Guide to lhs2TEX
(for version 1.17)

Ralf Hinze
Computing Laboratory, University of Oxford
ralf.hinze@comlab.ox.ac.uk

Andres Löh
Well-Typed LLP
mail@andres-loeh.de

March 17, 2011

Contents

1. About lhs2TEX

The program lhs2TEX is a preprocessor that takes a literate Haskell source file as input (or something sufficiently alike) and produces a formatted file that can be processed further by LATEX.

For example, consider the following input file:

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

If we run the following two commands on it

```
$ lhs2TeX -o HelloWorld.tex HelloWorld.lhs
$ pdflatex HelloWorld.tex
```

then the resulting PDF file will look similar to

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

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

The behaviour of lhs2TEX is highly customizable. The main mode of operation of lhs2TEX is called the **style**. By default, lhs2TEX operates in **poly** style. Other styles can be selected via command line flags. Depending on the selected style, lhs2TEX can perform quite different tasks. Here is a brief overview:

- **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**

The name of the style is also the name of the flag you have to pass to lhs2TEX in order to activate the style. For example, call `lhs2TeX --newcode` to use lhs2TEX in **newcode** style.

2. Installing lhs2TEX

There are three options for installing lhs2TEX (ordered by ease):

- Using Hackage
- Using Cabal.
- Classic configure/make.

2.1. Using Hackage to install lhs2TEX

The Haskell Platform [7] is the easiest way to get started with programming Haskell. It is also the easiest way to build, install, and manage Haskell packages, through Hackage [6]:

```
$ cabal update
$ cabal install lhs2tex
```

The first command downloads the latest package list, and the second installs (along with any dependencies) the latest version of lhs2TEX.

2.2. Using Cabal to install lhs2TEX

If you have downloaded a source distribution, which is a valid Cabal package, you can install lhs2TEX using Cabal (this requires Cabal 1.2 or later). Begin by unpacking the archive. Assuming that it has been unpacked into directory `/somewhere`, then say

```
$ cd /somewhere/lhs2TeX-1.17
$ runghc Setup configure
$ runghc Setup build
$ runghc Setup install
```

The install step requires write access to the installation location and the LaTeX filename database. (Hint: use `sudo` if necessary.)

2.3. configure/make

The following instructions apply to Unix-like environments. However, lhs2TEX does run on Windows systems, too. (If you would like to add installation instructions or facilitate the installation procedure for Windows systems, please contact the authors.)

Begin by unpack the archive. Assuming that it has been unpacked into directory `/somewhere`, then say

```
$ cd /somewhere/lhs2TeX-1.17
$ ./configure
$ make
$ make install
```

You might need administrator permissions to perform the `make install` step. Alternatively, you can select your own installation location by passing the `--prefix` argument to `configure`:

```
$ ./configure --prefix=/my/local/programs
```

There are a couple of library files that come with lhs2TEX (containing basic lhs2TEX formatting directives) that need to be found by the lhs2TEX binary. The default search path is as follows:

```
.
{HOME}/lhs2tex-1.17//
{HOME}/lhs2tex//
{HOME}/lhs2TeX//
{HOME}/.lhs2tex-1.17//
{HOME}/.lhs2tex//
{HOME}/.lhs2TeX//
{LHS2TEX}//
/usr/local/share/lhs2tex-1.17//
/usr/local/share/lhs2tex-1.17//
/usr/local/lib/lhs2tex-1.17//
/usr/share/lhs2tex-1.17//
/usr/lib/lhs2tex-1.17//
/usr/local/share/lhs2tex//
/usr/local/lib/lhs2tex//
/usr/share/lhs2tex//
/usr/lib/lhs2tex//
/usr/local/share/lhs2TeX//
/usr/local/lib/lhs2TeX//
/usr/share/lhs2TeX//
/usr/lib/lhs2TeX//
```

Here, `{HOME}` and `{LHS2TEX}` denote the current values of the environment variables `HOME` and `LHS2TEX`. The double slash at the end of each dir means that subdirectories are also scanned. If lhs2TEX is installed to a non-standard path, you might want to set the environment variable `LHS2TEX` to point to the directory where `lhs2TeX.fmt` and the other library files have been installed to.

IMPORTANT: To be able to use "poly" style, the two LaTeX packages `polytable.sty` and `lazylist.sty` are required!

Both are included in the lhs2TEX distribution (they are not part of standard LaTeX distributions, although they are available from CTAN [1, 2]), and are usually installed during the normal procedure. The `configure` script will determine whether a suitably recent version of `polytable` is installed on your system, and if necessary, install both `polytable.sty` and `lazylist.sty` to your TEX system. If this is not desired

or fails (because the script cannot detect your TEX installation properly), the installation of these files can be disabled by passing the option `--disable-polytable` to `configure`. In this case, the two files must be manually installed to a location where your TEX distribution will find them. Assuming that you have a local TEX tree at `/usr/local/share/texmf`, this can usually be achieved by placing the files in the directory `/usr/local/share/texmf/tex/latex/polytable` and subsequently running

```
$ mktexlsr
```

filename database.

3. How to hit the ground running with lhs2TEX

When run on a literate Haskell source file, lhs2TEX classifies the input into different blocks.

Bird-style code blocks In the Bird-style of literate Haskell programming, all lines starting with > are interpreted as code. (To be good literate code, you must always leave a blank line before and after the code block.)

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

These lines are considered by lhs2TEX as **code blocks** and are processed as such.

Lines beginning with > will be treated as code to be formatted by lhs2TEX and code to be compiled by the compiler. If you wish to hide code from the compiler, but not from lhs2TEX, you can flip the > characters around.

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

There is no change in the output of lhs2TEX (with the exception of code extraction through the **code** and **newcode** styles).

LATEX-style code blocks The LATEX-style of literate programming is to surround code blocks with `\begin{code}` and `\end{code}`.

```
\begin{code}
main  ::  IO ()
main  =   putStrLn "Hello, world!"
\end{code}
```

These lines will be treated by lhs2TEX (and a Haskell compiler) in the same way as lines beginning with >. The equivalent to lines beginning with <, is to surround the lines with `\begin{spec}` and `\end{spec}`.

```
\begin{spec}
main  ::  IO ()
main  =   putStrLn "Hello, world!"
\end{spec}
```

Unlike a Haskell compiler, lhs2TEX does not care if both styles of literate programming are used in the same file. *But*, if you are using the **code** and **newcode** styles to produce Haskell source files, the initial characters > and < will be replaced by spaces, which means that you have to indent `code` environments in order to create a properly indented Haskell module.

