Pickler Combinators – Explained

Benedikt Grundmann
`benedikt-grundmann@web.de`

Software Engineering Chair (Prof. Zeller)
Saarland University

Programming Systems Lab (Prof. Smolka)
Saarland University

Advanced Functional Programming – WS 2005/2006

Martin Elsman.
Type-specialized serialization with sharing.
In *Sixth Symposium on Trends in Functional Programming (TFP'05)*, September 2005.

Andrew Kennedy.
Pickler combinators.
J. Funct. Program., 14(6):727–739, 2004.

Guido Tack, Leif Kornstaedt, and Gert Smolka.
Generic pickling and minimization.
Electronic Notes in Theoretical Computer Science, 148(2):79–103, March 2006.

Outline

Outline

Example

- primitive Spellchecker application

Example

- primitive Spellchecker application
- words stored in binary search tree

Example

- primitive Spellchecker application
- words stored in binary search tree

Example

```
type Word = String

data Tree
  = N (Word, Tree, Tree)
  | E
```

Problem

How to store a tree?

```
createFile :: String -> String -> IO ()
loadFile :: String -> IO String
```

Problem

How to store a tree?

```
createFile :: String -> String -> IO ()
loadFile :: String -> IO String
```

Therefore we need:

```
toString :: Tree -> String
fromString :: String -> Tree
```

Writing those by hand is NO fun

- Synchronize
  - Type declaration
  - `toString` implementation
  - `fromString` implementation

Writing those by hand is NO fun

- Synchronize
  - Type declaration
  - `toString` implementation
  - `fromString` implementation
- extensibility?

Writing those by hand is NO fun

- Synchronize
  - Type declaration
  - `toString` implementation
  - `fromString` implementation
- extensibility?
- Implementation is not declarative

Outline

Solution: Pickling Combinators

```
word  :: PU String
word  = string

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

str = pickle tree (N ("foo", E, E))
N ("foo", E, E) = unpickle tree str
```

Outline

What is a Pickler Combinator Library?

- A combinator library to create picklers

What is a Pickler Combinator Library?

- A combinator library to create picklers
- We know what a combinator library is

What is a Pickler Combinator Library?

- A combinator library to create picklers
- We know what a combinator library is
  - Idea: Primitive functions + Combinator Functions = Powerful Functions

What is a Pickler Combinator Library?

- A combinator library to create picklers
- We know what a combinator library is
  - Idea: Primitive functions + Combinator Functions = Powerful Functions
  - “Higher-Order Functions for Parsing”

What is a Pickler Combinator Library?

- A combinator library to create picklers
- We know what a combinator library is
  - Idea: Primitive functions + Combinator Functions = Powerful Functions
  - “Higher-Order Functions for Parsing”
  - “Embedding an interpreted language using higher-order functions and types”

What is a Pickler Combinator Library?

- A combinator library to create picklers
- We know what a combinator library is
  - Idea: Primitive functions + Combinator Functions = Powerful Functions
  - “Higher-Order Functions for Parsing”
  - “Embedding an interpreted language using higher-order functions and types”
- So what is a pickler?

What is a Pickler?

A pair of a pickling and an unpickling function for values of a certain type.

What is a Pickler?

A pair of a pickling and an unpickling function for values of a certain type.

Definition (Pickling)

$$\text{Value} \mapsto \text{Byte*}$$

What is a Pickler?

A pair of a pickling and an unpickling function for values of a certain type.

Definition (Pickling)

$$\text{Value} \mapsto \text{Byte*}$$

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value}$$

What is a Pickler Combinator?

It is a pickler...

Definition (Pickling)

$$\text{Value} \mapsto \text{Byte*}$$

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value}$$

What is a Pickler Combinator?

It is a pickler extended to be composable.

Definition (Pickling)

$$\text{Value} \mapsto \text{Byte*}$$

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value}$$

What is a Pickler Combinator?

It is a pickler extended to be composable.

Definition (Pickling)

$$\text{Value} \times \text{Byte*} \mapsto \text{Byte*}$$

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value}$$

What is a Pickler Combinator?

It is a pickler extended to be composable.

