Typesetting Haskell and more with `lhs2TeX`

Andres Löh
Universiteit Utrecht
andres@cs.uu.nl

September 8, 2004

About lhs2TeX

- lhs2TeX is a preprocessor
  - Input: a literate Haskell source file
  - Output: a formatted file, depending on style of operation
- Possible input:

```
\documentclass{article}
%include lhs2TeX.fmt
%include lhs2TeX.sty
\begin{document}
This is the famous ‘‘Hello world’’ example,
written in Haskell:
\begin{code}
main  ::  IO ()
main  =   putStrLn "Hello, world!"
\end{code}
\end{document}
```

Hello, world!

- `lhs2TeX` is a preprocessor
  - Input: a literate Haskell source file
  - Output: a formatted file, depending on selected style
- Possible output:

This is the famous "Hello world" example, written in Haskell:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

- From input to output:

```
$ lhs2TeX --poly HelloWorld.lhs > HelloWorld.tex
$ pdflatex HelloWorld.tex
```

Styles

- `lhs2TeX` has several styles with different behaviour:
  - **verb** (verbatim): format code completely verbatim
  - **tt** (typewriter): format code verbatim, but allow special formatting of keywords, characters, some functions, ...
  - **math**: mathematical formatting with basic alignment, highly customizable
  - **poly**: mathematical formatting with mutliple alignments, highly customizable, supersedes **math**
  - **code**: delete all comments, extract sourcecode
  - **newcode** (new code): delete all comments, extract sourcecode, but allow for formatting, supersedes **code**

Example of "verb" style

```
zip                        :: [a] -> [b] -> [(a,b)]
zip                        =  zipWith  (\a b -> (a,b))

zipWith                    :: (a->b->c) -> [a]->[b]->[c]
zipWith z (a:as) (b:bs)    =  z a b : zipWith z as bs
zipWith _ _      _         =  []

partition                  :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs             =  foldr select ([],[]) xs
   where select x (ts,fs) | p x       = (x:ts,fs)
                          | otherwise = (ts,x:fs)
```

Example of “tt” style

```
zip                          :: [a] → [b] → [(a,b)]
zip                          =  zipWith  (λa b → (a,b))

zipWith                      :: (a→b→c) → [a]→[b]→[c]
zipWith z (a:as) (b:bs)      =  z a b : zipWith z as bs
zipWith _ _      _           =  []

partition                    :: (a → Bool) → [a] → ([a],[a])
partition p xs               =  foldr select ([],[]) xs
   where select x (ts,fs) | p x       = (x:ts,fs)
                          | otherwise = (ts,x:fs)
```

Differences from **verb** style:

- ➙ Some of Haskells symbols can be expressed more naturally.
- ➙ Keywords can be highlighted.

Drawback of formatting

```
zip :: [a] → [b] → [(a,b)]
zip = zipWith (λa b → (a,b))

zipWith :: (a → b → c) → [a] → [b] → [c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []

partition :: (a → Bool) → [a] → ([a],[a])
partition p xs = foldr select ([],[]) xs
    where select x (ts,fs) | p x = (x:ts,fs)
                           | otherwise = (ts,x:fs)
```

➙ Alignment information is lost.

Example of "math" style

$$
\begin{array}{lcl}
zip & :: & [a] \to [b] \to [(a,b)] \\
zip & = & zipWith\ (\lambda a\ b \to (a,b)) \\
zipWith & :: & (a \to b \to c) \to [a] \to [b] \to [c] \\
zipWith\ z\ (a:as)\ (b:bs) & = & z\ a\ b : zipWith\ z\ as\ bs \\
zipWith\ \_\ \_\ \_ & = & [\,] \\
partition & :: & (a \to Bool) \to [a] \to ([a],[a]) \\
partition\ p\ xs & = & foldr\ select\ ([\,],[\,])\ xs \\
\quad \textbf{where}\ select\ x\ (ts,fs) & \multicolumn{2}{l}{\mid p\ x = (x:ts,fs)} \\
 & \multicolumn{2}{l}{\mid otherwise = (ts,x:fs)}
\end{array}
$$

➙ Only one alignment column, plus indentation.

