函数式编程与 Haskell

Introduction to functional programming

Vani

Peking University

December 12, 2012

什么是函数式编程？

你一定知道面向对象

什么是函数式编程？

你一定知道面向对象

- 封装

什么是函数式编程？

你一定知道面向对象

- 封装
- 继承

什么是函数式编程？

你一定知道面向对象

- 封装
- 继承
- 多态

什么是函数式编程？

你一定知道面向对象

- 封装
- 继承
- 多态
- 总之好厉害

什么是函数式编程？

你一定知道面向对象

- 封装
- 继承
- 多态
- 总之好厉害

函数式

什么是函数式编程？

你一定知道面向对象

- 封装
- 继承
- 多态
- 总之好厉害

函数式

- 这是什么东西？？？

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

- 不可变量

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

- 不可变量
- 惰性求值

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

- 不可变量
- 惰性求值
- 高阶函数

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

- 不可变量
- 惰性求值
- 高阶函数
- 无副作用

其实...

函数式编程是一种完全不同的编程形式，与之对应的是指令式编程。

特点:

- 不可变量
- 惰性求值
- 高阶函数
- 无副作用
- 一切皆函数

从停机问题开始

什么是停机问题？

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

作用？

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

作用？

轻松愉悦地证明:

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

作用？

轻松愉悦地证明:

- 哥德巴赫猜想

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

作用？

轻松愉悦地证明:

- 哥德巴赫猜想
- 费马大定理

从停机问题开始

什么是停机问题？

给定任意一个程序及起输入，判断该程序是否能在有限的计算以内结束

作用？

轻松愉悦地证明:

- 哥德巴赫猜想
- 费马大定理
- ……

如果真有这个算法

如果真有这个算法

假如真的作出了这个算法，你只要给任意一个函数和这个函数的输入，它就能告诉你这个函数能不能结束

如果真有这个算法

假如真的作出了这个算法，你只要给任意一个函数和这个函数的输入，它就能告诉你这个函数能不能结束

我们用下面代码描述:

如果真有这个算法

假如真的作出了这个算法，你只要给任意一个函数和这个函数的输入，它就能告诉你这个函数能不能结束

我们用下面代码描述:

Code

```
bool halting(func, input) {
    return if_func_will_halt_on_input;
}
```

充分利用停机判定

充分利用停机判定

我们构造另一个函数：

充分利用停机判定

我们构造另一个函数：

Code

```
void evil(func) {
    if (halting(func, func)) {
        for(;;);
    }
}
```

接下来，调用：evil(evil)

悖论

悖论

到底停不停机？？

悖论

到底停不停机？？

所以停机问题不可判定

λ 演算法

基本语法：

λ 演算法

基本语法：

- $< expr > ::= < identifier >$

λ 演算法

基本语法：

- $< expr > ::= < identifier >$
- $< expr > ::= \ \lambda \ < identifier - list > . \ < expr >$

λ 演算法

基本语法：

- $< expr > ::= < identifier >$
- $< expr > ::= \ \lambda \ < identifier - list > . \ < expr >$
- $< expr > ::= \ (< expr > < expr >)$

λ 演算法

基本语法:

- $< expr > ::= < identifier >$
- $< expr > ::= \ \lambda \ < identifier - list > . \ < expr >$
- $< expr > ::= \ (< expr > < expr >)$

比如:

λ 演算法

基本语法：

- $< expr > ::= < identifier >$
- $< expr > ::= \ \lambda \ < identifier - list > . \ < expr >$
- $< expr > ::= \ (< expr > < expr >)$

比如：

$$\lambda\, x\, y.\, x + \, y$$

λ 演算公理

λ 演算公理

置换公理

λ 演算公理

置换公理

- $\lambda\ x\ y.\ x\ +\ y\ =>\ \lambda\ a\ b.\ a\ +\ b$

λ 演算公理

置换公理

- $\lambda\ x\ y.\ x\ +\ y\ => \ \lambda\ a\ b.\ a\ +\ b$

代入公理

λ 演算公理

置换公理

- $\lambda\ x\ y.\ x + y \ => \ \lambda\ a\ b.\ a + b$

代入公理

- $(\lambda\ x\ y.\ x + y)\ a\ b \ => \ a + b$

λ 演算公理

置换公理

- $\lambda\ x\ y.\ x + y \ => \ \lambda\ a\ b.\ a + b$

代入公理

- $(\lambda\ x\ y.\ x + y)\ a\ b \ => \ a + b$

以上就是 λ 演算的全部公理系统

函数生成器

函数生成器

我们可以使用 λ 演算法定义各种各样的语法

函数生成器

我们可以使用 λ 演算法定义各种各样的语法

"and" operator:

