Haskell Web Application from scratch

Yann Esposito

Introduction

Functional Programming oriented to make a web application

Tooling choices

- macOS Sierra
- spacemacs
- stack (not using ghc-8.0.1, there is a bug with macOS)

High quality application

Even if an application only print "hello world" there are a lot of subtle way such an app could fail or have problems. See for example the [changelogs to GNU Hello](https://github.com/avar/gnu-hello/blob/master/ChangeLog).

The goal of this tutorial is not to provide a "see what we can do with Haskell" but more, how could we enforce production quality development with Haskell. Unfortunately, the tooling is very important in these matters.

To reach such goal we should at least provide:

- Documentation
- Unit Tests
- Generative Tests
- Benchmarks
- Profiling
- CI
- Deployment

It's easy to have one tutorial for each of these concepts, here that won't be a deep dive, but a first introduction on how to achieve all these goals.

Tooling and preparing to code

blog-image("stillsuit.jpg","Stillsuit")

Warning

If you never installed Haskell before, it should be a bit long to setup a correct working environment. So please follow me, don't give up because something doesn't work the first time. I made my best to make my environment work for most people.

Installing Haskell Compiler

Install Haskell etc... In my opinion the easiest way to start is to install `stack`. Then you need to choose a great name for your project, why not `shai-hulud`?

blog-image("shai-hulud.jpg","Shai Hulud")

```
> stack new shai-hulud tasty-travis
```

Yeah now you have a new directory, let use git:

```
> cd shai-hulud
> git init .
```

Now we have some source code, let's try it[1].

[1]: If you are on a Mac, please, modify the line `resolver: lts-7.18` by `resolver: nightly-2017-01-25` to be certain to use `ghc-8.0.2` because there is a bug with `ghc-8.0.1`.

```
> stack setup && stack build && stack exec -- shai-hulud-exe
42
```

Dependencies & Libraries

As we want to make a web application let's add the needed dependencies to help us. Typically we want a web server [warp](https://hackage.haskell.org/package/warp) and also a Web Application Interface [WAI](https://hackage.haskell.org/package/wai). We'll also need to use [http-types](https://hackage.haskell.org/package/http-types).

In the `shai-hulud.cabal` file, in the `shai-hulud-exe` section, add to the build-depends `http-types`, `wai` and `warp` like this:

```
executable shai-hulud-exe
  default-language:  Haskell2010
  ghc-options:       -Wall -Werror -O2 -threaded -rtsopts -with-rtsopts=-N
  hs-source-dirs:    src-exe
  main-is:           Main.hs
  build-depends:     base >= 4.8 && < 5
                   , shai-hulud
                   , http-types
                   , wai
                   , warp
```

Then we modify the `src-exe/Main.hs` file to contains:

```haskell
{-# LANGUAGE OverloadedStrings #-}
import Network.Wai
import Network.HTTP.Types
import Network.Wai.Handler.Warp (run)

app :: Application
app _ respond = do
  putStrLn "I've done some IO here"
  respond $ responseLBS status200 [("Content-Type","text/plain")] "Hello, Web!"

main :: IO ()
main = do
  putStrLn "http://localhost:8080/"
  run 8080 app
```

We'll go in detail later about what everything means.

```
> stack build && stack exec -- shai-hulud-exe
...
... Lot of building logs there
...
http://localhost:8080/
```

Yeah! It appears to work, now let's try it by going on http://localhost:8080/ in a web browser. You should see `Hello, Web!` in your browser and each time you reload the page a new message is printed in the console because some IO were performed.

So can we start yet?

Hmmm no sorry, not yet.

We should not use the default prelude.

While this article is a tutorial, it is not exactly a "very basic" one. I mean, once finished the environment would be good enough for production. There will be tests, ability to reproduce the build on a CI, and so, for such a program I should prevent it to have runtime errors.

In fact, certainly one of the main reason to use Haskell is that it helps prevent runtime errors.

In order to do that we'll use a prelude that doesn't contain any partial functions. So I choosed to use `protolude`[2].

