GHCi for LuaTeX

A persistent ghci session in LuaTeX

Alice Rixte

July 4, 2025

Contents

1 Introduction

GHCi for LuaTeX provides a persistent GHCi session within a LaTeX document. Using the `ghci` package via `\usepackage{ghci}`, it mainly provides the `ghci` environment and the `hask` command which can be used as follows:

```
\begin{ghci}
x :: Int
x = 4

y :: Int
y = 5
\end{ghci}

The sum of $x$ and $y$ when $x = \hask{x}$
and $y = \hask{y}$ is $\hask{x + y}$.
```

2 Getting started

In order to execute the Haskell code, the `ghci4luatex` server must be running. If you are concerned with security issues, you are encouraged to verify the source code at github.com/AliceRixte/ghci4luatex/. In particular, you can make sure that

- the `ghci.sty` package can only connect to the local address `127.0.0.1` and will not attempt to connect to any external service

- the `ghci4luatex` server only processes the commands sent by `ghci.sty`

2.1 Installing the `ghci4luatex` server

You can install `ghci4luatex` either using Cabal, Stack, or directly from source. In all cases, you must have Haskell installed as well as cabal or stack.

To check that `ghci4luatex` is properly installed, run

```
ghci4luatex --version
```

Modulo the version, this should produce de following output `ghci4luatex v0.1, (C) Alice Rixte`

Using cabal

```
cabal install ghci4luatex
```

Using stack

```
stack install ghci4luatex
```

From source

```
git clone https://github.com/AliceRixte/ghci4luatex.git
cd ghci4luatex
stack install
```

Verifying the installation was successful   To check that `ghci4luatex` is properly installed, run

```
ghci4luatex --version
```

Modulo the version, this should produce the following output

```
ghci4luatex v0.1, (C) Alice Rixte
```

2.2   Installing the `ghci` package

To install the `ghci` package for LuaTeX, you can use either your package manager or install it from source.

Using TeX Live (not yet supported).

```
tlmgr install ghci
```

Using MiKTeX (not yet supported).

```
mpm --admin --install=ghci
```

From source   Copy both `ghci.sty` and `dkjson.lua` inside the root directory of the Latex file you want to use `ghci4luatex` in.

Both these files can be found at the root of the `ghci4luatex` repository: github.com/AliceRixte/ghci4luatex/

2.3   Running the `ghci4luatex` server

Once both `ghci4luatex` and the `ghci` package are installed, simply run the following in the same directory you will use LuaTeX:

```
ghci4luatex
```

The server should remain active while you are working on your file. You should not close it between consecutive compilations as it performs memoization to make the compilation faster.

> **♀ Tip**
>
> Always have a terminal with `ghci4luatex` running, it will give you clearer error messages from GHCi, and will show you which Haskell expressions are recomputed and which are not.

Once `ghci4luatex` is running, you can execute LuaTeX with

```
lualatex -shell-escape myFile.tex
```

or use `latexmk` using the luatex option:

```
latexmk -lualatex -shell-escape myFile.tex
```

> **⚠ Warning**
>
> Without the `-shell-escape` option, the compilation will fail, complaining about not finding the `'socket'` file.

3 The `ghci4luatex` server

The `ghci4luatex` provides a few options that can be listed by invoking `ghci4luatex --help`. In particular:

–command Allows using cabal or stack to run GHCi. For instance, you can run

- `ghci4luatex --command="cabal repl"`
- `ghci4luatex --command="stack ghci"`

–verbose This will show which commands were memoized.

–quiet Except for errors due to the server (not GHCi error dumps), `ghci4luatex` will be completely silent.

–host and –port Change the host address and port. Notice that this requires you to also change the host and port in `ghci.sty`. Using these options is discouraged.

4 The `ghci.sty` package

The `ghci` package can both execute Haskell code snippets and GHCi commands thanks to the `ghci` environment and print the result to LaTeX with the `\hask` command.

4.1 Running Haskell code: the `ghci` environment

To execute some Haskell code without printing anything to LaTeX, you can use the `ghci` environment. `ghci4luatex` will always surround the code between \begin{ghci} and \end{ghci} by `:{` and `:}` so that GHCi knows this is a multiple line command.

```
\begin{ghci}
x :: Int
x = 4

y :: Int
y = 5
\end{ghci}
```

Using GHCi commands You can also send GHCi commands (i.e. starting with ':'), for instance to load extensions:

```
\begin{ghci}
:set -XOverloadedStrings
\end{ghci}
```

> ⚠ **Warning**
>
> Since the code is always enclosed within `:{` and `:}`, this means you can only give one GHCi command (i.e. starting with ':') at a time. The following will fail with the message `unrecognised flag: :set`
>
> ```
> \begin{ghci}
> :set -XOverloadedStrings
> :set -XOverloadedLists
> \end{ghci}
> ```

Importing modules You can use the `ghci` environment to import any module you need, it will be available throughout the whole file.

```
\begin{ghci}
import Data.Functor
import Control.Monad
\end{ghci}
```

If you need to import modules on Hackage, you can use `:set -package some-package`. To load your own modules, use `:l`.

