Yogesh Sajanikar

# Working with Pure Functions and Data Types

In the last recipe, we have looked at data types and defined a simple boolean expression data type. In this section, we will look at functions.

In this section, we will work with pure functions in Haskell. Haskell is a **pure** functional language. By **pure** we mean that when applied same arguments, the function always results in the same value. And the result does not depend upon any hidden state or information in the function. And also that the function, while producing its output does not produces any "side effect". The side effect means that any semantic change in the environment due to function execution. A change such as value of a mutable variable, or writing to console or printer is a side effect.

## Function and Value

In haskell, a function is a value, and can be used as an argument to another function. Haskell does not distinguishes between values constructed using data costructor and function.

Hence functions are values which are bound to the expression used in function definition. Since a function is a value, it also has a type, which we will see in the recipe.

data Either a b = Left a | Right b deriving Show

In the above

## Quadratic Equation Solver

In this recipe, we will build a program to solve quadratic equation.

A quadratic equation is represented by an equation The roots of the equation can be given by the following formula

### Representing Quadratic Equation

We will create a data type to represent a quadratic equation.

data Quadratic = Quadratic Double Double Double

With each double precision in the data constructor representing 'a', 'b' and 'c' in the quadratic equation respectively.

But it would be difficult to attribute each double in the data type a meaning. Hence we will use record syntax for defining data. In the record syntax, each value type is given a name, and each such attribute is called a field.

data Quadratic = Quadratic { a :: Double, b :: Double, c :: Double } deriving Show

Now we can create a Quadratic equation by either assigning each field with name as follows,

Quadratic { a = 1.0, b = 2.0, c = 3.0 }

or by just calling data constructor with values.

Quadratic 1.0 2.0 3.0

In fact, in the above case, **a**, **b** and **c** are also functions, viz., and has a signature of returning the field value, given a type value. For example, if you enter ":type a", GHCi will return,

a :: Quadratic -> Double

i.e. Given a value of a quadratic type, you wll get a 'double', which represents a coefficient of second degree in the quadratic polynomial.

### Representing the solution

We know that a quadratic equation will have two roots. The roots can be complex as well. Here we will use complex numbers to represent both real as well as complex roots as complex data type.

Rather than we creating data type for representing Complex number is possible, we rather use the existing data type **Complex**. This data type is defined in a module 'Data.Complex'.

To use a module 'Data.Complex', we have to import it. Once imported we can use all the types and functions exported by the module Data.Complex.

import Data.Complex  
  
type RootT = Complex Double  
  
data Roots = Roots RootT RootT deriving Show

The data constructor 'Roots' takes two Complex numbers to represent two roots of a quadratic equation. Since we are dealing with only "Double" polynomial, the roots that we are interested in are also Double. Complex takes a type argument to represent whether we are defining Complex number in the context of Integers, Floats or Doubles.

Since we are always interested in "Complex Double", we have created a shorthand type by using

type RootT = Complex Double

This creates a synonym for the type constructor "Complex Double". And can be used in place of "Complex Double".

### Implementing the solver

Now we carefully construct the solution. Here in this recipe we will write the quadratic solver, providing insight into how an haskell code can structured in different ways. We will also see how pattern matching works in haskell.

1. Writing function declaration

* Before getting into implementation of the function, we need to know what is the type of the function.
* We know that we would like to find the roots of the quadratic equation, hence the input should be qudratic polynomial, Quadratic, and return value should be roots of the polynomial, represented by Roots.
* Hence the function type should be,
* roots :: Quadratic -> Roots

1. Function implementation in several parts

* You will notice a peculiar thing in Haskell. We will be splitting our function definitions in multiple parts like,
* roots <some criteria> = <some definition>  
  roots <other criteria> = <other definition>  
  roots <another criteria> = <another definition>
* By doing so, we greatly simplify code. Internally, the haskell compiler is able to unify all the above definitions in a single definition by doing something like,
* roots <input> =  
   if <some criteria> then  
   <some definition>  
   else if <other criteria> then  
   <other definition>  
   else if <another criteria> then  
   <another definition>  
   else  
   <compiler generated error>

