Tutorial

FRP browser programming with Reflex

Niklas Hambüchen

niklas@nh2.me

niklas@fpcomplete.com

Introduction

ghcjs
Compiles Haskell code to JavaScript.

JavaScript
Exotic assembly language that doesn't have integers.
It was invented so that Haskell can run in the browser.

FRP
"Functional Reactive Programming". In our case, this means values that can depend on other values and automatically update appropriately, GUI widgets that automatically update as well, and real world events to trigger updates.

reflex - a Haskell FRP library that I find easy to learn
reflex-dom - a library that lets us do FRP on HTML elements

Tutorial overview

- Development environment setup
- Brief therory on Reflex
- **Live coding**: Learning Reflex by building GUI widgets
  - Hello world, basic JavaScript FFI calls
  - Events
  - Dynamic values
  - Connecting events and dynamic values
  - Cyclic value dependencies, `RecusiveDo` notation
  - Making composeable widgets
  - Dynamic element rendering

Not in this talk

This is a beginner talk for writing GUIs in an FRP style.

We won't have time to cover but you may be interested in following up with:

- How to make web requests with `reflex-dom`
- How `reflex` is implemented
- How to reuse code (e.g. functions and data types) between your Haskell GUI app and your Haskell web server (you can do that easily)
- Cyclic dependency problems and how to solve them
- Investigating whether Reflex is fast enough to run your production website

Development environment setup

I don't know if this works on non-Linux OSs. A VM may help.

1. `git clone https://github.com/nh2/reflex-platform`
2. Inside, `git checkout haskellerz-reflex-tutorial-2017`
3. Do what it says in the **Setup** section of the `README`:
    - Install `nix`
    - Run `./try-reflex`

Now `git log` contains the tutorial steps we'll go through.

But if you're in the live audience, you can just stay at the last commit, open `haskellerz-reflex-tutorial.hs`, delete everything below `main :: IO ()`, and type along.

Compiling, docs and cheat sheets

Check out the top comments of `haskellerz-reflex-tutorial.hs`.

I've put some commands there that can

- compile your code
- link you to the API documentation

There are also links to cheat sheets for `reflex` and `reflex-dom`. They are extremely useful.

Reflex theory

There are only 2 types you need to know about:

```
myEv :: Event t a -- May occur 0 or many times. When
                  -- it occurs, it contains something
                  -- of type `a`. E.g. Event t Int
                  -- contains an `Int` if it occurs.

myDyn :: Dynamic t a -- A value of type `a`.
                     -- You can query its current value
                     -- or listen for changes.
```

(There's also a third type `Behavior t a`; you can query its current value but not listen for when it changes. We won't use it.)

The `t` is a type variable standing for "time" that only gets substituted when our FRP network of widgets is run.
We never instantiate it to anything, it always stays `t` in our code.

Reflex remarks

- A `Dynamic` has a value at any given time. Even at program startup. So you typically have to give it a default value.
  For example, a `Dynamic` that holds the contents of a text box may start as the empty string `""`.
- In contrast, an `Event` does *not* represent a current value.
  It's more like a message you get passed.
  For example, "the user tapped on screen coordinates (34,50)" might send an `Event t (Int,Int)` to you.
- You typically create `Dynamic` values by listening to some events and remembering its value when it occurs, or by combining events into some accumulator state.
  We'll do that a lot in the live coding.

Live coding

When you're not in the audience and can't watch the video recording either

Use `git log` in the repository and `git checkout` go through the tutorial steps one after the other.

I've commented the interesting bits so that you can follow what's going on.

As you go along, I recommend that when you encounter new functions, look them up in the docs.

The FRP network and the reflex monad

When you write code in `reflex`'s monad, e.g. `MonadWidget m`, you are essentially constructing a directed graph that describes how values flow.

This graph is only constructed *once*, e.g. at startup / page load, and then "runs", with value updates travelling between the nodes.
If you put a `trace` into your monadic code, you'll see it prints something only once.
So that's like a "static network" of value propagation.

Functions like `dyn` allow you to change the graph during its execution, by taking `Dynamic`s that can contain `MonadWidget m` actions, and running those graph construction actions.

My personal opinion when comparing reflex to non-FRP web GUI approaches

I started using `Knockout`, one of the earlier "observable value" JS framewoks, in 2010. It was cool, but it was not easy to keep track of which values were observable and which ones were "normal" JavaScript values, and there was no concept of reusable widgets.

`AngularJS` and `React` aimed to address those issues, but they still don't compose effortlessly in my opinion, and require significant convention and boilerplate to be used correctly. They didn't feel like the eventual solution to the GUI problem to me. I felt JavaScript lacked the language to talk about reactive GUIs.

Reflex solves this in my opinion. The types capture the concepts precisely, reusing widgets is easy, it feels lightweight to type down.

Current problems with Reflex / ghcjs

- Lack of guidance / community experience on how to write larger sites with it.
- The hello-world example compiles to a 5 MB JavaScript blob (compresses well though, e.g. `xz` makes 200 KB out of it).
- Nobody has benchmarked how fast it runs a reasonably sized web page. (Now you can do it and tell me about it.)
- Some specific things, like `aeson` based JSON parsing, are known to be extremely slow (as in "page stutters when you parse a 200 KB JSON snippet" slow). Likely fixable.
- Don't even try to make a non-Haskeller use it.
  (But getting proficient in Haskell is easy compared to in JavaScript, which confuses the best of us after years of usage.)

Pro points for using Reflex in your next project

- Actively maintained.
- Easy to understand (in my opinion)
- Good documentation.
- Code reuse really works, even across GUI & server.
- *Theoretically* faster than any dirty-checking approach (looking at you, *AngularJS*).
- And finally ...

Hip points. You'll have so many hip points.
You'll be disrupting the future with Haskell, FRP and nix while the JavaScript crowd is still busy evolving through a chain of JS frameworks that haven't even been invented yet.

Questions?