Rewinding the stack for parsing and pretty printing

Mathieu Boespflug

McGill University

26 July 2011

A little primer on HASKELL

- The polymorphic type of lists is written $[a]$.
- *head* of type $[a] \to a$ is written $head :: [a] \to a$.
- $() :: ()$
- new datatype:

$$\textbf{data}\ Maybe\ a = Nothing \mid Just\ a$$

- type synonym:

$$\textbf{type}\ Env\ a = [(Int, a)]$$

The Problem

What is a parser?

$$\textbf{type}\ P\ a = String \rightarrow Either\ Error\ (a, String)$$

What is a parser?

$$\textbf{type}\ P\ a = String \to Either\ Error\ (a, String)$$

$$
\begin{aligned}
&fail :: Error \to Either\ Error\ (a, String)\\
&success :: (a, String) \to Either\ Error\ (a, String)
\end{aligned}
$$

$$
\begin{aligned}
&lit :: P\ ()\\
&lit\ x = P\ (\lambda s \to \textbf{case}\ stripPrefix\ x\ s\ \textbf{of}\\
&\qquad\qquad Nothing \to fail\ \texttt{"Parse error."}\\
&\qquad\qquad Just\ s' \to success\ ((), s'))
\end{aligned}
$$

$$
\begin{aligned}
&true :: P\ Bool\\
&true = P\ (\lambda s \to \textbf{case}\ stripPrefix\ \texttt{"true"}\ s\ \textbf{of}\\
&\qquad\qquad Nothing \to fail\ \texttt{"Parse error."}\\
&\qquad\qquad Just\ s' \to success\ (True, s'))
\end{aligned}
$$

$$
\begin{aligned}
&false :: P\ Bool\\
&false = P\ (\lambda s \to \textbf{case}\ stripPrefix\ \texttt{"false"}\ s\ \textbf{of}\\
&\qquad\qquad Nothing \to fail\ \texttt{"Parse error."}\\
&\qquad\qquad Just\ s' \to success\ (False, s'))
\end{aligned}
$$

Combining parsers

- Parsers are monadic actions.
- Can be built compositionally from existing parser *combinators*, which are also monadic actions.

$$
\begin{aligned}
&\mathit{pure}\, f = P\,(\lambda s \rightarrow (f, s)) \\
&P\, m \otimes P\, k = P\,(\lambda s \rightarrow \textbf{case}\; m\; s\; \textbf{of} \\
&\qquad\qquad\qquad\qquad (f, s') \rightarrow \textbf{case}\; k\; s'\; \textbf{of} \\
&\qquad\qquad\qquad\qquad\quad (x, s'') \rightarrow (f\, x, s''))
\end{aligned}
$$

Example parser

$$
\begin{aligned}
&pure :: a \to P\,a \\
&(\otimes) :: P\,(a \to b) \to P\,a \to P\,b \\
&(\oplus) :: P\,a \to P\,a \to P\,a
\end{aligned}
$$

$$E ::= \textbf{true} \mid \textbf{false} \mid \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E$$

$$
\begin{aligned}
&\textbf{data}\ Tm = Boolean\ Bool \mid If\ Tm\ Tm\ Tm \\
&tm :: P\,Tm \\
&tm = pure\ Boolean \otimes true \\
&\quad \oplus pure\ Boolean \otimes false \\
&\quad \oplus pure\ (\lambda\_\,x\ \_\,y\ \_\,z \to If\ x\ y\ z) \otimes lit\ \texttt{"if"} \otimes tm \otimes \texttt{"then"} \\
&\qquad\qquad \otimes tm \otimes lit\ \texttt{"else"} \otimes tm
\end{aligned}
$$

Example Pretty Printer

$$E ::= \textbf{true} \mid \textbf{false} \mid \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E$$

$$\textbf{data}\ Tm = Boolean\ Bool \mid If\ Tm\ Tm\ Tm$$

$$\begin{aligned}
&tm\ t = \textbf{case}\ t\ \textbf{of}\\
&\quad Boolean\ True \rightarrow \texttt{"true"}\\
&\quad Boolean\ False \rightarrow \texttt{"false"}\\
&\quad If\ x\ y\ z \rightarrow \texttt{"if "} +\!\!+\ x +\!\!+ \texttt{" then "}\\
&\qquad\qquad\qquad +\!\!+\ y +\!\!+ \texttt{" else "} +\!\!+\ z
\end{aligned}$$

Objective:

- Write the parser once, get the pretty printer for free.
- Write the pretty printer once, get the parser for free.