1. Separating base case by pattern matching

* Haskell provides a really good way of separating cases that avoids using 'if ... else ... ' clauses by matching the data constructor patterns.
* In a quadrilateral polynomial, you will notice that user can provide a **Quadratic** with all coefficients in the polynomial as zero. In this case, any value of the root will solve the polynomial, (nothing to solve ... really) and hence we would like to create an error.
* Rather than checking each field of Quadratic for zero value, we can use data constructor in the function expression.
* roots (Quadratic 0 0 \_) = error "Not a quadratic polynomial"
* Note how we have used '(Quadratic 0 0 \_)' as an expression in the function. Whenever the function, roots, is given a value that matches above constructor pattern, the expression on the right side is invoked.
  1. \_ as unevaluated argument
  + Note the usage of \_ in the pattern above. \_ is used specially in haskell to specify wild card or unused parameter. An identifier that starts with \_ can also be used to signify unused parameter.

1. Flagging an error

* We have used a library function, "error" to flag an error. The error function has an interesting signature,
* error :: String -> a
* The error method takes a string as an argument, and evaluates to any type 'a'. This function is used to flag an error. The error does not materialize itself as an exception, unless we call it in a IO monad. Note that GHCi is a special IO monad, and when we execute our "roots" function in GHCi, you will see the error.

1. Matching partial patern and Using let..in

* We have solved the base case. Another case that we need to handle is when coefficient of second degree is zero. In this case, the quadratic becomes a linear equation. And we can solve it trivially.
* roots (Quadratic 0.0 b c) = \_
* Since we are interested in matching only 'a' part of Quadratic, we have provided value for only 'a' in the data constructor pattern above. We would like to use other values 'b' and 'c' are required in our calculation, and hence we have used identifiers there to which the remaining coefficient values in the input will be bound.
* To create a binding inside a function, we use "let ... in" construct. This construct has following
* let identifier = <expression>  
   identifier1 = <expression1>  
   ...  
  in result-expression
* "let" allows us to create several identifiers and bind values to it. We can use these values in the following expressions in "let" block and in the result expression. The result of the let block is the value of the expression that comes after "in".
* We can now go ahead and complete the "linear equation" case.
* roots (Quadratic 0.0 b c) = let root = ( (-c) / b :+ 0)  
   in Roots root root
* Note that ':+' is a data constructor for Complex. A constructor can start with capital letter or ':'. It is also an 'infix' constructor (i.e. it takes two arguments and used as an operator).

1. Completing the high level solution

* Now that we have handled base cases, we can solve the equation. In the solution we need to calculate . The term is also called discriminant of the quadratic equation. If its value is zero, then all roots are equal and real. If the discriminant is negative, then all roots are complex. Otherwise, we get real roots.
* Since discriminant is required again, we create following solution
* roots (Quadratic a b c) =  
   let discriminant = b \* b - 4 \* a \* c  
   in rootsInternal (Quadratic a b c) discriminant
* We use "let" to bind discriminant, and use it in another function, rootsInternal (yet to be defined). You will see that this way of 'top down' programming is very helpful in creating simple and readable solutions.
* Since rootsInternal takes addtional double value, its declaration is,
* rootsInternal :: Quadratic -> Double -> Roots

1. Solve zero discriminant case by using guard

* Let us now use a different constructs for guard. Here rather than doing pattern matching we use conditional expressions to differentiate top level function definition.
* Here we check for zero discriminant.
* rootsInternal q d | d == 0 = let r = (-(b q) / 2.0 / (a q))  
   root = r :+ 0  
   in Roots root root
* The condition is presented after the arguments to the function, and separated by a vertical bar '|'. if the condition is satisfied, (in this case 'd == 0'), then the expression on the right hand side is evaluated.
* In the above example, we use field functions "a, b and c" to access the fields in Quadratic to calculate root.

