lhs2TEX

Andres Löh

Dutch HUG meeting – October 12, 2009

What lhs2TEX is ...

A preprocessor.

- Input: a LaTeX document containing directives and Haskell-like code.
- Output: a LaTeX document where the code is formatted as LaTeX as well; or Haskell code that can be run.

What lhs2TEX is ...

A preprocessor.

- Input: a LaTeX document containing directives and Haskell-like code.
- Output: a LaTeX document where the code is formatted as LaTeX as well; or Haskell code that can be run.

Useful for:

- LaTeX documents containing Haskell code – papers, documentation, presentations, ...
- LaTeX documents containing other kinds of aligned code
- many things you might want a Haskell preprocessor for
- managing different versions of a document

What lhs2TEX is not . . .

- The conversion is *not* fully automatic.
- You have lots of freedom, but you have to make some choices.

Hello world

Input

```
\documentclass{article}
%include polycode.fmt
\begin{document}

> main = putStrLn "Hello world"

\end{document}
```

Output

$$main = putStrLn\ \texttt{"Hello world"}$$

History

- Created by Ralf Hinze in 1997. Most of the functionality is due to Ralf.
- I picked up development in 2002. New features:
  - better code alignment,
  - using lhs2TEX as a preprocessor also to generate code,
  - improved possibilities of calling GHC from within a document.

Inline code

Inline code is surrounded by vertical bars.

Input

```
The function |map| takes two arguments,
a function |f :: a -> b| and a value |x| of type |a|.
```

Output

The function *map* takes two arguments, a function $f :: a \to b$ and a value x of type a.

Inline code – contd.

Vertical bars occurring in inline code have to be escaped.

Input

```
The function |or| can be defined using |foldr|,
namely as |foldr (||||) True|.
```

Output

The function *or* can be defined using *foldr*, namely as *foldr* $(\vee)$ *True*.

As can be seen, some operators are by default formatted as symbols.

Haskell syntax

The parser is very liberal. It only approximates the Haskell syntax. Generally, Haskell constructs should be typeset nicely.

Input

```
|let x = 2 in x * x|\par
|case x of Foo -> Bar|\par
|[ x * x || x <- xs ]|\par
|\ x -> x x|
```

Output

let $x = 2$ **in** $x * x$
case x **of** $Foo \rightarrow Bar$
$[x * x \mid x \leftarrow xs]$
$\lambda x \rightarrow x\ x$

Verbatim

Much as inline Haskell, we can also produce verbatim code by surrounding it in @s. Again, escaping other @s is necessary.

Input

```
Typing @foo@ yields |foo|. Here's an escaped @@.
```

Output

Typing `foo` yields *foo*. Here's an escaped @.

Formatting

Directives are lines starting with a % immediately followed by a recognized lhs2TEX command. The directive `%format` can be used to change the appearance of tokens.

Input

```
%format True  = "\top "
%format foldr = "{\color{blue}\textbf{foldr}} "
|foldr (||||) True|
```

Output

foldr ($\vee$) $\top$

Formatting – contd.

Formatting directives can also be used to undo predefined formattings. The default formatting of variables and constructors makes use of `\Varid` and `\Conid`, respectively.

Input

```
|not x|
%format not = "\Varid{not}"
|not x|
\let\Varid\mathbf
|not x|
```

Output

¬ *x*, *not x*, **not x**

Implicit formatting

For indices there are special cases where no right hand side has to be given. The directive itself is still required.

Input

```
|a1|, |a_1|, |a_not|

%format a1
%format a_1
%format a_not
|a1|, |a_1|, |a_not|
```

Output

$a1$, a_1, a_not

a_1, a_1, $a_\neg$

Parameterized formatting

Input

```
%format <> = "\diamond "
%format Instr x = "{\let\Conid\texttt " x "}"
%format eval x = "\llbracket " x "\rrbracket "

> eval (Add x y) = eval x <> eval y <> [Instr ADD]
```

Output

$$[\![(Add\ x\ y)]\!] = [\![x]\!] \diamond [\![y]\!] \diamond [\mathtt{ADD}]$$

Too many parentheses!

