Types and Classes

Important 0.1: Immutable

In Haskell, everything is immutable!

1 Types

- Mathematics and programming rely on the notion of types
- Tell us how to interpret a variable
- Provide restrictions on valid operations

```java
int a = 4;
int b = 3;

double c = a/b;

/* c=1.0 */
```

```java
double a = 4;
double b = 3;

double c = a/b;

/* c=1.33333... */
```

The result depends on input types
Since computers represent everything as sequences of bits, types are also required to define what these bit streams mean.

Types are required to know what a bit sequence means

- **Implementation** - Find the correct implementation of, for example '+'
- **Correctness** - Check whether an operation on some data is valid and/or well-defined or check whether a code fragment is correct (type safety)
- **Documentation** - Document the code's semantics (for the reader)

1.1 Types in Haskell

Haskell is **strongly, statically typed** - every well-formed expression has exactly one type, these types are known at compile time

- `Bool` the two logical values `True` and `False`
- `Bool -> Bool` the set of all functions that take `Bool` as input and produce a `Bool` as output
- We will see more standard types soon

1.1.1 Attaching Types

Haskell's notation for "e is of type T" is spelt

```haskell
e :: T
-- False is of type Bool
False :: Bool
-- not is of type Bool -> Bool
not :: Bool -> Bool
```

1.1.2 What type does X have?

Every valid expression is Haskell must have a valid type.
You can ask GHCi what the type of the expression is with the command `type expr`

```
Prelude> :type sum
sum :: Num a => [a] -> a
```

1.2 All the types in Haskell

`Bool` - Logical values
`Char` - Single Characters
`String` - Strings of characters
`Int` - fixed-precision integers
`Integer` - Arbitrary-precision integers
`Float` - Floating-point numbers

1.3 List Types

A list is a sequence of values of the same type
The type of the elements is unrestricted, for example we can have lists of lists

```
[['a'],['b','c']] :: [[Char]]
```

1.4 Tuple types

A tuple is a sequence of values of different types

```
(False,True) :: (Bool, Bool)
(False, 'a', True) :: (Bool, Char, Bool)
```

2 Type Checking

2.1 Statically Typed Language

We check correctness at translation time. Invalid types mean ”translation error”

```
-- Invalid
foo = 1 + "f"
```

2.2 Dynamically Typed Language

We check correctness at run time
Invalid types only detected if we ”use them”

```
def foo():
  return 1 + "f"
```

2.3 How the translator determines the type of expression

Explicit Annotation: Programmer annotates all variables with type information (C/Java)
Type inference: Translator infers the types of variables based on the operations used (e.g. Haskell/ML)
Duck typing: Translator/runtime just tries the operation, if it succeeds, that was a valid type

3 Functions Have Types

3.1 Programming With Functions

Functions of one argument ”unary” Map from one type to another

```
not :: Bool -> Bool
and :: [Bool] -> Bool
```

Functions of two arguments ”binary Map from two types to another

```
add :: (Int, Int) -> Int
```

4 Currying

Currying - Turn a function of n arguments into a function of n-1 arguments
Why currying?

- Easier to reason about and prove things with functions of only one variable
- Flexibility in programming makes composing functions simpler
- Related to partial evaluation where we bind some variables in an n-ary function to a value

4.1 Function types

A function is a mapping from values of one type to values of another type, for example

```
not :: Bool -> Bool
add :: (Int, Int) -> Int
```

4.2 Curried Functions

Functions with multiple arguments are also possible by returning functions as results

```
add' :: Int -> (Int -> Int)
add' x y = x+y
```

Add and add' produce the same final result, but add takes its two arguments at the same time, whereas add' takes them one at a time