Objective:

- Write the parser once, get the pretty printer for free.
- Write the pretty printer once, get the parser for free.

Why?

- Synchrony!
- Synchrony means easier to maintain.
- Synchrony means less code.
- Less code means fewer bugs.
- Pollack consistency.

Objective:

- Write the parser once, get the pretty printer for free.
- Write the pretty printer once, get the parser for free.

Why?

- Synchrony!
- Synchrony means easier to maintain.
- Synchrony means less code.
- Less code means fewer bugs.
- Pollack consistency.

How?

- Write both at the same time.

The Solution

A Cassette

A *Kassette* in Haskell

$$\textbf{data}\ \mathit{K7}\ a\ b = \mathit{K7}\ \{\mathit{sideA} :: a, \mathit{sideB} :: b\}$$

A *Kassette* in Haskell

$$\textbf{data}\ K7\ a\ b = K7\ \{sideA :: a, sideB :: b\}$$

$$\begin{aligned}
(\diamondsuit)\ &::\ K7\ (b \to c)\ (a \to b) \\
&\to K7\ (a \to b)\ (b \to c) \\
&\to K7\ (a \to c)\ (c \to a) \\
\sim\!(K7\ f\ f')\ &\diamondsuit \sim\!(K7\ g\ g') = K7\ (f \circ g)\ (g' \circ f')
\end{aligned}$$

The category of cassettes

Can overload $(\circ)$ with $(\diamondsuit)$:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Category}\ \kappa\ \textbf{where} \\
\quad \mathit{id} :: \kappa\ a\ a \\
\quad (\circ) :: \kappa\ b\ c \rightarrow \kappa\ a\ b \rightarrow \kappa\ a\ c \\
\textbf{instance}\ \mathit{Category}\ \mathit{K7}\ \textbf{where} \\
\quad \mathit{id} = \mathit{K7}\ \mathit{id}\ \mathit{id} \\
\quad (\circ) = (\diamondsuit)
\end{array}
$$

Sequencing

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \to Either\ Error\ (a, String))$$
$$(??)$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \to Either\ Error\ (a, String))\ ((a, String) \to String)$$

A tentative parsing and pretty printing cassette

$$\begin{aligned}\textbf{type}\ PP\ a = K7\ &(String \to Either\ Error\ (a, String))\\ &(Either\ Error\ (a, String) \to String)\end{aligned}$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \to Either\ Error\ (a, String))\\ \qquad ((a, String) \to String)$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \to Either\ Error\ (a, String))$$
$$(a \to String \to String)$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \rightarrow Either\ Error\ (a, String))$$
$$(a \rightarrow String \rightarrow String)$$

$$pure\ (\lambda x\ y\ z \rightarrow If\ x\ y\ z) \quad :: P\ (Tm \rightarrow Tm \rightarrow Tm \rightarrow Tm)$$
$$pure\ (\lambda x\ y\ z \rightarrow If\ x\ y\ z) \otimes tm :: P\ (\quad Tm \rightarrow Tm \rightarrow Tm)$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \to Either\ Error\ (a, String))\ (a \to String \to String)$$

$$pure\ (\lambda x\ y\ z \to If\ x\ y\ z) :: P\ (Tm \to Tm \to Tm \to Tm)$$
$$pure\ (\lambda x\ y\ z \to If\ x\ y\ z) \otimes tm :: P\ (Tm \to Tm \to Tm)$$

$$K7\ (pure\ (\lambda x\ y\ z \to If\ x\ y\ z))\ (??)$$
$$:: K7\ (String \to (Tm \to Tm \to Tm \to Tm, String))$$
$$((Tm \to Tm \to Tm \to Tm) \to String \to String)$$