Parameterized formatting and parentheses

Input

```
%format eval (x) = "\llbracket " x "\rrbracket "

> (eval (Add x y))

%format (eval (x)) = "\llbracket " x "\rrbracket "

> (eval (Add x y))
```

Output

$(\llbracket Add\ x\ y \rrbracket)$

$\llbracket Add\ x\ y \rrbracket$

Blocks of code

Blocks of code can be typeset using a code-environment or by prefixing every line with a >:

Input

```
This is a |let| expression:

> let x = 2
> in x * x
```

Output

This is a **let** expression:

let $x = 2$
in $x * x$

Blocks of code – contd.

Input

```
This is a |let| expression:
\begin{code}
let x = 2
in x * x
\end{code}
```

Output

This is a **let** expression:

let $x = 2$
in $x * x$

Unused code

Code starting with < or in a spec-environment is also typeset – for code that should be included in the output, but not run.

Input

```
This is a |let| expression:
\begin{spec}
let x = 2
in x * x
\end{spec}
```

Output

This is a **let** expression:

let $x = 2$
in $x * x$

Comments are text

Comments are typeset as text. Use < or spec for larger blocks of commented code that should be shown.

Input

```
> 0 :: Num a => a -- not of type |Int|, but overloaded
```

Output

$0 :: \mathit{Num}\ a \Rightarrow a$ -- not of type *Int*, but overloaded

Alignment

Alignment is lhs2TEX's strong point: a token that is prefixed by two or more spaces is aligned with other tokens on the same column.

Input

```
> map  f  []      =  []
> map  f  (x:xs)  =  f x : map f xs
```

Output

$$
\begin{aligned}
&map\ f\ []\ \ \ \ \ \ \ = []\\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

Alignment and Indentation

Indentation is with respect to aligned columns.

Input

```
%format ... = "\dots "

> instance (Ord a) => Ord [a] where
>   ...
>   compare (x:xs)  (y:ys)  =  case compare x y of
>                                EQ     ->  compare xs ys
>                                other  ->  other
```

Output

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Ord}\ a) \Rightarrow \mathit{Ord}\ [a]\ \textbf{where} \\
\quad \ldots \\
\quad \mathit{compare}\ (x : xs)\ (y : ys) = \textbf{case}\ \mathit{compare}\ x\ y\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{EQ} \quad \rightarrow \mathit{compare}\ \mathit{xs}\ \mathit{ys} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{other} \rightarrow \mathit{other}
\end{array}
$$

Alignment – contd.

Alignment does not have to affect subsequent lines.

Input

```
> consTree a (Deep s (Two b c)  m sf) =
>   Deep (size a + s) (Three  a b c)  m sf
> consTree a (Deep s (One b)    m sf) =
>   Deep (size a + s) (Two    a b)    m sf
```

Output

$$
\begin{aligned}
&\mathit{consTree}\ a\ (\mathit{Deep}\ s\ (\mathit{Two}\ b\ c)\ m\ \mathit{sf}) = \\
&\quad \mathit{Deep}\ (\mathit{size}\ a + s)\ (\mathit{Three}\ a\ b\ c)\ m\ \mathit{sf} \\
&\mathit{consTree}\ a\ (\mathit{Deep}\ s\ (\mathit{One}\ b)\quad m\ \mathit{sf}) = \\
&\quad \mathit{Deep}\ (\mathit{size}\ a + s)\ (\mathit{Two}\quad a\ b)\quad m\ \mathit{sf}
\end{aligned}
$$

Watch out that code is not aligned by accident!

Alignment – contd.

Alignment is computed by LaTeX, using the `polytable` package that was written specifically for lhs2TeX.

Input

```
%format i = "\Varid{iiiiiiiiiiiiii}"

> xxx  yyy  zzz
> aaaaa  bbbb
> i    jjjjjjj
> c      dddd
```

Output

xxx yyy zzz
aaaaa bbbb
iiiiiiiiiiiiii jjjjjjj
c dddd

Reusing alignment

Alignment information can be shared for multiple code blocks.