函数生成器

我们可以使用 λ 演算法定义各种各样的语法

"and" operator:

```
let And =
        \ True False. False
        \ True True. True
        \ False True. False
        \ False False. False
```

函数生成器

我们可以使用 λ 演算法定义各种各样的语法

"and" operator:

```
let And =
        \ True False. False
        \ True True. True
        \ False True. False
        \ False False. False
```

"if" statement:

函数生成器

我们可以使用 λ 演算法定义各种各样的语法

"and" operator:

```
let And =
        \ True False. False
        \ True True. True
        \ False True. False
        \ False False. False
```

"if" statement:

```
let if = \cond tv fv. (cond and tv) or (not cond and fv)
```

似乎还有缺陷

似乎还有缺陷

如何实现递归？

似乎还有缺陷

如何实现递归？

写一个计算阶乘的函数...

似乎还有缺陷

如何实现递归？

写一个计算阶乘的函数...

Code

```
let fac = \n. if n == 0 then 1 else n * fac(n-1)
```

似乎还有缺陷

如何实现递归？

写一个计算阶乘的函数...

Code

```
let fac = \n. if n == 0 then 1 else n * fac(n-1)
```

Error! "fac" has not been defined

把自身参数化

把自身参数化

为了实现自己调用自己，不妨传一个参数进入

把自身参数化

为了实现自己调用自己，不妨传一个参数进入

Code

```
let fac = \self n. if n == 0 then 1 else n * self(self,n-1)
```

只不过传了一个函数重复调用而已

柯里化

柯里化

思考，如果一个两个参数的函数，我们只传给它一个参数，会得到什么？

柯里化

思考, 如果一个两个参数的函数，我们只传给它一个参数，会得到什么？

柯里化 是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术

柯里化

思考, 如果一个两个参数的函数，我们只传给它一个参数，会得到什么？

柯里化 是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术

有没有想到 C++ 里的函数适配器 bind1st, bind2nd

不动点

不动点

- 假如我们已经得到了一个完美的阶乘函数 FAC

不动点

- 假如我们已经得到了一个完美的阶乘函数 FAC
- 考虑一个有缺陷的阶乘函数:

Code

```
let fac = \self n. if n == 0 then 1 else n * self(n-1)
```

不动点

- 假如我们已经得到了一个完美的阶乘函数 FAC
- 考虑一个有缺陷的阶乘函数:

Code

```
let fac = \self n. if n == 0 then 1 else n * self(n-1)
```

- 如果把 FAC 传给 fac 会发生什么？

不动点

- 假如我们已经得到了一个完美的阶乘函数 FAC
- 考虑一个有缺陷的阶乘函数:

Code

```
let fac = \self n. if n == 0 then 1 else n * self(n-1)
```

- 如果把 FAC 传给 fac 会发生什么？
- $\mathrm{FAC} = \mathrm{fac}(\mathrm{FAC})$

不动点

- 假如我们已经得到了一个完美的阶乘函数 FAC
- 考虑一个有缺陷的阶乘函数:

Code

```
let fac = \self n. if n == 0 then 1 else n * self(n-1)
```

- 如果把 FAC 传给 fac 会发生什么？
- $\mathrm{FAC} = \mathrm{fac}(\mathrm{FAC})$
- FAC 是有缺陷的 fac 的不动点！

构造真正的递归函数

构造真正的递归函数

- 如果我们有一个神奇的函数 Y，可以得到所有伪递归函数的真正递归函数

构造真正的递归函数

- 如果我们有一个神奇的函数 Y，可以得到所有伪递归函数的真正递归函数
- 即：$\mathsf{Y}(\mathsf{F}) = \mathsf{f}$

构造真正的递归函数

- 如果我们有一个神奇的函数 Y，可以得到所有伪递归函数的真正递归函数
- 即：$Y(F) = f$
- $Y(F) = f = F(f) = F(Y(F))$

构造真正的递归函数

- 如果我们有一个神奇的函数 Y，可以得到所有伪递归函数的真正递归函数
- 即：$\mathrm{Y(F)} = \mathrm{f}$
- $\mathrm{Y(F)} = \mathrm{f} = \mathrm{F(f)} = \mathrm{F(Y(F))}$
- 如何构造这样的 Y 呢？

Y 组合子

Y 组合子

- 考虑下面的一个函数：

Y 组合子

- 考虑下面的一个函数：

Code

```
let FAC_gen = \self. fac(self(self))
```

Y 组合子

- 考虑下面的一个函数：

Code

```
let FAC_gen = \self. fac(self(self))
```

- 嵌套一下？

Y 组合子

- 考虑下面的一个函数：

Code

```
let FAC_gen = \self. fac(self(self))
```

- 嵌套一下？

Code

```
FAC_gen(FAC_gen) = fac(FAC_gen(FAC_gen))
```

Y 组合子

- 考虑下面的一个函数：

Code

```
let FAC_gen = \self. fac(self(self))
```

- 嵌套一下？

Code

```
FAC_gen(FAC_gen) = fac(FAC_gen(FAC_gen))
```

- 不妨令 FAC = FAC_gen(FAC_gen)

