An introduction to

Haskell2LaTeX

Ian Lynagh

October 2002

Literate programming

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: “Literate Programming.”

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

– Donald E. Knuth, “Literate Programming”, 1984

Objective

Transform this:

```
\begin{code}
qsort :: [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort xs_lt_x ++ [x] ++ qsort xs_greq_x
    where xs_lt_x   = [y | y <- xs, y < x]
          xs_greq_x = [y | y <- xs, y >= x]
\end{code}
```

into this:

$$qsort :: [\alpha] \longrightarrow [\alpha]$$

$$qsort\ [\,] \quad = [\,]$$

$$qsort\ (x{:}xs) = qsort\ xs\_lt\_x + \!\!+ \left[x\right] + \!\!+ qsort\ xs\_greq\_x$$

$$\textbf{where}\ xs\_lt\_x = \left[y \mid y \leftarrow xs, y < x\right]$$

$$xs\_greq\_x = \left[y \mid y \leftarrow xs, y \geq x\right]$$

Haskell

Haskell is...

- the language to be pretty-printed
  - The source format has some support for literate programming
- the language the pretty-printer will be written in
  - Well suited to parsing and tree manipulation
  - Could you seriously consider anything else?

TeX and LaTeX

- Literate Haskell makes targetting the TeX family slightly easier

```
\usepackage{verbatim}
\newenvironment{code}{\verbatim}{\endverbatim}
```

- The TeX family have the mathematical symbols commonly used in typesetting Haskell, e.g. $\longrightarrow$, $\geq$
- Using LaTeX allows you to write your documentation in exactly the same way as you (probably!) write your papers etc.

Parsing

Haskell is...

- Much of the art and science of parsing dates from the 70s and 80s
  - Lexical analysis then syntactic analysis
  - Top-down vs botom-up
- Parser combinators provide a nice way of writing top-down parsers in HOT languages
  - Take a list of symbols and return a value
  - Provide primitive parsers, e.g. Fail, Succeed, Match
  - Provide a number of combinators, e.g. Choice, Sequential Composition

Existing pretty-printers

- Haskell Style for LaTeX2e
  - For manual typesetting
- `haskell.sty`
  - Easily confused, e.g. `<=>` pretty-prints as $\leq>$
- λTeX
  - Forces certain styles on you or indents incorrectly
- `lhs2tex`
  - The most impressive, but still limited by only lexing the input
- Related programs
  - `detex`, `enscript`, HDoc, HaskellDoc, The Haskell Module Browser

Haskell2LaTeX Structure

- Parsing Haskell
  - Unlitting
  - Lexing
  - Layout rule
  - Fixity annotation
  - Syntax analysing
- Pretty-printing the abstract syntax tree

Unlitting

- *unlit* :: Bool ⟶ String ⟶ [Either String (String, [(Char, Position)], Position)]
- `.hs` files:
  - Do nothing!
- `.lhs` files:
  - Extract the contents of `\begin{code}`... `\end{code}` blocks
  - Take all lines beginning with > and replace the leading > with a space
  - Keep the position info!

Parser Combinator library types

- **type** Parser α = String $\longrightarrow$ Maybe (α, Int, String)
- **type** Parser α β = [α] $\longrightarrow$ Maybe (β, Int, [α])
- **type** Parser α β = [(α, Position)] $\longrightarrow$ Parsed α β

  data Parsed α β = Success β Int [(α, Position)]
  | Failure Position

Parser Combinator library

- Primitives:

  $\mathit{pPred} :: (\alpha \rightarrow \text{Bool}) \rightarrow \text{Parser}\ \alpha\ \alpha$

  $\mathit{pFail} :: \text{Parser}\ \alpha\ \beta$

  $\mathit{pSucceed} :: \beta \rightarrow \text{Parser}\ \alpha\ \beta$

  $(<|>) :: \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \beta$

  $(<*>) :: \text{Parser}\ \alpha\ (\beta \rightarrow \gamma) \rightarrow \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \gamma$

  $(<!>) :: \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \gamma \rightarrow \text{Parser}\ \alpha\ \beta$

- Derived:

  $(<*) :: \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \gamma \rightarrow \text{Parser}\ \alpha\ \beta$

  $(<\$>) :: (\beta \rightarrow \gamma) \rightarrow \text{Parser}\ \alpha\ \beta \rightarrow \text{Parser}\ \alpha\ \gamma$

  $(<\$) :: \beta \rightarrow \text{Parser}\ \alpha\ \gamma \rightarrow \text{Parser}\ \alpha\ \beta$

Parser Combinator library (cont.)

