Pickler Combinators – Explained

Benedikt Grundmann
benedikt-grundmann@web.de

[1] Software Engineering Chair (Prof. Zeller)
Saarland University
[2] Programming Systems Lab (Prof. Smolka)
Saarland University

Abstract. This paper summarizes the paper "Pickler Combinators" by Andrew J. Kennedy. Kennedy presents an purely library based approach to pickling similar in spirit to the well-known parser combinators. This approach to pickling is also compared to builtin pickling services as presented in the paper "Generic Pickling and Minimization" by Guido Tack et al.

1 Motivation

It is frequently necessary to externalize data in order to store it on disk or transmit it over the network. This process is also known as serialization or pickling. The reverse process is called deserialization or unpickling.

As long as the data is atomic such as a number or a simple sequence of atomic values of the same type serializing it is rather easy. But as soon as more complex heterogeneous data structures have to pickled doing so by hand easily gets very error prone.

One reason for that is that three different definitions have to be synchronized. These are the definitions of the datatype to be pickled, the definition of the pickling function and the definition of the unpickling function. And in most cases there is more than one datatype involved!

2 A pickler library: Kennedy's Pickler Combinators

In [3] Kennedy describes an solution to pickling purely based on an combinator library and therefore embeddable in any programming language which offers higher order functions.

A combinator library is based on the idea of combining higher order functions of very uniform type. The combinators are carefully designed higher order functions which act as the glue; they provide a variety of ways of composing functions together into more powerful functions.

In the case of Kennedy's Parser Combinators both the pickling and the unpickling function are composed at the same time. It is therefore impossible to create an inconsistent pair of pickling/unpickling functions. In the library such

pairs are provided for the built-in types of the programming language (e.g unit, booleans, characters, non-negative integers and integers between 0 and some upper bound). The pairs are given the type `PU` α, where α is the type of the value to be pickled. Kennedy refers to such a type as a "pickler for α". The definition of the type

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

is not made accessible outside the implementation of the library further ensuring that the construction of an inconsistent pickler is not possible.

As you can see in the definition above the type of the pickling and unpickling functions had to be extended to enable composition. The semantics of `appP` are defined like this `appP (v, s)` prepends a serialized representation of `v` to an existing stream of serialized values `s`. Whereas `appU s` returns the deserialized value and the remaining stream. Ignoring sharing and minimization (see section 2.1) the following equation holds for all cycle free values `v` and byte sequences `s`: `(v, s) = appU (appP (v, s))`.

As set of combinators – functions from and to picklers – are provided to generate picklers for composite types. Given experience with a combinator library they have the expected types mirroring the corresponding type constructor:

- The pickler combinators for tuples `pair :: PU a -> PU b -> PU (a, b)`, `triple :: PU a -> PU b -> PU c -> PU (a, b, c)`, ...
- The pickler for lists `list :: PU a -> PU [a]`
- Optional values `pMaybe :: PU a -> PU (Maybe a)`

All these combinators are defined by means of the two combinators `lift` and `sequ`. Pickling of fixed values is done by the `lift :: a -> PU a` combinator. Its implementation is simple as no value has to (de)serialized.

```
lift x = PU (\ (_, s) -> s) (\ s -> (x, s))
```

The combinator `sequ :: (b->a) -> PU a -> (a -> PU b) -> PU b` is more interesting. It encodes sequential composition of picklers, in particular sequential dependencies are allowed. Assuming two values `A :: a` and `B :: b`, a pickler `pa :: PU a` and the two functions `f :: (b->a)` and `k:: a -> PU b` the pickler `p = sequ f pa k` has the following semantics. The call `appP p (B, s)` precedes the encoding of `B` by the encoding of `A = f B`. Most notably this encoding can depend on the value `A` as the pickler for `B` is generated by the call `k A`. The call `appU p s` in turn decodes `A`, generates the pickler for `b` by calling `k A` and decodes and returns `B`.