Example of "poly" style

$$
\begin{array}{lll}
zip & :: & [a] \to [b] \to [(a,b)] \\
zip & = & zipWith\ (\lambda a\ b \to (a,b)) \\
zipWith & :: & (a \to b \to c) \to [a] \to [b] \to [c] \\
zipWith\ z\ (a:as)\ (b:bs) & = & z\ a\ b : zipWith\ z\ as\ bs \\
zipWith\ \_\ \_\ \_ & = & [\,] \\
partition & :: & (a \to Bool) \to [a] \to ([a],[a]) \\
partition\ p\ xs & = & foldr\ select\ ([\,],[\,])\ xs \\
\quad \textbf{where}\ select\ x\ (ts,fs) & \mid & p\ x \quad = (x:ts,fs) \\
 & \mid & otherwise = (ts,x:fs)
\end{array}
$$

➞ Complex layouts are possible.

History of `lhs2TeX`

- Ralf Hinze started development in 1997. Most of the hard work has been done by him!
- The program is based on `smugweb` and `pphs`, both of which are no longer available and I do not know.
- I picked up development in 2002, and added the **poly** and **newcode** styles.

`lhs2TeX` operation

- When given an input file, `lhs2TeX` does only look at the following constructs:
  - Directives.
  - Text between two `@` characters. Such text is considered inline verbatim. Any `@` in the source file needs to be escaped: `@@`.
  - Text between two `|` characters. Such text is considered inline code.
  - Lines indicating a Bird-style literate program (i.e. lines beginning with either > or <) are considered as code blocks.
  - Lines surrounded by `\begin{code}` and `\end{code}` statements, or by `\begin{spec}` and `\end{spec}` statements, are considered as code blocks.
- Everything else is considered plain text and either ignored (for **verb**, **tt**, **math**, and **poly**) or discarded (for **code** and **newcode**).

Directives

- `lhs2TeX` interprets a number of directives.
- Directives can occur on all non-code lines and start with a %, the TEX comment character, immediately followed by the name of the directive, plus a list of potential arguments.
- These are the directives we will learn about in this talk:

```
%include
%format
%{
%}
%let
%if
%else
%elif
%endif
%latency
%separation
%options
```

Including files

- Other files can be included by `lhs2TeX`.

  `%include` ⟨*filename*⟩

- Using `%include`, not only other sources, but also other directives can be included.
- The specified file is searched for in the `lhs2TeX` source path which can be modified using environment variables or the `-P` command line option.
- Included files are inserted literally at the position of the `%include` directive. The `lhs2TeX` inclusion is therefore entirely independent of TEX or Haskell includes/imports.

The `lhs2TeX` "prelude"

- Several aspects of the behaviour of `lhs2TeX` are not hardcoded, but configurable via directives.
- A minimal amount of functionality has to be defined so that `lhs2TeX` can operate usefully.
- Essential definitions are collected in two files, `lhs2TeX.fmt` (containing basic directives) and `lhs2TeX.sty` (containing basic LaTeX setup). These two files should be included – directly or indirectly – in every file to be processed by `lhs2TeX`!

```
%include lhs2TeX.fmt
%include lhs2TeX.sty
```

- It is perfectly possible to design own libraries that replace or extend these basic files and to include those own libraries instead.

Formatting

- Using the `%format` directive, tokens can be given a different appearance.

`%format` ⟨*token*⟩ `=` ⟨*fmttoken*⟩* (format single tokens)
`%format` ⟨*lhs*⟩ `=` ⟨*fmttoken*⟩* (parametrized formatting)
`%format` ⟨*name*⟩ (implicit formatting)

⟨*lhs*⟩ ::= ⟨*name*⟩ ⟨*arg*⟩* | `(`⟨*name*⟩`)` ⟨*arg*⟩*
⟨*name*⟩ ::= ⟨*varname*⟩ | ⟨*conname*⟩
⟨*arg*⟩ ::= ⟨*varname*⟩ | `(`⟨*varname*⟩`)`
⟨*fmttoken*⟩ ::= `"`⟨*text*⟩`"` | ⟨*token*⟩

- Let us look at a couple of examples.

