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

1. Hindley-Milner type system
2. type inference algorithm W
3. origins: Haskell Curry, typed lambda calculus, 1958 (wait a couple of lectures:-)
4. deduces most general type, even without type annotations (1969, Hindley)
5. complete (1982, Milner and Damas)
6. normally linear time, suitable for large programs
7. for bounded nesting of let-bindings: polynomial
8. 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:

class Graphical a where
    shape :: a -> Graphics

Instances:

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

instance Graphical a => Graphical [a] where
    shape = (foldr1 overGraphic) . (map shape)
Class inheritance

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 NilSet and ConsSet with types

> :t NilSet
NilSet :: Set a
> :t ConsSet
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.

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

Default definitions

class Eq a where
  (==), (!=) :: a -> a -> Bool
  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.
Derived instances

data Season = Spring | Summer | Autumn | Winter
deriving (Eq, Ord, Enum, Show, Read)

notWinter = [Summer..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

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

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

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

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

All instances of Monad should obey the following laws:

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:

fmap f xs = xs >>= return . f

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:

\[
\begin{align*}
\text{Age} & \quad : \quad \text{Int} \rightarrow \text{Age} \\
\text{unAge} & \quad : \quad \text{Age} \rightarrow \text{Int}
\end{align*}
\]

Numbers in Haskell

All numeric types are instances of the \textit{Num} class.

\[
\begin{align*}
\text{class} \quad (\text{Eq a, Show a}) & \rightarrow \text{Num a} \quad \text{where} \\
\quad (+), (-), (*) & \quad : \quad a \rightarrow a \rightarrow a \\
\quad \text{negate}, \text{abs}, \text{signum} & \quad : \quad a \rightarrow a \\
\quad \text{fromInteger} & \quad : \quad \text{Integer} \rightarrow a
\end{align*}
\]

Haskell 98 predefined type classes

Taken from Prelude

Numeric type classes
Main numeric types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Arbitrary-precision integers</td>
</tr>
<tr>
<td>Int</td>
<td>Fixed-precision integers</td>
</tr>
<tr>
<td>(Integral a) =&gt; Ratio a</td>
<td>Rational numbers</td>
</tr>
<tr>
<td>Float</td>
<td>Floating-point, single precision</td>
</tr>
<tr>
<td>Double</td>
<td>Floating-point, double precision</td>
</tr>
</tbody>
</table>

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**

```haskell
natPlus Zero y = y
natPlus (Succ x) y = Succ (natPlus x y)
natMinus x Zero = x
natMinus Zero y = error "Negative Natural"
natMinus (Succ x) (Succ y) = natMinus x y
```
**Functions on MyNatural**

- `natTimes Zero y = Zero`
- `natTimes (Succ x) y = natPlus y (natTimes x y)`
- `natSignum Zero = Zero`
- `natSignum (Succ x) = Succ Zero`

- `integerToNat 0 = Zero`
- `integerToNat (x+1) = Succ (integerToNat x)`

**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)`
  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?

\[
\begin{align*}
  f_1 \ x \ y &= \text{foldr} \ (\:) \ x \ y \\
  f_2 \ x \ y &= \text{foldr} \ (\text{const} \ (f_1 \ x)) \ [] \ y \\
  f_3 \ x \ y &= \text{foldr} \ (\text{const} \ (f_2 \ x)) \ [[]] \ y
\end{align*}
\]

Continue this definition:

```
instance Num ListNatural where ...
```

Note: requires ListNatural to be declared as a newtype!

\[
\begin{align*}
  \text{type ChurchNatural } a &= (a \to a) \to (a \to a) \\
  \text{zeroC, oneC, twoC : : ChurchNatural } a \\
  \text{zeroC } f &= \text{id} \quad \text{-- zeroC = const id} \\
  \text{oneC } f &= f \quad \text{-- oneC = id} \\
  \text{twoC } f &= f \circ f
\end{align*}
\]

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

\[
\begin{align*}
  \text{showC } x &= \text{show$ $(x \ (+1))$ 0} \\
  \text{pc } &= \text{showC$ $(\text{plusC twoC threeC})$} \\
  \text{tc } &= \text{showC$ $(\text{timesC twoC threeC})$} \\
  \text{xc } &= \text{showC$ $(\text{expC twoC threeC})$}
\end{align*}
\]