EDAN40: Functional Programming
Types and Type Classes (revisited)

Jacek Malec
Dept. of Computer Science, Lund University, Sweden
November 18th, 2015
In programming languages, a *type system* is a collection of rules that assign a property called *type* to various constructs a computer program consists of, such as variables, expressions, functions or modules.

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”
Haskell type system

Motivation:

- Static type system
- Type inferencing, thus run-time errors are rare
- Workflow: edit and typecheck instead of: edit and test run
- Actually, you will encounter run-time errors, too
Type theory

- Hindley-Milner type system
- type inference algorithm W
- origins: Haskell Curry, typed lambda calculus, 1958 (wait a couple of lectures:-)
- deduces most general type, even without type annotations (1969, Hindley)
- complete (1982, Milner and Damas)
- normally linear time, suitable for large programs
- for bounded nesting of let-bindings: polynomial
- for “pathological” inputs: exponential (1990)
Three kinds of type declarations

type Name = String

type synonym

data Season = Spring | Summer | Autumn | Winter

algebraic datatype

newtype Name = Nm String

renamed datatype
Same as data with a single unary constructor. Better performance as there is no runtime bookkeeping of a separate type.
Qualified types

> :type elem
elem :: (Eq a) => a -> [a] -> Bool

Qualification needed here to ensure that equality test is defined. Uses type classes. E.g.

> elem sin [sin,cos,tan,cot]

causes a type error.
Type classes

A structured way to introduce *overloaded* (or *polymorphic*) functions

class Example a where
    f1 :: a -> a -> String
    f2 :: a -> a
    f3 :: a

Usage: create *instances*

instance Example Int where
    f1 x y = show $(+) x y
    f2 = (+1)
    f3 = 0
Class and instance declaration

Class:

```haskell
class Graphical a where
    shape :: a -> Graphics
```

Instances:

```haskell
instance Graphical Box where
    shape = boxDraw -- assumed to be previously defined

instance Graphical a => Graphical [a] where
    shape = (foldr1 overGraphic) . (map shape)
```
class Graphical a => Enclosing a where
    encloses :: Point -> a -> Bool

Multiple constraints:

(Eq a, Show a) => ....

Multiple inheritance:

class (Eq a, Show a) => EqShow a
Another example

data Eq a => Set a = NilSet | ConsSet a (Set a)

Introduces two (data) constructors \texttt{NilSet} and \texttt{ConsSet} with types

\[
\begin{align*}
> &: \texttt{t NilSet} \\
\texttt{NilSet} &: \texttt{Set a} \\
> &: \texttt{t ConsSet} \\
\texttt{ConsSet} &: \texttt{Eq a} => \texttt{a} \rightarrow \texttt{Set a} \rightarrow \texttt{Set a}
\end{align*}
\]

Type inference will ensure that \texttt{ConsSet} can only be applied to values typed as instances of \texttt{Eq}.

\[
\begin{align*}
f \ (\texttt{ConsSet a s}) &= a \\
> &: \texttt{t f} \\
f &: \texttt{Eq a} => \texttt{Set a} \rightarrow \texttt{a}
\end{align*}
\]
Default definitions

class Eq a where
    (==), (!=) :: a -> a -> Bool
    x != y = not (x==y)
    x == y = not (x!=y)
data Season = Spring | Summer | Autumn | Winter
    deriving (Eq, Ord, Enum, Show, Read)

notWinter = [Spring..Autumn]

From Prelude only Eq, Ord, Enum, Bounded, Show and Read can be derived.
data Season = Spring | Summer | Autumn | Winter
  deriving (Eq, Ord, Enum, Show, Read)

notWinter = [Spring..Autumn]

From Prelude only Eq, Ord, Enum, Bounded, Show and Read can be derived.

“Classes defined by standard libraries may also be derivable.”
See “generic classes” in GHC (but not in pure Haskell 2010).
Haskell vs. Java

Haskell types ⇔ Java classes
Haskell class ⇔ Java interface

Java: A class implements an interface
Haskell: A type is an instance of a class

Java: An object is an instance of a class
Haskell: An expression has a type
Consider the following class (taken from the Prelude):

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

The `fmap` function generalizes the `map` function used previously.

```haskell
instance Functor [] where
    fmap = map
```
Functor laws (not enforced by Haskell):

\[
\text{fmap } \text{id} = \text{id} \\
\text{fmap } (f \cdot g) = (\text{fmap } f) \cdot (\text{fmap } g)
\]

The laws mean that \text{fmap} does not alter the structure of the functor.
Type class examples

Other instances:

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Note: higher-order class definitions Here Maybe and Tree, not Maybe a or Tree a, is a functor!

