J. Funct. Program.* To appear.

Benton, P. N., Kennedy, A. J. and Russo, C. V. (2004) Adventures in Interoperability: The SML.NET Experience. *Proceedings of 6th ACM – SIGPLAN International Conference on Principles and Practice of Declarative Programming*, (to appear).

Bird, R. S. (1984) Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, **21**, 239–250.

Clarke, D., Hinze, R., Jeuring, J., Löh, A. and de Wit, J. (2001) *Generic Haskell*. See `http://www.generic-haskell.org/`.

Elsman, M. (2004) Type-Specialized Serialization with Sharing. IT Universty of Copenhagen Technical Report Series, TR-2004-43.

Hinze, R. (2002) Polytypic values possess polykinded types. *Sci. Comput. Program.* **43**, 129–159.

Hutton, G. and Meijer, E. (1998) Monadic parsing in Haskell. *J. Func. Program.* **8**(4), 437–444.

Jansson, P. and Jeuring, J. (2002) Polytypic data conversion programs. *Sci. Comput. Program.* **43**(1), 35–75.

Leroy, X. (2003) *The Objective Caml System.* `http://caml.inria.fr`.

Morrisett, G. and Haroer, R. (1995) Compiling polymorphism using intensional type analysis. *ACM Symposium on Principles of Programming Languages*, pp. 130–141.

Paulson, L. C. (1996) *ML for the Working Programmer, 2nd ed.* Cambridge University Press.

Ramsey, N. (2003) Embedding an interpreted language using higher-order functions and type. *ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators.*

Sun Microsystems (2002) *Java Object Serialization Specification.* `http://java.sun.com/j2se/1.4.1/docs/guide.serialization/`.

Wadler, P. (1985) How to replace failure by a list of successes. *Proceedings of Functional Programming and Computer Architecture: LNCS 201.* Springer-Verlag.