Y 组合子

- 考虑下面的一个函数：

Code

```
let FAC_gen = \self. fac(self(self))
```

- 嵌套一下？

Code

```
FAC_gen(FAC_gen) = fac(FAC_gen(FAC_gen))
```

- 不妨令 FAC = FAC_gen(FAC_gen)
- fac(FAC) = FAC！

Y 组合子

Y 组合子

于是对于每个伪递归，都找出一个类似 FAC_gen 这样的函数就行了

Y 组合子

于是对于每个伪递归，都找出一个类似 FAC_gen 这样的函数就行了

Y combinator

```
let Y = \f. let gen = \self. f(self(self)); return gen(gen)
```

Y 组合子

于是对于每个伪递归，都找出一个类似 FAC_gen 这样的函数就行了

Y combinator

```
let Y = \f. let gen = \self. f(self(self)); return gen(gen)
```

展开看一下：

验证

验证

- 首先定义有缺陷版本：

验证

- 首先定义有缺陷版本：
- $let\ fac = \backslash\ self\ n.\ if\ n == 0\ then\ 1\ else\ n * self(n - 1)$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$
- $Y(fac)\ =>\ gen(gen)\ =>\ fac(gen(gen))$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$
- $Y(fac)\ =>\ gen(gen)\ =>\ fac(gen(gen))$
- $fac(gen(gen))\ (n)\ =>\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ \mathbf{gen(gen)}(n-1)$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$
- $Y(fac)\ =>\ gen(gen)\ =>\ fac(gen(gen))$
- $fac(gen(gen))\ (n)\ =>\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ \mathbf{gen(gen)}(n-1)$
- $gen(gen)\ =>\ fac(gen(gen))$

验证

- 首先定义有缺陷版本:
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$
- $Y(fac)\ =>\ gen(gen)\ =>\ fac(gen(gen))$
- $fac(gen(gen))\ (n)\ =>\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ \mathbf{gen(gen)}(n-1)$
- $gen(gen)\ =>\ fac(gen(gen))$
- $\mathbf{fac(gen(gen))}(n)\ =>$

验证

- 首先定义有缺陷版本：
- $let\ fac\ =\ \backslash\ self\ n.\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ self(n\ -\ 1)$
- $Y(fac)\ =>$

  $let\ gen\ =\ \backslash\ self.\ \mathbf{fac}(self(self));\ return\ gen(gen)$
- $Y(fac)\ =>\ gen(gen)\ =>\ fac(gen(gen))$
- $fac(gen(gen))\ (n)\ =>\ if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ \mathbf{gen(gen)}(n-1)$
- $gen(gen)\ =>\ fac(gen(gen))$
- $\mathbf{fac(gen(gen))}(n)\ =>$

  $if\ n\ ==\ 0\ then\ 1\ else\ n\ *\ \mathbf{fac(gen(gen))}(n-1)$

图灵等价

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

  可以在定义函数的过程中引用自身

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

  可以在定义函数的过程中引用自身

- 于是可以证明，λ 演算系统和图灵机是等价的

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

  可以在定义函数的过程中引用自身

- 于是可以证明，λ 演算系统和图灵机是等价的
- 那和图灵停机问题等价的问题是什么呢？

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

  可以在定义函数的过程中引用自身

- 于是可以证明，λ 演算系统和图灵机是等价的
- 那和图灵停机问题等价的问题是什么呢？

  不存在一个算法能够判定任意两个 λ 函数是否等价

图灵等价

- Y 组合子的推导成功相当于在 λ 演算公理中添加了一条公理

  可以在定义函数的过程中引用自身

- 于是可以证明，λ 演算系统和图灵机是等价的
- 那和图灵停机问题等价的问题是什么呢？

  不存在一个算法能够判定任意两个 λ 函数是否等价

- 证明？

初识 haskell

初识 haskell

- 特性:

初识 haskell

- 特性:
  - 部分求值 (柯里化)

初识 haskell

- 特性：
  - 部分求值 (柯里化)
  - 惰性求值

初识 haskell

- 特性：
  - 部分求值 (柯里化)
  - 惰性求值
  - 无副作用

初识 haskell

- 特性：
  - 部分求值 (柯里化)
  - 惰性求值
  - 无副作用
  - ....

