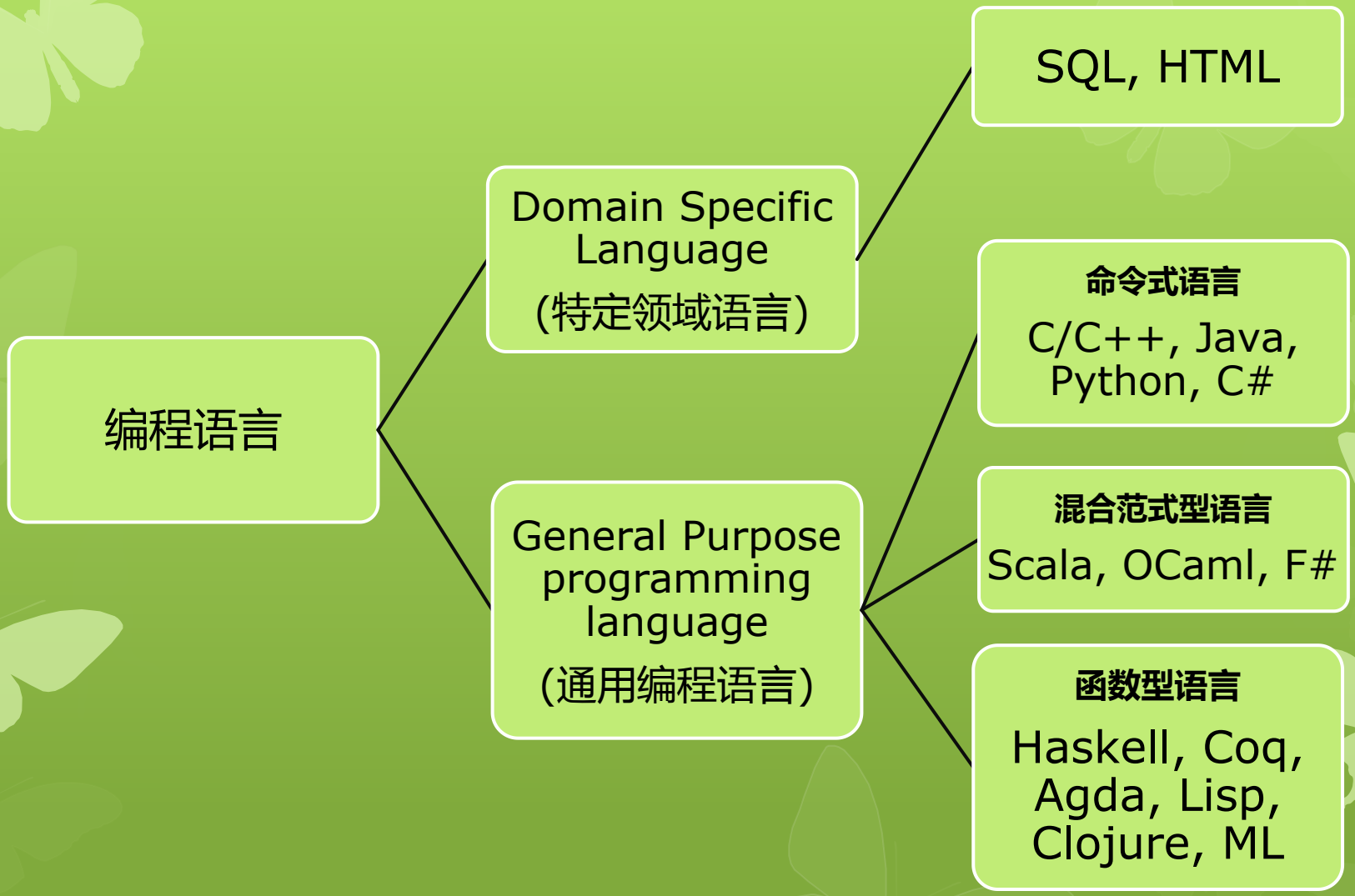
Haskell中的类型与类型系统

网易杭州研究院 张淞

@阅千人而惜知己

编程语言分类


编程语言
Domain Specific Language
(特定领域语言)
SQL, HTML
General Purpose programming language
(通用编程语言)
命令式语言
C/C++, Java, Python, C#
混合范式型语言
Scala, OCaml, F#
函数型语言
Haskell, Coq, Agda, Lisp, Clojure, ML

什么是Haskell？

- 一个强类型的、纯的、惰性的函数式编程语言
- 第一个编译器的版本于1990年4月发布
- 每年有相当多的博士在改进它，进化速度非常快
- see http://www.haskell.org