Inline verbatim Text between two `@` characters that is not in a code block is considered inline verbatim. If you actually want a `@` character to appear in the text, it needs to be escaped: `@@`. There is no need to escape `@`'s in code blocks. For example, `@id :: a -> a@` appears as `id :: a -> a`.

Inline code Text between two `|` characters that is not in a code block is considered inline code. Again, `|` characters that should appear literally outside of code blocks need to be escaped: `||`. For example, `|id :: a -> a|` appears as $id :: a \to a$.

Directives A `%` that is followed by the name of an lhs2TEX directive is considered as a **directive** and may cause lhs2TEX to take special actions. Directives are described in detail in Section 5.

Special commands Some commands are treated specially, such as occurrences of the TEX commands `\eval`, `\perform`, `\verb` or of the LATEX environment `verbatim`. The treatment of the `\eval` and `\perform` commands is covered in Section 12. The `\verb` command and the `verbatim` environment are intercepted by lhs2TEX, however, they will behave as they would without lhs2TEX.

Everything else Everything in the input file that does not fall into one of the above cases is is classified as **plain text** and will simply pass straight through lhs2TEX.

4. Using lhs2TEX with style

In this section, we will walk though an example to illustrate how to utilize the styles of lhs2TEX. As we noted in Section 1, lhs2TEX operates in the **poly** style by default. Appendix A contains summaries of the more simplistic and deprecated styles: **verb**, **tt** and **math**. For each style, there will also be a short summary. Some of the points listed in the summary are simply defaults for the particular style and can actually be changed.

4.1. Achieving complex layouts with the “poly” style

The **poly** style permits multiple alignments and thus it is possible to construct complex layouts. The style supersedes the **math** style and lifts the alignment restrictions that the **math** style has. We will demonstrate the **poly** style with the following example as our input to lhs2TEX:

```
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)
```

This results in the following output:

```
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)
```

If run in **poly** style, lhs2TEX produces LATEX code that makes use of the `polytable` package, a package that has been specifically designed to fit the needs that arise while formatting Haskell code. (If you are interested in the package or think that it might be useful for other purposes, you are welcome to look at the documentation for `polytable` [1, also distributed with lhs2TEX as `polytable.pdf` in the `polytable` directory].)

Beyond the advanced alignment options, **poly** style has all the functionality of its ancestor style. If **poly** style works for you, you should use it.

“poly” summary

- all formatting directives are obeyed
- conditionals and includes are handled
- inline verbatim is typeset as verbatim, whereas inline code and code blocks are typeset using a proportional font, using mathematical symbols to represent many Haskell operators.

– alignment can be flexibly specified; complex layouts are possible
– plain text is copied unchanged

4.2. Customizing the “poly” style

The following example demonstrates that the visual appearance of “poly” style is in no way dictated by lhs2TEX. There are several possibilities to modify the output by means of formatting directives. Here, we try to mimic the legacy **tt** style (see Section A.2) by choosing a typewriter font again and using the same symbols that are default in **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)
```

The spaces in the code of the source file are *not* preserved—the alignment is generated by the `polytable` package. This is in contrast to the **tt** style we are imitating, where the spacing of the output is the spacing of the input.

4.3. Producing code with the “code” and “newcode” styles

These two styles are not for producing a LATEX source file, but instead are for producing a Haskell file again. Everything that is not code is thrown away. In addition, the **newcode** style has a few extra features. It applies formatting directives, which can be used as simple macros on the Haskell source level, and it generates line pragmas for the Haskell compiler that will result in error messages pointing to the original file (before processing with lhs2TEX). The plain **code** style does not have this extra functionality. Again, **code** is mainly intended for compatibility with old documents. You should use **newcode** where possible.

“code” summary
– formatting directives are ignored
– conditionals and includes are handled
– code blocks that are not specifications are copied unchanged
– plain text, inline code, specification code, and inline verbatim are discarded

“new code” summary
– all formatting directives are obeyed

`%include`	include a file
`%format`	formatting directive for an identifer/operator
`%{`	begin of an `lhs2TeX` group
`%}`	end of an `lhs2TeX` group
`%let`	set a toggle
`%if`	test a condition
`%else`	second part of conditional
`%elif`	`else` combined with `if`
`%endif`	end of a conditional
`%latency`	tweak alignment in **poly** style
`%separation`	tweak alignment in **poly** style
`%align`	set alignment column in **math** style
`%options`	set options for call of external program
`%subst`	primitive formatting directive
`%file`	set filename

Table 1: All lhs2TEX directives

- conditionals and includes are handled
- code blocks that are not specifications are, after applying formatting directives, copied unchanged and prefixed by a line pragma indicating the original source location of the code block
- plain text, inline code, specification code, and inline verbatim are discarded

5. Directives

There are a number of directives that are understood by lhs2TEX. Some of these are specific to styles, and others are ignored in some styles. 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.

While lhs2TEX will remove directives that it has interpreted, it will simply ignore all normal TEX comments that are no directives. Therefore, if a directive is accidentally misspelled, no error message will be raised, in general.

Table 1 is a complete list of the directives that lhs2TEX knows about. Many of these directive will be explained in more detail in the following sections:

- See Section 6 for the `%include` directive.
- See Section 7 for the `%format` directive.
- See Section 7.6 for the `%{` and `%}` directives.
- See Section 8.3 for the `%separation` and `%latency` directives.
- See Section 9 for the `%let` directive.
- See Section 10 for the `%if`, `%elif`, `%else` and `%endif` directives.
- See Section 12 for the `%options` directive.
- See Section 13 for the `%subst` directive.

6. Including files

Other files can be included by lhs2TEX; this is what the `%include` directive is for:

```
%include ⟨filename⟩
```

The specified file is searched for in the lhs2TEX source path, which can be modified using environment variables or the `-P` command line option (see also page 5). The include directive causes the indicated file to be read and processed, exactly as if its contents had been inserted in the current file at that point. It is the lhs2TEX equivalent of the TEX command `\input`. The include mechanism of lhs2TEX is entirely independent of TEX or Haskell includes/imports.

WARNING: Although relative and absolute pathnames can be specified as part of a filename in an `%include` directive, the use of this feature is strongly discouraged. Set the search path using the `-P` command line option to detect files to include.

If the `-v` command line flag is set, lhs2TEX will print the paths of the files it is reading on screen while processing a file.

6.1. The lhs2TEX "prelude"

Several aspects of the behaviour of lhs2TEX are not hardcoded, but configurable via directives. As a consequence, a minimal amount of functionality has to be defined for lhs2TEX to be able to operate normally.

Essential definitions are collected in the file `polycode.fmt`. You should include this file at the start of your document:

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

\end{document}
```

This is the appropriate prelude to use for the default **poly** style and the **newcode** style. If you intend to use one of the other styles, you should instead include the file `lhs2TeX.fmt`.

```
%include lhs2TeX.fmt
```

The reason for this is that some of the defaults in `lhs2TeX.fmt` are sub-optimal for the **poly** or **newcode** styles; the `polycode.fmt` prelude file has been tailored specifically for them.