第一个函数

第一个函数

- 计算阶乘：

第一个函数

- 计算阶乘：

Code

```
fac :: Num a => a -> a
fac 0 = 1
fac n = n * fac(n - 1)
```

第一个函数

- 计算阶乘：

Code

```
fac :: Num a => a -> a
fac 0 = 1
fac n = n * fac(n - 1)
```

- 模式匹配

第一个函数

- 计算阶乘：

Code

```
fac :: Num a => a -> a
fac 0 = 1
fac n = n * fac(n - 1)
```

- 模式匹配
- 类型限制

列表

列表

- 类似 python 里的列表

列表

- 类似 python 里的列表
- ":" 操作符和模式匹配可以很方便地处理列表

列表

- 类似 python 里的列表
- ":" 操作符和模式匹配可以很方便地处理列表
- 让我们自己来实现一个 map 函数

Code

```
map' _ [] = []
map' f (x:xs) = f x:map f xs
```

列表

- 类似 python 里的列表
- ":" 操作符和模式匹配可以很方便地处理列表
- 让我们自己来实现一个 map 函数

Code

```
map' _ [] = []
map' f (x:xs) = f x:map f xs
```

- 试验一下

列表

- 类似 python 里的列表
- ":" 操作符和模式匹配可以很方便地处理列表
- 让我们自己来实现一个 map 函数

Code

```
map' _ [] = []
map' f (x:xs) = f x:map f xs
```

- 试验一下

Code

```
Prelude> map' (+3) [1,2,3]
[4,5,6]
```

列表产生器

列表产生器

- 列表产生器是什么？

列表产生器

- 列表产生器是什么？

Code

```
Prelude> [x | x<-[1..10], x `mod` 2 == 0]
[2,4,6,8,10]
```

列表产生器

- 列表产生器是什么？

Code

```
Prelude> [x | x<-[1..10], x `mod` 2 == 0]
[2,4,6,8,10]
```

- 学过数学的都能看明白吧

列表产生器

- 列表产生器是什么？

Code

```
Prelude> [x | x<-[1..10], x `mod` 2 == 0]
[2,4,6,8,10]
```

- 学过数学的都能看明白吧
- 自己来实现一个快速排序：

列表产生器

- 列表产生器是什么？

Code

```
Prelude> [x | x<-[1..10], x `mod` 2 == 0]
[2,4,6,8,10]
```

- 学过数学的都能看明白吧
- 自己来实现一个快速排序：

Code

```
qsort [] = []
qsort (x:xs) = qsort[y | y<-xs, y <= x] ++ [x] ++ qsort[y | y <-xs, y > x]
Prelude> qsort [1,3,5,3,2]
[1,2,3,3,5]
```

函数的高级用法

函数的高级用法

- 使用. 来复合两个函数：$f.gx = f(g(x))$

函数的高级用法

- 使用. 来复合两个函数：$f.gx = f(g(x))$
- 使用 $ 来改变函数求值顺序

函数的高级用法

- 使用. 来复合两个函数：$f.gx = f(g(x))$
- 使用 $ 来改变函数求值顺序
- 综合运用：

Code

```
main = interact $ concatMap(++"\n").takeWhile(/="42").lines
```

函数的高级用法

- 使用. 来复合两个函数：$f.gx = f(g(x))$
- 使用 $ 来改变函数求值顺序
- 综合运用：

Code

```
main = interact $ concatMap(++"\n").takeWhile(/="42").lines
```

Code

```
main=interact$(\[x,y]->zipWith(\a b->if a==b then '0' else '1')x y).lines
```

输入和输出

输入和输出

- haskell 里，用于执行输入和输出的函数被定义为 IO action 类型

输入和输出

- haskell 里，用于执行输入和输出的函数被定义为 IO action 类型

Code

```
main = do
    putStrLn "what's your name?"
    name <- getLine
    putStrLn ("Your name is " ++ name)
```

输入和输出

- haskell 里，用于执行输入和输出的函数被定义为 IO action 类型

Code

```
main = do
    putStrLn "what's your name?"
    name <- getLine
    putStrLn ("Your name is " ++ name)
```