For that that's quite easy, simply add `protolude` as a dependency to your cabal file. We'll modify the cabal file that way:

```
library
```

```
  default-language:  Haskell2010
  ghc-options:       -Wall -Werror -O2
  hs-source-dirs:    src
  exposed-modules:   {-# higlight #-}Lib{-# /highlight #-}
                   , ShaiHulud.App
  build-depends:     base >= 4.8 && < 5
                   , http-types
                   , protolude
                   , wai

executable shai-hulud-exe
  default-language:  Haskell2010
  ghc-options:       -Wall -Werror -O2 -threaded -rtsopts -with-rtsopts=-N
  hs-source-dirs:    src-exe
  main-is:           Main.hs
  build-depends:     shai-hulud
                   , base >= 4.8 && < 5
                   , http-types
                   , protolude
                   , wai
                   , warp
```

We move the `app` declaration in `src/ShaiHulud/App.hs`:

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

module ShaiHulud.App
  ( app )
where

import Protolude

import Network.Wai
import Network.HTTP.Types

app :: Application
app _ respond = do
  putText "I've done some IO here"
  respond $ responseLBS status200 [("Content-Type","text/plain")] "Hello, Web!"
```

And we remove it from `src-exe/Main.hs`:

```
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}
import Protolude

import Network.Wai.Handler.Warp (run)
```

```haskell
import ShaiHulud.App (app)

main :: IO ()
main = do
  putText "http://localhost:8080/"
  run 8080 app
```

So now the tooling around being able to start working seems done.

Not yet

Yes I talked about:

- Installation with `stack` that should take care of installing Haskell
- How to add dependencies by adding them to the cabal file
- Sane prelude with `protolude`
- Provided an overview of WAI Application type

But I forgot to mention part of the tooling that is generally very personal. I use spacemacs and to take advantages of many of the editor niceties I also use `intero` and `haddock`.

So other things to think about:

- Install `intero` with `stack install intero`.
- Also generate hoogle documentation: `stack hoogle data`
- You could also check the tests and benchmark suites: `stack test` and `stack bench`

So we should be done with prelimiaries

So we should be done with preliminaries, at least, I hope so...

If you started from scratch it was certainly a terrible first experience. But be assured that once done, most of the step you've taken won't be needed for your next project.

Web Application

So what is a web application?

WAI

So if you look again at the code you see that your application main function simply print `http://localhost:8080/` and then run the server on the port `8080` using `app`.

The type of `app` is `Application`, if we look at the type of Application in WAI, for example by using `SPC-h-h` on the Application keyword or by going in the [WAI documentation](https://www.stackage.org/haddock/lts-7.18/wai-3.2.1.1/Network-Wai.html).

We see that:

```
type Application = Request -> (Response -> IO ResponseReceived) -> IO ResponseReceived
```

Hmmmm.... What? So just remakr WAI is at it's third major version. So if we just take a look at WAI in its previous version we see that Application was defined as:

```
type Application = Request -> IO Response
```

Which look quite more intuitive. Because, what is the role of a web server if not sending response to requests? The IO here is just there to explain that in order to send a response the server might use IOs like reading in some DB or the file system.

So why let's take a bit to analyze the new definition of `Application` in WAI 3.

```
type Application = Request -> (Response -> IO ResponseReceived) -> IO ResponseReceived
```

It is explained:

> The WAI application.
>
> Note that, since WAI 3.0, this type is structured in continuation passing style to allow for proper safe resource handling. This was handled in the past via other means (e.g., ResourceT). As a demonstration:
>
> ```
> app :: Application
> app req respond = bracket_
>     (putStrLn "Allocating scarce resource")
>     (putStrLn "Cleaning up")
>     (respond $ responseLBS status200 [] "Hello World")
> ```

Great, so before it was difficult to handling some resources, now it appears to be easier to write using `bracket_`. Hmm... `bracket_`? What is this function? If you search it in [hoogle](https://www.haskell.org/hoogle/?hoogle=bracket_):

OK that's quite easy, you see it is a function of `Control.Exception.Base` that we could use like this:

```
bracket
  (openFile "filename" ReadMode)
  (hClose)
  (\fileHandle -> do { ... })
```

And `bracket_` is a variation of `bracket` which doesn't need the return value of the first computation to be used the the "closing" computation. (More details

here)[http://hackage.haskell.org/package/base-4.9.1.0/docs/Control-Exception-Base.html#v:bracket_].

So ok, an `Application` is "mostly" a function that take a `Request` an returns an `IO Response`.

Good, now let's take another look to the `app` code:

```
app :: Application
app _ respond = do
  putText "I've done some IO here"
  respond $ responseLBS status200 [("Content-Type","text/plain")] "Hello, Web!"
```

As you see we don't use the first parameter, the `Request`. So we could ask for some JSON data on `/foo/bar/` with a POST, it will still respond an HTTP 200 with content-type plain text containing the body `Hello, Web!`.

So what a web app should provide. And here we could go down the rabbit hole of the HTTP standard and all its subtleties. But the first thing to come in mind is "how to handle routing"?

One of the advantage of using a language with some types flexibility is to use the types as a high level specification.

Routing