One of the two files `lhs2TeX.fmt` or `polycode.fmt` should be included—directly or indirectly—in every file to be processed by lhs2TEX!

NOTE TO USERS OF PREVIOUS VERSIONS: There used to be a file `lhs2TeX.sty` that also contained a part of the prelude declarations. This file still exists for compatibility reasons, but it is now deprecated; it should *not* be included in any of your documents anymore.

It is perfectly possible to design your own libraries that replace or extend these basic files and to include these libraries instead. It is not recommended, though, to edit `polycode.fmt` or`lhs2TeX.fmt` files directly. If you are not satisfied with some of the default definitions, create your own file to redefine selected parts. This way, if lhs2TEX is updated, you will still be able to benefit from improvements and changes in the 'prelude' files.

It is possible to use lhs2TEX in a setup where a TEX document is split into several files. In this case, each of the files will be processed separately by lhs2TEX, so you should must include `polycode.fmt` (or `lhs2TeX.fmt`) in every single source file.

WARNING: Note that both `polycode.fmt` and `lhs2TeX.fmt` contain lhs2TEX directives, and therefore *cannot* be included using TEX or LATEX include mechanisms such as `\input` or `\usepackage`.

7. Formatting

The `%format` directive is a powerful tool for transforming the source file. The complete syntax that is supported by lhs2TEX is quite complex, but we will break it down by looking in detail at many different use cases.

```
%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⟩
```

There are three different forms of the formatting statement. The first can be used to change the appearance of most functions and operators and a few other symbols (cf. Section 7.1). The second form is restricted to named identifiers (both qualified and unqualified, but no symbolic operators); in turn, such formatting directives can be parametrized (cf. Section 7.3). Finally, the third form provides a syntactically lightweight way of formatting certain identifiers using some heuristics (cf. Section 7.7). Let us begin by looking at the first form.

7.1. Formatting single tokens

The most important use for `%format` is to assign a symbol to an identifier or an operator. The input

```
%format alpha = "\alpha"

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

produces output similar to the following:

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

The occurrences of `alpha` within the Haskell code portions of the input file are replaced by the TEX command `\alpha` and thus appear as "α" in the output.

A lot of formatting directives for frequently used identifiers or operators are already defined in the lhs2TEX prelude. For instance, `++` is formatted as "$+\!\!+$", `undefined` is formatted as "$\bot$", and `not` is formatted as "$\neg$". If you look at `lhs2TeX.fmt`, you will find the following directives that do the job:

```
%format ++         = "\plus "
%format undefined  = "\bot "
%format not        = "\neg "
```

Here, `\plus` refers to a LATEX macro defined in the lhs2TEX prelude:

```
\newcommand{\plus}{\mathbin{+\!\!\!+}}
```

If you are not satisfied with any of the default definitions, just redefine them (by overriding, not replacing them). A `%format` directive scopes over the rest of the input, and if multiple directives for the same token are defined, the last one is used. Thus, after

```
%format ++         = "\mathbin{\mathbf{+}}"
%format undefined  = "\Varid{undefined}"
%format not        = "!"
```

you get "$+$", "$\mathit{undefined}$", and "!", respectively. Note that `\Varid` is a macro defined in the lhs2TEX prelude that can be used to typeset identifier names. It is predefined to be the same as `\mathit`, but can be changed. Do not use identifier names in TEX replacements directly. For instance,

```
%% THE FOLLOWING IS BAD:
%format undefined  = "undefined"
```

will cause `undefined` to be typeset as "$undefined$", which looks by far less nice than "$\mathit{undefined}$". It is also possible to define a symbol for infix uses of a function. The file `lhs2TeX.fmt` contains:

```
%format `elem`     = "\in "
```

This causes `2 `elem` [1,2]` to be typeset as "$2 \in [1,2]$", whereas `elem 2 [1,2]` will still be typeset as "$\mathit{elem}\ 2\ [1,2]$".

7.2. Nested formatting

The right hand sides of formatting directives are not restricted to (TEX-)strings. They can in fact be sequences of such strings or other tokens, separated by space. Such other tokens will be replaced by their formatting again. For example, if you have already defined a specific formatting

```
%format ~> = "\leadsto "
```

then you can later reuse that formatting while defining variants:

```
%format ~>* = ~> "^{" * "}"
```

As you can see, in this definition we reuse both the current formatting for `~>` and for `*`. We now get "$\leadsto^*$" for `~>*`, but should we decide to define

```
%format * = "\star "
```

later, we then also get "$\leadsto^\star$". Of course, you can use the same mechanism for non-symbolic identifiers:

```
%format new      = "\mathbf{new}"
%format text0    = text
%format text_new = text "_{" new "}"
```

will cause `text0` to be typeset as "*text*", and `text_new` will appear as "$\mathit{text}_{\mathbf{new}}$".

WARNING: There is no check for recursion in the formatting directives. Formatting directives are expanded on-demand, therefore a directive such as

```
%% THE FOLLOWING IS BAD:
%format text = "\mathsf{" text "}"
```

will not produce "`text`**" for** `text`**, but rather cause an infinite loop in lhs2TEX once used.**

7.3. Parametrized formatting directives

Formatting directives can be parametrized. The parameters may occur one or more times on the right hand side. This form of the format directive is only available for alphanumeric identifiers. For example, the 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}
```

produces output similar to

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

$|-2| \rightsquigarrow 2$

If the function is used with too few arguments as in the text, a default symbol is substituted (usually a `\cdot`, but that is customizable, cf. Section 13).

7.4. (No) nesting with parametrized directives

You cannot use a parametrized directive on the right hand side of another directive. In summary, 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.

Note that the spaces between the tokens do not occur in the output. If you want spaces, insert them explicitly with quotes.

7.5. Parentheses

Sometimes, due to formatting an identifier as a symbol, 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. Let us look at another example:

```
%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}
```

The above input produces the following output:

$$\begin{array}{l} \mathit{ptest}\ a\ b\ c \\ \mathit{ptest}\ a\ (b)\ c \\ (\mathit{ptest}\ (a)\ ((b))\ (c)) \end{array}$$

In the first line there are no parentheses to drop. In the second line, the parentheses around the arguments `a` and `b` are dropped, as are the parentheses around the function `ptest`. In the third line, the source has double parentheses around each argument as well as the function. One set of parentheses are dropped in each case, except for the `b` argument.