Definition (Pickling)

$$\text{Value} \times \text{Byte*} \mapsto \text{Byte*}$$

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value} \times \text{Byte*}$$

Outline

API

```
data PU α
```

API

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

API

```
data PU α

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

API

```
data PU α

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

Example

```
True = unpickle bool (pickle bool True)
```

API

```
data PU α

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

Standard types

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

Basic Picklers & Combinators

- Constant values

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

unit = lift ()
```

- Small numbers

```
smallInt :: PU Int
smallInt = PU (\(c,s) -> (toEnum c : s))
              (\(c,s) -> (fromEnum c, s))
```

Sequential Composition

```
sequ :: (β->α) -> PU α -> (α->PU β) -> PU β
```

- pickles A followed by B
- A can be created from B
- pickled representation of B can depend on A

Example

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

More Combinators

- map on picklers

```
wrap :: (α-> β, β -> α) -> PU α -> PU β
bool = wrap (toEnum,fromEnum) (zeroTo 1)
```

- wrap & recursion

```
zeroTo :: Int -> PU Int
zeroTo 0 = lift 0
zeroTo n
 = wrap (\(h,l) -> h * 256 + l, (‘divMod‘ 256))
        (pair (zeroTo (n ‘div‘ 256)) smallInt)
```

Wrapping datatypes

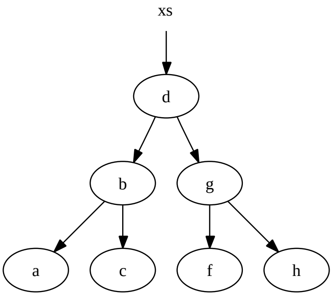
```
alt    :: (α -> Int) -> [PU α] -> PU α

wrap   :: (α -> β, β -> α) -> PU α -> PU β
```

Example

```
tree  = alt tag  [
          wrap (N, \(N d) -> d)
                 (triple word tree tree)
        , lift E
        ]
      where tag (N _) = 0
            tag E     = 1
```

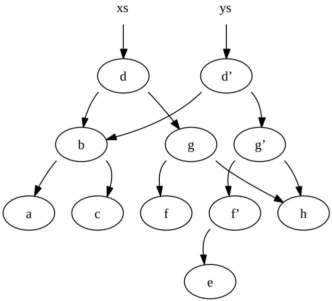
Outline

Sharing



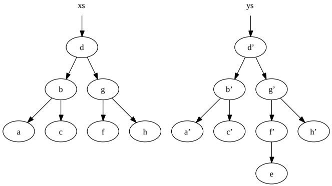
- We want sharing for efficiency
- Remember “Fun with binary heap trees”

Sharing



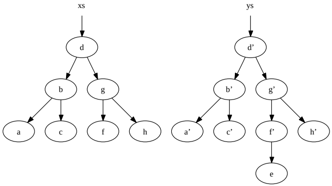
- We want sharing for efficiency
- Remember “Fun with binary heap trees”
- Example ys = insert (e, xs)

Sharing



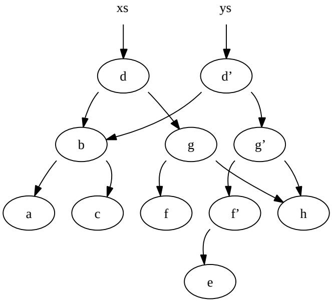
- We want sharing for efficiency
- Remember “Fun with binary heap trees”
- Example ys = insert (e, xs)
- (xs,ys) = unpickle
  (pickle (xs, ys))

Sharing



- We want sharing for efficiency
- Remember "Fun with binary heap trees"
- Example ys = insert (e, xs)
- (xs,ys) = unpickle
  (pickle (xs, ys))
- This is BAD!!

Sharing



- We want sharing for efficiency
- Remember “Fun with binary heap trees”
- Example ys = insert (e, xs)
- (xs,ys) = unpickle (pickle (xs, ys))
- This is BAD!!
- We want sharing!

Outline

Sharing Implementation Idea

On pickling

- Remember all values we pickled
- If we want to pickle it again store a reference

On unpickling

- Remember unpickled values
- On a reference return corresponding value

$\Rightarrow$ We need a dictionary!