- Larger parsers:

  pSym :: Eq $\alpha \Rightarrow \alpha \longrightarrow$ Parser α α

  pSyms :: Eq $\alpha \Rightarrow [\alpha] \longrightarrow$ Parser α $[\alpha]$

  opt :: Parser α $\beta \longrightarrow \beta \longrightarrow$ Parser α β

  pPredHolds :: ($\alpha \longrightarrow$ Bool) $\longrightarrow$ Parser α Bool

  pMaybe :: Parser α $\beta \longrightarrow$ Parser α (Maybe β)

  pList :: Int $\longrightarrow$ Parser α $\beta \longrightarrow$ Parser α $[\beta]$

  pSList :: Bool $\longrightarrow$ Int $\longrightarrow$ Parser α $\beta \longrightarrow$ Parser α $\gamma \longrightarrow$ Parser α ([Either β γ])

- Position handling:

  keep_pos :: Parser α $\beta \longrightarrow$ Parser α (β, Position)

  ($<$&$>$) :: ($\beta \longrightarrow \gamma$) $\longrightarrow$ Parser α $\beta \longrightarrow$ Parser α (γ, Position)

  ($<$&) :: $\beta \longrightarrow$ Parser α $\gamma \longrightarrow$ Parser α (β, Position)

Derived parsers implementation

```
opt :: Parser α β → β → Parser α β
p `opt` v = p <|> if accepts_empty p
                  then pFail
                  else pSucceed v

pList :: Int → Parser α β → Parser α [β]
pList _ p
    | accepts_empty p = error "Tried to make a list of empties"
pList 0 p      = pList 1 p `opt` []
pList (i +1) p = (:) <$> p <*> pList i p
```

Problem with position detection

```
Hugs session for:
/usr/share/hugs98/lib/Prelude.hs
Position.lhs
PCbase.lhs
PC.lhs
PC> id <$ pSym '{' <*> pList 0 (pSyms "abcd") <* pSym '}' $ posify "{abcdabcdabcd}"
Success ["abcd","abcd","abcd"] 14 []
PC> id <$ pSym '{' <*> pList 0 (pSyms "abcd") <* pSym '}' $ posify "{abcdabcdabd}"
Failure (Position 1 10)
```

Solution: Fix the datastructure

data Parsed α β = Success (β, Int, [(α, Position)], Position)
| Failure Position
type Parser α β = [(α, Position)] $\longrightarrow$ Parsed α β

pFail :: Parser α β
pFail = λ *inp* $\longrightarrow$ Failure (*get_pos inp*)

pSucceed :: β $\longrightarrow$ Parser α β
pSucceed v = λ *inp* $\longrightarrow$ Success (*v*, 0, *inp*, *get_pos inp*)

pPred :: (α $\longrightarrow$ Bool) $\longrightarrow$ Parser α α
pPred f = λ *inp* $\longrightarrow$ **case** *inp* **of**
[ ] $\longrightarrow$ Failure *end_of_file*
((*t*, *p*):*inp'*) $\longrightarrow$ **if** *f t*
then Success (*t*, 1, *inp'*, *get_pos inp'*)
else Failure *p*

```
(<|>) :: (Show α) ⇒ Parser α β → Parser α β → Parser α β
p <|> q = λ inp →
          case (p inp, q inp) of
               (Success s1, Success s2) → case num s1 `compare` num s2 of
                                              GT → succeed s1 (pos s2)
                                              LT → succeed s2 (pos s1)
                                              EQ → Failure (get_pos inp)
               (Success s1, Failure pos2) → succeed s1 pos2
               (Failure pos1, Success s2) → succeed s2 pos1
               (Failure pos1, Failure pos2) → Failure $! fp
                   where fp = furthest_pos pos1 pos2
    where succeed (v, n, inp, pos) pos' = fp `seq` Success (v, n, inp, fp)
              where fp = furthest_pos pos pos'

(<*>) :: Parser α (β → γ) → Parser α β → Parser α γ
p1 <*> p2 = λ inp →
            case p1 inp of
                 Failure pos1 → Failure pos1
                 Success s1 → case p2 (rinp s1) of
                                  Failure pos2 → Failure $! fp
                                      where fp = furthest_pos (pos s1) pos2
                                  Success s2 → Success (seq_comp s1 s2)
    where seq_comp (v1, n1, _, pos1) (v2, n2, r, pos2) = fp `seq` (v1 v2, n1 + n2, r, fp)
              where fp = furthest_pos pos1 pos2
```