> 💡 **Tip**
>
> You can directly load all the modules of your own package as well as all of the dependencies listed in your `.cabal` (or `package.yaml` file) by running `ghci4luatex --command="cabal repl"` or `ghci4luatex --command="stack ghci"`.

4.2 Printing the result: the `hask` command

You can use Haskell to output LaTeX code. For instance, `$\hask{1+2}$` will print 3. The result printed by GHCi can also be LaTeX expressions. For instance,

```
\hask{putStrLn "\\emph{I␣was␣written␣by␣GHCi}"}.
```

will produce *I was written by GHCi.*

4.3 Advanced usage: managing memoization

To reduce the compilation time of LuaTeX, the result of the execution of Haskell code snippets is stored in the server in order to avoid recomputing them. This is called *memoization.*

You can see this at work in the output of the `ghci4luatex` server: if you modify your LaTeX document without changing any of the commands, the server will only print

```
--- New session : "main"---
```

> 💡 **Tip**
>
> To see which results are memoized and which are recomputed, you can use the `--verbose` option when running `ghci4luatex`.

If you modify one of the Haskell snippets, `ghci4luatex` will have to recompute all of the snippets that appear after the one you modified.

The reason for this is that if you declare a variable, for instance by writing `x = 4`, and you then modify it to `x = 5`, all the subsequent code that uses `x` has to be updated, and therefore recomputed by `ghci4luatex`.

Using `\ghcisession`   When dealing with big documents that contain a lot of Haskell code that might generate some figures, it can become painfully slow to recompile a document when changing one of the first code snippets that appear in the document.

For this reason, you can tell `ghci4luatex` to create a new session, by using the command

```
\ghcisession{mySession}
```

This way, if you modify any of the code snippets before you started `mySession`, the server will not recompile the code snippets that appear *after* `\ghcisession`

> ⚠ **Warning**
>
> The `ghci4luatex` server actually *does not* spawn a new GHCi process for each new session.
> Instead, it only affects the memoization and there is actually only one GHCi process. This means that if you declare a variable `x = 4` then declare a new session, this variable will still be in the session scope.

Using `\ghcicontinue` If you want to continue a previously defined session, for instance the default session `"main"` you can use

```
\ghcicontinue{main}
```

Notice that this only affects memoization and does not actually switch between different GHCi processes.

> 💡 **Tip**
>
> If you want to make sure to recompile all Haskell code of your document, simply kill `ghci4luatex` and start a new one.

5   Usage with Haskell libraries

Any Haskell library can be used in conjunction with `ghci4luatex`. Here, we present only a selection along with common usage examples. Feel free to add your own suggestions by opening a pull request github.com/AliceRixte/ghci4luatex/.

Usage for all these libraries can be found in `examples/main.tex` at the `ghci4luatex` repository.

5.1   lhs2TeX and HaskinTeX

Any preprocessor can be used in conjunction with `ghci4luatex`, since it is a proper LaTeX package and not a preprocessor itself.

Simply run the preprocessor and use `ghci4luatex` as usual.

5.2   HaTeX

You can use HaTeX to generate some LaTeX code. To use it in conjunction with `ghci4luatex`, you need to print the latex code in GHCi.

A simple way to do so is to write the following:

```
\begin{ghci}
:set -XOverloadedStrings
\end{ghci}

\begin{ghci}
import Text.LaTeX

printTex = putStrLn . prettyLateX
\end{ghci}
```

You can then use the `printTex` function with the `\hask` command:

```
\hask{printTex (section "A␣section␣using␣HaTeX")}
```

5.3 Diagrams

Diagrams is a domain-specific language for drawing vector graphics. It already has a dedicated LaTeX package, diagrams-latex you should definitely consider using. Still, `ghci4luatex` has some advantage over `diagrams-latex`, mainly the persistency of the GHCi session. Here is a complete example with the svg package:

```
\begin{ghci}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE FlexibleContexts          #-}
{-# LANGUAGE TypeFamilies              #-}

import Diagrams.Prelude hiding (section)
import Diagrams.Backend.SVG

myDia = circle 1 # fc green
\end{ghci}

\begin{ghci}
  renderSVG "myDia.svg" (dims2D 400 300) myDia
\end{ghci}

\begin{figure}[h]
  \centering
  \includesvg[width=0.2\textwidth]{myDia}
  \caption{A circle using Diagrams}
\end{figure}
```