Contents

Types

- Haskell is strongly typed
  - Every expressoin/value has a well-defined type:
  - `myExpr :: MyType`
  - read: "Value `myExpr` has type `MyType`"
- Haskell supports *type-inference*: we don't have to declare types of functions in advance. The compiler can figure them out automatically
- Haskell's type system is *polymorphic*, which allows the use of arbitrary types in places where knowing the precise type is not necessary
- This is just like *generics* in Java or C++ - this `List<T>`, `Vector<T>`, etc.
- Some Literals have simple pre-determined types
  - `'a' :: Char`
  - `"ab" :: String`
- Numeric literals are more complicared
  - `1 :: ?`
  - Depending on context, 1 by integer, float or double depending on context
- This is common with many other languages where notation for numbers are often "overloaded"

- Haskell has a standard powerful way of handling overloading (the `class` mechanism)

Function Types

- A function type consists of the input type, followed by a right-arrow and then the output type
  - `myFun :: MyInputType -> MyOutputType`
- Given a function decleration like `f x = e`, if `e` has type `b` and we know that the usage of `x` in `e` has type `a`, then `f` must have type `a -> b`.

$$\frac{\texttt{x :: a} \qquad \texttt{e :: b} \qquad \texttt{f x = e}}{\texttt{f :: a -> b}}$$

- Given a function application `f v`, if `f` has type `a -> b`, the `v` must have type `a`, and `f v` will have type `b`.

$$\frac{\texttt{f :: a -> b} \qquad \texttt{v :: a}}{\texttt{f v :: b}}$$

Type Checking with Inference

When a type is provided the compiler checks to see that it is consistent with the equations of a function

```
notNull ::: [Char] -> Int
notNull xs = (length xs) > 0
```

The function `notNull` is valid, but the compiler rejects it. Why? The compiler knows the types of `(>)` and `length`

```
length :: [Char] -> Int     -- Not quite
(>) :: Int -> Int -> Bool   -- Not quite
```

We can correct things:

```
notNull xs = ((>) (length xs)) 0

notNull :: t1
xs :: t2
(>) :: Int -> Int -> Bool
length :: [Char] -> Int
0 :: Int
```

```
length :: [Char] -> Int        xs :: [Char]
-----------------------------------------
          length xs :: Int
```

```
(>) (length) :: Int -> Bool
0 :: Int

(>) (length) (0) -> Bool
```

```
xs:: [Char]    (>) (length) 0 :: Bool    notNull xs = (>) (length) 0
-------------------------------------------------------------------
              notNull :: [Char] -> Bool
```

```
notNull:: [Char] -> Bool
notNull xs = (length xs) > 0
```

Now the compiler accepts the code because the written and inferred types match

Parametric Polymorphism

What type is this function?

```
length [] = 0
length (x:xs) = 1 + length xs
```

In Haskell, we are allowed to give that function a general type

```
length :: [a] -> Integer
```

The type states that the function `length` takes a list of values and returns an integer. There is no contraint on the kind of values that much be contained in the list, except that they must all have the same type `a`.

What about this:

```
head (x:xs) = x
```

This takes a list of values and returns one of them. There is no constraint on the types of things that can be in the list, but the kind of thing that is returned must be the same type.

```
head :: [a] -> a
```

Revisisting notNull

Reminder

```
notNull xs = (length xs) > 0
```

The comiler knows the types of `(>)` and `length`

```
length:: [a] -> Int
(>):: Int -> Int -> Bool -- still not quite
```

Type inference will deduce

```
notNull :: [a] -> Bool
notNull xs = (length xs) > 0
```

What is the type of `sameLength`

```
sameLength[][] = True
sameLength (x:xs) [] = False
sameLength [] (y:ys) = False
sameLength (x:xs) (y:ys) = sameLength xs ys
```

Could it be

```
sameLength :: [a] -> [a] -> Bool
```

This type states that `sameLength` takes a list of values of type `a` and another list of values of *that same type* `a` and returns a `Bool`

A type signature can use more than one type variable (it can vary in more than one type). Again, we consider:

```
sameLength[][] = True
sameLength (x:xs) [] = False
sameLength [] (y:ys) = False
sameLength (x:xs) (y:ys) = sameLength xs ys
```

What would the most general type that could work be?

```
sameLength :: [a] -> [b] -> Bool
```

The two lists do not have to contain the same type of elements for `length` to work. `sameLength` has two *type parameters*. When doing type inference, Haskell will *always* infer the **most general type** for expressions

Prelude

- The "Standard Prelude" is a library of functions loaded automatically by any Haskell program
- Contains most commonly used datatypes and functions

Infix Declarations

```
infixr 9 .
infixr 8, ^, ^^, ..
infixr 7 *, /, `quot`, `rem`, `div`, `mod`
infixr 6 +, -
infixr 5 :
infixr 4 ==, /=, <, <=, >=, >
infixr 3 &&
infixr 2 ||
infixr 1 >>, >>=
infixr 0 $, $!, `seq`
```

Higher precedence numbers bind tighter. Function applicat binds tightest of all

Numeric Functions

```
subtract  :: (Num a) => a -> a -> a
even, odd :: (Integral a) => a -> Bool
gcd       :: (Integral a) => a -> a -> a
lcm       :: (Integral a) => a -> a -> a
(^)       :: (Num a, Integral b) => a -> b -> a
(^^)      :: (Fractional a, Integral b) => a -> b -> a
```

The `Num`, `Integral` and `Fractional` annotations have to do with *type-classes*

Function Functions

```
id        :: a -> a
const     :: a -> b -> a
(.)       :: (b -> c) -> (a -> b) -> a -> c
flip      :: (a -> b -> c) -> b -> a -> c
seq       :: a -> b -> b
($), ($!) :: (a -> b) -> a -> b
```

Boolean Type and Functions

```
data Bool  =  False | True
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool
```

List Functions

```
map    :: (a -> b) -> [a] -> [b]
(++)   :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
concat :: [[a]] -> [a]
head   :: [a] -> a
tail   :: [a] -> [a]
null   :: [a] -> Bool
length :: [a] -> Int
(!!)   :: [a] -> Int -> a
repeat :: a -> [a]
take   :: Int -> [a] -> [a]
drop   :: Int -> [a] -> [a]
elem   :: Eq a => a -> [a] -> Bool
```