A tentative parsing and pretty printing cassette

$$\textbf{type}\ PP\ a = K7\ (String \rightarrow Either\ Error\ (a, String))$$
$$(a \rightarrow String \rightarrow String)$$

$$pure\ (\lambda x\ y\ z \rightarrow If\ x\ y\ z) \quad :: P\ (Tm \rightarrow Tm \rightarrow Tm \rightarrow Tm)$$
$$pure\ (\lambda x\ y\ z \rightarrow If\ x\ y\ z) \otimes tm :: P\ (\qquad Tm \rightarrow Tm \rightarrow Tm)$$

$$K7\ (pure\ (\lambda(x,y,z) \rightarrow If\ x\ y\ z))\ (??)$$
$$:: K7\ (String \rightarrow ((Tm, Tm, Tm) \rightarrow Tm, String))$$
$$((Tm \rightarrow (Tm, Tm, Tm)) \rightarrow String \rightarrow String)$$

The problem

To summarize:

- Need uncurried functions so that type to parse and type to pretty print match.
- Can inductively construct curried function type $a_1 \to (a_2 \to (\ldots \to a_n))$.
- Uncurried function type $(a_1, a_2, \ldots, a_{n-1}) \to a_n$ cannot be inductively constructed.
- Cannot feed arguments to an uncurried function incrementally.
- Tuples as arguments and returning tuples breaks composability.

Recovering symmetry with continuation passing style

Type of consumer in CPS:

$$(a_1 \to \ldots \to a_n \to r) \to r$$

Type of producer in CPS:

$$r \to a_1 \to \ldots \to a_n \to r$$

Recovering symmetry with continuation passing style

Type of parser in CPS:

$$(String \to a_1 \to \ldots \to a_n \to r) \to String \to r$$

Type of pretty printer CPS:

$$(String \to r) \to String \to a_1 \to \ldots \to a_n \to r$$

Recovering symmetry with continuation passing style

Type of parser in CPS:

$$(String \to a_1 \to \ldots \to a_n \to r) \to String \to r$$

Type of pretty printer CPS:

$$(String \to r) \to String \to a_1 \to \ldots \to a_n \to r$$

- Both producer and consumer can be curried!
- Complete symmetry.

Recovering symmetry with continuation passing style

Type of parser in CPS:

$$(String \to a_1 \to \ldots \to a_n \to r) \to String \to r$$

Type of pretty printer CPS:

$$(String \to r) \to String \to a_1 \to \ldots \to a_n \to r$$

Type of parser in CPS:

$$(String \to a_1 \to \ldots \to a_n \to r) \to (String \to r)$$

Type of pretty printer in CPS:

$$(String \to r) \to (String \to a_1 \to \ldots \to a_n \to r)$$

- Both producer and consumer can be curried!
- Complete symmetry.

Composing parsers in CPS

$$
\begin{aligned}
f &:: (String \to b \to r_1) \to (String \to r_1) \\
g &:: (String \to a \to r_2) \to (String \to r_2) \\
f \circ g &:: (String \to a \to b \to r_1) \to (String \to r_1)
\end{aligned}
$$

Unification constraints: $r_2 = b \to r_1$.

Composing pretty printers in CPS (Danvy, 1998)

$$
\begin{aligned}
f' &:: (String \to r_1) \to (String \to b \to r_1) \\
g' &:: (String \to r_2) \to (String \to a \to r_2) \\
g' \circ f' &:: (String \to r_1) \to (String \to a \to b \to r_1)
\end{aligned}
$$

Unification constraints: $r_2 = b \to r_1$.

