HATEX 3

The User's Guide, version 1.3.1.2 (using HaTeX 3.16.2.0)

https://github.com/Daniel-Diaz/HaTeX

Main author: *Daniel Díaz (dhelta.diaz@gmail.com)*

Contributors:

GetContented

Date of creation: May 21, 2016.

Contents

1 Preface

1.1 Introduction

If you are here to learn more about HaTeX, or are just curious, you are in the right place. First of all, note that this guide is aimed at those who already know the basics of both Haskell and LaTeX. If you don't, first try to learn some of each (both are quite useful). To learn Haskell, start with some tutorials and suggestions at the excellent Haskell website [`http://haskell.org`]. To learn LaTeX, start with *The not so short introduction to LaTeX* [`http://tobi.oetiker.ch/lshort/lshort.pdf`].

The HaTeX library aspires to be the tool with which Haskellers want to construct their LaTeX documents while working within their beloved language. HaTeX tries to be as comprehensive and well-constructed as possible. Do you still think something could be better? Is something lacking, perhaps? If so, go to the HaTeX mailing list [`http://projects.haskell.org/cgi-bin/mailman/listinfo/hatex`] and complain without mercy! Or, if you are a GitHub user, create an issue [`https://github.com/Daniel-Diaz/HaTeX/issues`] or, to be even more awesome, create a patch and send a pull request. This is one of the great things about open source projects!

1.2 What is HaTeX?

Before explaining *how* HaTeX works, let's state *what* HaTeX actually is.

HaTeX is a Haskell library that provides functions to create, manipulate and parse LaTeX code.

People often say that *HaTeX is a LaTeX DSL*, or Domain Specific Language. With it, you can enjoy all the many advantages of Haskell while creating LaTeX documents. A common use is the automatic creation of such documents, perhaps from a Haskell data source. A more exotic one would be to render chessboard situations. Possibilities are limited only by the imagination. The goal is: if you can do it with LaTeX, you can do it with HaTeX, while taking advantage of all that Haskell offers.

2 Basics

Through this section you will learn the basics of HaTeX. Essentially, *how* it works.

2.1 The Monoid class

If you are already familiar with the `Monoid` class, jump to the next point. The `Monoid` class is something that you must get used to in Haskell. But don't worry, it is quite simple (despite having a similar name to the `Monad` class). A *monoid* in Mathematics is an algebraic structure consisting of a set of objects, an *associative* operation and a *neutral element*. Phew! But what is the meaning of this? By *associative* we mean that, if you have three elements a, b and c, then $a * (b * c) = (a * b) * c$. A *neutral element* is one that does not change other values when operated with, because it means nothing with respect to the operation! To say, e is a *neutral element* if $e * a = a * e = a$, given any object a. As an example, you may take the *real numbers* as objects and the ordinary multiplication as operation, in which case the *neutral element* would be the number one.

Now that you know the math basics behind the `Monoid` class, let's see its definition:

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
```

See that `mappend` corresponds to the monoid operation and `mempty` to its neutral element. The names of the methods may seem unsuitable, but they correspond to a particular case of monoid: the lists with the appending `(++)` operation. What is the neutral element here? The empty list:

```
xs ++ [] = [] ++ xs = xs
```

This class plays a significant role in HaTeX. Keep reading.

2.2 LaTeX blocks

Suppose we have a well-formed[1] piece of LaTeX code, call it a. Now, let `LaTeX` be a Haskell type in which each element represents a well-formed piece of LaTeX code. Then, a can be seen as a Haskell expression `a` of type `LaTeX`. We can say that `a` is a `LaTeX` **block**. What happens if we append,

[1] By *well-formed* we mean all braces, environments, math expressions, ... are closed.

by juxtaposition, two `LaTeX` blocks? As both are well-formed, so is the result. Thus, two blocks appended form another block. This way, we can define an operation over the `LaTeX` blocks. If we consider that a totally empty code is a well-formed piece of LaTeX code, we can speak about the empty block. And, as the reader may notice, these blocks with the append operation form a monoid. Namely, `LaTeX` is an instance of the `Monoid` class.

Of course, our objective when using HaTeX is to create a `LaTeX` block that fits our purpose. The way to achieve this is to create a multitude of `LaTeX` blocks and use the `Monoid` operation to collapse them into a single block.

2.3 Creating blocks

We now have a universe of blocks that form a monoid. What we need is a way to create these blocks. As we said, a block is the representation of a well-formed piece of LaTeX code. Let `a` be the block of the LaTeX expression `\delta{}`[2]. Since this is a constant expression, it has a constant value in Haskell, named `delta`. Calling this value will generate the desired block.

Other LaTeX expressions depend on a given argument. For example `\linespread{x}`, where `x` is a number. How do we use these? As you would expect, with functions. We can create blocks that depend on values with functions that take these values as arguments, where these arguments can be blocks as well. For instance, we have the function `linespread` with type:

```
linespread :: Float -> LaTeX
```

As you may know, a title in LaTeX can itself contain LaTeX code. So the type for the Haskell function `title` is:

```
title :: LaTeX -> LaTeX
```

And this is essentialy the way we work with HaTeX: **to create blocks and combine them**. Once you have your final block ready, you will be able to create the LaTeX code that corresponds to it (we will see how later). Note that there is LaTeX code for every block, but not every piece of LaTeX has a block, because a malformed (in the sense of the negation of our well-formed concept) code doe **not** have a corresponding block. This fact has a practical consequence: **we cannot create malformed LaTeX code** using HaTeX. *And that's a good thing!*

[2] Please, note that the `LaTeX` block is **not** the same that the LaTeX expression. The former is a Haskell value, not the LaTeX code itself.

2.3.1 From strings

Inserting text in a LaTeX document is a constant task. You can create a block with text given an arbitrary `String` with the `fromString` function, method of the `IsString` class:

```
class IsString a where
 fromString :: String -> a
```

Since there is a set of characters reserved to create commands or another constructions, HaTeX takes care to avoid using them, replacing each with a command whose output looks like the originally intended character. For example, the backslash `\` is replaced with the `\backslash{}` command.

The function that avoids reserved characters is exported with the name `protectString`. Also, there is a variant for `Text` values called `protectText`.

The use of the `IsString` class is because the *Overloaded Strings* extension. This is similar to the *Overloaded Numbers* Haskell feature, which translates the number `4` to `fromInteger 4`. In a similar way, with `OverloadedStrings` enabled, the string `"foo"` is translated to `fromString "foo"`. If we now apply this to our blocks, the string `"foo"` will be automatically translated to a `latex` block with *foo* as content. Quite handy! We will assume that the `OverloadedStrings` extension is enabled from now on.

2.3.2 More blocks

There are a lot of functions to create blocks. In fact, we can say this is the primary purpose of the library. LaTeX has a lot of commands: they can set font attributes, create tables, insert graphics, include mathematical symbols, etc. So HaTeX has a function for each command defined in LaTeX (to tell the truth, only for a small subset). Please go to the API documentation to read about particular functions - you can either build it locally or find it on Hackage: `http://hackage.haskell.org/package/HaTeX`. You will find the class constraint `LaTeXC l` in every entity. `LaTeX` is an instance of this class, so you can think of `l` as the `LaTeX` datatype without any problem. There is more about this in the section about the `LaTeXC` class.

2.4 Putting blocks together

Once you have your blocks, as we said before, you need to join them. The `mappend` method of the `Monoid` class will do this. If `a` and `b` are two blocks, `mappend a b`, or `a `mappend` b`, or even `a <>`

`b`[3], return the juxtaposition of `a` and `b`. For lists of blocks, you can use `mconcat` instead as follows:

```
mconcat [ "I can see a "  , textbf "rainbow"
        , " in the blue " , textit "sky" , "." ]
```

2.5 Rendering

This is the last step in our LaTeX document creation. When we have our final LaTeX block `a`, the function `renderFile` can output it to a file, in the form of its correspondent LaTeX code.

Say we have this definition:

```
short =
    documentclass [] article
 <> title "A short message"
 <> author "John Short"
 <> document (maketitle <> "This is all.")
```

Then, after calling `renderFile "short.tex" short`, the following file will appear in the current working directory (line breaks added for easier visualization):

```
\documentclass{article}
\title{A short message}
\author{John Short}
\begin{document}
\maketitle{}
This is all
\end{document}
```

Finally, you may use commands like `latex` or `pdflatex` in your command line environment to compile the LaTeX output to dvi or pdf.

The function `renderFile` is not only for `LaTeX` values. Let's see its type:

```
renderFile :: Render a => FilePath -> a -> IO ()
```

[3] From **GHC 7.4**, `(<>)` is defined as a synonym for `mappend`. For previous versions of GHC, HaTeX exports the synonym.

The `Render` class that appears in the context is defined as:

```
class Render a where
 render :: a -> Text
```

So, it is the class of types that can be rendered to a `Text` value. The type `LaTeX` is an instance, but other types, like `Int` or `Float`, are too. These instances are useful for creating blocks from other values. With the function `rendertex`, any value in the `Render` class can be transformed to a block. First, the value is converted to `Text`, and then to `LaTeX` in the same way as we did with strings. But, **be careful** because `rendertex` does **not** escape reserved characters.

2.6 Try it yourself

As always, the best way to learn something well is to try it for yourself. Since looking at code examples can help you greatly, HaTeX comes with several examples at [`https://github.com/Daniel-Diaz/HaTeX/tree/master/Examples`] so you can see for yourself how to accomplish various tasks.

The API reference is also a good reference to keep in mind. Descriptions of functions allow you to know exactly how they work. Even when these are not present, just the function names and their type signatures can be very helpful and descriptive.

3 LaTeX blocks and the Writer monad

3.1 The Writer Monad

For any given monoid, `M`, the `M`-writer monad is just all possible pairs of elements from `M` and elements from other types. Thus, the Haskell declaration is as follows[4]:

```
data W m a = W m a
```

Note that to get the monad we need to fix the type `m` (the kind of monads is `* -> *`). To inject an arbitrary value into the monad (the Haskell `return` function) we use the neutral element (`mempty`) of the monoid.

```
inject :: Monoid m => a -> W m a
inject a = W mempty a
```

Think about the element of `m`: there is only one element that it could be! Like any other monad, `W m` is also a `Functor`. We just apply the function to the value.

```
instance Functor (W m) where
 fmap f (W m a) = W m (f a)
```

Every `Monad` instance can be given by the two monad operations `inject` and `join`. We already defined the `inject` function. The other one deletes one monad type constructor.

```
join :: Monoid m => W m (W m a) -> W m a
join (W m (W m' a)) = W (mappend m m') a
```

In this function we use the other `Monoid` method to combine both values. It is important to note that in both monad operations `inject` and `join` we used `mempty` and `mappend` respectively. In practice, this is because they act similarly to each other. Indeed, they are equal if we forget the `a` value. Now, we are ready to define the `Monad` instance:

```
instance Monoid m => Monad (W m) where
 return  = inject
 w >>= f = join (fmap f w)
```

[4] Some authors write it using tuples, like this: `data W m a = W (a,m)`.

There is nothing to say about this instance. It is a standard definition, valid for any monad.

What we have done here is to hide a monoid in a monad, with all its operations. We have created a machine that operates on monoidal values. To insert a value into the machine we need the `tell` function:

```
tell :: m -> W m ()
tell m = W m ()
```

When we execute the machine, it returns the result of operating on all the values we have put into it.

```
execute :: W m a -> m
execute (W m a) = m
```

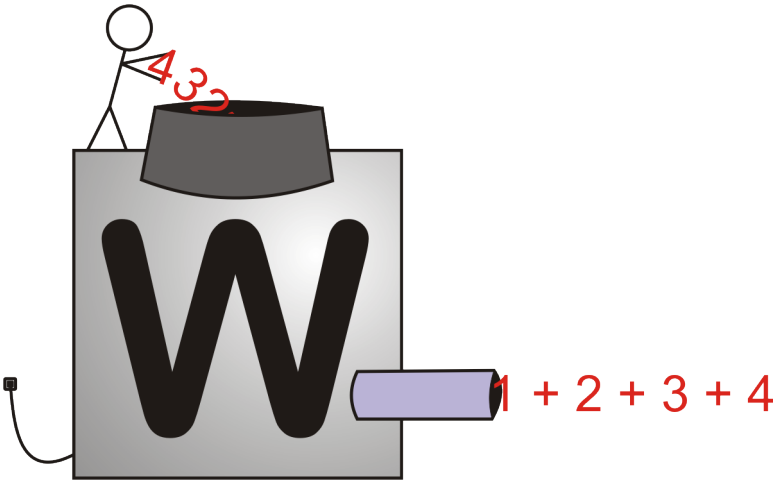
Let's see the machine working. For example, the `Int` type with addition forms a `Monoid`.

```
instance Monoid Int where
 mempty = 0
 mappend = (+)

example :: Int
example = execute $ do
  tell 1
  tell 2
  tell 3
  tell 4
```

When we evaluate `example` we get `10`, as expected. Using `mapM_` we can rewrite `example`.

```
example :: Int
example = execute $ mapM_ tell [ 1 .. 4 ]
```



3.2 The LaTeX Monad

Let's go back to the `LaTeX` type. Since `LaTeX` is an instance of `Monoid` we can construct its corresponding `Writer` monad.

```
type LaTeXW = W LaTeX
```

The `W` machine is now waiting for `LaTeX` values.

```
example :: LaTeX
example = execute $ do
  tell $ documentclass [] article
  tell $ author "Monads lover"
  tell $ title "LaTeX and the Writer Monad"
```

We put all these blocks into the machine, and it returns the concatenated block for us. We just saved a lot of `mappend`'s, but we now have a lot of `tell`'s instead. No problem, just redefine each function of blocks using `tell` and `execute`.

