# Live Haskell: An Integrated Editing and Debugging Tool for Haskell

CS240H Final Project

Vivek Jain, Neeral Dodhia, Giovanni Campagna

Dept. of Computer Science
Stanford University

## Abstract

Fast turnaround is key to modern programmer productivity: reducing the time from writing code to seeing it running means quicker testing and more chances to catch bugs early. To this extent, various languages support interactive and continuous evaluation directly in the editor. We propose Live Haskell as an application of this concept to the Haskell language, leveraging the purely functional aspect of the language to allow safe and repeatable execution of arbitrary code.

***Categories and Subject Descriptors*** D.2.6 [*Programming Environments*]: Interactive Environments

***Keywords*** toolchains, IDEs, code editors

## 1. Introduction and Motivation

Throughout the course, and especially while working on the assignments, we noticed that there seems to be a lack of a coherent, integrated and easy-to-use solution around developing Haskell, one that Java has in the form of Eclipse, Objective-C with XCode and C# with Visual Studio.

While existing tools are available for Emacs or Vi, they often lack build system integration with Stack, or they offer only some of the features we expect from a modern IDE, like continuous recompilation and type-error feedback or inline type information.

In addition, we want to make Haskell easier to learn and use for the developer. In his essay Learnable Programming [6], Bret Victor outlines the core principles that enable a programming system (both the programming language and the programming environment) to be easy to learn and understand. Among other concepts, he discusses the importance of being able to see the state, follow the flow of the program, and create by reacting (”start somewhere, then sculpt”). These are the concepts we explore in Live Haskell.

While nothing exists today that supports all of the features Victor explains, several have been inspired by the essay and his work to create systems that are easier to understand. Two of the more popular ones are Swift playgrounds [4] and Light Table [3]. However, Swift playgrounds do not support existing projects, thus making them only useful when you want to explore small isolated snippets of code. And while LightTable does support existing projects, its Haskell plugin does not integrate well with Stack projects (it uses Cabal), and only supports evaluating individual expressions, which is not particularly helpful for tracing through your code.

The goal of this project thus is to have a development environment that supports the modern Haskell toolset, gives continuous feedback to the programmer in the form of static type checking and dynamic execution traces, and does so with no changes to the program, such as the introduction of trace statements.

In addition, to our knowledge, none of the existing systems try to make repeatable IO easy. An environment that provides for a good testing and “continuous execution” platform that is repeatable, without incurring excessive changes to final production code base,

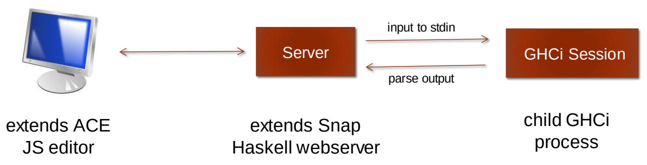


**Figure 1.** The Live Haskell Architecture

enables a much quicker feedback loop for your code. But existing tools fall into one of two buckets: either they explicitly disallow performing file IO (as in Swift playgrounds, which are run in a sandbox), or they run it as normal and thus make it difficult to re-run your code multiple times if it performs any modifications (as in LightTable). We suspect this is mostly because of the complexity of redirecting IO in other languages, such that execution is truly repeatable with no impact on the rest of the developer machine. This complexity is greatly reduced in Haskell, thanks to the fact that side-effectful code is explicitly annotated with the IO type in Haskell, and that it is easy to redefine primitives of the language arbitrarily.

## 2. Description of the system

Our system consists of an interactive editor for Haskell. The editor provides syntax highlighting, continuous checking of type and syntax errors, the ability to query the type of an arbitrary expression under the cursor, as well as to run the code in a sandboxed environment, which allows for repeatable testing with no changes to the program and no test-only code.

Furthermore, we support running the program in tracing mode, which lets the user see the values of all variable bindings at specified line numbers, for all the times that line is reached when the code is executed.

