Concepts of Higher Programming Languages

Types and Class Constraints

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

What is a Type?

Definition

A **type** is a name for a collection of related values.

Example

In Haskell the basic type

```
Bool
```

contains the two logical values:

```
False    True
```

Type Errors

Definition

Applying a function to one or more arguments of the wrong type is called a **type error**.

Example

```
> 1 + False
error ...
```

1 is a number and `False` is a logical value, but + requires two numbers.

Types in Haskell

Definition

If evaluating an expression `e` would produce a value of type `t`, then `e` has type `t`, written

```
e :: t
```

Every well formed expression has a type, which can be automatically calculated **at compile time** using a process called **type inference**.

Types in Haskell

All type errors are found at compile time, which makes programs **safer and faster** by removing the need for type checks at run time. Haskell's type system is therefore **static** and **strong** (for additional explanation see lecture notes [1]).

In GHCi, the `:type` or `:t` command calculates the type of an expression, without evaluating it:

```
> not False
True

> :type not False
not False :: Bool
```

[1] `https://homepages.fhv.at/thjo/lecturenotes/concepts/classes-of-type-systems.html`

Basic Types

Haskell has a number of basic types, including:

`Bool`	-	logical values
`Char`	-	single characters
`String`	-	strings of characters
`Int`	-	fixed-precision integers
`Integer`	-	arbitrary-precision integers
`Float`	-	floating-point numbers

List Types

A list is sequence of values of the same type:

Definition

[t] is the type of lists with elements of type t.

List Examples

```
[False,True,False] :: [Bool]

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

List Types

The type of a list says **nothing** about its **length**:

```
[False,True] :: [Bool]

[False,True,False] :: [Bool]
```

The type of the elements is unrestricted. For example, we can have **lists of lists**:

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

Tuple Types

A tuple is a sequence of values of **different types**:

Definition

`(t1,t2,...,tn)` is the type of n-tuples whose ith components have type `ti` for any `i` in `1...n`.

Tuple Examples

```
(False,True) :: (Bool,Bool)

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

Tuple Types

The type of a tuple encodes its **size**:

```
(False,True) :: (Bool,Bool)

(False,True,False) :: (Bool,Bool,Bool)
```

The type of the components is **unrestricted**:

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

(True,['a','b']) :: (Bool,[Char])
```

Function Types

A function is a **mapping** (transformation) from values of one type to values of another type:

Definition

$t1 \to t2$ is the type of functions that **map** values of type `t1` to values to type `t2`.

Examples

```haskell
not :: Bool -> Bool

even :: Int -> Bool
```

Function Types

The arrow $\rightarrow$ is typed at the keyboard as `->`

The argument and result types are **unrestricted**. For example, functions with multiple arguments or results are possible using lists or tuples:

```
add :: (Int,Int) -> Int   Typ
add (x,y) = x + y         Implementierung

zeroto :: Int -> [Int]
zeroto n = [0..n]
```

Curried Functions

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

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

add' takes an integer x and returns a function add' x. In turn, this function takes an integer y and returns the result x+y.

Curried Functions

`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**:

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

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

Definition

Functions that take their arguments **one at a time** are called **curried** functions, celebrating the work of Haskell Curry on such functions.

Curried Functions

Functions with more than two arguments can be **curried** by **returning nested functions**:

```haskell
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x * y * z
```

`mult` takes an integer `x` and returns a function `mult x`, which in turn takes an integer `y` and returns a function `mult x y`, which finally takes an integer `z` and returns the result `x*y*z`.

Why is Currying useful?

Curried functions are **more flexible** than functions on tuples, because useful functions can often be made by **partially applying** a curried function.

Examples

```
add' 1 :: Int -> Int

take 5 :: [Int] -> [Int]

drop 5 :: [Int] -> [Int]

const 42 :: b -> Int
```

Currying Conventions

To avoid excess parentheses when using curried functions, two simple conventions are adopted:

Definition

The arrow `->` **associates to the right**:

```
Int -> Int -> Int -> Int
```

Means: `Int -> (Int -> (Int -> Int))`

Currying Conventions

Definition

As a consequence, it is then natural for **function application** to **associate to the left**.

`mult x y z`

Means: `((mult x) y) z`

Unless tupling is explicitly required, **all** functions in Haskell are normally defined in **curried** form.

Polymorphic Functions

Definition

A function is called **polymorphic** ("of many forms") if its type contains one or more **type variables**.

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

For any type `a`, `length` takes a list of values of type `a` and returns an integer.

Polymorphic Functions

Type variables can be **instantiated** to different types in different circumstances:

```
> length [False,True] -- NOTE: a = Bool
2

> length [1,2,3,4]     -- NOTE: a = Int
4
```

Type variables must begin with a **lower-case** letter, and are usually named `a`, `b`, `c`, etc.

Polymorphic Functions

Many of the functions defined in the standard prelude are **polymorphic**. For example:

```haskell
fst :: (a,b) -> a

head :: [a] -> a

take :: Int -> [a] -> [a]

zip :: [a] -> [b] -> [(a,b)]

id :: a -> a
```

Overloaded Functions

A polymorphic function is called **overloaded** if its type contains one or more **class constraints**.

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

For any numeric type `a`, `(+)` takes two values of type `a` and returns a value of type `a`.

Overloaded Functions

Constrained type variables can be **instantiated** to any types that satisfy the constraints:

```
> 1 + 2        -- NOTE: a = Int
3

> 1.0 + 2.0    -- NOTE a = Float
3.0

> 'a' + 'b'    -- Note: Char is not a numeric type
ERROR
```

Overloaded Functions

Haskell has a number of **Type Classes,** including:

- `Num` - Numeric types
- `Eq` - Equality types
- `Ord` - Ordered types

Examples

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

(==) :: Eq a => a -> a -> Bool

(<) :: Ord a => a -> a -> Bool
```

Hints and Tips

- When defining a new function in Haskell, it is useful to **begin** by writing down its **type**;
- Within a script, it is good practice to **state the type** of every new function defined;
- When stating the types of **polymorphic** functions that use numbers, equality or orderings, take care to include the necessary **class constraints**.