Note that in this example, a special purpose operator, `^^`, is used to facilitate the insertion of spaces on the right hand side of a formatting directive. You can read more about influencing spacing using formatting directives in Section 11.1.

Let us consider another example involving parentheses with the following 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}
```

This results in

$$\begin{array}{l} \mathit{size}\ (\llbracket (2 + 2) \rrbracket) \\ \mathit{size}\ \llbracket 2 + 2 \rrbracket \end{array}$$

In the second format directive we have redefined the eval function to drop the redundant parentheses.

7.6. Local formatting directives

Usually, formatting directives scope over the rest of the input. If that is not desired, formatting directives can be placed into **groups**. Groups look as follows:

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

Formatting directives that are defined in a group only scope over the remainder of that group. Groups can also be nested. (Groups in lhs2TEX do not interact with TEX groups, so these different kinds of groups do not have to occur properly nested.)

Let us demonstrate the effect of groups with the following 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|.
```

This is appears as:

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

On the first line, the string "one" has been formatted in italics as lhs2TEX has treated it, by default, as a Haskell identifier. On the second line of output, the first format directive from the source file has come into effect, so "one" has been rendered as a numeral in a sans-serif font. On the third line, the corresponding source is inside the group and second formatting directive is in effect. Thus, "one" has been rendered in a sans-serif font. Finally, on the fourth line, the group has closed, along with the scope of the second format directive. The original format directive applies again (as its scope extends to the end of the source file), thus, "one" has again been rendered as a numeral in a sans-serif font.

7.7. Implicit formatting

The third syntactic form of the formatting directive, which lacks a right hand side, can be used to easily format a frequently occurring special case, where a token is to be given a numeric subscript, or is primed. Only a variable (or constructor) name that ends in a number or a prime ' can be used in an implicit formatting statement. The prefix will then 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.

Let us demonstrate implicit formatting with the follow input:

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

The corresponding output is:

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

Another form of implicit formatting only takes place only if the token to be formatted does not end in primes, and only if digits at the end are immediately preceded by an underscore. The reason for these conditions is compatibility. If the conditions are met, then the token is split at underscores, and the part to the right of an underscore is typeset as subscript to the part on the left, recursively. Again, let us look at an example:

```
%format a_i
%format a_j
%format left  = "\leftarrow "
%format right = "\rightarrow "
%format a_left
%format a_right
%format a_let
%format a_where
%format a_x_1
%format a_x_2
%format y_1
%format y_2
%format a_y_1
%format a_y_2
%format a_y1
%format a_i'
|[a_i,a_j,a_left,a_right,a_let,a_where,a_x_1,a_x_2,a_y_1,a_y_2,a_y1,a_i']|
```

And its output:

$[a_i, a_j, a_{\leftarrow}, a_{\rightarrow}, a_{\textbf{let}}, a_{\textbf{where}}, a_{x\_1}, a_{x\_2}, a_{y_1}, a_{y_2}, a\_y_1, a'_i]$

7.8. Formatting behaviour in different styles

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

A document can be prepared for processing in different styles using conditionals (cf. Section 10).

8. Alignment in "poly" style

While the ability to transform the appearance of the source file is probably the most important feature of lhs2TEX, certainly the next most important is the ability to maintain alignment of code elements, while using a proportional font.

Using this feature is relatively simple:

- Alignment is computed per code block.
- All tokens that start on the same column and are preceded by at least **2** spaces will appear beginning from the same vertical axis in the output.

Using these simple rules, (almost) everything is possible, but it is very important to verify the results and watch out for accidental alignments (i.e. tokens that get aligned unintentionally).

8.1. An example

The following example shows some of the potential. This is the 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)
```

Look at the highlighted (gray) tokens. The `lt` will not appear aligned with the two equality symbols, because it is preceded by only one space. Similarly, the `m` in the first line after the `Leaf` constructor will not be aligned with the declarations and the body of the let-statement, because it is preceded by only one space. Note furthermore that the equality symbols for the main functions `rep_alg` and `replace_min'` are surrounded by two spaces on both sides, also on the right. This causes the comma and the closing parenthesis to be aligned correctly. The output looks as follows:

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

8.2. Accidental alignment

The main danger of the alignment heuristic is that it may result in some tokens being aligned unintentionally. The following example contains illustrates this possibility:

```
%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 gray tokens will be unintentionally aligned because they start on the same column, with two or more preceding spaces each. The output looks as follows:

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

The "::" and the "=" have been aligned with the declarations of the where-clause. This results in too much space between the two *options* tokens and the symbols. Another problems is that in this case the *centering* of the two symbols is destroyed by the alignment (cf. Section 8.7). As a result, "::" and "=" appear left-aligned, but not cleanly, because TEX inserts a different amount of whitespace around the two symbols.

The solution to all this is surprisingly simple: just insert extra spaces in the input to ensure that unrelated tokens start on different columns:

```
%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)
```

This produces the correct output:

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

8.3. The full story

If you want to customize the alignment behaviour further, you can. Here is exactly what happens:

- Alignment is computed per code block.
- Per code block there are a number of **alignment columns**.
- If a token starts in column n and is prefixed by at least "*separation*" spaces, then n is an **alignment column** for the code block.
- If a token starts in an alignment column n and is prefixed by at least "*latency*" spaces, then the token is **aligned** at column n.
- All tokens that are aligned at a specific column will appear aligned (i.e. at the same horizontal position) in the output.

Both latency and separation can be modified by means of associated directives:

```
%separation ⟨natural⟩
%latency ⟨natural⟩
```

It can occasionally be useful to increase the default settings of 2 and 2 for large code blocks where accidental alignments can become very likely! It does not really make sense to set latency to a value that is strictly smaller than the separation, but you can do so—there are no checks that the specified settings are sensible.

8.4. Indentation in "poly" style

Sometimes, lhs2TEX will insert additional space at the beginning of a line to reflect indentation. The rule is as follows.

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 indented with respect to the first aligned token in the previous line to the left of column n.

The first example demonstrates the first case:

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

In this example, there is an aligned token in the previous line at the same column, so everything is normal. The two highlighted parentheses are aligned, causing the second line to be effectively indented:

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

The next example demonstrates the second case. It is the same example, with one space before the two previously aligned parentheses removed:

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

Here, 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, which in this case happens to be the `xs`:

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

Sometimes, this behaviour might not match the intention of the user, especially in cases as above, where there really starts a token at the same position in the previous line, but is not preceded by enough spaces. Always verify the output if the result looks as desired.