## 3. Architecture

We use a client-server architecture that allows us to leverage web technologies for the frontend, giving it great portability across platforms and ease of development with modern web toolkits.

In our architecture, shown in figure 1, the server runs on the system where the code is (in the form of an existing Haskell project or a newly created one), and takes care of running the interactive evaluation session, while the browser client hosts the editor (with usual editing features such as syntax highlighting). The network protocol transmits the code, the interactive statements and the result of the evaluation.

Despite the client-server architecture, we are not intending this to be a distributed or multi-user system. Instead, we envision the server running on the same machine as the client, essentially operating as a single-user application that just happens to use the web. Thus security is not a primary concern for us, and we allow the user's code to read any files on the server, simplifying development for the user.

## 4. Implementation

### 4.1 Client

The client is written in standard web technologies: HTML, CSS and JavaScript. We briefly considered using a Haskell to JavaScript compiler so that the frontend could also be written in Haskell, but we decided to use JavaScript, as the Haskell to JavaScript compilers are likely slower than using raw JavaScript, and integrating existing plugins is much easier with raw JavaScript.

We extend the Ace web editor [1] to allow users to edit their Haskell code. We chose Ace because developing our own editor is outside the scope of this project, and Ace seems to be the most mature web-based editor and includes support for Haskell syntax highlighting and basic auto-completion.

While Ace proved to be a great starting point, extending it to get the behavior we wanted was not always easy due to the minimal documentation available. Due to Ace's popularity, we were sometimes able to find other users' additions to Ace that we could leverage. We added the following features to the frontend: keybinding to save and run the file, keybinding to display the type of the selected expression, selecting lines to trace and displaying traced output, showing compile errors inline, and entering what command to run/trace.

### 4.2 Server

The server in this architecture receives and responds to HTTP requests from the client. In most cases this involves parsing the request parameters, invoking the appropriate function on the GHCI backend, and then converting the output from GHCI to JSON for sending back to the client.

The initial attempt was to write a server program using the Network module and building on the RockPaperScissors example program from class. While it was possible to write the initial functionality using this approach, it was unable to serve HTTP requests from the client's browser. Snap, a Haskell framework for a webserver [5] was chosen to allow us to work on functionality and leverage existing code for the web server components. Snap has good documentation and an easy to use interface. Handlers to URL endpoints are wrapped in its Snap Monad data type.

For formatting the results in JSON, several options were tried before settling on Aeson. The main problems were resolving dependency issues and understanding how to convert between our custom data types and the Aeson ones. This was accomplished by extending the ToJSON class and providing implementations for our custom data types.

### 4.3 GHCi Backend

Our first attempt to implement the execution environment for this project tried to use the GHC API directly in the process. Unfortunately, we stumbled upon the large complexity of GHC, and the need to reimplement several features such as loading files, keeping a context of imported modules and obtaining the types of arbitrary expressions.

Instead, we opted for using an interactive REPL running as a separate process, with a translation layer converting from high level calls to low-level pipe communication and string parsing. We used GHCi-ng, which is a version of GHCi extended with new command, such as "type-at", which shows the type of an arbitrary expression in a file even if not available at the top level. We also enable `-fdefer-type-errors` so that GHCi-ng processes the type information for expressions even if the program would not otherwise compile.

We built a small custom parser module for parsing the GHCi-ng output. While string parsing works well for our purposes and given our time constraints, it is not as robust as using GHC directly because we require the output to be of a particular format. We ran into a few instances where the output differed from what we expected, resulting in unexpected behavior and requiring modifying our server to support the new output.

Each time the user enters the command to save their file on the frontend, we write the file to disk, compile it and run tracing. The behaviour we desired was `:reload`, where GHCi only recompiles parts which have changed and so is much faster than a regular `:load`. However, `:reload` does not re-import our modified packages. Hence, we use `:load`.

