Rapid Applications Prototyping in Haskell

Workshop Session 1

2018-07-19

Objectives

- Understand what we mean by statically typed functional programming.
- Be able to use the basic command line tools: stack & ghci.
- Know where to find web resources (guides, libraries, docs).
- Introduction to the RIO prelude.
- Write “Hello, world.”

You will need

- A copy of stack from https://docs.haskellstack.org/en/stable/README/
- An account on gitlab.com

Don't use distro managers to install Haskell packages. Always use stack.

Run `stack ghci` to get into the interpreter.

Check you can run

```
1 + 1
reverse "Hello"
map (+1) [1,2,3]
```

What is a type?

A type is a object that indicates how a value might behave.

Every term or value has a type.

We denote a value x that has type A by

$$x :: A$$

Some simple type denotations in Haskell might be

```
5     :: Int
"foo" :: String
True  :: Bool
```

What happens if you type

```
"foo" :: Int
```

Type inference

Haskell has an inference engine that allows it to infer the type of a given expression. If it can't resolve the types, it will fail your code and will not execute it. This stops us from running programs that make no sense. We call this static typing.

You very often don't need to specify the types of your terms since the inference engine will figure it out. You can ask Haskell for the type of a term `x` with `:t x`.

What does ghci say about the following terms?

```
"foo"
True
("foo", True)
5
5 :: Int
```

What is a function?

A function is a value that takes values of one type, and turns them into values of another type. Since functions are values, they have types. For a function that takes values of type A and turns them into values of type B, we write:

$$f :: A \to B$$

We call $A \to B$ the *signature*. Some simple function signatures in haskell are

```
not     :: Bool -> Bool
even    :: Int -> Bool
(+)     :: Int -> Int -> Int
```

You can retrieve the type signature for any function by calling `:t`

Calling functions

We can call functions by supplying them with an argument. Try the following.

```
not True
even 4
1 + 1
```

Parametric polymorphism

When we talk about concrete types in function signatures we use upper case, for example `Int` and `String`. When we talk about polymorphic types we use lowercase letters such as `a`. What is the type signature for the function `reverse`? What about `map` and `filter`?

Parametric polymorphism

```
reverse :: [a] -> [a]
map     :: (a -> b) -> [a] -> [b]
filter  :: (a -> Bool) -> [a] -> [a]
```

Function composition

Functions *compose*.

Say we have two functions $f :: a \to b$, and $g :: b \to c$, we can compose them together to make a function $g \circ f :: a \to c$.

Function composition is *associative*. That is if $h :: c \to d$

$$h \circ (g \circ f) = (h \circ g) \circ f$$

Function composition

Function composition in Haskell is done with the dot `.` operator.

```
odd = not . even
```

```
firstWord = head . words
```

We build large applications by composing small modular parts.

Prototyping

This is your toolkit for massive success in life:

- stack, hlint, ghcid, brittany, tintin
- gitlab (https://gitlab.com)
- stackage (https://www.stackage.org/)
- zenhaskell (https://zenhaskell.gitlab.io)
- The RIO prelude

stack can install globally, or locally.

When installing a tool like hlint globally, just sit in your home directory and run

```
stack install hlint
```

A stack project

Run `stack new foo` to make a new project

This will make an application with a package.yaml, a stack.yaml, with app, src (lib) and test directories.

cd into it and run `stack build`, followed by `stack exec --foo-exe`

Adding basic CI

Make a .gitlab-ci.yaml file and add the follwing:

```
image: zenhaskell/foundation:lts-12.1

build:
  stage: build
  script:
  - stack build
```

Haskell 2010

The classic standard library can be found at

- http://hackage.haskell.org/package/base

It's perfectly adequate, but we're going to be using something better.

Super Haskell 2018

Super Haskell 2018 uses RIO and a bunch of language extensions that are considered non-breaking.

- http://hackage.haskell.org/package/rio

RIO is a custom standard library specifically aimed at powering user-facing applications.

RIO

RIO is named after a type

```
type RIO env a = ReaderT env IO a
```

Which in layman's means "take a config, and then do something with it". This is the basic template for all applications, put together a config ("the environment") type, declare the config's values, and then do something with it.

RIO Environments

The `env` part of the RIO will often be described in terms of certain capabilities that it possesses, denoted by 'HasX' typeclasses. You can think of the env part of RIO like a toolkit that descibes what your app can guarantee access to. For example, most envs will have a `HasLogFunc`, indicating that you can log things to the console or to a file.

Bringing in RIO

Add `- rio` to the `dependencies:` section of your package.yaml file.

You'll also want to add a section underneath that reads

```
default-extensions:
- NoImplicitPrelude
- OverloadedStrings
```

And then in `app/Main.hs` and `src/Lib.hs` add in a line that says

```
import RIO
```

Our first app

We'll use the `SimpleApp` from the `RIO.Prelude.Simple` module which is exported by default. SimpleApp has a predefined logging function and a predefined process context for running external processes so we don't have to specify our own just yet. For a real app, we would define our own environment.

We'll change main to log "Hello, world!". Your app/Main.hs should look like this:

```
module Main where

import RIO

main :: IO ()
main = runSimpleApp $ do
  logInfo "Hello World!"
```

Our first app

Build with `stack build` and run with `stack exec -- foo-exe`.

export the environment variable `export RIO_VERBOSE=true` and run it again.

Further resources:

- A non linear guide to haskell
  http://locallycompact.gitlab.io/ANLGTH/
- Category theory for programmers
  https://www.youtube.com/playlist?list=
  PLbgaMIhjbmEnaH_LTkxLI7FMa2HsnawM_
- Parallel and Concurrent Haskell
  https://www.youtube.com/playlist?list=
  PLbgaMIhjbmEm_51-HWv9BQUXcmHYtl4sw
- Data61 Course https://github.com/data61/fp-course

Roadmap

Coming Next

- Algebraic Data Types
- Semigroups, Monoids, Functors
- Category Theory
- Logging strategies
- More on Typeclasses
- Custom environments
- Decoding YAML

On the horizon

- Applicative Functors, Monads, Lenses
- Command Line Interfaces
- Making REST servers
- GTK GUIs
- GL with gloss
- SQL
- Games with SDL2