Formatting identifiers

- Input:

```
%format alpha = "\alpha"

\begin{code}
tan alpha = sin alpha / cos alpha
\end{code}
```

- Output:

$$tan\ \alpha = sin\ \alpha\ /\ cos\ \alpha$$

Parametrized formatting directives

- Formatting directives can be parametrized. The parameters may occur once or more on the right hand side.
- Input:

```
%format abs (a) = "\mathopen{|}" a "\mathclose{|}"
%format ~>      = "\leadsto"
The |abs| function computes the absolute value of
an integer:
\begin{code}
abs(-2) ~> 2
\end{code}
```

- Output:

The $|\cdot|$ function computes the absolute value of an integer:

$|-2| \leadsto 2$

Parentheses

- Sometimes, due to formatting, parentheses around arguments or the entire function become unnecessary.
- Therefore, `lhs2TeX` can be instructed to drop parentheses around an argument by enclosing the argument on the left hand side of the directive in parentheses.
- Parentheses around the entire function are dropped if the entire left hand side of the directive is enclosed in parentheses.

Parentheses – example

- Input:

```
%format ^^                    = "\;"
%format (ptest (a) b (c)) = ptest ^^ a ^^ b ^^ c
\begin{code}
ptest a b c
(ptest (a) (b) (c))
((ptest((a)) ((b)) ((c))))
\end{code}
```

- Output:

ptest a b c
ptest a (*b*) *c*
(*ptest* (*a*) ((*b*)) (*c*))

Parentheses – example

- Input:

```
%format eval a = "\llbracket " a "\rrbracket "
\begin{code}
size (eval (2 + 2))
\end{code}
%format (eval (a)) = "\llbracket " a "\rrbracket "
\begin{code}
size (eval (2 + 2))
\end{code}
```

- Output:

$size\ (\llbracket (2 + 2) \rrbracket)$
$size\ \llbracket 2 + 2 \rrbracket$

Local formatting directives

- Usually, formatting directives scope over the rest of the input.
- Formatting directives can be placed into **groups**.

```
%{
...
%}
```

- Formatting directives that are defined in a group scope only over the rest of the current group.

Local formatting directives – example

- Input:

```
In the beginning: |one|.\par
%format one = "\mathsf{1}"
Before the group: |one|.\par
%{
%format one = "\mathsf{one}"
Inside the group: |one|.\par
%}
After the group: |one|.
```

- Output:

In the beginning: *one*.
Before the group: 1.
Inside the group: one.
After the group: 1.

Nested applications of formatting directives

The right-hand sides of formatting directives are processed as follows:

- A string, enclosed in ", will be reproduced literally (without the quotes).
- A name, if it is the name of a parameter, will be replaced by the actual (formatted) argument.
- A name, if it is the name of a non-parametrized formatting directive, will be replaced by that directive's replacement.
- Any other name will be replaced by its standard formatting.

Implicit formatting

- A variable (or constructor) name that ends in a number or a prime ’ can be used in an implicit formatting statement.
- The prefix will be formatted as determined by the formatting directives in the input so far. The number will be added as an index, the prime character as itself.

Implicit formatting – example

- Input:

```
%format omega = "\omega"
|[omega, omega13, omega13']|\par
%format omega13
|[omega, omega13, omega13']|\par
%format omega13'
|[omega, omega13, omega13']|
```

- Output:

$[\omega, omega13, omega13']$
$[\omega, \omega_{13}, omega13']$
$[\omega, \omega_{13}, \omega'_{13}]$

Formatting in the various styles

- Formatting directives are applied in **math**, **poly**, and **newcode** styles.
- In **tt** style, only non-parametrized apply.
- In **verb** and **code** styles, formatting directives are ignored.

Alignment in “poly” style

- Alignment is computed per code block.
- All tokens that start on the same column and are preceded by at least **2** spaces are horizontally aligned in the output.
- (Almost) everything is possible, but watch out for accidental alignments!

Alignment example

- Input:

```
> rep_alg          =  (\  _           -> \m ->  Leaf m
>                     ,\  lfun rfun   -> \m ->  let  lt  = lfun m
>                                               rt  = rfun m
>                                       in    Bin lt rt
>                     )
> replace_min' t   =  (cata_Tree rep_alg t) (cata_Tree min_alg t)
```