什么是Haskell？

- 每个值都有严格的类型，不能任意转型，也没有隐式转型。
- 对于纯函数只要输入确定结果必然确定，与计算机当前的状态无关。
- Haskell可以严格求值，7.10前需要我们手动控制，但7.12中会有-XStrict扩展

演示

- 用SMT Solver解百鸡问题

公鸡每只5元，母鸡每只3元，小鸡3只1元，用100元钱买100只鸡，求公鸡、母鸡、小鸡各有多少只。

$$\begin{cases} x \geq 0, y \geq 0, z \geq 0 \\ x + y + z = 100 \\ 15x + 9\,y + z = 100 \end{cases}$$

- 函数反应式编程模拟小球下落

$$y = v_0 + \int -g\,dt \qquad s = y_0 + \int v_0\,dt$$

Haskell中的主要概念

Kind（多态kind、实体kind、data kind）

类型(多态类型、类型类限定类型、实体类型)

类型类

函数/运算符（位置、结合性、优先级）

值

Haskell中的值

- ‘1’, True
- 函数
  - not :: Bool -> Bool
  - id :: a -> a
  - show :: Show a => a -> String
  - 运算符只是有优先级的函数，它们可以相互转化
  - (+) 1 2与1 + 2相同
  - mod 19 7与19 `mod` 7相同

Haskell中的类型

- 基本类型：Int, Integer, Char, type String = [Char]
- 基于基本类型的函数
  - ord :: Char -> Int
  - chr :: Int -> Char
- 多态类型：[a], Maybe a, (a,b)
- 多态类型函数：
  - id :: a -> a,
  - const :: a -> b -> a
  - length :: [a] -> Int
  - size :: Tree a -> Int
  - fst :: (a,b) -> a

所以的值都有精确的类型，‘1’ :: Char, True :: Bool

代数类型

枚举类型

```
data Bool = False | True
        deriving (Eq, Show, Ord, Enum, Bounded, Read)
```

参数化类型　类似于Java中的泛型

```
data Maybe a = Nothing | Just a
        deriving (Eq, Show, Ord, Read)
         -- 解决了Tony Hoare的十亿美元的null问题!
```

递归类型

```
data Nat = Zero | Succ Nat
        deriving (Eq, Show, Ord, Read)
```

构造类型

```
data Pair a b = Pair a b    -- 类型构造器与数据构造器名字一样
data Person = Person {name :: String, age:: Int}

data List a = Nil | Cons a (List a)  -- data [] a = [] | a : [a]
        deriving (Eq, Show, Ord, Read)
```

基于这些类型的函数

每个模式实际上是匹配的构造器与参数。

```
not :: Bool -> Bool
not False = True
not True = False

fromJust :: Maybe a -> a
fromJust (Just x) = x
fromJust Nothing = error “cannot get value from nothing”
```

error的类型是什么？

基于这些类型的函数

每个模式实际上是匹配的构造器与参数。

```
not :: Bool -> Bool
not False = True
not True = False

fromJust :: Maybe a -> a
fromJust (Just x) = x
fromJust Nothing = error “cannot get value from nothing”
```

error的类型是什么？ `error :: String -> a`

基于这些类型的函数

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ n) m = add n (Succ m)

foo :: Person -> String
foo (Person n s) = n ++ if s > 18
                        then “ is an adult”
                        else “ is a child”

sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs
```

类型类

- 相当于多种类型的公共属性

```
(==) :: Eq a => a -> a -> Bool
```

- 类型类间的有依赖关系

```
class Eq a => Ord a
```

```
(>=),(>),(<=),(<) :: Ord a => a -> a -> Bool
```

- `show :: Show a => a -> String`
- 函数的重载 `(+) :: Num a => a -> a -> a`
  - Int, Integer, Double, Float都可相加
- *Show, Eq, Ord, Enum, Read, Bounded*

定义类型类

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    (==) x y = not (x /= y)
    (/=) x y = not (x == y)
    {-# MINIMAL (==) | (/=) #-}
class Show a where
    show :: a -> String
data Person = Person Name Int
instance Eq Person where
    (Person n1 i1) == (Person n2 i2) = n1 == n2 && i1 == i2
instance Show Person where
    show (Person name age) = name ++ show age
```

Functor函子类型类中的fmap函数

```
data List a = Nil | Cons a (List a)
class List<T>{
    T e = null;
    List<T> list = null;
}
data Tree a = Leaf | Node a (Tree a) (Tree a)
list :: List Int
list = Cons 10 (Cons 11 (Cons 6 (Cons 1 Nil)))

tree :: Tree Int
tree = Node 10 (Node 11 Leaf Leaf) (Node 6 (Node 1 Leaf Leaf)
Leaf)
```