Optimisation 1

```
(<|) :: Parser α β → Parser α β → Parser α β
p1 <| p2 = λ inp →
            case (p1 inp, p2 inp) of
                 (Success r1, _) → Success r1
                 (Failure pos1, Success (v2, n2, inp2, pos2)) → fp `seq` Success (v2, n2, inp2, fp)
                     where fp = furthest_pos pos1 pos2
                 (Failure pos1, Failure pos2) → Failure $! fp
                     where fp = furthest_pos pos1 pos2

(|>) :: Parser α β → Parser α β → Parser α β
p1 |> p2 = p2 <| p1
```

Optimisation 2

```
type SParsed α β = (β, Int, [(α, Position)], Position)
data Parsed α β = Success (SParsed α β)
                | Failure Position
type SParser α β = [(α, Position)] → SParsed α β
type Parser α β = [(α, Position)] → Parsed α β

pList :: Int → Parser α β → Parser α [β]
pList 0 p      = pList0 p
pList (i +1) p = (:) <$> p <*> pList i p

pList0 :: Parser α β → Parser α [β]
pList0 p = λ inp → Success (pListS p inp)

pListS :: Parser α β → SParser α [β]
pListS p = λ inp →
          case p inp of
              Success (v1, n1, inp1, pos1) → fp `seq` (v1:v2, n1 + n2, inp2, fp)
                  where (v2, n2, inp2, pos2) = pListS p inp1
                        fp = furthest_pos pos1 pos2
              Failure pos → ([ ], 0, inp, pos)
```

Lexing

$$ascDigit \quad \rightarrow \quad 0 \mid 1 \mid \ldots \mid 9$$

```
lexc_ascDigit :: Parser Char Char
lexc_ascDigit = pPred isDigit
```

$$charesc \quad \rightarrow \quad a \mid b \mid f \mid n \mid r \mid t \mid v \mid \backslash \mid \text{"} \mid \text{'} \mid \&$$

```
lexs_charesc :: Parser Char String
lexs_charesc = wrap <$> lexc_charesc

lexc_charesc :: Parser Char Char
lexc_charesc = pSym 'a' <| pSym 'b' <| pSym 'f' <| pSym 'n' <| pSym 'r' <| pSym 't' <| pSym 'v' <|
               pSym '\\' <| pSym '"' <| pSym '\'' <| pSym '&'
```

More lexing

$$
\begin{array}{lcl}
\mathit{varid} & \rightarrow & (\mathit{small}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \text{'}\})_{<\mathit{reservedid}>} \\
\mathit{conid} & \rightarrow & \mathit{large}\ \{\mathit{small} \mid \mathit{large} \mid \mathit{digit} \mid \text{'}\}
\end{array}
$$

id_body :: Parser Char Char
id_body = *lexc_small* <| *lexc_large* <| *lexc_digit* <| *pSym* '\''

lexs_varid :: Parser Char String
lexs_varid = (:) <$> *lexc_small* <*> *pList* 0 *id_body* <!> *lext_reservedid*

lexs_conid :: Parser Char String
lexs_conid = (:) <$> *lexc_large* <*> *pList* 0 *id_body*

$$
\mathit{reservedop} \quad \rightarrow \quad \texttt{..} \mid \texttt{:} \mid \texttt{::} \mid \texttt{=} \mid \texttt{\textbackslash} \mid \texttt{|} \mid \texttt{<-} \mid \texttt{->} \mid \texttt{@} \mid \texttt{\textasciitilde} \mid \texttt{=>}
$$

lext_reservedop :: Parser Char (Token, Position)
lext_reservedop = ReservedOp <&> *lexs_reservedop*

lexs_reservedop :: Parser Char String
lexs_reservedop = *pSyms* ".." |> *pSyms* ":" |> *pSyms* "::" |> *pSyms* "=" |> *pSyms* "\\" |> *pSyms* "|"
|> *pSyms* "<−" |> *pSyms* "−>" |> *pSyms* "@" |> *pSyms* "~" |> *pSyms* "=>"

