EECS 776

Functional Programming and Domain Specific Languages

Professor Gill

The University of Kansas

Feb 3rd 2020

Types

Types are the distinguishing feature of Haskell-like languages

- What are types?

```
42 :: Int
```

- What is type-checking and type-inference?
  - Type-checking is checking if the types are **self-consistent**
  - Type-inference is checking **without being told** what the types are

Most modern languages have some form of type-checking, some have type-inference

Robin Milner

Robin was an outstanding and well-rounded computer scientist

- Machine-assisted proof construction (LCF)
- **Design of typed programming languages** (ML)

  “Well-typed programs don't go wrong.”

- Models of concurrent computation (CCS, π-calculus)

He was awarded the Turing Award in 1991

Type systems in modern languages

Java - static typing

```
public int example(int
x,double y) {
  String z = "Hello";
  ...
}
```

Statically typed languages are dependable but rigid

JavaScript - dynamic typing

```
function example(x,y) {
  var z = "Hello";
  ...
}
```

Dynamically typed languages are flexible but unreliable

The type system in Haskell

In Haskell, you can give the types of the values ...

```
sphereArea :: Double -> Double
sphereArea r = 4 * pi * r^2
```

... or let Haskell infer it ...

```
sphereArea r = 4 * pi * r^2
```

The type says "take a **`Double`**, return a **`Double`**"
So **`r`** is a **`Double`**, and **`4 * pi * r^2`** is a **`Double`**

```
Prelude> :l Example.hs
*Main> sphereArea 5
314.1592653589793
```

The type system in Haskell (GHCi)

You can also give the type in GHCi …

```
Prelude> let sphereArea :: Double -> Double ; sphereArea r
= 4 * pi * r^2
Prelude> :t areaOfSphere
areaOfSphere :: Double -> Double
```

… or let GHCi infer it …

```
Prelude> let sphereArea r = 4 * pi * r^2
Prelude> :t
????
```

Type inference

Parametric polymorphism is a sweet spot on the typing landscape.

- Static typing,
- with Polymorphic values (give you dynamic-like typing when you need it)

The type inference in Haskell is really powerful.

It is considered good form (and documentation) to write some types, and let Haskell figure the rest out.

Haskell is not guessing the types, it is inferring them.

An inferred type is a high form of truth, and inference is a crowning achievement of centuries of mathematics.

Caveat: In order to be work within this powerful system, many primitives
in Haskell have non-obvious types. There is always a reason why.

Everything has a type

Everything has a type, and GHCi can tell you, using **:t**.
Basic characters have type **Char**.

```
Prelude> 'c'
'c'
Prelude> 'c' :: Char
'c'
Prelude> :t 'c'
'c' :: Char
```

Strings have type **[Char]**, which means many chars. Strings are literally lists of characters.

```
Prelude> :t "Hello"
"Hello" :: [Char]
```

Type of a Number

```
Prelude> 1 :: Int
1
```

This is a C-style 32 or 64 bit number. (The Haskell spec says at least 29 bits + sign bit.)

```
Prelude> 1.0 :: Double
1
Prelude> 1.0 :: Float
1
```

`Double` and **`Float`** are 64 bit and 32 bit floating point numbers.

```
Prelude> 1 :: Integer
1
```

`Integer` has an arbitrary precision.

Type-inference of a Number

```
Prelude> :t 1
1 :: Num a => a
```

What can this mean? There is clearly more than meets the eye.

You can always use the **`::`** notation to fix a number as an **`Int`**, **`Float`**, etc.

Let us see some other examples, get back to basics, and come back to this.

Type-inference of a Function

```
Prelude> let f x = x
Prelude> :t f
???
```

What can you know about **x**. Nothing at all?

Literally, the type of **f** is ∀t . t →t. Haskell assumes the ∀ in this example.

```
Prelude> :t f
f :: t -> t
```

f takes anything, and returns (the same) anything.

Terminology: **t** is polymorphic, and **f** is a polymorphic function.

In the type syntax, polymorphic arguments are lower case.