This looks quite complicated but as mentioned above it allows for simple definitions of pickler combinators such as `pair`

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

This definition is easy to understand if read in reverse order. In line three the values `a` and `b` are fixed and to pickle a pair of fixed values we can simple use lift. Now we precede this empty encoding of the fixed pair by the encodings of `b` and `a`.

Another combinator called `wrap :: (b->a, b->a) -> PU a -> PU b` is also implemented in terms of `sequ` and `lift` to provide mapping on picklers. Given an implementation of a fixed range cardinal number pickler `zeroTo :: Int -> PU Int` all ranged ordinal type picklers can be defined in terms of `wrap` and `zeroTo`. For example the definition of a pickler for boolean values looks like this
`bool = wrap (toEnum, fromEnum) (zeroTo 1)`.

A number of combinators make use of recursion. A good example is the previously mentioned `zeroTo :: Int -> PU Int`. `zeroTo n` creates a pickler for integers in the range $[0, n]$. It separates the representation into $\lceil \log_{256} n \rceil$ digits. Each digit has 256 possible values thereby making maximum use of the available storage.

Pickling of custom datatypes is done by combining the combinator `alt` with the `wrap` combinator. The `alt` combinator is used to combine several distinct picklers for values of the same type. Each pickler handles a disjunct set of possible values contained in the type. The user must also specify a tagging function which is used to determine which pickler to use.

Therefore for each constructor a separate pickler is defined by either lifting the constructor into a pickler or by wrapping a pickler for the argument of the constructor. These are then combined using `alt`, as seen in the example below.

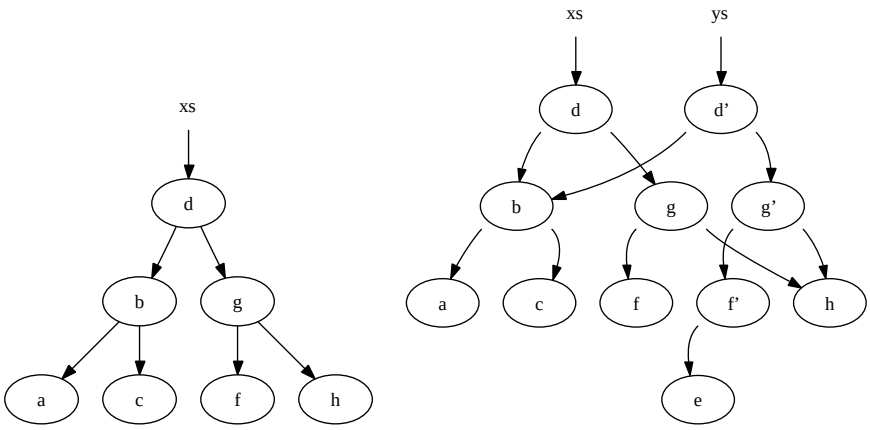
```
data Tree
  = Node (String, Tree, Tree)
  | Empty

tree  :: PU Tree
tree  = alt tag  [
          wrap (Node, \\(Node d) -> d)
                 (triple string tree tree)
        , lift Empty
        ]
      where tag (Node _)  = 0
            tag Empty     = 1
```

2.1 Sharing and Minimization

Thanks to persistence programs written a functional programming language usually make extensive use of sharing. A popular example are binary search trees. After inserting an element into the tree shown in figure 1 (a) we do not end up with two separate trees `xs` and `ys = insert (e, xs)`, but rather two trees which share a large number of nodes (see figure 1 (b)). Asides from memory consumption this difference is normally not observable from within the programming language. But there are two points which make sharing so important. One if we

had copied the elements upon insertion the runtime cost of insert would have been a lot worse. Second and even more important with the increased memory consumption it would be impossible to keep several versions of the tree in memory.



(a) before (b) after

Fig. 1. tree(s) before and after insertion

The library as described so far does not preserve sharing. In the example mentioned above this would matter as soon as more than one tree were to be pickled. In order to support sharing Kennedy implements the following idea. Sharing is detected by memorizing which values have already been pickled. If a value has not been pickled yet an numerical id is generated and added to the encoding. If the value is already part of the overall encoding just the id – which is part of the dictionary – will be included in the encoding.