The amount of space that is inserted can be modified. A call to the TEX control sequence `\hsindent` is inserted at the appropriate position in the output, which gets as argument the column difference in the source between the token that is indented, and the base token. In the situation of the above example, the call is `\hsindent{12}`. The default definition in the lhs2TEX prelude ignores the argument and inserts a fixed amount of space:

```
\newcommand{\hsindent}[1]{\quad}
```

Here is another example that shows indentation in action, the Haskell standard function *scanr1* written using only basic pattern matching:

```
scanr1        ::  (a -> a -> a) -> [a] -> [a]
scanr1 f xxs  =   case xxs of
                     x:xs ->  case xs of
                                 []  ->  [x]
                                 _   ->  let
                                            qs = scanr1 f xs
                                         in
                                            case qs of
                                               q:_ -> f x q : qs
```

And the associated output:

```
scanr1        :: (a → a → a) → [a] → [a]
scanr1 f xxs = case xxs of
                  x : xs → case xs of
                              [] → [x]
                              _ → let
                                     qs = scanr1 f xs
                                  in
                                     case qs of
                                       q: _ → f x q : qs
```

The third line, which begins with $x : xs$, is an indented line, but it does not start at an alignment column from the previous line. Thus, the second rule applies and it is

indented relative to the first aligned token to the left in the previous line, which is **case**. The same explanation applies for the pattern [ ]. The indentation of the line beginning with _ is an example of the first rule. It is indented so as to be aligned with the token [ ].

8.5. Interaction between alignment and indentation

In rare cases, the indentation heuristic can lead to surprising results. This is an example:

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

And its output:

```
test           1
verylongfoo bar
               2
```

Here, the large amount of space between *test* and 1 might be surprising. However, the 1 is aligned with the 2, but 2 is also indented with respect to *bar*, so everything is according to the rules. The "solution" is to verify if both the alignment between 1 and 2 and the indentation of the 2 are intended, and to remove or add spaces accordingly.

8.6. Interaction between alignment and formatting

If a token at a specific column is typeset according to a formatting directive, then the first token of the replacement text inherits the column position of the original token. The other tokens of the replacement text will never be aligned. Actual arguments of parametrized formatting directives keep the column positions they have in the input.

8.7. Centered and right-aligned columns

Under certain circumstances lhs2TEX decides to typeset a column centered instead of left-aligned. This happens if the following two conditions hold:

- There is *at most one* token per line that is associated with the column.
- *At least one* of the tokens associated with the column is a symbol.

In most cases, this matches the intention. If it does not, there still might be the possibility to trick lhs2TEX to do the right thing:

- Change the alignment behaviour of the column using `\aligncolumn` (see below).

`l`	left-align column
`c`	center column
`r`	right-align column
`p {`⟨*dimen*⟩`}`	make column of fixed width ⟨*dimen*⟩
`@@{`⟨*tex*⟩`}`	can be used before or after the letter specifying alignment to suppress inter-column space and typeset ⟨*tex*⟩ instead; note that this is usually achieved using just one `@`, but as `lhs2TeX` interprets the `@`, it must be escaped
`>{`⟨*tex*⟩`}`	can be used before the letter specifying the alignment to insert ⟨*tex*⟩ directly in front of the entry of the column
`<{`⟨*tex*⟩`}`	can be used after the letter specifying the alignment to insert ⟨*tex*⟩ directly after the entry of the column

Table 2: Column specifiers for `\aligncolumn`

- If the column is centered but should not be, add extra tokens that are formatted as nothing that will be associated with the column (see also Section 11.1 about spacing).
- If the column should be centered but is left-aligned, it is sometimes possible to use a symbol instead of an alphanumeric identifier, and add a formatting directive for that newly introduced symbol.

The syntax of the `\aligncolumn` command is:

```
\aligncolumn {⟨integer⟩} {⟨column-specifier⟩}
```

The ⟨*integer*⟩ denotes the number (i.e. as displayed by the editor) of a column. Note that `lhs2TeX` starts counting columns at 1. As ⟨*column-specifier*⟩ one can use about the same strings that one can use to format a column in a `tabular` environment using the LaTeX `array` [4] package. Table 2 has a short (and not necessarily complete) overview.

TODO: ADD EXAMPLE!!

8.8. Saving and restoring column information

It is possible to share alignment information between different code blocks. This can be desirable, especially when one wants to interleave the definition of a single function with longer comments. This feature is implemented on the TeX level (the commands are defined in the lhs2TeX prelude).

Here is an example of its use:

```
\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}
```

As output we get:

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

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

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

Compare this to the output that would be generated without the `\savecolumns` and `\restorecolumns` commands:

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

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

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

IMPORTANT: If this feature is used, it may require several runs of LaTeX until all code blocks are correctly aligned. Watch out for warnings of the `polytable` package that tell you to rerun LaTeX!

9. Defining variables

One can define or define flags (or variables) 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`), numerals (integers, but also decimal numbers) and previously defined variables using some fixed set of builtin operators. The expression will be evaluated completely at the time the `%let` directive is processed. If an error occurs during evaluation, lhs2TEX will fail.

Variables can also be passed to lhs2TEX from the operating system level by using the `-l` or `-s` command line options.

The main use of variables is in conditionals (cf. Section 10). At the moment, there is no way to directly use the value of a variable in a `%format` directive.

9.1. Predefined variables

In every run of lhs2TEX, the version of lhs2TEX is available as a numerical value in the predefined variable `version`. Similarly, the current style is available as an integer in the predefined variable `style`. There also are integer variables `verb`, `tt`, `math`, `poly`, `code`, and `newcode` predefined that can be used to test `style`.

It is thus possible to write documents in a way that they can be processed beautifully in different styles, or to make safe use of new lhs2TEX features by checking its version first.

10. Conditionals

Boolean expressions can be used in conditionals. The syntax of an lhs2TEX conditional is

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

where the `%elif` and `%else` directives are optional. There may be arbitrarily many `%elif` directives. When an `%if` directive is encountered, the expression is evaluated, and depending on the result of the evaluation of the expression, only the then or only the else part of the conditional is processed by lhs2TEX, the other part is ignored.

10.1. Uses of conditionals

These are some of the most common uses of conditionals:

- One can have different versions of one paper in one (set of) source file(s). Depending on a flag, lhs2TEX can produce either the one or the other. Because the flag can be defined via a command line option (cf. Section 9), 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.
- Alternatively, if Haskell code has to be annotated for lhs2TEX to produce aesthetically pleasing output, one can 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! Section 11.4 contains an example that puts this technique to use.