Jacek Malec, http://rss.cs.lth.se
Other instances:

```haskell
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Functor Tree where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

Note: higher-order class definitions
Here `Maybe` and `Tree`, not `Maybe a` or `Tree a`, is a functor!
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a

m >> k = m >>= \_ -> k
fail s = error s
Requirements on monadic types

All instances of \texttt{Monad} should obey the following laws:

\begin{align*}
\text{return } a \ >>=> \ k & = k \ a \\
\text{m} \ >>=> \ \text{return} & = \text{m} \\
\text{m} \ >>=> (\langle x \rightarrow k \ x \ >>=> \ h \rangle) & = (\text{m} \ >>=> \ k) \ >>=> \ h
\end{align*}

Instances of both \texttt{Monad} and \texttt{Functor} should satisfy also:

\begin{align*}
\text{fmap } f \ \text{x} & = \text{x} \ >>=> \ \text{return} \ . \ f
\end{align*}
Field labelling

Type definitions

data C = F Int Int Bool

and

data C = F { f1, f2 :: Int, f3 :: Bool}

are exactly the same (except that we get “deconstructor” functions)
Field labelling

Type definitions

data C = F Int Int Bool

and

data C = F { f1, f2 :: Int, f3 :: Bool}

are exactly the same (except that we get “deconstructor” functions)

Note that in pattern matching notation F {} matches every use of type F.
Type renaming

newtype Age = Age Int

or

newtype Age = Age {unAge :: Int}

Note 1: Just one field possible!
Note 2: the second variant brings into scope two functions, 
constructor and deconstructor:

Age :: Int -> Age
unAge :: Age -> Int
All numeric types are instances of the `Num` class.

```haskell
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger :: Integer -> a
```
Haskell predefined type classes

Taken from Prelude
Numeric type classes
# Main numeric types

<table>
<thead>
<tr>
<th>Type</th>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integral</td>
<td>Arbitrary-precision integers</td>
</tr>
<tr>
<td>Int</td>
<td>Integral</td>
<td>Fixed-precision integers</td>
</tr>
<tr>
<td>(Integral a)</td>
<td>RealFrac</td>
<td>Rational numbers</td>
</tr>
<tr>
<td>Float</td>
<td>RealFloat</td>
<td>Floating-point, single precision</td>
</tr>
<tr>
<td>Double</td>
<td>RealFloat</td>
<td>Floating-point, double precision</td>
</tr>
</tbody>
</table>
Main numeric classes

Num (Eq, Show)
  Fractional
    Floating

Real
  Integral
  RealFrac (Fractional)
    RealFloat (Floating)

(+), (-), (*), ...
(/)
exp, log, sin, cos, ...
toRational
quot, rem, mod, ...
round, truncate
exponent, significand, ...
Extended Example: MyNatural

numeric type based on Peano’s definition

data MyNatural = Zero | Succ MyNatural
  deriving (Eq, Show)

Some values of this type:

two = Succ $ Succ Zero
three = Succ two
Functions on MyNatural

\[
\begin{align*}
\text{natPlus} \ Zero \ y &= y \\
\text{natPlus} \ (\text{Succ} \ x) \ y &= \text{Succ} \ (\text{natPlus} \ x \ y) \\

\text{natMinus} \ x \ Zero &= x \\
\text{natMinus} \ Zero \ y &= \text{error} \ "\text{Negative Natural}" \\
\text{natMinus} \ (\text{Succ} \ x) \ (\text{Succ} \ y) &= \text{natMinus} \ x \ y
\end{align*}
\]
Functions on MyNatural

\[
\begin{align*}
natTimes \ Zero \ y & = \ Zero \\
natTimes \ (Succ \ x) \ y & = \ natPlus \ y \ (natTimes \ x \ y) \\
natSignum \ Zero & = \ Zero \\
natSignum \ (Succ \ x) & = \ Succ \ Zero \\
\end{align*}
\]

\[
\begin{align*}
integerToNat \ 0 & = \ Zero \\
integerToNat \ (x+1) & = \ Succ \ (integerToNat \ x)
\end{align*}
\]
Making MyNatural a number

instance Num MyNatural where
  (+) = natPlus
  (-) = natMinus
  (*) = natTimes
  negate = error "Negative natural"
  abs x = x
  signum = natSignum
  fromInteger = integerToNat
showNat n = show (intValue n)
  where
  intValue Zero = 0
  intValue (Succ x) = 1 + intValue x

instance Show MyNatural where
  show = showNat

and remove previous deriving of Show !
Another example: ListNatural

Natural numbers corresponding to lists (of nothing)

type ListNatural = [(())]

For example:

twoL = [(()),()]
threeL = [(()),(),()]

What is: (:)?
What is: (++)?
What is: map (const ())?
What do these functions do?

\[ f_1 \ x \ y = \text{foldr} (:) \ x \ y \]
\[ f_2 \ x \ y = \text{foldr} (\text{const} (f_1 \ x)) \ [] \ y \]
\[ f_3 \ x \ y = \text{foldr} (\text{const} (f_2 \ x)) \ [()] \ y \]

Continue this definition:

\[
\text{instance } \text{Num} \ \text{ListNatural} \ \text{where} \ \ldots
\]

Note: requires \text{ListNatural} to be declared as a \text{newtype}!
Church numbers

type ChurchNatural a = (a -> a) -> (a -> a)

zeroC, oneC, twoC :: ChurchNatural a
zeroC f = id -- zeroC = const id
oneC f = f -- oneC = id
twoC f = f.f
Church numbers

\[
\begin{align*}
\text{succC } n \ f &= f.(n \ f) \\
\text{threeC} &= \text{succC } \text{twoC} \\
\text{plusC } x \ y \ f &= (x \ f).(y \ f) \\
\text{timesC } x \ y &= x.y \\
\text{expC } x \ y &= y \ x
\end{align*}
\]
Church numbers

\[
\text{showC } x = \text{show } (x (+1)) 0
\]

\[
\text{pc} = \text{showC } \text{plusC twoC threeC}
\]
\[
\text{tc} = \text{showC } \text{timesC twoC threeC}
\]
\[
\text{xc} = \text{showC } \text{expC twoC threeC}
\]