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 (future topic)

- 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)

Type inference

\[
f : : A \rightarrow B \quad e : : A \\
\frac{}{f \ e : : B}
\]

Basic types

- Bool
- Char
- String
- Int
- Integer
- Float
- Double

\[
[Int] \\
[a] \\
(a, b) \\
f : : a \rightarrow b
\]
Basic types during exam

(+) (:) 
(+ 5) (1 :) (filter odd) (:[1,2,3])
-- below actual question
((.):)
(+0).(0+)
(.)(.)

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 inheritance**

```haskell
class Graphical a => Enclosing a where
    encloses :: Point -> a -> Bool
```

Multiple constraints:

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

Multiple inheritance:

```haskell
class (Eq a, Show a) => EqShow a
```

**Another example**

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

Introduces two (data) constructors `NilSet` and `ConsSet` with types

```haskell
NilSet :: Set a
ConsSet :: Eq a => a -> Set a -> Set a
```

Type inference will ensure that `ConsSet` can only be applied to values typed as instances of `Eq`.

```haskell
f (ConsSet a s) = a
f :: Eq a => Set a -> a
```

**Default definitions**

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

```haskell
x /= y = not (x==y)
x == y = not (x/=y)
```
Derived instances

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).

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

17(41)

-- Maybe type
data Maybe a = Nothing | Just a deriving (Eq,Ord,Read,Show)

maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing = n
maybe n f (Just x) = f x

How does deriving work?

Answer: “naturally” or “magically”:-)
Exact (somewhat) answer can be found in Haskell 2010 report, Chapter 11.
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

Type class example

Consider the following class (taken from the Prelude):

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

The fmap function generalizes the map function used previously.

instance Functor [] where
  fmap = map

A note

Functor laws (not enforced by Haskell):

fmap id = id
fmap (f . g) = (fmap f) . (fmap g)

The laws mean that fmap does not alter the structure of the functor

Type class examples

Other instances:

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)
**Type class examples**

Other instances:

```hs
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!

**One more important type class**

```hs
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 >>= (\x -> k x)
  fail s = error s
```

**Requirements on monadic types**

All instances of Monad should obey the following laws:

```hs
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Instances of both Monad and Functor should satisfy also:

```hs
fmap f xs = xs >>= return . f
```

**Field labelling**

Type definitions

```hs
data C = F Int Int Bool
```

and

```hs
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
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

Numbers in Haskell

All numeric types are instances of the Num class.

class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs, signum :: a -> a
  fromInteger :: Integer -> a

Haskell 98 predefined type classes

Taken from Prelude
### Numeric type classes

#### Main numeric types

- **Integer**
- **Integral**
- **Arbitrary-precision integers**
- **Int**
- **Fixed-precision integers**
- **(Integral a) => Ratio a**
  - **RealFrac**
  - **Rational numbers**
  - **Rational = Ratio Integer**
- **Float**
- **RealFloat**
  - **Floating-point, single precision**
- **Double**
- **RealFloat**
  - **Floating-point, double precision**

#### Main numeric classes

- **Num (Eq, Show) (+), (-), (*), ...**
- **Fractional (/)**
- **Floating exp, log, sin, cos, ...**
- **Real toRational**
- **Integral quot, rem, mod, ...**
- **RealFrac (Fractional) round, truncate**
- **RealFloat (Floating) exponent, significand, ...**

### Extended Example: MyNatural

A numeric type based on Peano’s definition

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

Some values of this type:

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

- **natPlus**:
  - `natPlus Zero y = y`
  - `natPlus (Succ x) y = Succ (natPlus x y)`

- **natMinus**:
  - `natMinus x Zero = x`
  - `natMinus Zero y = error "Negative Natural"`
  - `natMinus (Succ x) (Succ y) = natMinus x y`

Making MyNatural a number

- **instance Num MyNatural where**
  - `(+)= natPlus`
  - `(-)= natMinus`
  - `(*)= natTimes`
  - `negate = error "Negative natural"`
  - `abs x = x`
  - `signum = natSignum`
  - `fromInteger = integerToNat`

Better output

- **showNat n = show (intValue n)**
  - `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 ())`?

---

**ListNatural, Exercise**

- What do these functions do?
  ```
  f1 x y = foldr (:) x y
  f2 x y = foldr (const (f1 x)) [] y
  f3 x y = foldr (const (f2 x)) [] y
  ```

- Continue this definition:
  ```
  instance Num ListNatural where ...
  ```
  Note: requires `ListNatural` to be declared as a `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
```

```
succC n f = f.(n f)
threeC = succC twoC
```

```
plusC x y f = (x f).(y f)
timesC x y = x.y
expC x y = y x
```
Church numbers

showC x = show $ (x (+1)) 0

pc = showC $ plusC twoC threeC
tc = showC $ timesC twoC threeC
xc = showC $ expC twoC threeC