HATEX 3

The User's Guide, version 1.0 (using HaTeX 3.6.1)

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

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

Date of creation: July 3, 2013.

Contents

1 Preface

1.1 Introduction

If you are here because you want to learn more about HaTeX, or just feel curious, you are in the right place. First of all, note that this guide is addressed to that people that already knows the basics of both Haskell and LaTeX. Otherwise, try to learn first a bit of these languages (both are quite useful learnings). To learn Haskell, though I guess you already learned it since you are reading these lines, go to the Haskell web [`http://haskell.org`] and search for some tutorials or books. To learn LaTeX, you can start with *The not so short introduction to LaTeX* [`http://tobi.oetiker.ch/lshort/lshort.pdf`].

The HaTeX library aspires to be the tool that Haskellers could want to make their LaTeX things without exit of their language (we understand that is difficult to leave Haskell after the first date), trying to be the most comprehensive and well done as possible. Do you think, anyway, that something could be done better? Perhaps something is lacked? Go then to the HaTeX mailing list [`http://projects.haskell.org/cgi-bin/mailman/listinfo/hatex`] and leave your complain without mercy! Or, in the case you are a GitHub user, say your word in the issue list [`https://github.com/Daniel-Diaz/HaTeX/issues`] or, to be awesome, make yourself a patch and send a pull request. This is the great thing about open source projects!

1.2 What is HaTeX?

Before we explain *how* HaTeX works, it is convenient to say *what* actually HaTeX is.

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

People often says that *HaTeX is a LaTeX DSL*. With it you can enjoy all the advantages you already have in Haskell while creating LaTeX documents. A common purpose is to automatize the creation of such documents, perhaps from a source data in Haskell. A more exotic one is to render chess tables. Possibilities are in a wide range. The idea is the following: if you can do it with LaTeX, you can do it with HaTeX, but adding all the Haskell features.

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 (in spite of the similarity in the name with the `Monad` class). A *monoid* in Mathematics is an algebraic structure consisting of a set of objects with an operation between them, being this operation *associative* and with 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 the one that does not worth to operate with, because it does nothing! 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.

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 insuitable, but they correspond to an example of monoid: the lists with the appending `(++)` operation. Who 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

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

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, 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 its appending form a monoid. Namely, `LaTeX` can be done an instance of the `Monoid` class.

Of course, our mission 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, then, use the `Monoid` operation to collapse them all in a single block.

2.3 Creating blocks

We have now a universe of blocks forming a monoid. What we need now 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 we deal with this? As you 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 contain itself LaTeX code. So the type for the Haskell function `title` is:

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

And this is, essentialy, the way to work with HaTeX: **to create blocks and combine them**. Once you have your final block ready, you will be able to create its corresponding LaTeX code (we will see how later). Note that for every block there is a LaTeX code, but not for every code there is a block, because a malformed (in the sense of the negation of our well-formed concept) code

[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.

has **not** a block in correspondence. This fact has a practical consequence: **we cannot create malformed LaTeX code**. *And that's a good deal!*

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 and avoids them replacing each reserved character with a command which output looks like the original character. For example, the backslash `\` is replaced with the `\backslash{}` command.

The function that avoids reserved characteres 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 one 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 the `OverloadedStrings` extension enabled from now.

2.3.2 More blocks

There is a lot of functions for create blocks. In fact, we can say that this is the main purpose of the library. LaTeX has a lot of commands, in order to set font attributes, create tables, insert graphics, include mathematical symbols, etc. So HaTeX have 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. Build it locally or find it in 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 assume that `l` is the `LaTeX` datatype without any problem. More about this in section about the `LaTeXC` class.

2.4 Putting blocks together

Once you have the blocks, as we said before, you need to append them. The `mappend` method of the `Monoid` class does this work. If `a` and `b` are two blocks, `mappend a b`, or ``a `mappend` b``, or even `a <> b`[3], is the block with `a` and `b` juxtaposed. For long lists of blocks, you can try it with `mconcat` 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 into a file, in the form of its correspondent LaTeX code.

