An Introduction to Haskell

Prof. Susan Older

28 August 2019

Haskell Programs

We're covering material from Chapters 1-2 (and maybe 3) of the textbook.

A Haskell program is a series of comments and definitions:

- Each comment begins with `--` (or appears between `{-` and `-}`) and serves as documentation.
- Each definition contains a type declaration and one or more equations:

$$\mathit{name} :: t_1 \texttt{ -> } t_2 \texttt{ -> } \cdots \texttt{ -> } t_n \texttt{ -> } t$$
$$\mathit{name}\ x_1\ x_2 \cdots x_n = \mathit{exp}$$

- Each t_i is a type, each x_i is a formal parameter.
- The type declaration serves as a contract:
  - What the function expects to receive as input (x_i has type t_i)
  - What the function will deliver as a result (exp has type t)

What are Types?

Type = Collection of "similar" data values

Types are very important in Haskell:

- Every expression has a type.
- Types govern what/how we can combine.
- Nothing gets evaluated unless the types make sense.

Consider the following function definition:

```
isPositive :: Integer -> Bool
isPositive num =  (num > 0)
```

- `isPositive` expects to receive an `Integer`
- `isPositive` will return a `Bool` as a result.

⋆ Thus, for example: `isPositive 7` will have type `Bool`.

Some Very Basic Types

Haskell has lots of built-in types, including:

- `Bool`
  Boolean values: `True` and `False`
- `Integer`
  All possible integer values
- `Float`
  Floating-point numbers, such as `3.267` or `-81.09` or `12345.0`

We'll discuss these (and other types) in more detail later.

Types: Simple Examples

```
thrice :: Integer -> Integer
thrice n = 3*n

isPositive :: Integer -> Bool
isPositive num =  (num > 0)

mystery :: Integer -> Integer -> Integer
mystery x y =  (thrice x) + y
```

What are the types of these expressions?

1. `thrice 12 ::  Integer`
2. `thrice False ...Type error`
3. `isPositive (thrice 12) ::  Bool`
4. `mystery (thrice 12) 5 ::  Integer`

Terminology: Formal Parameters and Actual Parameters

Consider the following definition:

```
mystery :: Integer -> Integer -> Integer
mystery x y =  (thrice x) + y
```

- These lines define `mystery` to be a function that accepts two `Integer` values and returns an `Integer` result.
- The names `x` and `y` are the formal parameters of `mystery`.

  They appear in the function definition to represent the data that may eventually be passed into the function.
- Suppose we evaluate `mystery (4+2) 5`:

  `(4+2)` and `5` are the actual parameters (a.k.a. arguments) of the function call.

Evaluating Expressions in Haskell

Idea: Based on rewriting equations (just like in algebra!)

- Happens after types are checked: type errors mean no evaluation
- Lazy evaluation: expressions evaluated only when values needed

Recall `thrice n = 3*n` from earlier:

```
        thrice (5+2)  ⇝  3 * (5+2) ⇝ 3 * 7 ⇝ 21

thrice (4 - thrice 5)  ⇝  3 * (4 - thrice 5)
                       ⇝  3 * (4 - 3*5)
                       ⇝  3 * (4 - 15)
                       ⇝  3 * (-11)
                       ⇝  -33
```

Boolean values: `Bool`

- Exactly two values: `True`, `False`
- Standard operators:

```
 not :: Bool -> Bool
(&&) :: Bool -> Bool -> Bool   (==) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool   (/=) :: Bool -> Bool -> Bool
```

  - `not` e: evaluates to `True` when e evaluates to `False` (and vice versa)
  - e_1 `&&` e_2: evaluates to `True` when **both** e_1 **and** e_2 evaluate to `True` (evaluates to `False` when **at least one** e_i evaluates to `False`)
  - e_1 `||` e_2: evaluates to `True` when **at least one** e_i evaluates to `True` (evaluates to `False` when **both** e_1 **and** e_2 evaluate to `False`)
  - `==` and `/=` are equality and inequality (respectively)

Full-Precision Integers: Integer

- Integer: all possible integer values
- Standard operators and functions

```
            (+), (*), (-):: Integer -> Integer -> Integer
           div, mod, (^) :: Integer -> Integer -> Integer
               even, odd :: Integer -> Bool
             abs, negate :: Integer -> Integer
              (==), (/=) :: Integer -> Integer -> Bool
    (<), (<=), (>), (>=) :: Integer -> Integer -> Bool
```

Floating-point Numbers: Float

- Float: single-precision floating-point numbers

  Examples include: 543.874   -346.2   12.0
- Some standard operators and functions

```
    (+), (*), (-), /, (**) :: Float -> Float -> Float
                (==), (/=) :: Float -> Float -> Bool
      (<), (<=), (>), (>=) :: Float -> Float -> Bool
    ceiling, floor, round :: Float -> Integer
```

- More functions listed in Figure 3.2 of the textbook (page 58).

Let's Write Some Functions!

As time permits, let's write these functions:

- `average :: Float -> Float -> Float`

  Accepts two numbers and calculates their average
- `allPos :: Integer -> Integer -> Integer -> Bool`

  Accepts three integers and determines whether they're all positive
- `someNeg :: Integer -> Integer -> Integer -> Bool`

  Accepts three integers and determines whether at least one is negative

Details You'll Need to Know to Keep the Interpreter Happy

- Rules/requirements for names/identifiers
- Rules/requirements for indentation
- Functions versus operators
- Overloading of names/operators

What's in a Name?

Identifiers (i.e., names) begin with a letter, and can then be followed by any combination of letters, digits, underscores (_), and single quotes ('):

```
x    Number    a123_xy    alpha'''
```

Three important rules

1. Names of functions and variables must begin with a lowercase letter.
2. Names of types must begin with an uppercase letter.

   Later on, we'll see: constructors, module names, and type classes also must begin with an uppercase letter.
3. Haskell is case sensitive: `abcdef` and `abcDef` are two distinct names.

Convention: When names are built from multiple words, the second and subsequent words are capitalized.

`celsiusToFahr`, `isTooHot`

Another Gotcha: Layout (Indentation Matters!)

Layout determines where definitions start and stop.

The Rule:

A definition ends at the first piece of text that lies at the same indentation as (or to the left of) the start at that definition.

Guidelines:

- For top-level definitions, start at the leftmost column.
- When writing definitions, use the same indentation for each.

  (Emacs can help you with this task.)

Calling Functions and Using Operators

When calling a function, the name appears before its arguments:

```
div 17 4
thrice (thrice 7)
isPostive (mystery 5 (mod 18 5))
```

Operators have two arguments and appear between those arguments:

```
6 * (3+4)
(mystery 6 7) < (thrice (8-2))
```

Parentheses are needed only when the result of a function call (or operator usage) is itself being passed to a function:

- `isPositive thrice 4` will cause a type error.
- `isPositive (thrice 4)` will work.
- `mystery 6 7 < thrice (8-2)` is okay.
- `mystery 6 7 < thrice 8-2` will cause a type error.

Overloading: One Name, Multiple Meanings

We've seen that `==` and `+` have the following types (among others):

```
(==):: Bool -> Bool -> Bool
(==):: Integer -> Integer -> Bool
(==):: Float -> Float -> Bool
 (+):: Integer -> Integer -> Integer
 (+):: Float -> Float -> Float
```

- These are instances of overloading:

  The same name (or symbol) is used to represent different operations/functions on different types.
- Overloading provides a way to provide common naming for similar (but ultimately different) functions/operations.
- Haskell determines from context which definition is needed.
- Overloading is handled through type classes (a topic for the future).