Lecture 9: Haskell Types

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science, Faculty of Electrical Eng.
Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

April, 2020

What is a Type?

A **type** is a name for a collection of related values (e.g., basic, composed, functions, etc.). For example, in Haskell the basic type

$\boxed{\texttt{Bool}}$

contains the two logical values:

$\boxed{\texttt{True}}$ $\boxed{\texttt{False}}$

Type errors

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

```
Prelude> 1 + False

...  error ...
```

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

Errors prevented by types

- Applying a function to the wrong number of arguments
- Permuting the arguments of a function
- Forgetting the application of a (conversion) function

Types in functional languages

Java types: A function defined to return a string may return

- a String
- null
- an exception
- in addition to that, it can also return a "modified state of the world" (e.g. it can print a line of text to the console)
- not return at all (e.g. run into an infinite loop)

Functional languages can be (and often are) completely rigorous about types

- If it says it returns a String, it does that and only that
- No side effects!

Types in functional languages often more expressive

- They allow representing more complex properties of programs

Static typing

- Types are checked without executing a function
  - E.g., type error in an unreachable branch is still detected
  - infinite stream can have a clear type
- 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
  - \+ **clearer?:** can serve as documentation / specification
  - – sometimes more verbose code
  - – slower compilation
  - – more complex compiler implementation

Types in Haskell

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

$$\boxed{\texttt{e :: t}}$$

Every well-formed expression has a type automatically calculated at **compile time** using a process called **type inference**.

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

Tuple types

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

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

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 (not only basic)

Function types

Types of functions are denoted using ->

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

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

- The argument and result types are unrestricted
- It is encouraged to write types above each function

Curried functions

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

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

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

Curried functions

It 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]
```

Curying Conventions

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

1) The arrow operator `->` associates to the **right**.

`Int -> Int -> Int -> Int` means `Int -> (Int -> (Int -> Int))`

2) 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

A function is called **polymorphic** (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]        -- a = Bool
2
> length [1,2,3,4]           -- a = Int
4
```

Polymorphic functions

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          -- a = Int
3
> 1.0 + 2.0      -- a = Float
3.0
> 'a' + 'b'      -- Char is not numeric
ERROR
```

Type classes

Haskell has a number of type classes, including:

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

For example, you can verify by calling `:type`:

```
(+)  :: 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.

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

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

Type declarations make other types easier to read.

```haskell
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

Type declarations can be nested:

```haskell
type Pos = (Int,Int)

type Trans = Pos -> Pos
```

However, they cannot be recursive:

```haskell
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 **constructors** for the type

Type and constructor names begin with a **capital letter**

Data declarations

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 have parameters. Given

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

we can define:

```haskell
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 be decomposed by **pattern matching**

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

Parametric data declarations

One of the most common Haskell types

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

allows defining safe operations.

```haskell
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 **recursive**. (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)
...
```

Recursive types

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))
```

Recursive types

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)
```

Records

Purely positional data declarations are impractical with a large number of fields. Therefore, the fields can be named:

```
data Person = Person { firstName :: String,
                       lastName :: String,
                       age :: Int,
                       height :: Float,
                       phone :: String,
                       address :: String}
```

This allows to define records in arbitrary order

```
defaultPerson = Person {lastName="Smith",
                        firstName="John",...
```

And access fields using automatically generated functions, e.g.,

```
firstName :: Person -> String
```

Example: Arithmetic expressions

Recursive typed can represent tree structures, such as **expressions** 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))
```

Example: Arithmetic expressions

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

Summary

- Everything has a type known in compile time
  - basic values
  - functions
  - data structures
- Types are key for data structures in Haskell
- **Algebraic types**

  compose complex types from simpler as products and unions
- Types can be instances of classes
  - polymorphic functions