- putStrLn 和 getLine 都为 IO string 类型

输入和输出

输入和输出

- getLine 其实类似与一个盒子，我们要使用 <-从里面提取一个 string

输入和输出

- getLine 其实类似与一个盒子，我们要使用 <-从里面提取一个 string
- 使用 = 号是达不到预期效果的

输入和输出

- getLine 其实类似与一个盒子，我们要使用 <-从里面提取一个 string
- 使用 = 号是达不到预期效果的

Code

```
main = do
    name <- getLine
    name' = getLine
```

输入和输出

- getLine 其实类似与一个盒子，我们要使用 <-从里面提取一个 string
- 使用 = 号是达不到预期效果的

Code

```
main = do
    name <- getLine
    name' = getLine
```

- name' 其实是一个函数，相当于 getLine

输入和输出

- getLine 其实类似与一个盒子，我们要使用 <-从里面提取一个 string
- 使用 = 号是达不到预期效果的

Code

```
main = do
    name <- getLine
    name' = getLine
```

- name' 其实是一个函数，相当于 getLine
- 于是可以执行 name<-name' 来从 name' 里提取字符串了

输入和输出

输入和输出

- 我们把一个 IO action 绑定到 main 函数，并在程序开始执行时触发

输入和输出

- 我们把一个 IO action 绑定到 main 函数，并在程序开始执行时触发
- 并使用 do 语句把若干个 IO action 函数绑定为一个

输入和输出

- 我们把一个 IO action 绑定到 main 函数，并在程序开始执行时触发
- 并使用 do 语句把若干个 IO action 函数绑定为一个

Code

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ revverseWords line
            main
```

输入和输出

- 我们把一个 IO action 绑定到 main 函数，并在程序开始执行时触发
- 并使用 do 语句把若干个 IO action 函数绑定为一个

Code

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ revverseWords line
            main
```

- 注意 return 不是终止程序执行，而是返回一个 IO action

Type 和 typeclass

Type 和 typeclass

- Type 即为通常意义下的类型，比如 Int，Bool 等

Type 和 typeclass

- Type 即为通常意义下的类型，比如 Int，Bool 等
- 一个 typeclass 定义了一些函数，所有该 typeclass 的实例都支持这些函数

Type 和 typeclass

- Type 即为通常意义下的类型，比如 Int，Bool 等
- 一个 typeclass 定义了一些函数，所有该 typeclass 的实例都支持这些函数
- 比如 Eq 的实例都支持判断是否相等，Ord 的实例都支持比较大小

Type 和 typeclass

Type 和 typeclass

- 我们先试着定义一个自己的 Type

Type 和 typeclass

- 我们先试着定义一个自己的 Type

Code

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving(Show, Read, Eq)
```

- 此时 Tree 是什么类型呢？

Type 和 typeclass

- 我们先试着定义一个自己的 Type

Code

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving(Show, Read, Eq)
```

- 此时 Tree 是什么类型呢？
- Tree a 才是一个合法的类型

Type 和 typeclass

- 我们先试着定义一个自己的 Type

Code

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving(Show, Read, Eq)
```

- 此时 Tree 是什么类型呢？
- Tree a 才是一个合法的类型
- Tree 不过是一个类型构造子

kind：类型的类型

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind
- Bool 的结果为 *

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind
- Bool 的结果为 *
- 但是 Tree 的 kind 为 *->* （有没有想到函数？）

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind
- Bool 的结果为 *
- 但是 Tree 的 kind 为 *->* （有没有想到函数？）
- BTW，Tree 类型的一个插入函数：

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind
- Bool 的结果为 *
- 但是 Tree 的 kind 为 *->* （有没有想到函数？）
- BTW，Tree 类型的一个插入函数：

Code

```
ins x Empty = Node x Empty Empty
ins x (Node a l r)
    | x <= a = Node a (ins x l) r
    | otherwise = Node a l (ins x r)
```

kind：类型的类型

- kind 即为类型的类型，可用:k 命令来检测某类型或者构造子的 kind
- Bool 的结果为 *
- 但是 Tree 的 kind 为 *->*（有没有想到函数？）
- BTW，Tree 类型的一个插入函数：

Code

```
ins x Empty = Node x Empty Empty
ins x (Node a l r)
    | x <= a = Node a (ins x l) r
    | otherwise = Node a l (ins x r)
```

- 注意它是返回了一个新的 Tree

抽象：Functor typeclass

抽象：Functor typeclass

- Functor typeclass 的实例是可以被 map 的对象

抽象：Functor typeclass

- Functor typeclass 的实例是可以被 map 的对象
- 定义：

抽象：Functor typeclass

- Functor typeclass 的实例是可以被 map 的对象
- 定义：

Code

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

抽象：Functor typeclass

- Functor typeclass 的实例是可以被 map 的对象
- 定义：

Code

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

- 注意 f 并不是一个类型，而是一个类型构造子。其 kind 为 *->*

抽象：Functor typeclass

- Functor typeclass 的实例是可以被 map 的对象
- 定义：

Code

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

- 注意 f 并不是一个类型，而是一个类型构造子。其 kind 为 *->*
- fmap 接受一个函数，它把一个类型映射成另外一个

Functor 的例子

Functor 的例子

- 来看一下 list 是如何被定义为 Functor 的实例的

Functor 的例子

- 来看一下 list 是如何被定义为 Functor 的实例的

Code

