Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing

Tillmann Rendel   Klaus Ostermann

Marburg University

Haskell Symposium, September 30, 2010

Parsing and Pretty Printing

- parser combinator libraries
- pretty printing libraries

$$\textbf{data}\ Exp = \ldots$$

$$parseExp :: Parser\ Exp$$

$$printExp :: Exp \rightarrow Doc$$

Example language

- Concrete syntax

$$e ::= \texttt{"S"} \mid \texttt{"K"} \mid e\ e \mid \texttt{"("}\ e\ \texttt{")"}$$

- Abstract syntax

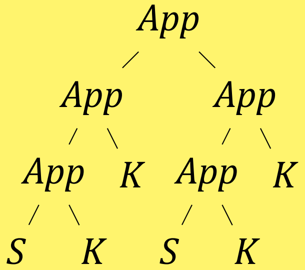
$$\textbf{data}\ SK = S \mid K \mid App\ SK\ SK$$

Left Associative Application

- String

```
"SKK(SKK)"
"((SK)K)((SK)K)"
```

- AST



Parser

```haskell
parseSK = exp1 where
  exp0 = S <$ tok "S"
      <|> K <$ tok "K"
      <|> tok "(" *> exp1 <* tok ")"
  exp1 = foldl App <$> exp0 <*> many exp0
  tok k = string k <* spaces
```

Pretty Printer

$$
\begin{aligned}
&\mathit{printSK}\ p\ x = \textbf{case}\ x\ \textbf{of} \\
&\quad S \qquad\quad\ \to \mathit{text}\ \texttt{"S"} \\
&\quad K \qquad\quad\ \to \mathit{text}\ \texttt{"K"} \\
&\quad \mathit{App}\ a\ b \to \\
&\qquad (\textbf{if}\ p\ \textbf{then}\ \mathit{parens}\ \textbf{else}\ \mathit{id}) \\
&\qquad\quad (\mathit{printSK}\ \mathit{False}\ a \diamond \mathit{printSK}\ \mathit{True}\ b)
\end{aligned}
$$

- Need to synthesize parentheses

Unified Syntax Description

```
syntaxSK = exp1 where
  exp0   = s <$> tok "S"
           <|> k <$> tok "K"
           <|> tok "(" *> exp1 <* tok ")"
  exp1   = foldl app <$> exp0 <*> many exp0
  tok k  = string k <* spaces
```

Unifying Choice

$$(\diamondsuit) :: \mathit{Parser}\ \alpha \rightarrow \mathit{Parser}\ \alpha \rightarrow \mathit{Parser}\ \alpha$$

$$\textbf{type}\ \mathit{Printer}\ \alpha = \alpha \rightarrow \mathit{Maybe}\ \mathit{String}$$
$$(\diamondsuit) :: \mathit{Printer}\ \alpha \rightarrow \mathit{Printer}\ \alpha \rightarrow \mathit{Printer}\ \alpha$$

$$(\diamondsuit) :: \mathit{Syntax}\ \delta \Rightarrow \delta\ \alpha \rightarrow \delta\ \alpha \rightarrow \delta\ \alpha$$

Unifying Mapping

$$(\langle\$\rangle) :: (\alpha \rightarrow \beta) \rightarrow (\mathit{Parser}\ \alpha \rightarrow \mathit{Parser}\ \beta)$$

$$(\langle\$\rangle) :: (\beta \rightarrow \alpha) \rightarrow (\mathit{Printer}\ \alpha \rightarrow \mathit{Printer}\ \beta)$$

$$(\langle\$\rangle) :: \mathit{Syntax}\ \delta \Rightarrow \mathit{Iso}\ \alpha\ \beta \rightarrow (\delta\ \alpha \rightarrow \delta\ \beta)$$

Partial Isomorphisms

$$
\begin{aligned}
&\textbf{data}\ \mathit{Iso}\ \alpha\ \beta \\
&\quad = \mathit{Iso}\ (\alpha \to \mathit{Maybe}\ \beta) \\
&\qquad\qquad (\beta \to \mathit{Maybe}\ \alpha)
\end{aligned}
$$

- $f\ x \equiv \mathit{Just}\ y \quad \Leftrightarrow \quad g\ y \equiv \mathit{Just}\ x$

Constructors

$$
\begin{aligned}
&app :: Iso\ (Exp, Exp)\ Exp \\
&app = Iso\ f\ g\ \mathbf{where} \\
&\quad f\ (a, b) = Just\ (App\ a\ b) \\
&\quad g\ (App\ a\ b) = Just\ (a, b) \\
&\quad g\ \_ \qquad\qquad\ \ = Nothing
\end{aligned}
$$