Tracing of execution is implemented using GHCi breakpoints: first we extract all toplevel declarations in a file (by parsing the file in process with the GHC API, because GHCi does not have a command to do so), then we set a breakpoint on each and execute the entry point of the program. When execution stops at the breakpoint, we collect information on variables in scope (as provided by GHCi), then step to the next reduction. If execution steps into a function call defined in a different file, we force evaluation of the result with no further stepping and then continue tracing.

### 4.4 Repeatable IO

One of the issues with interactive continuous evaluation is that different executions of the same code need to yield the same outcome, or the programmer might be left wondering if the different result is due to a code change or to some external cause.

For pure Haskell code this is attained by definition, but real programs necessarily depend on IO, so in order to provide this guarantee we developed a library of sandboxed repeatable IO. This is a new IO-like monad designed to be injected in place of the actual IO, that uses copy-on-write semantics for all operations (including in memory) and performs file IO in a sandbox directory. This allows the code that is run to see a consistent view of the world such that it doesn't know it is in a sandbox, but we can rerun it as often as desired, since each run is essentially idempotent.

We implemented support for equivalents of System.IO, Data.IORef and Data.Array.IO, and built a SafePrelude module that substitutes the Haskell Prelude with the new monad. In our interactive evaluator, we craft search paths so that the standard Haskell module names resolve to our variants. We also use Safe Haskell to make sure that no true IO can be executed.

## 5. Results

In figures 2 and 3, we show screenshots of a file in a Stack project, loaded in Live Haskell.

In figure 2, we see that compiler or type errors are displayed in-line, denoted by the small red cross in the left-hand side margin. Hovering over these displays the error. This is a more intuitive way of identifying where your errors are than by reading through a long list of compiler output. The type annotation for the function `test2` is displayed underneath its definition, again being visually close to where its source is.

In figure 3, we show tracing output where we display the equivalent of the output from imperative style `printf` statements, showing the values of the arguments on each call. This greatly aids developers by showing tracing output in-line, which is not only more integrated than having to run a debugger in a separate session, but also easier because it doesn't require manually specifying what they want to debug or what variables they want to examine.

## 6. Future work

While our editor achieves the goals we initially laid out, there are several possibilities for extending it to make it a more full-fledged tool to improve developer productivity. Most existing editors or IDEs only support a subset of the features that a full Haskell IDE should have [2].

This presents an opportunity for Live Haskell. Possible extensions include adding features such as integrating a build system, interactive debugging with breakpoints, refactoring support, Haskell-aware autocompletion, and analysis of the call graph. We currently call out to a child process running GHCi and it would be more tightly integrated and robust if we were using the GHC API directly.

Our current sandboxed IO is limited to opening, reading, writing and closing files. This is another area where further work could extend this functionality to allow support for more file-system IO, such as renaming or deleting files, and to networking. One way to sandbox networking would be to capture all network traffic and replay packets. Alternatively, we could allow the user to specify that network IO should be run as-is without any sandboxing.

Ease of use can also be improved. There are many cosmetic refinements that could be applied to the UI, for example having a file picker instead of typing the file path to open a project, or displaying the project directory tree next to the editor, as is commonly done in IDEs. Another area that would help users working on existing projects is being able to switch GHC and Stack resolver versions. This may pose some technical difficulties if our code relies on later features.

Given the short timespan of this project, we reduced the scope of what we would support to the subset of Haskell that is SafeHaskell. In fact, many of the assumptions we take are based on this. To be able to support the same behavior for unsafe Haskell code would be an interesting extension to be explored.

## 7. Conclusions

In conclusion, Live Haskell achieves the goals we set out at the start of our project. It provides useful and novel features, including type information and tracing output, that help developers fix errors and increase their understanding of their code. It enables much quicker testing of code by running on every save, but still keeps this safe with a low overhead sandbox that works at the language level.

Live Haskell is lacking in some respects, for example its limited file-system interface and inability to load multiple files into a session. This creates opportunity to pursue this project further, on its own or integrating the backend components into an existing editor.

