Functional Programming
Lecture 8

Rostislav Horčík
Niklas Heim

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz
heimnikl@fel.cvut.cz

Haskell type system

Types

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

`Bool` denotes the 2-element set `{True, False}`.

Applying a function to an argument of incorrect type causes a **type error**.

```
Prelude> :t 1 + False

<interactive>:1:1: error:
    • No instance for (Num Bool) arising from a use of ‘+’
    • In the expression: 1 + False
```

Type systems

Typed languages are classified as

- Strongly-typed — expression types are fixed
- Weakly-typed — allow some automatic type coercions, e.g.

  ```
  "5"+6
  => "56"
  => 11
  printf("56" + 1) => "6"
  ```

- Statically-typed — types are checked during the compile time
- Dynamically-typed — types are checked during run-time (type-errors are run-time errors)

Function types

Function types

Function types are created by function type constructor `->`. It associates to the right, e.g.

```
mult :: Int -> Int -> Int -> Int
    means Int -> (Int -> (Int -> Int))
```

Correspondingly, the function application associates to the left, e.g.

```
mult x y z  means ((mult x) y) z
```

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

Lambda abstractions

Functions can be also defined by lambda abstractions. The following definitions are equivalent.

```
f :: Int -> Int -> Int -> Int
f x y z = x + y * z

f x y = \z -> x + y * z

f x = \y -> (\z -> x + y * z)

f x = \y z -> x + y * z

f = \x -> (\y -> (\z -> x + y * z))
```

Partial application

```
f :: Int -> Int -> Int -> Int
f x y z = x + y * z

f 5 :: Int -> Int -> Int
f 5 4 :: Int -> Int
```

Partial applications of operators are called sections.

```
(2/) 1 => 2.0
(/2) 1 => 0.5
filter (>0) [-1,0,2,-3,1] => [2,1]
```

Polymorphisms

Polymorphisms

A function is called polymorphic if it applies to different types. There are two kinds of polymorphisms:

- parametric — the same general function definition works for different types
- ad hoc — assigning different function definitions to the same name (overloading it)

Parametric polymorphisms

A parametric polymorphism is a function using a type variable in its type declaration, e.g.

```
len :: [a] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

The type variable `a` can be instantiated into any type.

```
> len [True, False]
2
> len [1,2,3]
3
```

Examples of parametric polymorphisms

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

```
fst :: (a,b) -> a
snd :: (a,b) -> b
head :: [a] -> a
take :: Int -> [a] -> [a]
zip :: [a] -> [b] -> [(a,b)]
id :: a -> a
```

Type classes

Ad hoc polymorphisms are implemented via type classes.
A type class defines a set of functions that can have different implementations depending on the types they are applied to.

E.g. we want to test equality `==` for many types:

```
1 == 1 => True
'a' == 'b' => False
[1,2] == [1,2] => True
```

Types testable on equality are instances of `Eq` class:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Examples of type classes

- `Eq` — types testable on equality
- `Ord` — linearly ordered types `<,>,<=,>=,max,min`
- `Num` — numeric types implementing `+,-,*,fromInteger,abs,negate,signum`
- `Fractional` — as `Num` extended by division `/`
- `Show` — implements `show :: a -> String`

Types of polymorphic functions can contain one or more type constraints, e.g.

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
(>0) :: (Num a, Ord a) => a -> Bool
```

Type declarations

New names

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

```haskell
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)
```

Parametric 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 Trans = Pos -> Pos
```

but not recursive!

Algebraic data types

Algebraic data types

To define a completely new type, use algebraic data types.

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

`Answer` is called type constructor.
`Yes,No,Unknown` are called data constructors.
Constructors have to start with a capital letter.

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

flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
```

Parametric data constructors

The data constructors in a data declaration can have parameters, e.g.

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

`Circle` and `Rect` are functions that construct values of type `Shape`.

```haskell
square :: Float -> Shape
square n = Rect n n
```

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 type constructors

One of the most common Haskell types

```haskell
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)
```

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,
                           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
```

Recursive types

Algebraic data types can be recursive.

```
data List a = Nil | Cons a (List a) deriving Show
```

List `[1,2,3]` can be represented as

```
Cons 1 (Cons 2 (Cons 3 Nil)) :: Num a => List a
```

```
rev :: List a -> List a
rev lst = iter lst Nil where
    iter Nil acc = acc
    iter (Cons x l) acc = iter l (Cons x acc)
```

Show instance

We can make `List a` our own instance of `Show`

```
data List a = Nil | Cons a (List a)
```

Suppose we wish to display our lists as `<1, 2, 3>`

```
instance Show a => Show (List a) where
  show lst = "<" ++ disp lst ++ ">" where
    disp Nil = ""
    disp (Cons x Nil) = show x
    disp (Cons x l) = show x ++ ", " ++ disp l
```

Arithmetic expressions

Arithmetic expressions can be represented as

```
data Expr a = Val a
            | Add (Expr a) (Expr a)
            | Mul (Expr a) (Expr a)
```

It is easy to evalute them

```
eval :: (Num a) => Expr a -> a
eval (Val x) = x
eval (Add x y) = eval x + eval y
eval (Mul x y) = eval x * eval y
```

Show instance

```haskell
instance (Show a, Num a) => Show (Expr a) where
  show (Val a) = show a
  show (Add e1 e2) = "(" ++ show e1
                         ++ " + "
                         ++ show e2 ++ ")"
  show (Mul e1 e2) = "(" ++ show e1
                         ++ " * "
                         ++ show e2 ++ ")"
```

Num instance

```haskell
instance (Ord a, Num a) => Num (Expr a) where
    x + y           = Add x y
    x - y           = Add x (Mul (Val (-1)) y)
    x * y           = Mul x y
    negate x        = Mul (Val (-1)) x
    abs x           | eval x >= 0 = x
                    | otherwise = negate x
    signum          = Val . signum . eval
    fromInteger x = Val (fromInteger x)
```

Homework assignment 3 - λ-calculus evaluator

Aim: To practice λ-calculus and algebraic data types in Haskell

```haskell
type Symbol = String
data Expr = Var Symbol
          | App Expr Expr
          | Lambda Symbol Expr deriving Eq
```

Tricky point: fresh symbols in substitutions

Points: 12

Deadline: in 3 weeks (May 11)

Penalty: after deadline -1 points every day (at most -11)

Description: all details can be found in CW

What have we learned?

- Haskell is strongly-typed language, i.e., no automatic coercion
- Haskell is statically-typed language, i.e., types are checked in compilation time
- Function types, currying, lambda expressions, sections
- Parametric polymorphism - type variables
- Adhoc polymorphism - type classes
- Parametric types
- Algebraic data types
- Type class instances