- Template Haskell macro

Folding

$$foldl :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

$$foldl :: Iso\ (\alpha, \beta)\ \alpha \to Iso\ (\alpha, [\beta])\ \alpha$$

Folding

$$foldl :: (\alpha \to \beta \to \alpha) \to \alpha \to [\beta] \to \alpha$$

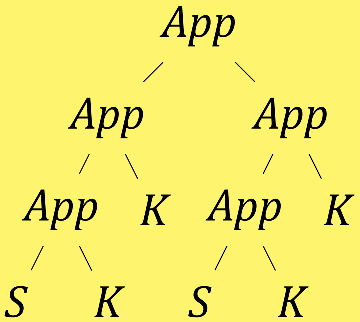
$$foldl :: Iso\ (\alpha, \beta)\ \alpha \to Iso\ (\alpha, [\beta])\ \alpha$$

- *inverse foldl* is *unfoldl*!

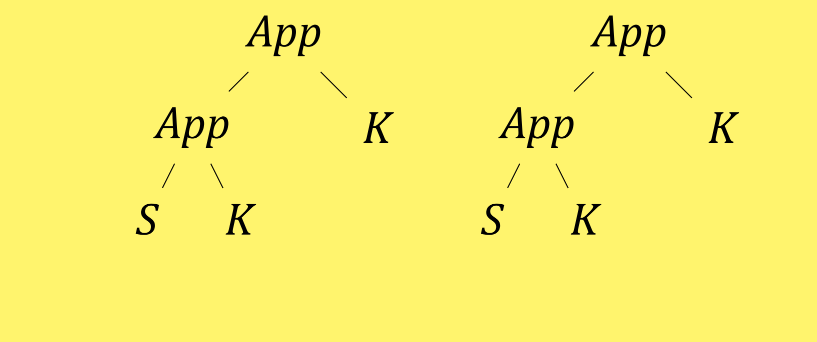
Unified Syntax Description

```
syntaxSK = exp1 where
  exp0  = s <$> tok "S"
        <|> k <$> tok "K"
        <|> tok "(" *> exp1 <* tok ")"
  exp1  = foldl app <$> exp0 <*> many exp0
  tok k = string k <* spaces
```

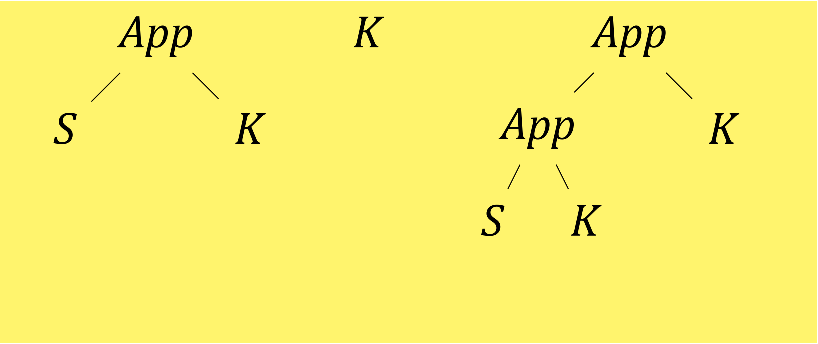
Unfolding



Unfolding

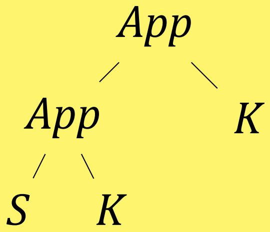


Unfolding



Unfolding

$S \quad K \quad K$

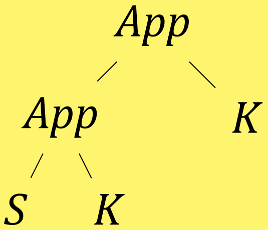


Unified Syntax Description

```
syntaxSK = exp1 where
  exp0  =  s <$> tok "S"
           <|> k <$> tok "K"
           <|> tok "(" *> exp1 <* tok ")"
  exp1  = foldl app <$> exp0 <*> many exp0
  tok k = string k <* spaces
```

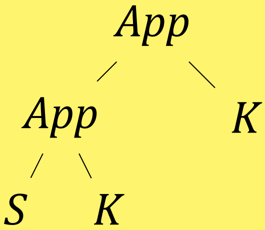
Atomic terms

$S \qquad K \qquad K$



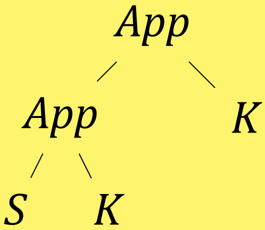
Atomic terms

`"S"` K K



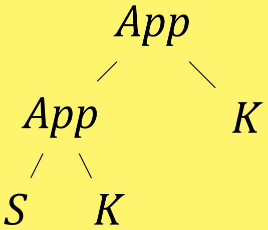
Atomic terms

`"S"` `"K"` K



Atomic terms

`"S"`   `"K"`   `"S"`

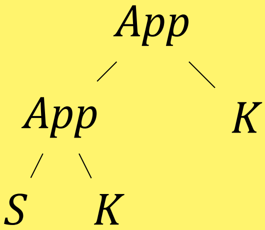


Unified Syntax Description

```
syntaxSK = exp1 where
  exp0  =  s <$> tok "S"
       <|> k <$> tok "K"
       <|> tok "("  *> exp1 <*  tok ")"
  exp1  = foldl app <$> exp0 <*> many exp0
  tok k = string k <*  spaces
```