Input

```
\savecolumns

> eval (Const n)     =  n
> eval (Neg x)       =  - (eval x)

And now addition:\restorecolumns

> eval (Add x y)     =  eval x + eval y
```

Output

$$[\![Const\ n]\!] = n$$
$$[\![Neg\ x]\!] = -[\![x]\!]$$

And now addition:

$$[\![Add\ x\ y]\!] = [\![x]\!] + [\![y]\!]$$

Including files

Using an `%include` directive, a file can be included. This is used for `.fmt` files that contain lhs2TEX libraries, but can be used for parts of the document instead of LaTeX commands.

Input

```
%include polycode.fmt
```

There are a number of useful files shipped with lhs2TEX.

Libraries

- `polycode.fmt` – standard library
- `colorcode.fmt` – some code styles using colored backgrounds
- `greek.fmt` – format greek identifiers
- `forall.fmt` – universal quantifier magic
- `spacing.fmt` – spacing hacks

Libraries – standard

Using the lhs2TEX standard library, you can easily adapt the look and feel of lhs2TEX.

Input

```
\renewcommand\hscodestyle{\small\rmfamily}

> foldr op e []       =  []
> foldr op e (x:xs)   =  x `op` foldr op e xs
```

Output

$$
\begin{aligned}
&\mathit{foldr}\ \mathit{op}\ e\ [\,] &&= [\,]\\
&\mathit{foldr}\ \mathit{op}\ e\ (x : \mathit{xs}) &&= x\ \text{`}\mathit{op}\text{`}\ \mathit{foldr}\ \mathit{op}\ e\ \mathit{xs}
\end{aligned}
$$

Libraries – standard

Input

```
\framedhs

> foldr op e []       =  []
> foldr op e (x:xs)   =  x `op` foldr op e xs
```

Output

$$
\begin{aligned}
&\mathit{foldr}\ \mathit{op}\ e\ [\,] &&= [\,] \\
&\mathit{foldr}\ \mathit{op}\ e\ (x : \mathit{xs}) &&= x\ `\mathit{op}`\ \mathit{foldr}\ \mathit{op}\ e\ \mathit{xs}
\end{aligned}
$$

Libraries – standard

Sometimes you want to have code as part of the module and still show it inline.

Input

```
We therefore define
\inlinehs

> mapM f = sequence . map f

and are done.
```

Output

We therefore define $mapM\ f = sequence \circ map\ f$ and are done.

Libraries – Greek identifiers

Input

```
%include greek.fmt

> gamma = alpha + beta
```

Output

$$\gamma = \alpha + \beta$$

Libraries – universal quantifier magic

If you use Haskell code with explicit quantifiers, you probably want to include `forall.fmt`:

Input

```
%include forall.fmt

> mapM :: forall m. (Monad m)  =>  (a -> m b)
>                              ->  [a] -> m [b]
> mapM f = sequence . map f
```

Output

$$\begin{aligned} &mapM :: \forall m.(Monad\ m) \Rightarrow (a \to m\ b) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \to [a] \to m\ [b] \\ &mapM\ f = sequence \circ map\ f \end{aligned}$$

Note the different formatting of the periods.

Conditionals

There are directives `%if`, `%else`, `%elif` and `%endif` that can be used to process parts of the document conditionally.

- Documentation, paper, presentation from the same sources.
- Process differently depending on mode.

Using `%let` or command line flags, we can set variables to boolean or integer values.

Testing current mode

Input

```
%if style == newcode
%format (RED (x)) = x
%else
%format (RED (x)) = "{\color{red}" x "}"
%endif

> return (12 + RED x)
```

Output

$$\mathit{return}\ (12 + x)$$

You can annotate your code and still run it.

What else?

- By using formatting directives and conditionals, you can typecheck your documents.
- Lhs2TEX is not limited to displaying Haskell code. Using formatting directives, you can use it to display a wide range of languages.

How to get it

- Current version is 1.14.
- Available from Hackage (i.e., `cabal install lhs2tex`).
- Version 1.15 should appear soon (mainly interesting for Windows users).
- Let me know if you're doing something cool with lhs2TEX.