1. Solve for complex root by using "where" clause

* Now we can use similar guard for negative discriminant to solve for complex roots. Rather than using "let .. in" construct. We use "where clause" where bindings are given after the result expression.
* The construct for where construct is.
* <result-expression>  
   where  
   identifier(s) = binding(s)
* both "let ... in" and ".. where .. " are expressions and can be used as an expression in a binding.
* The complex root finding can be represented by,
* rootsInternal q d | d < 0 = Roots (realpart :+ complexpart) (realpart :+ (-complexpart))  
   where plusd = -d  
   twoa = 2.0 \* (a q)  
   complexpart = (sqrt plusd) / twoa  
   realpart = - (b q) / twoa

1. Completing the solution

* The only case that is remaining is a case where discriminant is positive. Since we have exhausted all other conditions we can exclude the guard and solve all remaining cases in one go. (actually only one remaining).
* rootsInternal q d = Roots (root1 :+ 0) (root2 :+ 0)  
   where plusd = -d  
   twoa = 2.0 \* (a q)  
   dpart = (sqrt plusd) / twoa  
   prefix = - (b q) / twoa  
   root1 = prefix + dpart  
   root2 = prefix - dpart

### Complete solution

The complete solution is here. You can load it in GHCi, and try to call roots function with various Quadratic constructs!

module Quadratic where  
  
import Data.Complex  
  
  
data Quadratic = Quadratic { a :: Double, b :: Double, c :: Double } deriving Show  
  
type RootT = Complex Double   
  
data Roots = Roots RootT RootT deriving Show  
  
-- | Calculates roots of a polynomial and return set of roots  
roots :: Quadratic -> Roots  
  
-- Trivial, all constants are zero, error roots are not defined  
roots (Quadratic 0 0 \_) = error "Not a quadratic polynomial"  
  
-- Is a polynomial of degree 1, x = -c / b  
roots (Quadratic 0.0 b c) = let root = ( (-c) / b :+ 0)  
 in Roots root root  
  
-- b^2 - 4ac = 0  
roots (Quadratic a b c) =  
 let discriminant = b \* b - 4 \* a \* c  
 in rootsInternal (Quadratic a b c) discriminant  
  
  
rootsInternal :: Quadratic -> Double -> Roots  
-- Discriminant is zero, roots are real  
rootsInternal q d | d == 0 = let r = (-(b q) / 2.0 / (a q))  
 root = r :+ 0  
 in Roots root root  
  
-- Discriminant is negative, roots are complex  
rootsInternal q d | d < 0 = Roots (realpart :+ complexpart) (realpart :+ (-complexpart))  
 where plusd = -d  
 twoa = 2.0 \* (a q)  
 complexpart = (sqrt plusd) / twoa  
 realpart = - (b q) / twoa  
  
-- discriminant is positive, all roots are real  
rootsInternal q d = Roots (root1 :+ 0) (root2 :+ 0)  
 where plusd = -d  
 twoa = 2.0 \* (a q)  
 dpart = (sqrt plusd) / twoa  
 prefix = - (b q) / twoa  
 root1 = prefix + dpart  
 root2 = prefix - dpart

1. Running the solution

* You can check the solution by using your own inputs. The sample GHCi run is given here
* λ> roots (Quadratic 0 0 0)  
  \*\*\* Exception: Not a quadratic polynomial  
  λ> roots (Quadratic 0 1 2)  
  Roots ((-2.0) :+ 0.0) ((-2.0) :+ 0.0)  
  λ> roots (Quadratic 1 3 4)  
  Roots ((-1.5) :+ 1.3228756555322954) ((-1.5) :+ (-1.3228756555322954))  
  λ> roots (Quadratic 1 4 4)  
  Roots ((-2.0) :+ 0.0) ((-2.0) :+ 0.0)  
  λ> roots (Quadratic 1 0 4)  
  Roots ((-0.0) :+ 2.0) ((-0.0) :+ (-2.0))