Introduction to Functional Programming

Chris Purdy

March 4, 2024

1 Getting started with GHCi

To get started with Haskell quickly and follow along with the lecture, use my replit: ***https://replit.com/@ChrisPurdy1/IntroToFP***

Programs in Haskell are typically written in ".hs" files, and programs consist of type declarations and function definitions.

```
module Example where

add :: Int -> (Int -> Int)
add 0 m = m
add n m = 1 + (add (n - 1) m)

-- Main entry-point of compiled executable
main :: IO ()
main = print (add 7 8)
```

A basic template for a Haskell program (in a file called "Example.hs")

The first line declares our collection of functions as a Haskell module, and `--` (double-dash) can be used for a single line comment.

We use the Glasgow Haskell Compiler (GHC) to compile Haskell programs. GHC also comes with GHCi, an interactive REPL (read-eval-print loop).

GHCi is run with the `ghci` command; you can install the Haskell toolchain (including `ghci`) at `https://www.haskell.org/ghcup/`.

```
Prelude> :l Example.hs
...
*Example> f = add 10
```

```
*Example> f 5
15
*Example> g = \x -> x * 7
*Example> g 7
49
*Example> :t g
g :: Num a => a -> a
```

An example GHCi session

Notice how "add" was defined in Example.hs, and that I had access to the "add" function in GHCi after loading the file. Here are a few commands that you may find useful:

- `:l [filename]` - loads a given Haskell file
- `:q` - quits GHCi
- `:t [expression]` - gives the type of a given expression

In particular, the `:t` command will be useful to explore how Haskell infers the type of expressions. Some example types and what they mean:

Type	**Example expression/term**	**Description**
`Int`	`4`	Integers/whole numbers
`Char`	`'u'`	Individual characters (the single quotes are important)
`Int -> Char`	`\x -> if x == 0 then 'n' else 'y'`	Functions that take an `Int` as input, and produce a `Char` as output
`(Int, String)`	`(7, "lucky")`	Ordered pairs of strings and integers
`Num a => a -> a`	`\n -> n * 7`	Functions with type `a` as input and type `a` as output, where `a` is a numerical type (such as `Int` or `Float`)

Don't worry if you don't understand the last example above, but here is some further explanation if you are interested:

For the last type in the table `Num a => a -> a`, the `a` is called a *type variable*, and you use these to define *polymorphic* functions - these are functions that can have many different types for their input and/or output. The `Num a` to the left of the `=>` symbol is called a *type constraint* - it constrains the possible types that `a` could be to "numeric" ones.

2 Functions

Here are some example Haskell functions that operate on integers and pairs of integers:

```haskell
fst :: (Int, Int) -> Int
fst (a, b) = a

snd :: (Int, Int) -> Int
snd (a, b) = b

add :: Int -> (Int -> Int)
add 0 m = m
add n m = 1 + (add (n - 1) m)

add10 :: Int -> Int
add10 = add 10

fork :: (Int -> Int, Int -> Int) -> (Int -> (Int , Int))
fork (f , g) n = (f n, g n)

tens :: (Int, Int) -> Int
tens p = ((fst p) `div` 10) + ((snd p) `div` 10)

twice :: (Int -> Int) -> (Int -> Int)
twice f n = f (f n)
```

For these exercises, assume that all numerical inputs are non-negative integers (otherwise known as *natural numbers*) - I'll give some examples later of how you can handle erroneous inputs.

Exercises

- Explain what the "tens" function does in natural language.
- Define the function "thrice" that takes a function of type `Int -> Int`, and applies it three times in a row.
- Define multiplication and exponentiation recursively (similarly to how "add" is defined above).

  It helps if you think of, for example 3×5, as "3 added to itself 5 times".
- Define the function "applyTo5" with type `(Int -> Int) -> Int`, that takes as input a function, and returns the result of the function applied to 5.

  For example `applyTo5 add10` should evaluate to 15.

- Define "thrice" using function composition (the . operator)
- *Extension* - Show that `fork (f , g) . h = fork (f . h) (g . h)`, by unfolding the definitions on both sides of the equality.

  Start by considering an input `x`, and then unfolding/executing `((fork (f , g)) . h) x` and `fork ((f . h), (g . h)) x` until both reach the same point - this shows that they are equivalent.

3 More resources

A fantastic introductory course to Haskell, with plenty of exercises, can be found at `https://learnyouahaskell.com/chapters`.

If you are comfortable with Haskell already and want something *extremely* hardcore to study, here is a great introduction to category theory (one of my areas of research) by Bartosz Milewski that uses (pseudo-)Haskell in examples and exercises: `https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/`