Using `lhs2TeX` and `sproof`

January 12, 2016

This document explains how to make use of

- `lhs2TeX`, for typesetting Haskell code in LaTeX, and
- the `sproof` environment, for typesetting structured proofs.

The document itself explains most of what you need to know, but for even more insight you can take a look at the LaTeX source used to create this document as well.

1 Typesetting Haskell with `lhs2TeX`

`lhs2TeX` is a tool for nicely typesetting *literate Haskell* files with LaTeX. In fact—as with this very document—they do not really have to be literate Haskell per se, but just LaTeX documents with some Haskell code interspersed.

To install `lhs2TeX`, it should suffice to issue the command

```
cabal install lhs2TeX
```

To use it, start by creating a LaTeX document within a `.lhs` file (if you use `emacs`, you can tell it to use `latex-mode` instead of literate Haskell mode by putting `% -*- mode: LaTeX -*-` at the top of your file). At the top of your file, immediately following the `\documentclass` command, you should put

```
%include polycode.fmt
```

which will instruct `lhs2TeX` to import a standard library of tools and definitions.

Within your LaTeX document, you can include snippets of Haskell code by enclosing them in vertical bars. For example,

```
|length . map f == length|
```

gets typeset as $length \circ map\ f \equiv length$. Note that there are several styles for typeset Haskell code. The default is to use italics and nice "mathy" symbols like $\circ$ in place of `.`, $\equiv$ in place of `==`, $\leqslant$ in place of `<=`, and so on. This looks nice but can sometimes be harder to read if you are not used to it. If you would rather have good old typewriter font used for your Haskell code, you can pass the `--verb` option to `lhs2TeX`.

You can also include larger sections of Haskell code by enclosing them in

```
\begin{code} ... \end{code}
```

or

```
\begin{spec} ... \end{spec}
```

(The only difference between the two is that anything in a `spec` environment will be ignored when the file is loaded into `ghci` or compiled with `ghc`.) For example,

```
\begin{code}
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs)  = f x : map f xs
\end{code}
```

is typeset as

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f\ [\,] \qquad = [\,] \\
&map\ f\ (x : xs) = f\ x : map\ f\ xs
\end{aligned}
$$

To achieve the nice vertical alignment of the equals signs (or anything else), vertically align them in the source code, taking care to leave at least *two* spaces before each of the things to be aligned (note how `map f (x:xs)  = f x ...` actually has two spaces before the `=` sign).

`lhs2TeX` acts as a preprocessor that takes `.lhs` files and produces `.tex` files. You can use it with a command line like

```
lhs2TeX myfile.lhs > myfile.tex
```

followed by compiling the `.tex` file normally (*e.g.* with `pdflatex`). Recall that you can also pass the `--verb` option to `lhs2TeX` to use a typewriter font for your Haskell code.

There is much, much more you can do with `lhs2TeX`, including automatically typesetting variable names like `x2` as x_2, introducing your own custom formatting for certain functions or operators, and automatically evaluating Haskell expressions via `ghci` and inserting their output into your document. For more information, see `http://www.andres-loeh.de/lhs2tex/`. It appears there is as of yet no manual that specifically accompanies the latest release (1.19), but the 1.17 manual should work just fine (`http://www.andres-loeh.de/lhs2tex/Guide2-1.17.pdf`).

2 Typesetting structured proofs with `sproof`

The first step is to download `sproof.sty`. You can put this file somewhere LaTeX knows to look for it, or (much simpler) just put a copy of it in the same folder as any `.tex` files you want to use it with. To use it, include `\usepackage{sproof}` in your document preamble.

Structured proofs are created with the `sproof` environment. Each expression or statement is typeset with the `\stmt{...}` command, which automatically places its contents in math mode. In between the expressions/statements, you can use the `\reason{...}{...}` command. The first argument to `\reason` is a transitive binary operator (in math mode); the second argument is a justification (in text mode). So, for example,

```
\begin{sproof}
  \stmt{1 + 1 + 1}
  \reason{=}{arithmetic}
  \stmt{1 + 2}
  \reason{=}{more arithmetic}
  \stmt{3}
  \reason{\leq}{duh}
  \stmt{5}
\end{sproof}
```

produces

$$
\begin{array}{ll}
& 1 + 1 + 1 \\
= & \quad \{ \quad \text{arithmetic} \quad \} \\
& 1 + 2 \\
= & \quad \{ \quad \text{more arithmetic} \quad \} \\
& 3 \\
\leqslant & \quad \{ \quad \text{duh} \quad \} \\
& 5
\end{array}
$$

As another example, consider typesetting an inductive proof of the fact that for all lists xs and functions f, $\mathit{length}\ (\mathit{map}\ f\ xs) = \mathit{length}\ xs$. The empty list case is simple enough; consider the case when xs is a cons, $y : ys$. As our inductive hypothesis, we get to assume

$$\mathit{length}\ (\mathit{map}\ f\ ys) = \mathit{length}\ ys,$$

and we must show the same holds for $y : ys$. We reason as follows:

$$
\begin{array}{ll}
& \mathit{length}\ (\mathit{map}\ f\ (y : ys)) \\
= & \quad \{ \quad \text{defn of } \mathit{map} \quad \} \\
& \mathit{length}\ (f\ y : \mathit{map}\ f\ ys) \\
= & \quad \{ \quad \text{defn of } \mathit{length} \quad \} \\
& 1 + \mathit{length}\ (\mathit{map}\ f\ ys) \\
= & \quad \{ \quad \text{IH} \quad \} \\
& 1 + \mathit{length}\ ys \\
= & \quad \{ \quad \text{defn of } \mathit{length} \quad \} \\
& \mathit{length}\ (y : ys)
\end{array}
$$