- The red `lt` is not aligned (only one preceding space).
- Output:

$$
\begin{array}{lll}
rep\_alg & = (\lambda\_ & \to \lambda m \to Leaf\ m \\
 & \ \ ,\lambda lfun\ rfun & \to \lambda m \to \textbf{let}\ lt = lfun\ m \\
 & & \qquad\qquad\ \ rt = rfun\ m \\
 & & \qquad\quad \textbf{in}\ \ Bin\ lt\ rt \\
 &) & \\
replace\_min'\ t & = (cata\_Tree\ rep\_alg\ t)\ (cata\_Tree\ min\_alg\ t) &
\end{array}
$$

Accidental alignment example – input

```
%format <| = "\lhd "

> options  ::  [String] -> ([Class],[String])
> options  =   foldr (<|) ([],[])
>   where  "-align"     <| (ds,s:  as) = (Dir Align    s :  ds,     as)
>          ('-':'i':s)  <| (ds,    as) = (Dir Include  s :  ds,     as)
>          ('-':'l':s)  <| (ds,    as) = (Dir Let      s :  ds,     as)
>          s            <| (ds,    as) = (                 ds,s :  as)
```

- The red items will be unintentionally aligned because they start on the same column, with two or more preceding spaces each.
- To correct, insert extra spaces to ensure that unrelated tokens start on different columns.

Accidental alignment example – continued

- Output:

$$
\begin{array}{lll}
\mathit{options} & :: & [\mathit{String}] \to ([\mathit{Class}], [\mathit{String}]) \\
\mathit{options} & = & \mathit{foldr}\ (\lhd)\ ([\,], [\,]) \\
\quad \textbf{where}\ \texttt{"-align"} & \lhd\ (\mathit{ds}, s : \mathit{as}) & = (\mathit{Dir}\ \mathit{Align} \quad s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad \texttt{'-':'i'}:s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\mathit{Dir}\ \mathit{Include}\ s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad \texttt{'-':'l'}:s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\mathit{Dir}\ \mathit{Let} \quad\ s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\qquad\qquad\ \mathit{ds}, s : \mathit{as})
\end{array}
$$

- Corrected version:

$$
\begin{array}{lll}
\mathit{options} & :: & [\mathit{String}] \to ([\mathit{Class}], [\mathit{String}]) \\
\mathit{options} & = & \mathit{foldr}\ (\lhd)\ ([\,], [\,]) \\
\quad \textbf{where}\ \texttt{"-align"} & \lhd\ (\mathit{ds}, s : \mathit{as}) & = (\mathit{Dir}\ \mathit{Align} \quad s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad \texttt{'-':'i'}:s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\mathit{Dir}\ \mathit{Include}\ s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad \texttt{'-':'l'}:s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\mathit{Dir}\ \mathit{Let} \quad\ s : \mathit{ds}, \quad \mathit{as}) \\
\quad\quad s & \lhd\ (\mathit{ds}, \quad \mathit{as}) & = (\qquad\qquad\ \mathit{ds}, s : \mathit{as})
\end{array}
$$

Indentation in “poly” style

- If a line is indented in column n, then the **previous** code line is taken into account:
  - If there is an aligned token at column n in the previous line, then the indented line will be aligned normally.
  - Otherwise, the line will be indendet with respect to the first aligned token in the previous line to the left of column n.

Indentation in “poly” style – example

- Input:

```
unionBy           ::  (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys  =   xs ++ foldl  (flip (deleteBy eq))
                                   (nubBy eq ys)
```

- Output:

$$\begin{aligned}
&\mathit{unionBy} && :: (a \to a \to \mathit{Bool}) \to [a] \to [a] \to [a] \\
&\mathit{unionBy}\ \mathit{eq}\ \mathit{xs}\ \mathit{ys} && = \mathit{xs} +\!\!+ \mathit{foldl}\ (\mathit{flip}\ (\mathit{deleteBy}\ \mathit{eq})) \\
& && \qquad (\mathit{nubBy}\ \mathit{eq}\ \mathit{ys})
\end{aligned}$$

- In this example, there is an aligned token in the previous line at the same column, so everything is normal.