Users of the combinator library must indicate shared datatypes by using the `share` combinator on its pickler. This combinator extends the normal pickler by the algorithm outlined above. To do so the datatype must support the equality operation to detect whether it is already part of the dictionary or not. Also the definition of the pickler datatype has been changed into

```
PU a s = PU { appP :: (a, (s, [Char])) -> (s, [Char])
            , appU :: (s, [Char]) -> (a, (s, [Char])) }
```

As you can see a pickler now also threads the dictionary. As we can not know what the type of the dictionary is – it depends on which type the `share` combinator is applied on – this type is a new type parameter of the pickler. This also

implies that sharing of more than one type of value at the same time requires rewriting the `share` combinator.

Still as long as the value pickled is not cyclic (see section 2.2) this library can be used minimize the heap representation of a value by maximizing sharing with respect to one component. As an example one could use the `share` combinator to either share the nodes of the tree, or the values of the keys but not both at the same time.

2.2 Cyclic values

Pure functional programming languages such as Haskell[2] normally use a non-eager evaluation model and can therefore express infinite (cyclic) data structures. The algorithm outlined above could in principle be used to serialize cyclic values. But the implementation given in the paper can not do so as the equality test used would diverge. Some low level pointer based comparison, which does not diverge on cyclic values has to be used instead.

In a non pure functional programming language such as SML[4] cyclic data structure are introduced explicitly by using references. Martin Elsman[1] presents an SML variant of Kennedy's library which uses an adopted variant of the algorithm outlined above to serialize references.

3 Builtin pickling and the abstract store

A different approach to pickling was defined by Guido Tack et al [5]. They defined an language independent memory model – the so called abstract store – and introduced pickling as a runtime system service similar in spirit to the garbage collector. Which they implemented as part of the virtual machine of the programming language Alice, a variant of SML. They also defined and included a generic minimization algorithm based on graph minimization. In particular as this minimization algorithm works on the representation level, values of different types can be shared and true minimization is achieved.

4 Comparison and Conclusions

If we compare the two different approaches we have to realize that both have different strength and weaknesses. One the one hand the combinator based approach does not require any kind of runtime support and is easily extensible and adaptable by the programmer. The approach by Guido Tack et al is essentially a runtime service and not extensible by the programmer in any way. But it supports both arbitrary sharing and full minimization in the presence of cyclic values. In the standard case it is a even simpler to use than the combinator based library.

It is also interesting to compare the different ways used to embed a dynamically typed value into a statically typed language. In Alice the types of the

pickled values are included in the pickled representation and the language was extended by a dynamic typecheck facility, similar to the well known `typecase` instruction. Therefore it is not necessary to known exactly what type of value was pickled. In the pickler combinator library instead one has to specify the toplevel pickler and there are no checks at all whether the pickled representation was actually generated using the same pickler.

One advantage of the combinator based library not mentioned by Kennedy is that it could be extended to support different backends such as binary versus textual by just changing a small number of the combinators. Still the solution to sharing presented by Kennedy does not scale as the number and types of shared values have to be known in advance. Even worse using a standard equality test used by the sharing/minimization algorithm actually results in an quadratic runtime behavior. I am therefore not sure whether the library as described is ready for use in non toy programs.

Still the principles presented are interesting. I do not know any other combinator library which creates more than one function at the same time and found that idea inspiring.

References

1. Martin Elsman. Type-specialized serialization with sharing. In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.
2. S. Peyton Jones and et al, editors. *Haskell 98 Language and Libraries, the Revised Report.* CUP, April 2003.
3. Andrew Kennedy. Pickler combinators. *J. Funct. Program.*, 14(6):727–739, 2004.
4. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* August 1990.
5. Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. *Electronic Notes in Theoretical Computer Science*, 148(2):79–103, March 2006.