```
instance Functor [] where
    fmap = map
```

Functor 的例子

- 来看一下 list 是如何被定义为 Functor 的实例的

Code

```
instance Functor [] where
    fmap = map
```

- 另一个 Functor 的例子是 I/O action:

Functor 的例子

- 来看一下 list 是如何被定义为 Functor 的实例的

Code

```
instance Functor [] where
    fmap = map
```

- 另一个 Functor 的例子是 I/O action:

Code

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

Functor 的例子

Functor 的例子

- IO 就是一个类型构造子。

Functor 的例子

- IO 就是一个类型构造子。
- :t getLine 的结果为 IO string

Functor 的例子

- IO 就是一个类型构造子。
- :t getLine 的结果为 IO string
- :k IO 的结果为 *->*

Functor 的例子

- IO 就是一个类型构造子。
- :t getLine 的结果为 IO string
- :k IO 的结果为 *->*

Code

```
main = do line<-fmap reverse getLine
          putStr line
```

Functor 的例子

Functor 的例子

- (->)r 也是 Functor 的例子

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？
- a->r => (->) r a

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？
- a->r => (->) r a
- (->) r a 是一个类型，a 也是一个类型

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？
- a->r => (->) r a
- (->) r a 是一个类型，a 也是一个类型
- 接受一个类型，返回一个新类型？

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？
- a->r => (->) r a
- (->) r a 是一个类型，a 也是一个类型
- 接受一个类型，返回一个新类型？
- 所以 (->)r 是一个类型构造子

Functor 的例子

- (->)r 也是 Functor 的例子
- (->)r 究竟代表什么？
- a->r => (->) r a
- (->) r a 是一个类型，a 也是一个类型
- 接受一个类型，返回一个新类型？
- 所以 (->)r 是一个类型构造子
- 于是 (->)r 也可以是 Functor 的实例

Functor 的例子

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- 我们已经知道 fmap 形态为 fmap :: (a -> b) -> f a -> f b

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- 我们已经知道 fmap 形态为 fmap :: (a -> b) -> f a -> f b
- 把所有的 f 在心里替换为 (->) r

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- 我们已经知道 fmap 形态为 fmap :: (a -> b) -> f a -> f b
- 把所有的 f 在心里替换为 (->) r
- 于是就变成了 fmap :: (a -> b) -> ((->) r a) -> ((->) r b)

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- 我们已经知道 fmap 形态为 fmap :: (a -> b) -> f a -> f b
- 把所有的 f 在心里替换为 (->) r
- 于是就变成了 fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
- 换成中缀函数的形式：fmap :: (a->b)->(r->a)->(r->b)

Functor 的例子

Code

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

- 我们已经知道 fmap 形态为 fmap :: (a -> b) -> f a -> f b
- 把所有的 f 在心里替换为 (->) r
- 于是就变成了 fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
- 换成中缀函数的形式：fmap :: (a->b)->(r->a)->(r->b)
- 这正是函数的复合！

Functor law

Functor law

- Functor law:

Functor law

- Functor law:
  - fmap 的结果不改变原有 Functor 的拓扑顺序

Functor law

- Functor law:
  - fmap 的结果不改变原有 Functor 的拓扑顺序
- 思考：如何把我们的 Tree 实例化为 Functor

Functor law

- Functor law:
  - fmap 的结果不改变原有 Functor 的拓扑顺序
- 思考：如何把我们的 Tree 实例化为 Functor

Code

```
instance Functor Tree where
    fmap f Empty = Empty
    fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)