```
author' :: LaTeXW a -> LaTeXW ()
author' = tell . author . execute
```

If this is done in a similar way to `documentclass` and `title`, every `tell` in `example` disappears.

```
example :: LaTeX
example = execute $ do
  documentclass' [] article
  author' "Monads lover"
  title' "LaTeX and the Writer Monad"
```

And we can now use the `LaTeX` machine more comfortably. However, we have duplicated all of our functions. This is why the `LaTeXC` class exists. We'll talk about this later.

3.3 Composing monads

To add flexibility to HaTeX, the writer monad explained above is defined as a monad transformer, named `LaTeXT`. The way we use it is the same, with just a few small changes.

The first change is in the type signature. We need to carry an inner monad in every type.

```
foo :: Monad m => LaTeXT m a
```

However, in practice, we can avoid this by using type aliases. Say we're going to use a specific monad `M`.

```
type LaTeXW = LaTeXT M

foo :: LaTeXW a
```

Now, the type signatures go back to the way they were.

The other change is a new feature: the `lift` function. With it we can do any computation on our inner monad at any time. For example, suppose we want to output some code we have in the file *foo.hs*. Instead of copying all of its content, or reading and carrying it as an argument along in the code, you can simply read that file using `lift` wherever you want.

```
type LaTeXIO = LaTeXT IO

readCode :: FilePath -> LaTeXIO ()
readCode fp = lift (readFileTex fp) >>= verbatim . raw

example :: LaTeXIO ()
```

```
example = do
  "This is the code I wrote this morning:"
  readCode "foo.hs"
  "It was a funny exercise."
```

Different monads will each give different features. In the case where we're not interested in any of these features, we can simply use the Identity monad.

```
type LaTeXW = LaTeXT Identity
```

4 The LaTeXC class

HaTeX has two different interfaces. One uses blocks as `Monoid` elements and the other as `Monad` actions. If we want to keep both interfaces we have two choices: to duplicate function definitions[5] or to have a typeclass which unifies both interfaces. Since having duplicate definitions is hard work and can raise many problems[6], we took the second alternative and defined the `LaTeXC` typeclass. Both `LaTeX` and `LaTeXT m a` are instances of `LaTeXC` (the second one is a little tricky), so every function in HaTeX is defined using this typeclass. This way, we can have both interfaces with a single import, without being worried about maintaining duplicated code. The cost for this is that we must have class constraints in our type signatures. However, these constraints are only required in the package. At the user level, you choose your interface and write type signatures correspondingly.

[5] This was the approach taken in HaTeX 3 until the version 3.3, where the `LaTeXC` class was included.

[6] In fact, we had a problem with HaTeX-meta, the program that automatically generated the duplicated functions. The problem was described in the following blog post: `http://deltadiaz.blogspot.com.es/2012/04/hatex-trees-and-problems.html`.

5 Packages

LaTeX, in addition to its predefined commands, has a big number of packages that increase its power. HaTeX functions for some of these packages are defined in separate modules, with one or more modules per package. This way you can import only the functions you actually need. Some of these modules are explained below.

5.1 Inputenc

This package is of vital importance if you use non-ASCII characters in your document. For example, if my name is *Ángela*, the *Á* character will not appear correctly in the output. To solve this problem, use the `Inputenc` module.

```
import Text.LaTeX.Base
import Text.LaTeX.Packages.Inputenc

thePreamble :: LaTeX
thePreamble =
    documentclass [] article
 <> usepackage [utf8] inputenc
 <> author "Ángela"
 <> title "Issues with non-ASCII characters"
```

Don't forget to set to UTF-8 encoding in your Haskell source too.

5.2 Graphicx

With the `Graphicx` package, you can insert images in your document and do some other transformations to them. In order to insert an image, use the `includegraphics` function.

```
includegraphics :: LaTeXC l => [IGOption] -> FilePath -> l
```

The list of `IGOption`'s allows you to set some properties of the image like width, height, scaling or rotation. See the API documentation for details.

6 Epilogue

6.1 Notes about this guide

This guide is not static. It will certainly be changed as time passes. Any reader can also help participate in its writing, since the guide is itself open source (and written in Haskell!). The source repository can be found at: `https://github.com/Daniel-Diaz/hatex-guide`. You can read more detailed instructions in the README file.

If you think something is unclear, or hard to understand, please do take the time to report it. We really appreciate it.

6.2 Notes from the author

I would like to end this guide by saying thanks to all the people that have been interested in HaTeX in any way, especially to those who contributed to it with patches, opinions and\/or bug reports. **Thanks**.