The lhs2TEX library files use conditionals to include different directives depending on the style selected, but they also use conditionals to provide additional or modified behaviour if some flags are set. These flags are `underlineKeywords`, `spacePreserving`, `meta` (activate a number of additional formatting directives), `array` (use `array` environment instead of `tabular` to format code blocks in **math** style; use `parray` instead of `pboxed` in **poly** style), `latex209` (adapt for use with LATEX 2.09 (not supported anymore)), `euler`, and `standardsymbols`. It is likely that these flags will be replaced by a selection of library files that can be selectively included in documents in future versions of lhs2TEX.

11. Typesetting code beyond Haskell

11.1. Spacing

There is no full Haskell parser in lhs2TEX. Instead, the input code is only lexed and subsequently parsed by an extremely simplified parser. The main purpose of the parser is to allow a simple heuristic where to insert spaces into the output while in **math** or **poly** style.

The disadvantage is that in rare cases, this default spacing produces unsatisfying results. However, there is also a big advantage: dialects of Haskell can be processed by lhs2TEX, too. In theory, even completely different languages can be handled. The more difference between Haskell and the actual input language, the more tweaking is probably necessary to get the desired result.

An easy trick to modify the behaviour of lhs2TEX is to insert "dummy" operators that do not directly correspond to constructs in the input language, but rather provide hints to lhs2TEX on how to format something. For instance, spacing can be guided completely by the following two formatting directives:

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

Use ^ everywhere where *no* space is desired, but the automatic spacing of lhs2TEX would usually place one. Conversely, use ^^ everywhere where a space *is* desired, but lhs2TEX does usually not place one.

As described in Section 10, one can use conditionals to format such annotated input code in both **poly** (or **math**) and newcode style to generate both typeset document and code with annotation remove from a single source file. For this to work correctly, one would define

```
%if style == newcode
%format ^  =
%format ^^ = " "
%else
%format ^  = " "
%format ^^ = "\;"
%endif
```

as an extended version of the above. This instructs lhs2TEX to ignore ^ and replace ^^ by a single space while in **newcode** style, and to adjust spacing in other styles, as before.

The examples in the following subsections show these directives in use.

11.2. Inline TEX

Another possibility that can help to trick lhs2TEX into doing things it normally doesn't want to is to insert inline TEX code directly into the code block by using a special form of Haskell comment:

```
{-"⟨tex⟩"-}
```

If this construct appears in a code block, then ⟨*tex*⟩ is inserted literally into the output file. The advantage of this construct over a dummy operator is that if the input language is indeed Haskell, one does not need to sacrifice the syntactic validity of the source program for nice formatting. On the other hand, inline TEX tends to be more verbose than an annotation using a formatting directive.

11.3. AG code example

Here is an example that shows how one can typeset code of the Utrecht University Attribute Grammar (UUAG) ([3]) system, which is based on Haskell, but adds additional syntactic constructs.

The input

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

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

produces the following 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*

11.4. Generic Haskell example

Another example of a Haskell variant that can be typeset using lhs2TEX using some annotations is Generic Haskell [5].

This is a possible input file, including the directives necessary to be able to process it in both **newcode** and **poly** style.

```
%if style == newcode
%format ^
%format ^^    = " "
%format ti(a) = "{|" a "|}"
%format ki(a) = "{[" a "]}"
%else
%format ^     = " "
%format ^^    = "\;"
%format ti(a) = "\lty " a "\rty "
%format ki(a) = "\lki " a "\rki "
\newcommand{\lty}{\mathopen{\{\mskip-3.4mu|}}
\newcommand{\rty}{\mathclose{|\mskip-3.4mu\}}}
\newcommand{\lki}{\mathopen{\{\mskip-3.5mu[}}
\newcommand{\rki}{\mathclose{]\mskip-3.5mu\}}}
%format t1
%format t2
%format a1
%format a2
%format r_    = "\rho "
%format s_    = "\sigma "
%format k_    = "\kappa "
%format forall a = "\forall " a
%format .     = "."
%format mapa  = map "_{" a "}"
%format mapb  = map "_{" b "}"
%format :*:   = "\times "
%endif
\begin{code}
type Map^ki(*)         t1  t2       =   t1 -> t2
type Map^ki(r_ -> s_)  t1  t2       =   forall a1 a2. Map^ki(r_) a1 a2
                                          -> Map^ki(s_) (t1 a1) (t2 a2)

map^ti(t :: k_)                     ::  Map^ki(k_) t t
map^ti(Unit)             Unit       =   Unit
map^ti(Int)              i          =   i
map^ti(Sum)   mapa mapb  (Inl  a)   =   Inl  (mapa a)
map^ti(Sum)   mapa mapb  (Inr  b)   =   Inr  (mapb b)
map^ti(Prod)  mapa mapb  (a :*: b)  =   mapa a :*: mapb b
\end{code}
```

Processed in **poly** style, the output looks as follows:

$$
\begin{array}{lll}
\textbf{type}\ Map\{\![*]\!\} & t_1\ t_2 & = t_1 \to t_2 \\
\textbf{type}\ Map\{\![\rho \to \sigma]\!\}\ t_1\ t_2 & & = \forall a_1\ a_2.Map\{\![\rho]\!\}\ a_1\ a_2 \\
 & & \quad \to Map\{\![\sigma]\!\}\ (t_1\ a_1)\ (t_2\ a_2) \\
map\{\!|t :: \kappa|\!\} & & :: Map\{\![\kappa]\!\}\ t\ t \\
map\{\!|Unit|\!\} & Unit & = Unit \\
map\{\!|Int|\!\} & i & = i \\
map\{\!|Sum|\!\}\ map_a\ map_b\ (Inl\ a) & & = Inl\ (map_a\ a) \\
map\{\!|Sum|\!\}\ map_a\ map_b\ (Inr\ b) & & = Inr\ (map_b\ b) \\
map\{\!|Prod|\!\}\ map_a\ map_b\ (a \times b) & & = map_a\ a \times map_b\ b
\end{array}
$$

11.5. Calculation example

The following example shows a calculational proof. The input

```
\def\commentbegin{\quad\{\ }
\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}
```

produces

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

12. Calling `hugs` or `ghci`

It is possible to call `ghci` or `hugs` using the `%options` directive. In all but the two **code** styles, lhs2TEX looks for calls to the **TEX commands** `\eval` and `\perform` and feeds their arguments to the Haskell interpreter selected.

The current input file will be the active module. This has a couple of consequences: on the positive side, values defined in the current source file may be used in the expressions; on the negative side, the feature will only work if the current file is accepted as legal input by the selected interpreter.