As a group, we believe it to be a good editor that we would personally use. One particularly appropriate use case is the in-class exercises during CS240H lectures. This is where Live Haskell works best: for a single file and when learning new concepts, where tracing and type information are great as teaching aids.

## References

[1] Ace, the code editor for the web. URL `https://ace.c9.io`.

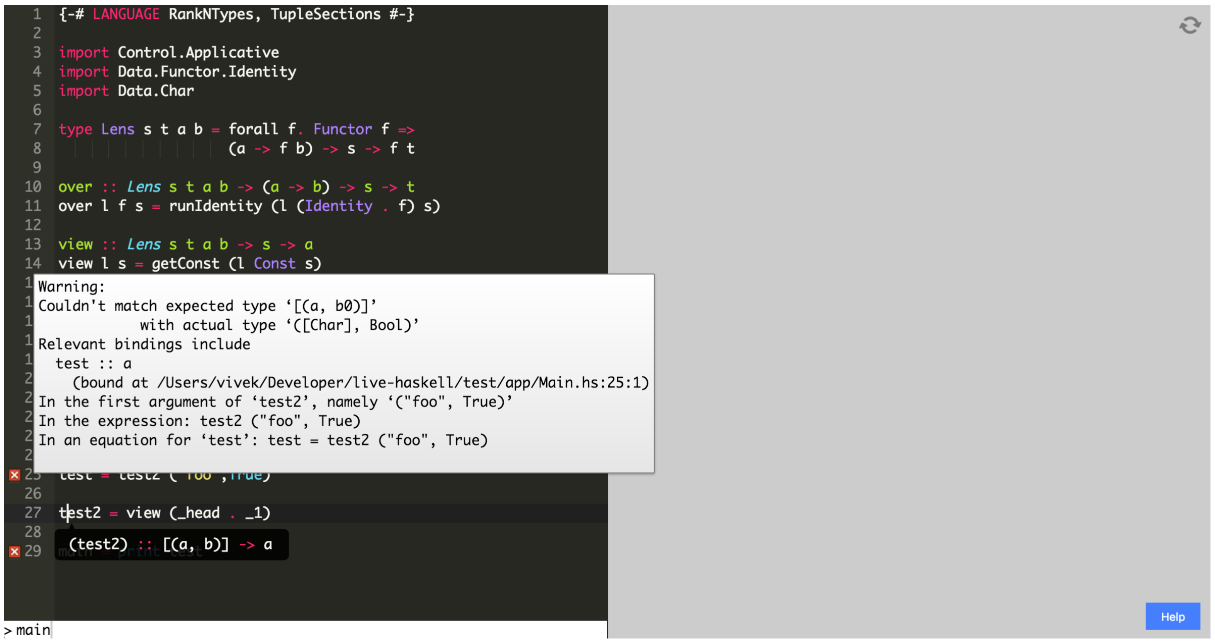
[2] Haskell Wiki. URL `https://wiki.haskell.org/IDEs`.

[3] Light Table. URL `http://lighttable.com`.

[4] Playgrounds - an interactive Swift coding environment. URL `https://developer.apple.com/library/ios/recipes/Playground_Help/Chapters/AboutPlaygrounds.html`.

[5] Snap Web Framework. URL `http://snapframework.com`.

[6] B. Victor. Learnable programming. Sept. 2012. URL `http://worrydream.com/LearnableProgramming`.



```
{-# LANGUAGE RankNTypes, TupleSections #-}

import Control.Applicative
import Data.Functor.Identity
import Data.Char

type Lens s t a b = forall f. Functor f =>
                    (a -> f b) -> s -> f t

over :: Lens s t a b -> (a -> b) -> s -> t
over l f s = runIdentity (l (Identity . f) s)

view :: Lens s t a b -> s -> a
view l s = getConst (l Const s)
```

```
Warning:
Couldn't match expected type ‘[(a, b0)]’
            with actual type ‘([Char], Bool)’
Relevant bindings include
  test :: a
    (bound at /Users/vivek/Developer/live-haskell/test/app/Main.hs:25:1)
In the first argument of ‘test2’, namely ‘("foo", True)’
In the expression: test2 ("foo", True)
In an equation for ‘test’: test = test2 ("foo", True)
```