Sharing Pickler Combinator

Need to memorize pickled values

Definition (Pickling)

$$\text{Value} \times \text{Byte}^* \mapsto \text{Byte}^*$$

Need to memorize unpickled values

Definition (Unpickling)

$$\text{Byte}^* \mapsto \text{Value}$$

Sharing Pickler Combinator

Need to memorize pickled values

Definition (Pickling)

$$\text{Value} \times \text{Byte*} \times \text{Dict} \mapsto \text{Byte*} \times \text{Dict}$$

Need to memorize unpickled values

Definition (Unpickling)

$$\text{Byte*} \mapsto \text{Value}$$

Sharing Pickler Combinator

Need to memorize pickled values

Definition (Pickling)

$$\text{Value} \times \text{Byte*} \times \text{Dict} \mapsto \text{Byte*} \times \text{Dict}$$

Need to memorize unpickled values

Definition (Unpickling)

$$\text{Byte*} \times \text{Dict} \mapsto \text{Value} \times \text{Dict}$$

Sharing continued

```
share :: Eq α => PU α [α] -> PU α [α]
share p = memorizing logic as outlined before

tree  = share $ alt tag  ...
```

- Sharing limited to values of one type

Sharing continued

```
share :: Eq α => PU α [α] -> PU α [α]
share p = memorizing logic as outlined before

tree  = share $ alt tag  ...
```

- Sharing limited to values of one type
- Normal equality test maximizes sharing

Sharing continued

```
share :: Eq α => PU α [α] -> PU α [α]
share p = memorizing logic as outlined before

tree  = share $ alt tag  ...
```

- Sharing limited to values of one type
- Normal equality test maximizes sharing
- Cyclic values

Sharing continued

```
share :: Eq α => PU α [α] -> PU α [α]
share p = memorizing logic as outlined before

tree  = share $ alt tag  ...
```

- Sharing limited to values of one type
- Normal equality test maximizes sharing
- Cyclic values
    - equality test diverges

Sharing continued

```
share :: Eq α => PU α [α] -> PU α [α]
share p = memorizing logic as outlined before

tree  = share $ alt tag  ...
```

- Sharing limited to values of one type
- Normal equality test maximizes sharing
- Cyclic values
    - equality test diverges
    - pointer based test would work

Outline

Pickler Combinator

Pro

- Declarative syntax – easy to use

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype
- Extensible

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype
- Extensible
- Language implementation independent

Contra

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype
- Extensible
- Language implementation independent

Contra

- either no cycles

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype
- Extensible
- Language implementation independent

Contra

- either no cycles
- or no minimization

Pickler Combinator

Pro

- Declarative syntax – easy to use
- Synchronization problem solved
  - only one code for both directions
  - Type checker checks consistency of pickler and datatype
- Extensible
- Language implementation independent

Contra

- either no cycles
- or no minimization
- sharing only values of one type

More Samples

$$
\begin{array}{lll}
\texttt{list} & \texttt{::} & \texttt{PU}\ \alpha\ \texttt{->}\ \texttt{PU}\ [\alpha] \\
\texttt{pair} & \texttt{::} & \texttt{PU}\ \alpha\ \texttt{->}\ \texttt{PU}\ \beta\ \texttt{->}\ \texttt{PU}\ (\alpha,\ \beta) \\
\texttt{triple} & \texttt{::} & \texttt{PU}\ \alpha\ \texttt{->}\ \texttt{PU}\ \beta\ \texttt{->}\ \texttt{PU}\ \gamma\ \texttt{->}\ \texttt{PU}\ (\alpha,\ \beta,\ \gamma) \\
\texttt{maybe} & \texttt{::} & \texttt{PU}\ \alpha\ \texttt{->}\ \texttt{PU}\ (\texttt{Maybe}\ \alpha)
\end{array}
$$

More Samples

```
list   :: PU α -> PU [α]
pair   :: PU α -> PU β -> PU (α, β)
triple :: PU α -> PU β -> PU γ -> PU (α, β, γ)
maybe  :: PU α -> PU (Maybe α)
```

Example

```
type URL      = (String, String, Maybe Int, String)
type Bookmark  = (String, URL)

string     = list char
url        = quad string string (maybe nat) string
bookmark   = pair string url
```