Type-inference of a Function (2)

If we are more specific about arguments or results, the function will have a more specific type to reflect this.

```
Prelude> let f x = (x :: Int)
Prelude> :t f
f :: Int -> Int
```

Alternatively (Uses an extension **ScopedTypeVariables** ; originally not considered good form):

```
Prelude> :set -XScopedTypeVariables
Prelude> let f (x :: Int) = x
Prelude> :t f
f :: Int -> Int
```

Key observation: the original polymorphic function is the most general version of the function.

Types of key arithmetic functions

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a
```

This means

- **(+)** takes two **a** values,
- and returns an **a** value,
- and **a** is a Num-thing.
- "**Num a =>**" means this is my constraint.
- "**a ->**" means this is what I pass as an argument.

Now, addition does add two numbers, to give a number.

Types of arithmetic (2)

```
Prelude> :t (*)
????
```

```
Prelude> :t (*)
(*) :: Num a => a -> a -> a
```

```
Prelude> negate 4
-4
Prelude> :t negate
????
```

```
Prelude> :t negate
negate :: Num a => a -> a
```

Types of arithmetic (3)

What does this mean?

```
Prelude> :t (^)
(^) :: (Integral b, Num a) => a -> b -> a
```

- A number can be raised to an **`Integral`** power using ^.

```
Prelude> 2 ^ 10
1024
Prelude> 2.2 ^ 10
2655.992279142402
Prelude> 25 ^ 0.5

 <interactive>:4:4:
    No instance for (Integral b0) arising from a use of `^'
...
 <interactive>:4:6:
    No instance for (Fractional b0) arising from the literal `0.5'
...
Prelude>
```

The type of a number revisited

```
Prelude> :t 1
1 :: Num a => a
Prelude> :t 1.0
1.0 :: Fractional a => a
```

This makes more sense now!

`1` is a **`Num`**, any **`Num`**.

`1.0` is a **`Fractional`**, any **`Fractional`**.

Kinds of Numbers

```
Prelude> let sphereArea r = 4 * pi * r^2
Prelude> :t sphereArea
sphereArea :: Floating a => a -> a
```

- **`Num`** is basic arithmetic **`(+)`**, **`(*)`**
- **`Fractional`** is **`Num`** **and** (floating-point style) division.
- **`Floating`** is **`Fractional`** **and** trig functions, pi, sqrt, log.

Recap: Types

Types are shorthand descriptions of things

```
42 :: Int
```

- Type-checking vs. type-inference?
  - Type-checking is checking if the types are self-consistent
  - Type-inference is checking without being told what the types are
- Haskell supports both. In practice:
  - Most types are inferred by the compiler
  - Types given by Haskell users (you!) are a mild form of documentation
- Question to ask: **Does adding types add clarity?**

Primitives in Haskell

All primitive types start with an **upper case** letter

`Int, Integer, Float, Double, Bool, Char`

- **`Int`** - Signed value, the size of the machine **`int`**
- **`Integer`** - Arbitrary precision signed value
- **`Float`** - IEEE 32-bit floating point number
- **`Double`** - IEEE 64-bit floating point number
- **`Bool`** - result of a comparison, **`True`** or **`False`**.
- **`Char`** - a single character

There are ways of defining new types

Structure in Haskell

There are two built-in structures

- Lists - arbitrary length, every element has the same type

```
[1,2,3]::[Int]
```

- Tuples - specific length, every element can have a different type

```
(1,'c',1.0)::(Int,Char,
Float)
```

Both are used extensively in programs

(There are also ways of defining new structures)

Lists

Lists are conceptually linked-lists. You can build them directly, or build a list out of a smaller list.

```
GHCi> let xs = [1,2,3]
GHCi> xs
[1,2,3]
GHCi> length xs
3
```

The type is written **`[ type ]`**, and every element must be of that type.
What is the type of these bindings?

```
GHCi> let xs = [1,2,3::Float]
GHCi> let xs = [1] :: [Int]
GHCi> let xs = [1..100 :: Float]
GHCi> let xs = []
```