Indentation in "poly" style – example

- Input:

```
unionBy           ::  (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy eq xs ys  =   xs ++ foldl (flip (deleteBy eq))
                                  (nubBy eq ys)
```

- Output:

$$\begin{aligned}
&\mathit{unionBy} && :: (a \to a \to \mathit{Bool}) \to [a] \to [a] \to [a] \\
&\mathit{unionBy}\ \mathit{eq}\ \mathit{xs}\ \mathit{ys} &&= \mathit{xs} +\!\!+ \mathit{foldl}\ (\mathit{flip}\ (\mathit{deleteBy}\ \mathit{eq})) \\
& && \qquad (\mathit{nubBy}\ \mathit{eq}\ \mathit{ys})
\end{aligned}$$

- In this example, there is no aligned token in the previous line at the same column. Therefore, the third line is indented with respect to the first aligned token in the previous line to the left of that column.

Indentation in “poly” style – example

- Input:

```
%format foo = verylongfoo
\begin{code}
test  1
foo  bar
      2
\end{code}
```

- Output:

test 1
verylongfoo bar
2

- In rare cases, the indentation heuristic can lead to surprising results. Here, the 1 is aligned with the 2, but 2 is also indented with respect to *bar*.

Advanced alignment topics

- Some columns (containing symbols) are centered by `lhs2TeX` (all other columns are left-aligned).
- It is possible redefine the alignment of a specific column.
- It is possible to customize the output environment (using `%subst` directives). Using this, one can produce effects such as putting all code blocks into yellow boxes.
- It is possible to save (and restore) column information.

Saving and restoring column information example – input

```
\savecolumns
\begin{code}
intersperse                 ::  a -> [a] -> [a]
intersperse  _     []       =   []
intersperse  _     [x]      =   [x]
\end{code}
The only really interesting case is the one for lists
containing at least two elements:
\restorecolumns
\begin{code}
intersperse  sep  (x:xs)  =   x : sep : intersperse sep xs
\end{code}
```

Saving and restoring column information example – output

$$
\begin{array}{lll}
\mathit{intersperse} & & :: a \to [a] \to [a] \\
\mathit{intersperse}\ \_ & [\,] & = [\,] \\
\mathit{intersperse}\ \_ & [x] & = [x]
\end{array}
$$

The only really interesting case is the one for lists containing at least two elements:

$$\mathit{intersperse}\ \mathit{sep}\ (x : \mathit{xs}) = x : \mathit{sep} : \mathit{intersperse}\ \mathit{sep}\ \mathit{xs}$$

Spacing

- `lhs2TeX` does not really have a Haskell parser.
- Because of this, it can be used for dialects of Haskell, too!
- Spacing is handled automatically so that it works for correctly for pure Haskell most of the time.
- A good trick is to define the following two pseudo-operators to correct wrong automatic spacing:

```
%format ^  = " "
%format ^^ = "\;"
```

  - Use `^` where you do **not** want a space, but `lhs2TeX` would place one.
  - Use `^^` where you **do** want a space, but `lhs2TeX` does not place one.

AG code example – input

```
%format ^  = " "
%format ^^ = "\;"
%format ATTR = "\mathbf{ATTR}"
%format SEM  = "\mathbf{SEM}"
%format lhs  = "\mathbf{lhs}"
%format .  = "."
%format *  = "\times "
\begin{code}
ATTR Expr Factor  [ ^^ | ^^ | numvars   : Int  ]
ATTR Expr Factor  [ ^^ | ^^ | value     : Int  ]

SEM Expr
  |  Sum
             lhs   .  value     =  @left.value    +  @right.value
                   .  numvars   =  @left.numvars  +  @right.numvars
SEM Factor
  |  Prod
             lhs   .  value     =  @left.value    *  @right.value
                   .  numvars   =  @left.numvars  +  @right.numvars
\end{code}
```

AG code example – output

```
ATTR Expr Factor [ | | numvars : Int ]
ATTR Expr Factor [ | | value   : Int ]

SEM Expr
  | Sum
      lhs.value   = @left.value   + @right.value
          .numvars = @left.numvars + @right.numvars
SEM Factor
  | Prod
      lhs.value   = @left.value   * @right.value
          .numvars = @left.numvars + @right.numvars
```

Calculation example – input

```
\def\commentbegin{\{\ }
\def\commentend{\}}
\begin{spec}
    map (+1) [1,2,3]

==  {- desugaring of |(:)| -}

    map (+1) (1 : [2,3])

==  {- definition of |map| -}

    (+1) 1  :  map (+1) [2,3]

==  {- performing the addition on the head -}

    2       :  map (+1) [2,3]

==  {- recursive application of |map| -}

    2       :  [3,4]

==  {- list syntactic sugar -}

    [2,3,4]
\end{spec}
```

