Functional Programming
Lecture 9: Haskell Types

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

What is a Type?

A type is a name for a collection of related values. For example, in Haskell the basic type

```
Bool
```

contains the two logical values:

```
False
```

```
True
```

Type Errors

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

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

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

Types in Haskell

If evaluating an expression e would produce a value of type t, then e <u>has type</u> t, written

```
e :: t
```

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

Static typing

- All type errors are found at compile time
  - safer: if it compiles, there is not type mismatch
  - faster: no need for type checks at run time
- In GHCi, the `:type` command calculates the type of an expression, without evaluating it:

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

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 a sequence of values of the same type:

```
[False,True,False] :: [Bool]
['a','b','c','d'] :: [Char]
```

In general, for any type a

`[a]` is the type of lists with elements of type a

The type of a list says nothing about its length

The type of the elements can be arbitrary

Tuple Types

A <u>tuple</u> is a sequence of values of <u>different</u> types:

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

The type of n-tuples whose i-th element has type `ti` is

`(t1,t2,…,tn)`

The type of a tuple encodes its size

The type of the components is unrestricted

Function Types

A <u>function</u> is a mapping from values of one type to values of another type:

```
not :: Bool → Bool
even :: Int → Bool
```

In general:

> t1 → t2 is the type of functions that map values of type t1 to values to type t2.

Function Types

The arrow → is typed at the keyboard as ->

The argument and result types are unrestricted

It is encouraged to write types above each function

```
add :: (Int,Int) → Int
add (x,y) = x+y

zeroto :: Int → [Int]
zeroto n = [0..n]
```

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 take arguments in a different form

```
add :: (Int,Int) → Int

add’ :: Int → (Int → Int)
```

Curried Functions

Transparently works for multiple arguments

```
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.

Partial function application

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

```
add' 1 :: Int → Int

take 5 :: [Int] → [Int]

drop 5 :: [Int] → [Int]
```

Currying Conventions

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

- The arrow $\rightarrow$ associates to the <u>right</u>.

```
Int → Int → Int → Int
```

means Int $\rightarrow$ (Int $\rightarrow$ (Int $\rightarrow$ Int))

- As a consequence, it is then natural for function application to associate to the <u>left</u>.

```
mult x y z
```

means ((mult x) y) z

Polymorphic Functions

A function is called <u>polymorphic</u> ("of many forms") if its type contains type variables.

```
length :: [a] → Int
```

Type variables can be instantiated to different types in different circumstances

```
> length [False,True]
2
> length [1,2,3,4]
4
```

a = Bool

a = Int

Many of the functions defined in the standard prelude are polymorphic.

```
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.

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

```
> 1 + 2
3

> 1.0 + 2.0
3.0

> ’a’ + ’b’
ERROR
```

a = Int

a = Float

Char is not a numeric type

Haskell has a number of type classes, including:

`Num` - Numeric types

`Eq` - Equality types

`Ord` - Ordered types

For example:

```
(+)  :: 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.

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

```
type String = [Char]
```

String is a synonym for the type [Char].

Type declarations make other types easier to read.

```
type Pos = (Int,Int)

left :: Pos → Pos
left (x,y) = (x-1,y)
```

Parametrized Types

Like function definitions, type declarations can also have parameters. With

```
type Pair a = (a,a)
```

we can define:

```
mult :: Pair Int → Int
mult (m,n) = m*n

copy :: a → Pair a
copy x = (x,x)
```

Type declarations can be nested:

```
type Pos = (Int,Int)

type Trans = Pos → Pos
```

However, they cannot be recursive:

```
type Tree = (Int,[Tree])
```

Data Declarations

Define a completely new type by specifying its values

```
data Bool = False | True
```

Values False and True are the <u>constructors</u> for the type

Type and constructor names begin with a <u>capital letter</u>

Values of new types can be used in the same ways as those of built in types. Given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers :: [Answer]
answers = [Yes,No,Unknown]

flip :: Answer → Answer
flip Yes     = No
flip No      = Yes
flip Unknown = Unknown
```

Parametric Constructors

The constructors in a data declaration can also have parameters. Given

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

we can define:

```
square :: Float → Shape
square n = Rect n n
```

Circle and Rect can be viewed as functions that construct values of type Shape

New composed data types can still be decomposed by pattern matching

```
area :: Shape → Float
area (Circle r) = pi * r^2
area (Rect x y) = x * y
```

Parametric Data Declarations

One of he most common Haskell types

```
data Maybe a = Nothing | Just a
```

allows defining safe operations.

```
safediv :: Int → Int → Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

safehead :: [a] → Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

Recursive Types

New types can be declared in terms of themselves. That is, types can be <u>recursive.</u> (just not with type keyword)

```
data Nat = Zero | Succ Nat
```

A value of type Nat is either Zero, or Succ n where n :: Nat. Nat contains infinite sequence of values:

```
Zero
```

```
Succ Zero
```

```
Succ (Succ Zero)
```

We can use pattern matching and recursion to translate from `Int` to `Nat` and back.

```
nat2int :: Nat → Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int → Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

Two naturals can be added by converting them to integers, adding, and then converting back:

```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

However, using recursion the function add can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Example: Arithmetic Expressions

Recursive typed can represent tree structures, such as <u>expressions</u> from numbers, plus, multiplication.

```
data Expr = Val Int
          | Add Expr Expr
          | Mul Expr Expr
```

$$1 + 2 * 3$$

```
Add (Val 1) (Mul (Val 2) (Val 3))
```

Using recursion, it is now easy to define functions that process expressions. For example:

```
size :: Expr → Int
size (Val n)   = 1
size (Add x y) = size x + size y
size (Mul x y) = size x + size y

eval :: Expr → Int
eval (Val n)   = n
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Homework assignment 4

Evaluating a log of card game Sedma

- we provide the basic types to use
- just implementing the function (no I/O)
- will need implementing instances
  - next lecture
  - use deriving for now
- deadline is two weeks form your lab

Summary

- Everything has a type known in compile time
  - basic values
  - functions
  - data structures
- Types are key for data structures in Haskell
- Types can be instances of classes
  - polymorphic functions