Putting it all together

$$\begin{aligned} K7\,f\,f' \circ K7\,g\,g' :: K7\,&((String \to a \to b \to r) \to (String \to r)) \\ &((String \to r) \to (String \to a \to b \to r)) \end{aligned}$$

$\{0,1\}$-parsers and $\{0,1\}$-printers

Existentially pack answer type:

$$\begin{aligned}
&\textbf{type}\ \mathit{PPP}\ a = \forall r.\ \mathit{K7}\ ((\mathit{String} \to a \to r) \to (\mathit{String} \to r)) \\
&\qquad\qquad\qquad\qquad\quad ((\mathit{String} \to r) \to (\mathit{String} \to a \to r)) \\
&\textbf{type}\ \mathit{PPP0} = \forall r.\ \mathit{K7}\ ((\mathit{String} \to r) \to (\mathit{String} \to r)) \\
&\qquad\qquad\qquad\qquad\quad ((\mathit{String} \to r) \to (\mathit{String} \to r))
\end{aligned}$$

- Not closed under composition!
- Compose n-parser with (pure) n-consumer to get 1-parser.
- Compose n-printer with (pure) n-producer to get 1-printer.
- Parser-consumer and printer-producer composition written using $(\longrightarrow)$ (alias for $(\Diamond)$, but with lower precedence).

Example: parsing and printing pairs

$$
\begin{aligned}
&\mathit{lit} :: \mathit{String} \rightarrow \mathit{PPP0} \\
&\mathit{lit}\ x = \mathit{K7}\ (\lambda k\ s \rightarrow \textbf{case}\ \mathit{stripPrefix}\ x\ s\ \textbf{of}\ \mathit{Just}\ s' \rightarrow k\ s') \\
&\qquad\qquad (\lambda k\ s \rightarrow k\ (x \mathbin{+\!\!+} s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{anyChar} :: \mathit{PPP}\ \mathit{Char} \\
&\mathit{anyChar} = \mathit{K7}\ (\lambda k\ s \rightarrow k\ (\mathit{tail}\ s)\ (\mathit{head}\ s))\ (\lambda k\ s\ x \rightarrow k\ ([x] \mathbin{+\!\!+} s))
\end{aligned}
$$

Example: parsing and printing pairs

$$
\begin{aligned}
&\mathit{lit} :: \mathit{String} \rightarrow \mathit{PPP0}\\
&\mathit{lit}\ x = \mathit{K7}\ (\lambda k\ s \rightarrow \mathbf{case}\ \mathit{stripPrefix}\ x\ s\ \mathbf{of}\ \mathit{Just}\ s' \rightarrow k\ s')\\
&\qquad\qquad (\lambda k\ s \rightarrow k\ (x +\!\!+\ s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{anyChar} :: \mathit{PPP}\ \mathit{Char}\\
&\mathit{anyChar} = \mathit{K7}\ (\lambda k\ s \rightarrow k\ (\mathit{tail}\ s)\ (\mathit{head}\ s))\ (\lambda k\ s\ x \rightarrow k\ ([x] +\!\!+\ s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{kpair} :: \mathit{K7}\ ((\mathit{String} \rightarrow (a, b) \rightarrow r) \rightarrow (\mathit{String} \rightarrow b \rightarrow a \rightarrow r))\\
&\qquad\qquad ((\mathit{String} \rightarrow b \rightarrow a \rightarrow r) \rightarrow (\mathit{String} \rightarrow (a, b) \rightarrow r))\\
&\mathit{kpair} = \mathit{K7}\ (\lambda k\ s\ y\ x \rightarrow k\ s\ (x,y))\ (\lambda k\ s\ (x,y) \rightarrow k\ s\ y\ x)
\end{aligned}
$$