If the command line in the `%options` directive starts with `ghci`, then lhs2TEX assumes that `ghci` is called; otherwise, it assumes that `hugs` is called. Depending on the interpreter, lhs2TEX will use some heuristics to extract the answer from the output of the interpreter. After this extraction, the result will either be printed as inline verbatim (for a `\perform`) or as inline code (for `\eval`), to which formatting directives apply.

WARNING: This feature is somewhat fragile: different versions of `ghci` and `hugs` show different behaviour, and the extraction heuristics can sometimes fail. Do not expect too much from this feature.

12.1. Calling `ghci` – example

The following input shows an example of how to call `ghci`:

```
%options ghci -fglasgow-exts

> fix    ::  forall a. (a -> a) -> a
> 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':))}.
```

The option `-fglasgow-exts` is necessary to make `ghci` accept the `forall` keyword (it only serves as an example here how to pass options to the interpreter). The output will look similar to this:

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

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

Note that it is possible to pass interpreter commands such as `:t` to the external program.

12.2. Calling `hugs` – example

The same could be achieved using `hugs` instead of `ghci`. For this simple example, the output is almost indistinguishable, only that `hugs` usually does not print type signatures using explicit quantification and tends to use different variable names.

```
%options hugs -98

> fix    ::  forall a. (a -> a) -> a
> 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':))}.
```

The input is the same except for the changed `%options` directive. The output now looks as follows:

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

This function is of type **?hugs not found?**, and *take* 10 (*fix* (`'x':`)) evaluates to **?hugs not found?**.

12.3. Using a preprocessor

The situation is more difficult if the current lhs2TEX source file is not valid input to the interpreter, because annotations were needed to format some Haskell extensions satisfactory. The following input file makes use of Template Haskell, and uses the formatting directives for both **newcode** and **poly** style. The `%options` directive instructs `ghci` to use lhs2TEX itself as the literate preprocessor, using the `-pgmL` option of `ghci`. The lhs2TEX binary itself acts as a suitable literate preprocessor if the `--pre` command line option is passed, which is achieved using the `-optL--pre` option:

```
%format SPL(x) = $ ( x )
%if style == newcode
%format QU(x)  = [ | x | ]
%format ^^     = " "
%else
%format QU(x)  = "\llbracket " x "\rrbracket "
%format ^^     = "\; "
%endif

%options ghci -fth -pgmL ../lhs2TeX -optL--pre

This is a rather stupid way of computing |42| using
Template Haskell:

> answer = SPL(foldr1 (\x y -> QU(SPL(x) + SPL(y))) (replicate 21 ^^ QU(2)))

The answer is indeed \eval{answer}.
```

This is the corresponding output:

This is a rather stupid way of computing 42 using Template Haskell:

$$answer = \$(foldr1\ (\lambda x\ y \to [\![\$(x) + \$(y)]\!])\ (replicate\ 21\ [\![2]\!]))$$

The answer is indeed 42.

13. Advanced customization

There is one directive that has not yet been described: `%subst`. This directive is used by lhs2TEX to customize almost every aspect of its output. The average user will and should not need to use a `%subst` directive, but if one wants to influence the very nature of the code generated by lhs2TEX, the `%subst` directives provide a way to do it.

If one would, for instance, want to generate output for another TEX format such as plainTEX or ConTEXt, or if one would want to use a different package than `polytable` to do the alignment on the TEX side, then the `%subst` directives are a good place to start. The default definitions can be found in `lhs2TeX.fmt`.

Table 3 shows only a short description of the approximate use of each of the categories.

14. Pitfalls/FAQ

The document consists of multiple files. Can lhs2TEX be used? One option is to use `%include` rather than LATEX commands to include all files in the master file. The other is to process all files that contain code *and* the master file with lhs2TEX. All files to be processed with lhs2TEX must contain an `%include lhs2TeX.fmt` (or `%include polycode.fmt`) statement. From version 1.11 on, including `lhs2TeX.sty` is no longer necessary.

Yes, but the master file should be pure LATEX. Create a file `mylhs2tex.lhs` with just one line, namely `%include lhs2TeX.fmt`. Process that file with lhs2TEX, using the options you also use for the other included files. Call the resulting file `mylhs2tex.sty` and say `\usepackage{mylhs2tex}` at the beginning of your master file.

The spacing around my code blocks is bad (nonexistent) in "poly" style. Add the line `%include polycode.fmt` to the preamble of your document.

LATEX complains when using lhs2TEX in "poly" style with the `beamer` package. Add the line `%include polycode.fmt` to the preamble of your document.

LATEX complains when using lhs2TEX in "poly" style with the `jfp` class. Add the line `%include jfpcompat.fmt` to the preamble of your document.

LATEX claims that the package `polytable` (or `lazylist`) cannot be found, or that the version installed on your system is too old. Did you install `polytable.sty` (or `lazylist.sty`) in your TEX system manually? If you have absolutely no idea how to

`thinspace`	how to produce a small quantity of horizontal space
`space`	how to produce a normal horizontal space
`newline`	how to produce a new line inside a code block
`verbnl`	how to produce a new line in lhs2TEX generated verbatim
`blankline`	how to translate a blank line in a code block
`dummy`	how to display a missing argument in a formatted function
`spaces` *a*	how to format the whitespace contained in *a*
`special` *a*	how to format the special character *a*
`verb` *a*	how to format the (already translated) inline verbatim text *a*
`verbatim` *a*	how to format an (already translated) verbatim block *a*
`inline` *a*	how to format (already translated) inline code *a*
`code` *a*	how to format an (already translated) code block *a*
`conid` *a*	how to format an identifier starting with an upper-case character *a*
`varid` *a*	how to format an identifier starting with a lower-case character *a*
`consym` *a*	how to format a constructor symbol *a*
`varsym` *a*	how to format a variable symbol *a*
`backquoted` *a*	how to format a backquoted operator *a*
`numeral` *a*	how to format a numeral *a*
`char` *a*	how to format a character literal *a*
`string` *a*	how to format a literal string *a*
`comment` *a*	how to format an (already translated) one-line comment *a*
`nested` *a*	how to format an (already translated) nested comment *a*
`pragma` *a*	how to format an (already translated) compiler pragma *a*
`tex` *a*	how to format inlines TEX code
`keyword` *a*	how to format the Haskell keyword *a*
`column1` *a*	how to format an (already translated) line *a* in one column in **math** style
`hskip` *a*	how to produce a horizontal skip of *a* units
`phantom` *a*	how to produce horizontal space of the width of the (already translated) text *a*
`column3` *a*	how to format an (already translated) line *a* in three columns in **math** style
`fromto` *b e a*	how to format a column starting at label *b*, ending at label *e*, containing the (already translated) code *a* in **poly** style
`column` *n a*	how to define a column of label *n* with (already processed) format string *a* in **poly** style
`centered`	the format string to use for a centered column
`left`	the format string to use for a left-aligned column
`dummycol`	the format string to use for the dummy column (a column that does not contain any code; needed due to deficiencies of the `polytable` implementation)
`indent` *n*	how to produce an indentation (horizontal space) of *n* units

Table 3: A short description of the `%subst` directives

do this, you may try to copy both `polytable.sty` and `lazylist.sty` from the lhs2TEX distribution into your working directory.