```
data ShaiHuludApp = Routes -> Application
```

That's easy, provided a "Routes" representation we should be able to "generate" a WAI Application. Now how should we represent a set of `Routes`?

We should split them by:

- HTTP Verb: `GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, ...
- Path: `/`, `/users/:userID` ...
- Content-Type: `application/json`, `text/plain`, `text/html`, `text/css`, `application/javascript`...

Hmmm....

So it is immediately very difficult. And these are just the basic requirement, what about all subtelties about Standardized Headers (CORS, ETags, ...), Custom Headers...

Is that FP compatible?

As a functional programmer, and more generally as a scientis, math lover I immediately dislike that profusion of details with a lot of ambiguities.

For example, REST is still ambiguous, should you use POST / PUT to update? Should you put a parameter in:

- part of the path like `/user/foo`
- in the query param of the URI `/user?userID=foo`
- in the body? Then what parser should we use? FORM param, JSON, XML?
- in the headers?
- Why not as an anchor? =/user#foo
- How should I provide a parameter that is a list? A set? A Hash-map? Something more complex?

The problem of web programming come from the tooling. Browsers and HTTP evolved together and some feature existed in browser before people had really the time to think corectly about them.

That's called real-life-production-world. And it sucks! For a better critique of web standards you should really read [the chapter «A Long digression into how standards are made» in Dive into HTML5](http://diveintohtml5.info/past.html#history-of-the-img-element).

So how could we get back our laws, our composability? Our maths and proofs?

We have a lot of choices, but unfortunately, all the tooling evolved around the existing standards. So for example, using GET will be cached correctly while POST won't. And a lot of these details.

FP Compatible Web Programming?

Let's re-invent web programming with all we know today.

First, one recent trends has changed a lot of things. Now a web application is splitted between a frontend and backend development.

The frontend development is about writing a complete application in a browser. Not just a webpage. The difference between the two notions is blurred.

Once consequence is that now, backend application should only present Web API and should never send rendering informations. Only datas. So this is a simplification, the backend should simply expose "procedures", the only things to think about are the size of the parameter to send and the size of the response. As every of these objects will go through the wire.

But there are interresting rules:

- `GET` for read only functions
- `POST` generic read/write functions
- `PUT` idempotent read/write functions
- `DELETE` like `PUT` but can delete things

But there are also HTTP code with so many different semantics.

- `1xx`: technical detail
- `2xx`: Successful

- `3xx`: Redirection
- `4xx`: Client Error
- `5xx`: Server Error

So there are some similarities with the HTTP 1.1 reference and the control on functions we try to achieve with Haskell.

One thing I'd like to imagine is simply that a Web API should simply be like a library. We could simplify everything a lot by removig most accidental complexity.

If we consider a web application to be split between a frontend application and a backend application it changes a lot of things. For example, we could mostly get rid of urls, we can consider to use the backend as a way to expose procedures.

Let's for example decide to use only POST, and send parameters only in the body.

In Haskell we could write:

```
foo :: IO X -- ⇒ POST on /foo
bar :: A -> IO X -- ⇒ POST on /foo with a body containing an A
```

And that's it.

Appendix

Haskell Fragmentation vs Di

There are many other prelude, one of my personal problem with Haskell is fragmentation.

Someone could see "diversity" another one "fragmentation".

Diversity is perceived as positive while fragmentation isn't.

So is diversity imply necessarily fragmentation? Could we cure fragmentation by making it hard for people to compete?

I don't think so. I believe we could have the best of both world.

Then fragmentation occurs. And fragmentation is bad, because if you have an issue with your choice, the number of people that could help you is by nature reduced.

I would say that there is fragmentation when there is no one obvious choice. But having an obvious choice for a library for example doesn't really prevent diversity. Fragmentation:

- NixOS, raw cabal + Linux, stack
- preludes
- editor

- stream library
- orientation of the language "entreprisy ready, production oriented" make it work being dirty, add dirty choices for research people working in the language, "research oriented" make it beautiful or don't make it, block entreprisy people.

bracket_

[3]: Also if you are curious and look at its implementation it's quite short and at least for me, easy to inuit.

```
bracket :: IO a         -- ^ computation to run first (\"acquire resource\")
        -> (a -> IO b)  -- ^ computation to run last (\"release resource\")
        -> (a -> IO c)  -- ^ computation to run in-between
        -> IO c         -- returns the value from the in-between computation
bracket before after thing =
  mask $ \restore -> do
    a <- before
    r <- restore (thing a) `onException` after a
    _ <- after a
    return r

bracket_ :: IO a -> IO b -> IO c -> IO c
bracket_ before after thing = bracket before (const after) (const thing)
```

Very nice