

Overview

This lab gives you some more practice with types, especially as they relate to functions and function applications. Every valid Haskell expression has some predetermined type as determined by Haskell's typing rules. Here is a brief recap of some of these rules.

1. If a function expression `f` has type $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$ is applied to arguments $e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k$, (where $k \leq n$), then the resulting expression `(f e1 e2 ... ek)` has type

$$t_{k+1} \rightarrow t_{k+2} \rightarrow \dots \rightarrow t_n \rightarrow t.$$

For example, suppose that we have a function `mystery` with type `Int -> Float -> Bool -> Char`. The following expressions would therefore have the indicated types:

```
mystery 7           :: Float -> Bool -> Char
(mystery 7)         :: Float -> Bool -> Char
(mystery 7) 3.2     :: Bool -> Char
mystery 7 3.2      :: Bool -> Char
mystery 7 3.2 False :: Char
(mystery 7 3.2) False :: Char
```

2. If an expression `e` has a polymorphic type (e.g., `a -> [(a,b)]`), then `e` also has every possible *instance* of that type (i.e., anything that can be obtained by substituting in for the polymorphic variables). For example, suppose that we have a function `riddle` with the type `riddle :: (a,b) -> [a] -> [(b,a)]`. The function `riddle` necessarily also has the following types:

```
riddle :: (Int,b) -> [Int] -> [(b,Int)]
riddle :: (a,Char) -> [a] -> [(Char,a)]
riddle :: (Float,Bool) -> [Float] -> [(Bool,Float)]
riddle :: (Int,Int) -> [Int] -> [(Int,Int)]
riddle :: (b,a) -> [b] -> [(a,b)]
riddle :: (Int -> Bool,b) -> [Int -> Bool] -> [(b,Int -> Bool)]
riddle :: ((c,d),a->c) -> [(c,d)] -> [(a->c,(c,d))]
```

3. When you define a new algebraic type via the `data` construction, its constructors create new functions for which the above rules remain applicable. For example, consider the following definition:

```
data Shape = Circle Float
           | Rect Float Float
```

As a result of this definition, `Circle` and `Rect` have the following types:

```
Circle :: Float -> Shape
Rect   :: Float -> Float -> Shape
```

Your Mission

Grab a copy of: www.cis.syr.edu/courses/cis252/code/lab10.hs.

The Rules of the Game. Your task is to use those definitions—and any others you choose to add—to build valid Haskell expressions with specific types (given below). Here are the rules:

- You **may not** alter any of the definitions or type signatures (i.e., contracts) we have provided.
- You **may** add additional definitions to the file if you want, but you **may not** introduce any type signatures for your definitions. Thus, for example, you can define a function `foo x = x + 1`, but you cannot add the type signature `foo :: Int -> Int`.
- You **may** use any valid Haskell expressions at all to build your expressions, including anonymous functions.
- You **may not** use type annotations in your expressions themselves (e.g., `reverse ([] :: [Bool])` is not allowed).
- You **may not** use `error` or `undefined` or anything else from the prelude of type `a`.
- Each expression you come up with must have *exactly* the indicated type, as confirmed by using the `:type` command in Hugs. Thus, for example, the expression `[]` does not count as a representative for `[[Int]]`, because Hugs claims the type is `[[a]]`. Likewise, `3` does not count as an `Int`, because Hugs claims the type is `Num a => a`.

An Example. Consider the task of building an expression with the type `[Float]`. Here's one way to approach the problem:

1. Look at the definitions that already exist (including any you may already have written), to see if they include the types you're looking for. You might also consider what built-in Haskell functions work over that specific type.

In the case of `[Float]`, we can see that the `Shape` definition involves `Float`. Unfortunately, most of the built-in functions that work on type `Float` accept *all* members of the `Floating` type class, so they're not apt to be immediately helpful. Even a number such as `7.3` has type `Fractional a => a`. To force Hugs to view `7.3` as a `Float`, we'll have to be clever.

2. `Shape` seems the most viable, so let's look at how we might use it.

`Circle` is a type constructor that accepts a `Float` and returns a `Shape`. If we feed `Circle` the value `7.3`, Hugs will have to accept `7.3` as a `Float`.

3. We're close now, but `Circle 7.3` is a `Shape`, not the `Float` that we need.

Here's where we get clever: let's imagine writing a function that accepts a value (such as `7.3`), applies `Circle` to it (which will force that value to be a `Float`), but then returns the original value (which we now know must be a `Float`). One way to do this is as follows:

```
getFloat v = snd (Circle v, v)
```

If you type `:type getFloat 7.3` at the Hugs prompt, you'll see that it indeed has type `Float`. Whereas the role of `Circle` is to ensure that `v` is a `Float`, the role of `snd` is to return the value that we really care about.

4. What we really need is a value of type `[Float]`: we can simply put the above expression within brackets (e.g., `[getFloat 7.3]`), and we're done.

Alternatively, we can use a similar idea and `map` to get a list of `Floats` directly, using the function:

```
getList vs = snd (map Circle vs, vs)
```

In Search Of... To receive full credit on this lab, follow the above rules to find legitimate expressions for `ten` of the following `fourteen` types. (*Warning:* Some of them require you to be clever; a couple require you to be **very** clever. However, all are solvable using things we've seen in lectures and assignments.)

1. `Char`
2. `[Float] -> Bool`
3. `Bool -> Int`
4. `Char -> Char -> Bool`
5. `(Bool,Int,Char) -> Bool`
6. `((Char -> Int) -> Char) -> Int`
7. `(Int, [Char -> Bool])`
8. `[[Float] -> Bool]`
9. `a -> b -> a`
10. `(a -> b -> c) -> b -> a -> c`
11. `(a -> b -> c) -> (a -> b) -> (a -> c)`
12. `(a -> b) -> (b -> c) -> (a -> c)`
13. `(a,b) -> (b,c) -> (a,b)`
14. `(a,b,c) -> [c] -> a`

What to Hand In: Hand in a copy of the file `lab10.hs` with your additions, and a transcript demonstrating the correctness of your answers. *This lab is due by noon on Friday, April 4* in the bin in CST 3-212.