Haskell strings are displayed without double quotes. This is a result from using an old `lhs2TeX.fmt` file together with a new version of lhs2TEX. Usually, this stems from the fact that there is an old version in the working directory. Now, lhs2TEX maintains a search path for included files, thus usually a local old copy of `lhs2TeX.fmt` can be removed.

In "math" style, I have aligned several symbols on one column, but lhs2TEX still won't align the code block. Did you set the alignment column correctly using the `%align` directive? Note also that lhs2TEX starts counting columns beginning with 1, whereas some editors might start counting with 0.

Large parts of the formatted file look completely garbled. Passages are formatted as code or verbatim, although they are plain text. Conversely, things supposed to be code or verbatim are typeset as text. You probably forgot multiple `|` or `@` characters. Because lhs2TEX identifies both the beginning and end of inline code or inline verbatim via the same character, one missing delimiter can confuse lhs2TEX and cause large passages to be typeset in the wrong way. You should locate the first position in the document where something goes wrong and look for a missing delimiter at the corresponding position in the source file.

LATEX complains about a "nested `\fromto`" in "poly" style. This usually is a problem with one of your formatting directives. If you start a TEX group in one of your directives but do not close it, then this error arises. You should not write such unbalanced formatting directives unless you make sure that they do never span an aligned column.

References

[1] Andres Löh. *The* `polytable` *package.* http://ctan.org/tex-archive/macros/latex/contrib/polytable/

[2] Alan Jeffrey. *The* `lazylist` *package.* http://ctan.org/tex-archive/macros/latex/contrib/lazylist/

[3] Arthur Baars, S. Doaitse Swierstra, Andres Löh. *The UU AG System User Manual.* http://www.cs.uu.nl/~arthurb/data/AG/AGman.pdf

[4] Frank Mittelbach and David Carlisle. *The* `array` *package.* http://www.ctan.org/tex-archive/macros/latex/required/tools/array.dtx

[5] Andres Löh. *Exploring Generic Haskell.* PhD Thesis, Utrecht University, 2004.

[6] Hackage http://hackage.haskell.org

[7] The Haskell Platform. http://hackage.haskell.org/platform/

A. Deprecated styles

In this Appendix, we will cover the styles that were omitted from Section 4. We will demonstrate them with the same common example. As before, each style will include a short summary. Some of the points listed in the summary are simply defaults for the particular style and can actually be changed.

A.1. Verbatim: "verb" style

In **verb** style, the code shows up in the formatted document exactly as it has been entered, i.e. verbatim. All spaces are preserved, and a non-proportional font is used.

```
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)
```

One does not need lhs2TEX to achieve such a result. This style, however, does not make use of an internal TEX verbatim construct. The implementation of verbatim environments in TEX is somewhat restricted, and the preprocessor approach may prove more flexible in some situations. For example, it is easier to apply additional formatting instructions to the output as a whole, such as placing the code in a colored box.

Verbatim summary

- formatting directives are ignored
- conditionals and includes are handled
- inline code, inline verbatim, and code blocks are all typeset completely verbatim, using a typewriter font
- all spaces in code blocks are preserved
- plain text is copied unchanged

A.2. Space-preserving formatting with "tt" style

The **tt** style is very similar to **verb** style, but applies a tiny bit of formatting to the code and allows for more customizabilty:

```
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)
```

By default, some of the Haskell symbols are expressed more naturally. For instance, special symbols are being used for the arrows or the lambda. In addition, the user can specify additional formatting directives to affect the appearance of certain identifiers. In this way, keywords can be highlighted, user-defined Haskell infix operators can be replaced by more appropriate symbols etc. In this style, the layout and all spaces from the source file are still preserved, and a non-proportional font is used, as in **verb** style.

Typewriter summary

- non-recursive formatting directives are obeyed
- conditionals and includes are handled
- inline verbatim is typeset as verbatim, whereas inline code and code blocks are typeset almost verbatim, after formatting directives are applied, in a typewriter font using some special symbols to "beautify" some Haskell operators.
- all spaces in code blocks are preserved
- plain text is copied unchanged

A.3. Proportional vs. Monospaced

Usually, there is a tradeoff between restricting oneself to the use of a typewriter font and not using any formatting and using a proportional font, at the same time replacing operators with mathematical symbols, using different font shapes to highlight keywords etc. While the latter offers far more flexibility, the proportional font might destroy (at least part of) the layout that the programmer has employed in order to make the source code more readable.

Compare, for example, the previous two examples with the following result (this is a negative example, lhs2TEX can do far better than that!!):

```
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)
```

While the indentation is kept (otherwise, for the layout sensitive Haskell it would be even disastrous, because the code might no longer be valid), alignment that has been present in the code lines has been lost. For example, in the input the user had decided to align all equality symbols of all three function definitions, and also align them with the "has-type" operator ::.

Without support from a tool like lhs2TEX, the horizontal positions of the equality symbols in the formatted code are totally unrelated. A solution to this problem is of course to put the Haskell code in a LATEX table. Doing this manually, though, is very cumbersome and in some case still quite hard. The task of the formatted styles of lhs2TEX is thus to spare the user the burden of cluttering up the code with formatting annotations. Most of the time, completely un-annotated code can be used to achieve good results, using the fonts you like while maintaining alignment information in the code!

A.4. Alignment and formatting with "math" style

In prior versions of lhs2TEX, **math** style was the mode to use for formatted Haskell code. There is one alignment column, often used to align the equality symbols of several equations. Additionally, indentation is handled automatically. User-defined formatting directives can be used to alter the formatting of identifiers, operators and symbols in many places.

```
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)
```

The example shows that there is still a loss of alignment information compared to the original verbatim example. The three arguments of the *zipWith* function as well as the two guarded equations in the definition of *select* are not aligned. At the moment,

math style exists mainly to maintain compatibility with old documents. New features may be added to **poly** style only.

"math" summary

- all formatting directives are obeyed
- conditionals and includes are handled
- inline verbatim is typeset as verbatim, whereas inline code and code blocks are typeset using a proportional font, using mathematical symbols to represent many Haskell operators.
- indentation in code blocks is preserved; furthermore, alignment on a single column is possible
- plain text is copied unchanged