Layout rule

2 stages:

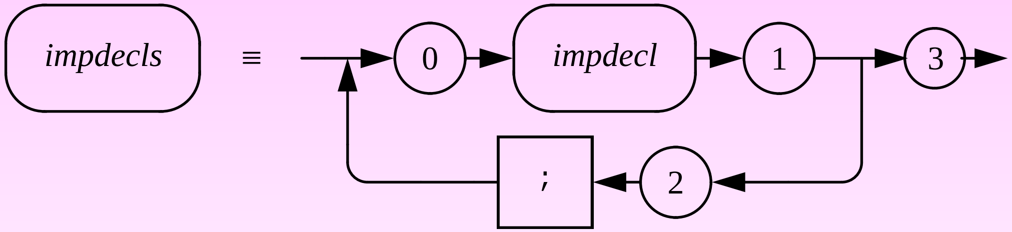
- ☞ Indent marking: Almost-code given in the report
- ☞ Applying the rule: Almost-code given in the report

But! One of the conditions is parse-error(t)!

Solution: Make an automaton for the (slightly simplified) Haskell grammar and step it each symbol

Automaton constructed automatically from a CFG-like description

Can also automatically create diagrams of the sub-automata, e.g.:



Fixity information

- Conceptually very simple
- Walk through the tree of blocks looking for fixity declarations, then apply them to the block and all child blocks
- Current code is a mess, but could be tidied up and made more efficient

Syntactic analysis

Similar to lexical analysis, but need to optimise a number of cases, e.g.:

Report:

$$\begin{array}{lcl} exp & \rightarrow & exp0 :: [context =>]\ type \\ & | & exp0 \end{array}$$

Optimised:

$$exp \rightarrow exp0\ [:: [context =>]\ type]$$

Pretty-printing

Straightforward recursion over the abstract syntax tree

```
s_op ">=" = "$\\ge$"

s_tyvar "a" = "$\\alpha$"

text = concatMap escape

escape '^' = "\\^{}"

pp_VarID v = "\\textit{" ++ text v ++ "\\/}"

pp_Idecl (Idecl_var var rhs, semi) = pp_Var var ++ pp_Rhs rhs semi

pp_Fixity Fixity_infixl = "\\HaskellToLaTeXkeyword{infixl}"

pp_Gd (Gd exp_i) = "\\HaskellToLaTeXguard{" ++ pp_Exp_i exp_i ++ "}"
```

Some numbers

Size	CPU time taken for lexical analysis
355k	0m44.240s
710k	1m29.760s
1420k	3m07.750s

Using normal or optimised *pList*	CPU time taken
Normal	10m39.840s
Optimised	6m16.180s

Source size	CPU time taken	Processing speed (variable size heap)
44k	1m20.600s	551 bytes per second
88k	2m43.620s	543 bytes per second
176k	5m31.870s	535 bytes per second
352k	11m14.850s	526 bytes per second
704k	23m24.520s	505 bytes per second

Source size	CPU time taken	Processing speed (fixed size heap (300M))
44k	1m14.170s	598 bytes per second
88k	2m25.540s	610 bytes per second
176k	4m52.170s	608 bytes per second
352k	9m42.200s	610 bytes per second
704k	19m22.840s	611 bytes per second

The path

Original input:

```
foo x = y + z
    where y = 2 * x
          z = x + 3
```

After indent marking:

```
{1}foo x = y + z
    <5>where {11}y = 2 * x
          <11>z = x + 3
```

After layout:

```
{foo x = y + z
    where {y = 2 * x
          ;z = x + 3
}}
```

Parseable tokens:

```
{foox=y+zwhere{y=2*x;z=x+3}}
```

Code:

foo x = *y* + *z*
    where *y* = 2 × *x*
        z = *x* + 3

That's all folks!