Say we have the next definition:

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

Then, after call `renderFile "short.tex" short` it appears the following file in the current working directory (line formatting added for easier visualization):

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

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:

```
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`, so 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` the same way we did with strings. But, **be careful!** Because `rendertex` does **not** escape reserved characters.

2.6 Try yourself

As always, the best way to learn something well is to try it by yourself. Since to see code examples can give you a great help, HaTeX comes with several examples where you can see by yourself how to get the work done.

The API reference is also a good point to keep in mind. Descriptions of functions make you know how exactly they works. And, when they are not present, function names with type signatures may be very helpful and descriptive.

3 LaTeX blocks and the Writer monad

3.1 The Writer Monad

Fixed a 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` (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 that no other element of `m` is possible to think: it is the only element we know of it! 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 equal. 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 and standard definition valid to any monad.

What we have done here is to hide in a monad a monoid with all its operations. We have created a machine that operates monoid 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 to us the result of operate all the values we have put on it.

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

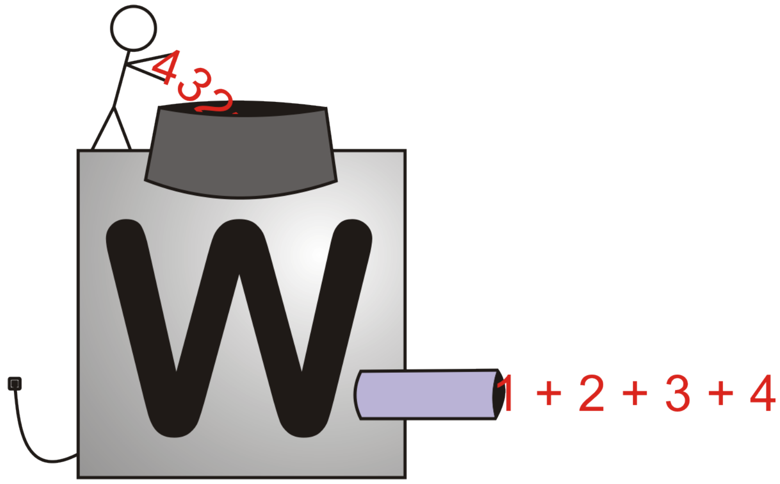
Let's see the machine working. For example, the `Int` type with addition forms a `Monoid`.

```
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 correspondent `Writer` monad.

```
type LaTeXW = W LaTeX
```

The `W` machine is waiting now 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 that blocks in the machine, and it returns the concatenated block. We saved a lot of `mappend`'s, but we now have a lot of `tell`'s. No problem. Just redefine each function of blocks with `tell` and `execute`.

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

If it is done in a similar way with `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 all functions duplicated. This is why the `LaTeXC` class exists. We are going to talk about it 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 to use it is the same, there are just a few changes.

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

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

However, in practice, we can avoid it. Say we going to use an specific monad `M`.

```
type LaTeXW = LaTeXT M

foo :: LaTeXW a
```

Now, type signatures remain unchanged.

The other change is a new feature: the `lift` function. With it we can do any computation of our inner monad at any time. For example, suppose we want to output some code we have in the file *foo.hs*. Instead of copy all its content, or read and carry it as an argument along 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 give different features. In the case we are not interested in any of these features, it is enough to 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 duplicate definitions is a hard work and can arise 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 the typeclass. This way, we have both interfaces with a single import, without being worry about maintaining duplicated code. The cost is to have class constraints in type signatures. But these constraints are only required in the package. At the user level, you choose your interface and write type signatures in consequence.

[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 a 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 separated modules, one module per package. This way, you can import only those functions you actually need. Some of these modules are below explained.

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 your Haskell source too.

5.2 Graphicx

With the `Graphicx` package you can insert images in your document and do some other transformations. 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 be changed, extended and improved with the time. If you think there is something unclear, something hard to understand, please, report it.

6.2 Notes from the author

I would like to end this guide saying thanks to all the people that has been interested in HaTeX somehow, especially to those who contributed to it with patches, opinions or bug reports. **Thanks**.