Functor函子类型类中的fmap函数

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

instance Functor List where
    fmap f Nil = Nil
    fmap f (Cons a l) = Cons (f a) (fmap f l)

instance Functor Tree where
    fmap f Leaf = Leaf
    fmap f (Node v l r) = Node (f v) (fmap f l) (fmap
f r)
```

可以使用DeriveFunctor编译器扩展自动生成

数学对于Functor的指导意义

$F(g) \circ F(f) = F(g \circ f)$

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

fmap g (fmap f v) = fmap (g.f) v

{-# RULE “fmap”
      forall g f v fmap g (fmap f v) = fmap (g.f) v #-}

fmap (+10) (fmap (*2) [1..10]) = fmap ((+10).(*2)) [1..10]
```

```
int[] arr
for(int i = 0 ; i < arr.length; i++){
    arr[i] = arr[i] * 2;
}

for(int i = 0 ; i < arr.length; i++){
    arr[i] = arr[i] + 10;
}
```

类型类与Java的接口的不同

1、接口在Haskell相当于是一个字典，函数调用会根据不同的类型来传递。多几个类型类限定相当于是多传入了几个字典。Java中要把接口合并起来有些啰嗦。

2、由于Haskell的类型是代数数据类型，即全部可以用单位元、类型加法、类型乘法、类型复合来定义，所以类型类实例可以自动实现能力极强，而且方法特别多——通用编程、摒弃模板化编程、元编程。

3、还有很多其他不同。

演示：处理JSON数据的aeson库、漂亮打印的generic-pretty库，类型的序列化。

```
interface Show<T>{ String show (T a);}
interface Eq<T>{ boolean eq(T a);}
interface ShowEq<T> extends Eq<T>, Show<T>{}
class Person implements Eq<Person>, Show<Person>{
    String name ; int age;
    Person(String name, int age){
        this.name = name; this.age = age;}
    public boolean eq(Person a) {
        return this.name.equals(a.name) && this.age == a.age;}
    public String show() {
        return this.name + " " + this.age;  }
}
foo :: (Show a, Eq a) => a -> a -> (Bool, String)
```

类型系统与Java的不同

1、Java中没有“高阶泛型” 高阶Kind

```
class Mk<T,K>{  T<K> m ; }
new Mk<Vector, Integer>()
new Mk<ArrayList, Node>()
```

2、没有“多态泛型” Kind多态

类型的Kind

类型的类型为kind，类型把值分类，class给类型赋予了属性，kind把类型分类。*代表一个实体的kind如：

```
3 :: Int , Int :: *
Just 3 :: Maybe Int,  Maybe :: * -> *, Maybe Int :: *
    -- 只有kind为*的类型下面才有值
    -- 没有一个值的类型是Maybe，只有Maybe Int或Maybe Char等
Either :: * -> * -> * -- 同理也没有一个值的类型是Either
(,) :: * -> * -> *  -- 元组类型的构造器
```

高阶Kind

```
data RoseTree a = RLeaf a
                | RNode a [RoseTree a]
                -- RNode  a ([] (RoseTree a))
>:kind []
* -> *
data BinTree a = BLeaf a
               | BNode a (Pair (BinTree a))
data Pair a = MkPair a a  -- (a,a)
>:kind Pair
* -> *
```

高阶Kind

```
data Tree k a = Leaf a
              | Node a (k (Tree k a))

type RoseTree a = Tree [] a
type BinTree a = Tree Pair a
type AnnTree a = Tree AnnPair a

data Pair a = MkPair a a
data AnnPair a = AnnPair String a a
>:kind Tree
(* -> *) -> * -> *
```

Kind多态 Typeable

```
class Typeable a where   -- a :: *
    typeOf :: a -> String
instance Typeable Int where -- Int :: *
    typeOf _ = “Int”

instance Typeable Maybe where -- ??  Maybe :: * -> *
solution:
class Typeable2 t where
    typeOf2 :: t a -> String
class Typeable3 t where
    typeOf3 :: t a b -> String
    -- 显然这个方案不好！
```

Kind多态 Typeable

```
class Typeable (t :: k) where
    typeOf :: t -> String
    -- 但是t类型下没有值，因为它的kind为多态的k而非*，所以会有类型错误

data Proxy (a :: k) = Proxy
> :k Proxy
Proxy :: k -> *
class Typeable (t :: k) where
    typeOf :: Proxy t -> String

instance Typeable Maybe where
    typeOf Proxy = “Maybe”
```

更多关于类型的内容

- RankNType
- 基于类型的运算
- 类型的角色
- 依赖类型