Example: parsing and printing pairs

$$
\begin{aligned}
&\mathit{lit} :: \mathit{String} \to \mathit{PPP0} \\
&\mathit{lit}\ x = \mathit{K7}\ (\lambda k\ s \to \mathbf{case}\ \mathit{stripPrefix}\ x\ s\ \mathbf{of}\ \mathit{Just}\ s' \to k\ s') \\
&\qquad\qquad (\lambda k\ s \to k\ (x +\!\!+\ s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{anyChar} :: \mathit{PPP}\ \mathit{Char} \\
&\mathit{anyChar} = \mathit{K7}\ (\lambda k\ s \to k\ (\mathit{tail}\ s)\ (\mathit{head}\ s))\ (\lambda k\ s\ x \to k\ ([x] +\!\!+\ s))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{kpair} :: \mathit{K7}\ ((\mathit{String} \to (a, b) \to r) \to (\mathit{String} \to b \to a \to r)) \\
&\qquad\qquad ((\mathit{String} \to b \to a \to r) \to (\mathit{String} \to (a, b) \to r)) \\
&\mathit{kpair} = \mathit{K7}\ (\lambda k\ s\ y\ x \to k\ s\ (x, y))\ (\lambda k\ s\ (x, y) \to k\ s\ y\ x)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{pair} :: \mathit{PPP}\ (\mathit{Char}, \mathit{Char}) \\
&\mathit{pair} = \mathit{lit}\ \texttt{"("} \circ \mathit{anyChar} \circ \mathit{lit}\ \texttt{","} \circ \mathit{anyChar} \circ \mathit{lit}\ \texttt{")"} \longrightarrow \mathit{kpair}
\end{aligned}
$$

Choice

Choice for parsing/printing algebraic datatypes

- Need to add throwing and catching exceptions side effect:
  1. abort on malformed input.
  2. backtrack to last choice point if parsing/printing failure.
- Can model exceptions through the exception monad.
- Parsing is a monad.
  $\longrightarrow$ **can** *compose monads to compose effects.*
- Printing is not a monad.
  $\longrightarrow$ **cannot** *compose monads to compose effects.*

Choice for parsing/printing algebraic datatypes

- Need to add throwing and catching exceptions side effect:
  1. abort on malformed input.
  2. backtrack to last choice point if parsing/printing failure.
- Can model exceptions through the exception monad.
- Parsing is a monad.
  $\longrightarrow$ **can** *compose monads to compose effects.*
- Printing is not a monad.
  $\longrightarrow$ **cannot** *compose monads to compose effects.*
- Answer type must be polymorphic — cannot lift to monadic type:

$$f :: (String \to b \to m\ r_1) \to (String \to m\ r_1)$$
$$g :: (String \to a \to m\ r_2) \to (String \to m\ r_2)$$
$$f \circ g :: ??$$

  Unsatisfiable unification constraint: $m\ r_2 = b \to m\ r_1$.

Solution: CPS transform a second time!

- Obtain 2-CPS 1-parser and 1-printer. Types:

$$K7\ ((String \to a \to (r \to t) \to t) \to String \to (r \to t) \to t)$$
$$((String \to (r \to t) \to t) \to String \to a \to (r \to t) \to t)$$

- Now have a continuation *and* a meta-continuation.
- Pass continuation, meta-continuation first and make meta-continuation constant:

$$K7\ ((t \to String \to a \to t) \to t \to String \to t)$$
$$((t \to String \to t) \to t \to String \to a \to t)$$

- Cannot be composed! Infinite type during unification: $t = a \to t$.

Solution: CPS transform a second time!

- Must weaken meta-continuation argument of continuation of parser.
- Conversely, must strengthen meta-continuation argument of continuation of printer.
- Obtained type:

$$\begin{aligned}&K7\ (((a \to t) \to String \to a \to t) \to (t \to String \to t))\\&\quad ((t \to String \to t) \to ((a \to t) \to String \to a \to t))\end{aligned}$$

- Composition of cassettes is still pairwise functional composition of components, as before.

The choice combinator

$$\textbf{type}\ PPP\ a = \forall r.\ K7\ (((a \to t) \to String \to a \to t) \to (t \to String \to t))\ ((t \to String \to t) \to ((a \to t) \to String \to a \to t))$$

$$\begin{aligned}&(\oplus) :: PPP\ a \to PPP\ a \to PPP\ a\\ &K7\ f\ f' \oplus K7\ g\ g' =\\ &\quad K7\ (\lambda k\ k'\ s \to f\ k\ (g\ k\ k'\ s)\ s)\\ &\qquad (\lambda k\ k'\ s\ x \to f'\ k\ (g'\ k\ k'\ s)\ s\ x)\end{aligned}$$

- Reset meta-continuation (aka failure continuation) of f, f'.