Recursive call

`"S"    "K"    "S"`



Recursive call

```
"S"   "K"   "S"   "(SKK) "
```

In the paper

- Proof-of-concept Implementation
- Algebra of Partial Isomorphisms
- Operator Priorities
- Derivation of *foldl*
- Related Work
- Full Code

Future Work

- *Syntax* instances for existing libraries
- Larger case-study
- proof-carrying partial isomorphisms
- proof of round-trip property

Haskell wish list

- Better support for *Category* (*Functor* etc. between different categories)
- Arrow notation for nearly-arrows (no $arr :: (\alpha \to \beta) \to (Iso\ \alpha\ \beta)$)
- Generic isomorphisms for constructors

Summary

- unified combinator library for parsing and pretty printing
- based on functor from partial isomorphisms
- looks like existing parser combinator parsers
- pretty printing for free!

Summary

- unified combinator library for parsing and pretty printing
- based on functor from partial isomorphisms
- looks like existing parser combinator parsers
- pretty printing for free!

Thank you!

Invertible Folding

- Entry point

$$
\begin{aligned}
&\mathit{foldl} :: \mathit{Iso}\ (\alpha, \beta)\ \alpha \to \mathit{Iso}\ (\alpha, [\beta])\ \alpha \\
&\mathit{foldl}\ i = \mathit{inverse}\ \mathit{unit} \\
&\qquad\qquad \circ\ (\mathit{id} \times \mathit{inverse}\ \mathit{nil}) \\
&\qquad\qquad \circ\ \mathit{iterate}\ (\mathit{step}\ i)
\end{aligned}
$$

Invertible Folding

- Invertible step function

$$\begin{aligned} \mathit{step}\ i = {} & (i \times \mathit{id}) \\ & \circ\ \mathit{associate} \\ & \circ\ (\mathit{id} \times \mathit{inverse}\ \mathit{cons}) \end{aligned}$$

- Exchange top-level *i* and *cons*

Invertible Folding

- Iteration of partial isomorphisms

$$
\begin{aligned}
&\mathit{iterate} :: \mathit{Iso}\ \alpha\ \alpha \to \mathit{Iso}\ \alpha\ \alpha\\
&\mathit{iterate}\ \mathit{step} = \mathit{Iso}\ f\ g\ \textbf{where}\\
&\quad f = \mathit{Just} \circ \mathit{driver}\ (\mathit{apply}\ \ \ \ \mathit{step})\\
&\quad g = \mathit{Just} \circ \mathit{driver}\ (\mathit{unapply}\ \mathit{step})
\end{aligned}
$$

Invertible Folding

- Iteration of partial functions

$$
\begin{array}{l}
\mathit{driver} :: (\alpha \to \mathit{Maybe}\ \alpha) \to (\alpha \to \alpha) \\
\mathit{driver}\ \mathit{step}\ \mathit{state} \\
\quad = \mathbf{case}\ \mathit{step}\ \mathit{state}\ \mathbf{of} \\
\qquad\quad \mathit{Just}\ \mathit{state}' \to \mathit{driver}\ \mathit{step}\ \mathit{state}' \\
\qquad\quad \mathit{Nothing} \quad\ \to \mathit{state}
\end{array}
$$

Type classes

$$
\begin{array}{l}
\textbf{class}\ \mathit{IsoFunctor}\ f\ \textbf{where} \\
\quad (\langle\$\rangle) :: \mathit{Iso}\ \alpha\ \beta \rightarrow (f\ \alpha \rightarrow f\ \beta) \\
\textbf{class}\ \mathit{Alternative}\ f\ \textbf{where} \\
\quad (\langle|\rangle) :: f\ \alpha \rightarrow f\ \alpha \rightarrow f\ \alpha \\
\quad \mathit{empty} :: f\ \alpha \\
\textbf{class}\ \mathit{ProductFunctor}\ f\ \textbf{where} \\
\quad (\langle*\rangle) :: f\ \alpha \rightarrow f\ \beta \rightarrow f\ (\alpha, \beta)
\end{array}
$$