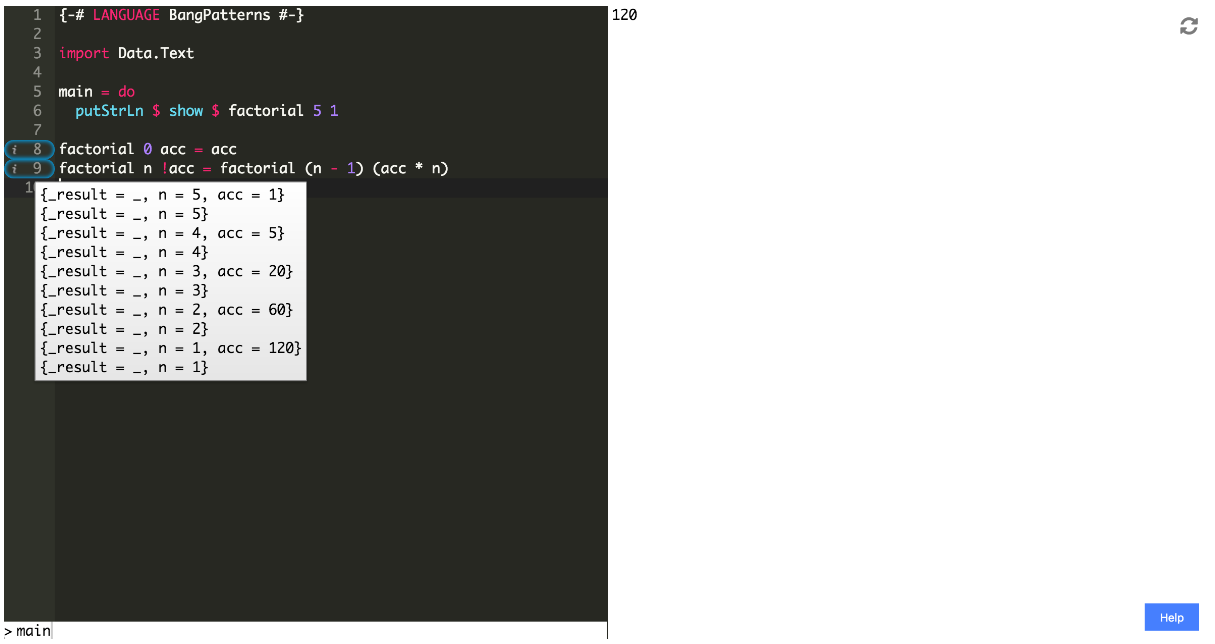
```
test = test2 ("foo",True)

test2 = view (_head . _1)
```

```
(test2) :: [(a, b)] -> a
```

```
> main
```

**Figure 2.** Live Haskell showing a type error when compiling the file. To help understand the issue, the user has also displayed the type of the function test2, where the cursor is located.



```
{-# LANGUAGE BangPatterns #-}

import Data.Text

main = do
  putStrLn $ show $ factorial 5 1

factorial 0 acc = acc
factorial n !acc = factorial (n - 1) (acc * n)
```

```
{_result = _, n = 5, acc = 1}
{_result = _, n = 5}
{_result = _, n = 4, acc = 5}
{_result = _, n = 4}
{_result = _, n = 3, acc = 20}
{_result = _, n = 3}
{_result = _, n = 2, acc = 60}
{_result = _, n = 2}
{_result = _, n = 1, acc = 120}
{_result = _, n = 1}
```

```
120
```

```
> main
```

**Figure 3.** Live Haskell showing tracing output for a tail-recursive version of factorial, which uses the BangPatterns language extension to make the function strict in its second argument