Calculation example – output

$$
\begin{aligned}
& \mathit{map}\ (+1)\ [1,2,3] \\
\equiv\ & \{\ \text{desugaring of } (:)\ \} \\
& \mathit{map}\ (+1)\ (1 : [2,3]) \\
\equiv\ & \{\ \text{definition of } \mathit{map}\ \} \\
& (+1)\ 1 : \mathit{map}\ (+1)\ [2,3] \\
\equiv\ & \{\ \text{performing the addition on the head}\ \} \\
& 2 \quad\quad : \mathit{map}\ (+1)\ [2,3] \\
\equiv\ & \{\ \text{recursive application of } \mathit{map}\ \} \\
& 2 \quad\quad : [3,4] \\
\equiv\ & \{\ \text{list syntactic sugar}\ \} \\
& [2,3,4]
\end{aligned}
$$

Defining variables

- `lhs2TeX` allows flags (or variables) to be set by means of the `%let` directive.

`%let` ⟨*varname*⟩ `=` ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*application*⟩ ⟨*operator*⟩ ⟨*application*⟩*
⟨*application*⟩ ::= `not?` ⟨*atom*⟩
⟨*atom*⟩ ::= ⟨*varid*⟩ | `True` | `False` | ⟨*string*⟩ | ⟨*numeral*⟩ | `(`⟨*expression*⟩`)`
⟨*operator*⟩ ::= `&&` | `||` | `==` | `/=` | `<` | `<=` | `>=` | `>` | `++` | `+` | `-` | `*` | `/`

- Expressions are built from booleans (either `True` or `False`), integers, strings and previously define variables using some predefined, Haskell-like operators.
- Variables can also be defined by using the `-l` or `-s` command line options.
- `lhs2TeX`'s version is available as predefined `version` variable, and the current style is available as predefined `style` variable.

Conditionals

- (Boolean) expressions can also be used in conditionals:

```
%if ⟨expression⟩
...
%elif ⟨expression⟩
...
%else
...
%endif
```

  The `%elif` and `%else` directives are optional.

- Depending on the result of the evaluation of the expression, only the then or the else part are processed by `lhs2TeX`.

Uses of conditionals

- Have different versions of one paper in one source. Depending on a flag, produce either the one or the other. Because the flag can be defined via a command line option, no modification of the source is necessary to switch versions.
- Code that is needed to make the Haskell program work but that should not appear in the formatted article (module headers, auxiliary definitions), can be enclosed between `%if False` and `%endif` directives, **or:**
- If Haskell code has to be annotated for `lhs2TeX` to produce the right output, define different formatting directives for the annotation depending on style (**poly** or **newcode**). Both code and TEX file can then still be produced from a common source!

Calling `ghci`

- It is possible to call `ghci` (or `hugs`) using the `%options` directive.
- `lhs2TeX` looks for calls to the **TEX commands** `\eval` and `\perform` and feeds their arguments to the interpreter.
- The current input file will be the active module. Therefore, this feature works only if the current file really is legal Haskell.

Calling `ghci` – example

- Input:

```
%options ghci -fglasgow-exts

> fix f = f (fix f)

This function is of type \eval{:t fix},
and |take 10 (fix ('x':))|
evaluates to \eval{take 10 (fix ('x':))}.
```

- Output:

$fix \quad :: \forall a.(a \to a) \to a$
$fix\, f = f\ (fix\, f)$

This function is of type $fix :: \forall a.(a \to a) \to a$, and $take\ 10\ (fix\ (\texttt{'x':}))$ evaluates to `"xxxxxxxxxx"`.

Implementation and distribution

- `lhs2TeX` is written in Haskell
- **poly** style makes use of a specifically written LaTeX package `polytable`, which is included in the distribution
- License is GPL.
- There has not been an official release for a long time, so get the most recent version from the Subversion repository.
- It is reported to work on Linux, Mac OS X, and Windows.
- It has been used for several papers and seems to be quite stable.

Future work

- More language independence (customizable lexer).
- Clean up (and extend) the formatting directives language.
- Allow directives during code blocks.
- Add more features to `polytable` package.
- ...

Future development is relatively low priority, though. If you want it, do it yourself or try to convince me that it is urgent!