Type classes

$$
\begin{aligned}
&\textbf{class}\ (\ \mathit{IsoFunctor}\ \delta,\\
&\qquad\quad \mathit{ProductFunctor}\ \delta,\\
&\qquad\quad \mathit{Alternative}\ \delta)\\
&\ \ \Rightarrow \mathit{Syntax}\ \delta\ \textbf{where}\\
&\quad \mathit{pure}\ \ :: \mathit{Eq}\ \alpha \Rightarrow \alpha \to \delta\ \alpha\\
&\quad \mathit{token} :: \delta\ \mathit{Char}
\end{aligned}
$$

Parser Implementation

$$\textbf{newtype}\ Parser\ \alpha$$
$$\quad = Parser\ (String \rightarrow [(\alpha, String)])$$
$$parse :: Parser\ \alpha \rightarrow String \rightarrow [\alpha]$$
$$parse\ (Parser\ p)\ s = [x \mid (x, \texttt{""}) \leftarrow p\ s]$$

Parser Implementation

$$
\begin{array}{l}
\textbf{instance}\ \mathit{IsoFunctor}\ \mathit{Parser}\ \textbf{where} \\
\quad \mathit{iso} \mathbin{\diamond\!\!\!\!\$\,} \mathit{Parser}\ p \\
\qquad = \mathit{Parser}\ (\lambda s \rightarrow \begin{array}[t]{l} [\ (y, s') \\ \mid (x, s') \leftarrow p\ s \\ ,\ \mathit{Just}\ y \leftarrow [\mathit{apply}\ \mathit{iso}\ x]]) \end{array}
\end{array}
$$

Parser Implementation

$$
\begin{aligned}
&\textbf{instance } \mathit{ProductFunctor}\ \mathit{Parser}\ \textbf{where}\\
&\quad \mathit{Parser}\ p \diamond\!\!\!\!*\ \ \mathit{Parser}\ q\\
&\qquad = \mathit{Parser}\ (\lambda s \rightarrow [\ ((x,y), s'')\\
&\qquad\qquad\qquad\qquad\ \ \mid (x, s') \leftarrow p\ s\\
&\qquad\qquad\qquad\qquad\ \ ,\ (y, s'') \leftarrow q\ s'])
\end{aligned}
$$

Parser Implementation

instance *Alternative Parser* **where**

$$Parser\ p \diamondsuit Parser\ q$$
$$= Parser\ (\lambda s \rightarrow p\ s + q\ s)$$
$$empty = Parser\ (\lambda s \rightarrow [\,])$$

Parser Implementation

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Syntax}\ \mathit{Parser}\ \textbf{where} \\
\quad \mathit{pure}\ x = \mathit{Parser}\ (\lambda s \rightarrow [\,(x, s)\,]) \\
\quad \mathit{token}\ \ = \mathit{Parser}\ f\ \textbf{where} \\
\qquad f\ [\,] \qquad\quad = [\,] \\
\qquad f\ (t : \mathit{ts}) = [\,(t, \mathit{ts})\,]
\end{array}
$$

Printer Implementation

$$\textbf{newtype}\ \mathit{Printer}\ \alpha = \mathit{Printer}\ (\alpha \to \mathit{Maybe}\ \mathit{String})$$
$$\mathit{print} :: \mathit{Printer}\ \alpha \to \alpha \to \mathit{Maybe}\ \mathit{String}$$
$$\mathit{print}\ (\mathit{Printer}\ p)\ x = p\ x$$

Printer Implementation

$$
\begin{array}{l}
\textbf{instance}\ \mathit{IsoFunctor}\ \mathit{Printer}\ \textbf{where} \\
\quad \mathit{iso} \mathbin{\langle\$\rangle} \mathit{Printer}\ p \\
\qquad = \mathit{Printer}\ (\lambda b \rightarrow \mathit{unapply}\ \mathit{iso}\ b \gg\!= p)
\end{array}
$$

Printer Implementation

```
instance ProductFunctor Printer where
  Printer p <*> Printer q
    = Printer (λ(x,y) → liftM2 (+) (p x) (q y))
```

Printer Implementation

$$
\begin{array}{l}
\textbf{instance } \mathit{Alternative\ Printer} \textbf{ where} \\
\quad \mathit{Printer}\ p \diamond\!\!\!\mid\ \mathit{Printer}\ q \\
\qquad = \mathit{Printer}\ (\lambda s \rightarrow \mathit{mplus}\ (p\ s)\ (q\ s)) \\
\quad \mathit{empty} = \mathit{Printer}\ (\lambda s \rightarrow \mathit{Nothing})
\end{array}
$$

Printer Implementation

```
instance Syntax Printer where
  pure x = Printer (λy → if x ≡ y
                           then Just ""
                           else Nothing)
  token = Printer (λt → Just [t])
```