Pickler Combinators – Explained

Benedikt Grundmann
benedikt-grundmann@web.de

Software Engineering Chair (Prof. Zeller)
Saarland University

Programming Systems Lab (Prof. Smolka)
Saarland University

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

Andrew Kennedy.
Pickler combinators.

Guido Tack, Leif Kornstaedt, and Gert Smolka.
Generic pickling and minimization.
Outline

Motivation
   Spellchecker
   Solution preview

Pickler Combinator
   Introduction
   API & Implementation

Sharing
   Problem
   Solution

The End
   Wrap-Up Pickler Combinator
Outline

Motivation
  Spellchecker
  Solution preview

Pickler Combinator
  Introduction
  API & Implementation

Sharing
  Problem
  Solution

The End
  Wrap-Up Pickler Combinator
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

Motivation
  Spellchecker
  Solution preview

Pickler Combinator
  Introduction
  API & Implementation

Sharing
  Problem
  Solution

The End
  Wrap-Up Pickler Combinator
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

Motivation
  Spellchecker
  Solution preview

Pickler Combinator
  Introduction
  API & Implementation

Sharing
  Problem
  Solution

The End
  Wrap-Up Pickler Combinator
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)

Value $\mapsto$ Byte*

Definition (Unpickling)

Byte* $\mapsto$ 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)**

\[ \text{Value} \rightarrow \text{Byte}^* \]

**Definition (Unpickling)**

\[ \text{Byte}^* \leftarrow \text{Value} \]
What is a Pickler Combinator?

It is a pickler extended to be composable.

**Definition (Pickling)**

\[
\text{Value} \times \text{Byte}^* \rightarrow \text{Byte}^*
\]

**Definition (Unpickling)**

\[
\text{Byte}^* \leftrightarrow \text{Value}
\]
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

Motivation
  Spellchecker
  Solution preview

Pickler Combinator
  Introduction
  API & Implementation

Sharing
  Problem
  Solution

The End
  Wrap-Up Pickler Combinator
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)
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**
  
  \[
  \text{lift} :: \alpha \rightarrow PU \alpha
  \]
  
  \[
  \text{lift } x = PU \ \text{snd} \ (\lambda s \rightarrow (x, s))
  \]

  \[
  \text{unit} = \text{lift} ()
  \]

- **Small numbers**
  
  \[
  \text{smallInt} :: PU \text{ Int}
  \]
  
  \[
  \text{smallInt} = PU \ (\lambda (c, s) \rightarrow (\text{toEnum } c : s))
  \]
  \[
  (\lambda (c, s) \rightarrow (\text{fromEnum } c, s))
  \]
### Sequential Composition

\[
\text{sequ} :: (\beta \rightarrow \alpha) \rightarrow PU \alpha \rightarrow (\alpha \rightarrow PU \beta) \rightarrow PU \beta
\]

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

#### Example

\[
\text{pair} :: PU \alpha \rightarrow PU \beta \rightarrow PU (\alpha, \beta)
\]

\[
\text{pair} \ pa \ pb = \text{sequ} \ \text{fst} \ pa (\lambda \ a \rightarrow \text{sequ} \ \text{snd} \ pb (\lambda \ b \rightarrow \text{lift} \ (a, b)))
\]
More Combinators

• map on picklers

\[
\text{wrap} :: (\alpha \to \beta, \beta \to \alpha) \to \text{PU} \alpha \to \text{PU} \beta \\
\text{bool} = \text{wrap} (\text{toEnum, fromEnum}) (\text{zeroTo} 1)
\]

• wrap & recursion

\[
\text{zeroTo} :: \text{Int} \to \text{PU} \text{Int} \\
\text{zeroTo} 0 = \text{lift} 0 \\
\text{zeroTo} n \\
= \text{wrap} (\\( (h, l) \to h \times 256 + l, (\text{\'divMod\' 256}) \) (pair (\text{zeroTo} (n \text{\'div\' 256}) \text{smallInt}))
\]
Wrapping datatypes

\[
\begin{align*}
alt &:: (\alpha \to \text{Int}) \to [\text{PU} \ \alpha] \to \text{PU} \ \alpha \\
\text{wrap} &:: (\alpha \to \beta, \beta \to \alpha) \to \text{PU} \ \alpha \to \text{PU} \ \beta
\end{align*}
\]

Example

\[
\begin{align*}
tree &= \text{alt} \ \text{tag} \ [ \\
&\quad \text{wrap} \ (N, \ \text{\textbackslash} (N \ d) \to d) \\
&\quad \quad \text{(triple word tree tree)} \\
&\quad , \ \text{lift} \ E \\
&\quad ] \\
\text{where} \ \text{tag} \ (N \ _) &= 0 \\
\text{tag} \ E &= 1
\end{align*}
\]
Outline

Motivation
  Spellchecker
  Solution preview

Pickler Combinator
  Introduction
  API & Implementation

Sharing
  Problem
  Solution

The End
  Wrap-Up Pickler Combinator
Motivation

Sharing

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

Remember “Fun with binary heap trees”

Example $ys = \text{insert} \ (e, \ xs)$
We want sharing for efficiency

Remember “Fun with binary heap trees”

Example $ys = insert(e, xs)$

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

Motivation
   Spellchecker
   Solution preview

Pickler Combinator
   Introduction
   API & Implementation

Sharing
   Problem
   Solution

The End
   Wrap-Up Pickler Combinator
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
⇒ We need a dictionary!
Sharing Pickler Combinator

Need to memorize pickled values

\textbf{Definition (Pickling)}

\[ \text{Value} \times \text{Byte}^* \mapsto \text{Byte}^* \]

Need to memorize unpickled values

\textbf{Definition (Unpickling)}

\[ \text{Byte}^* \mapsto \text{Value} \]
Sharing Pickler Combinator

Need to memorize pickled values

Definition (Pickling)

\[ \text{Value} \times \text{Byte}^* \times \text{Dict} \rightarrow \text{Byte}^* \times \text{Dict} \]

Need to memorize unpickled values

Definition (Unpickling)

\[ \text{Byte}^* \leftrightarrow \text{Value} \]
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

Motivation
Spellchecker
Solution preview

Pickler Combinator
Introduction
API & Implementation

Sharing
Problem
Solution

The End
Wrap-Up Pickler Combinator
Pickler Combinator

Pro

- Declarative syntax – easy to use

Contra

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

**Pro**

- Declarative syntax – easy to use
- Synchronization problem solved

**Contra**

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

list :: PU \alpha \rightarrow PU [\alpha]

pair :: PU \alpha \rightarrow PU \beta \rightarrow PU (\alpha, \beta)

triple :: PU \alpha \rightarrow PU \beta \rightarrow PU \gamma \rightarrow PU (\alpha, \beta, \gamma)

maybe :: PU \alpha \rightarrow PU (Maybe \alpha)
More Samples

list :: PU \alpha \rightarrow PU \{\alpha\}
pair :: PU \alpha \rightarrow PU \beta \rightarrow PU (\alpha, \beta)
triple :: PU \alpha \rightarrow PU \beta \rightarrow PU \gamma \rightarrow PU (\alpha, \beta, \gamma)
maybe :: PU \alpha \rightarrow PU (Maybe \alpha)

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