*Main> fmap (*2) (ins 3 (Node 2 Empty Empty))
Node 4 Empty (Node 6 Empty Empty)
```

进一步抽象:Applicative functors

进一步抽象:Applicative functors

- 如果我们对一个 Functor 实例 A fmap 一个多参函数会发生什么？

进一步抽象:Applicative functors

- 如果我们对一个 Functor 实例 A fmap 一个多参函数会发生什么？
- 得到一个函数集合，函数之间保留 A 的拓扑顺序，不妨称这个函数集合类型为 B

进一步抽象:Applicative functors

- 如果我们对一个 Functor 实例 A fmap 一个多参函数会发生什么？
- 得到一个函数集合，函数之间保留 A 的拓扑顺序，不妨称这个函数集合类型为 B
- 怎么把 B 上的每个函数都对应地作用于 A 上？

Applicative functor 的定义

Applicative functor 的定义

- 先围观 Applicative functor 的定义：

Applicative functor 的定义

- 先围观 Applicative functor 的定义：

Code

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Applicative functor 的定义

- 先围观 Applicative functor 的定义：

Code

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- f 被型别限定为 Functor 类型

Applicative functor 的定义

- 先围观 Applicative functor 的定义：

Code

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- f 被型别限定为 Functor 类型
- pure 接受一个值，返回一个包含那个值的 Applicative functor

Applicative functor 的定义

- 先围观 Applicative functor 的定义：

Code

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- f 被型别限定为 Functor 类型
- pure 接受一个值，返回一个包含那个值的 Applicative functor
- <*> 接受一个装有函数的 Functor 和另一个 functor，然后取出第一个 functor 里的函数对第二个做 map

Applicative functor 例子

Applicative functor 例子

- 考虑如何把 Maybe 类型作为 Applicative 实例的

Applicative functor 例子

- 考虑如何把 Maybe 类型作为 Applicative 实例的

Code

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Applicative functor 例子

- 考虑如何把 Maybe 类型作为 Applicative 实例的

Code

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

- pure 的定义使用了柯里化噢

Applicative functor 例子

- 考虑如何把 Maybe 类型作为 Applicative 实例的

Code

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

- pure 的定义使用了柯里化噢
- 如果左边是 Just，那么 <*> 会抽出其中的函数来 map 右面的值

Applicative functor 例子

- 考虑如何把 Maybe 类型作为 Applicative 实例的

Code

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

- pure 的定义使用了柯里化噢
- 如果左边是 Just，那么 $<*>$ 会抽出其中的函数来 map 右面的值
- 如果任何一个函数是 Nothing，那结果就为 Nothing

Applicative functor 例子

Applicative functor 例子

- 试一下效果吧：

Applicative functor 例子

- 试一下效果吧：

Code

```
Prelude> Just (+3) <*> Just 9
Just 12
```

Applicative functor 例子

- 试一下效果吧：

Code

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> Just (++"hahah") <*> Nothing
Nothing
```

Applicative functor 例子

- 试一下效果吧：

Code

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> Just (++"hahah") <*> Nothing
Nothing
Prelude> ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Applicative functor 例子

- 试一下效果吧：

Code

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> Just (++"hahah") <*> Nothing
Nothing
Prelude> ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

- <\$> 为一个语法糖，类似于中缀版的 fmap

Applicative functor 例子

- 试一下效果吧：

Code

```
Prelude> Just (+3) <*> Just 9
Just 12
Prelude> Just (++"hahah") <*> Nothing
Nothing
Prelude> ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

- <$> 为一个语法糖，类似于中缀版的 fmap

Code

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

从 Functor, Applicative functor 到 Monad

从 Functor, Applicative functor 到 Monad

- 因为许多的形态都都可以被 map，所以抽象出了 Functor 这个 typeclass

从 Functor, Applicative functor 到 Monad

- 因为许多的形态都都可以被 map，所以抽象出了 Functor 这个 typeclass

Code

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

从 Functor, Applicative functor 到 Monad

- 因为许多的形态都都可以被 map，所以抽象出了 Functor 这个 typeclass

Code

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

- 然后只要针对 Functor 撰写实例就行了

从 Functor, Applicative functor 到 Monad

- 因为许多的形态都都可以被 map，所以抽象出了 Functor 这个 typeclass

Code

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

- 然后只要针对 Functor 撰写实例就行了
- 比如 Maybe a, [a], IO a, 甚至是 (->)r a

从 Functor, Applicative functor 到 Monad

从 Functor, Applicative functor 到 Monad

- 接下来我们发现如果 a->b 也被包含在一个 Functor 里呢

从 Functor, Applicative functor 到 Monad

- 接下来我们发现如果 a->b 也被包含在一个 Functor 里呢
- 比如 Just (*3) 如何将它应用到 Just 5 上从而得到 Just 15

从 Functor, Applicative functor 到 Monad

- 接下来我们发现如果 a->b 也被包含在一个 Functor 里呢
- 比如 Just (*3) 如何将它应用到 Just 5 上从而得到 Just 15
- 于是我们得到了 Applicative functor:

从 Functor, Applicative functor 到 Monad

- 接下来我们发现如果 a->b 也被包含在一个 Functor 里呢
- 比如 Just (*3) 如何将它应用到 Just 5 上从而得到 Just 15
- 于是我们得到了 Applicative functor:

Code

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

从 Functor, Applicative functor 到 Monad

从 Functor, Applicative functor 到 Monad

- 现在问题来了：

从 Functor, Applicative functor 到 Monad

- 现在问题来了：
  - 如何将一个具有 context 的值 m a，丢进一个只接受普通值 a 的函数中

从 Functor, Applicative functor 到 Monad

- 现在问题来了：
  - 如何将一个具有 context 的值 m a，丢进一个只接受普通值 a 的函数中
- 换句话说如何套用一个形态为 a -> m b 的函数至 m a

从 Functor, Applicative functor 到 Monad

- 现在问题来了：
  - 如何将一个具有 context 的值 m a，丢进一个只接受普通值 a 的函数中
- 换句话说如何套用一个形态为 a -> m b 的函数至 m a
- 我们要求的函数为：

从 Functor, Applicative functor 到 Monad

- 现在问题来了：
  - 如何将一个具有 context 的值 m a，丢进一个只接受普通值 a 的函数中
- 换句话说如何套用一个形态为 a -> m b 的函数至 m a
- 我们要求的函数为：

Code

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Monad

Monad

- 我们先来看一下 Monad 的实例

Monad

- 我们先来看一下 Monad 的实例

Code

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

Monad

Monad

- return 是包装一个 monadic value

Monad

- return 是包装一个 monadic value
- $(>>=)$ 运算符是把 monadic value 传给一个接受普通值的函数 f，f 的返回值是一个 monadic value

Monad

- return 是包装一个 monadic value
- (>>=) 运算符是把 monadic value 传给一个接受普通值的函数 f，f 的返回值是一个 monadic value
- fail 用于处理错误的情况

Monad

- return 是包装一个 monadic value
- (>>=) 运算符是把 monadic value 传给一个接受普通值的函数 f，f 的返回值是一个 monadic value
- fail 用于处理错误的情况
- list，Maybe，IO action 都是一个 Monad

Maybe Monad

Maybe Monad

- Maybe 的例子:

Maybe Monad

- Maybe 的例子:

Code