Example: repeating cassettes

$$
\begin{aligned}
kcons = K7\ (&\lambda k\ k'\ s\ xs\ x \rightarrow k\ (const\ (k'\ xs\ x))\ s\ (x : xs))\\
(&\lambda k\ k'\ s\ xs \rightarrow \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad x : xs \rightarrow k\ (\lambda\_\ \_ \rightarrow k'\ xs)\ s\ xs\ x\\
&\qquad \_ \rightarrow k'\ xs)
\end{aligned}
$$

$$
\begin{aligned}
knil = K7\ (&\lambda k\ k'\ s \rightarrow k\ (const\ k')\ s\ [\,])\\
(&\lambda k\ k'\ s\ xs \rightarrow \mathbf{case}\ xs\ \mathbf{of}\\
&\qquad [\,] \rightarrow k\ (k'\ xs)\ s\\
&\qquad \_ \rightarrow k'\ xs)
\end{aligned}
$$

Example: repeating cassettes

$$
\begin{aligned}
kcons = K7\ (&\lambda k\ k'\ s\ xs\ x \rightarrow k\ (const\ (k'\ xs\ x))\ s\ (x : xs)) \\
(&\lambda k\ k'\ s\ xs \rightarrow \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad x : xs \rightarrow k\ (\lambda\_\ \_ \rightarrow k'\ xs)\ s\ xs\ x \\
&\qquad \_ \rightarrow k'\ xs)
\end{aligned}
$$

$$
\begin{aligned}
knil = K7\ (&\lambda k\ k'\ s \rightarrow k\ (const\ k')\ s\ [\,]) \\
(&\lambda k\ k'\ s\ xs \rightarrow \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad [\,] \rightarrow k\ (k'\ xs)\ s \\
&\qquad \_ \rightarrow k'\ xs)
\end{aligned}
$$

$$
\begin{aligned}
&many :: PPP\ a \rightarrow PPP\ [a] \\
&many\ ppp = (ppp \circ many\ ppp \longrightarrow kcons) \oplus knil
\end{aligned}
$$

- *many* is a derived combinator.
- Need lazy semantics to avoid non-termination.
- Essential use of answer type polymorphism.

Playing cassettes

$$
\begin{aligned}
&\mathit{play} :: (\mathit{K7}\ a\ b \to c) \to \mathit{K7}\ a\ b \to c\\
&\mathit{play}\ f\ \mathit{csst} = f\ \mathit{csst}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{parse} :: \mathit{PPP}\ a \to \mathit{String} \to \mathit{Maybe}\ a\\
&\mathit{parse}\ \mathit{csst} = \mathit{play}\ \mathit{sideA}\ \mathit{csst}\ (\lambda\_\ \_\ x \to \mathit{Just}\ x)\ \mathit{Nothing}\\
&\mathit{pretty} :: \mathit{PPP}\ a \to a \to \mathit{Maybe}\ \mathit{String}\\
&\mathit{pretty}\ \mathit{csst} = \mathit{play}\ \mathit{sideB}\ \mathit{csst}\ (\lambda\_\ s \to \mathit{Just}\ s)\ (\mathit{const}\ \mathit{Nothing})\ \text{""}
\end{aligned}
$$

Conclusion

Literature

- "Functional unparsing" (Danvy, 1998)
  ⟶ *CPS, only printf, no ADTs.*
- "There and back again" (Alimarine et al., 2005)
  ⟶ *arrows, needs binary encoding of alternatives, arrows must respect isomorphism laws.*
- "Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing" (Rendel and Ostermann, 2010)
  ⟶ *applicative functor but not quite, packs all arguments in nested tuples.*

Future work

- Fix order of arguments.
- Implementation in direct style.
- Port all Parsec combinators to cassette framework.
- Study initial vs final.