```
Prelude> Just 9 >>= (\x->return(x+3))
Just 12
Prelude> Just 9 >>= (\x->return(x+3)) >>= (\x->Nothing)
Nothing
```

Maybe Monad

- Maybe 的例子:

Code

```
Prelude> Just 9 >>= (\x->return(x+3))
Just 12
Prelude> Just 9 >>= (\x->return(x+3)) >>= (\x->Nothing)
Nothing
```

- 可以很好的处理 Nothing 这种情况

list Monad

list Monad

- list 其实也是一个 Monad

list Monad

- list 其实也是一个 Monad

Code

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

list Monad

- list 其实也是一个 Monad

Code

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

- 例子：

list Monad

- list 其实也是一个 Monad

Code

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

- 例子：

Code

```
Prelude> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

list Monad

- list 其实也是一个 Monad

Code

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

- 例子：

Code

```
Prelude> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

- 我们从 monadic value 取出普通值给函数，$>>=$ 会帮我们处理好一切关于 context 的问题

do 语句

do 语句

- 可以使用 do 语句把一些 Monad 的操作绑定在一起

do 语句

- 可以使用 do 语句把一些 Monad 的操作绑定在一起

Code

```
f = do
    x <- Just 4
    y <- Just 5
    return (x, y)
```

do 语句

- 可以使用 do 语句把一些 Monad 的操作绑定在一起

Code

```
f = do
    x <- Just 4
    y <- Just 5
    return (x, y)
```

- 有没有想到 IO 操作

函数的副作用

函数的副作用

- haskell 里 monad 最重要的作用就是引入函数的副作用

函数的副作用

- haskell 里 monad 最重要的作用就是引入函数的副作用
- 比如 Maybe 类型的 Nothing 情况

函数的副作用

- haskell 里 monad 最重要的作用就是引入函数的副作用
- 比如 Maybe 类型的 Nothing 情况
- 比如 IO action 的输入输出操作

函数的副作用

- haskell 里 monad 最重要的作用就是引入函数的副作用
- 比如 Maybe 类型的 Nothing 情况
- 比如 IO action 的输入输出操作
- 比如 Error 类型的出错信息

Monad law

Monad law

- 三条 Monad 定律:

Monad law

- 三条 Monad 定律：
  - $(returnx) >>= f == fx$

Monad law

- 三条 Monad 定律:
  - $(return x) >>= f == f x$
  - $m >>= return == m$

Monad law

- 三条 Monad 定律:
  - $(returnx) >>= f == fx$
  - $m >>= return == m$
  - $(m >>= f) >>= g == m >>= (\backslash x- > fx >>= g)$

Monad law

- 三条 Monad 定律:
  - $(return x) >>= f == f x$
  - $m >>= return == m$
  - $(m >>= f) >>= g == m >>= (\backslash x -> f x >>= g)$
- 第一二条很显然

Monad law

Monad law

- 第三条很类似于函数的复合

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Code

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Code

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

- 于是 f.(g.h) 和 (f.g).h 显然等价的

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Code

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

- 于是 f.(g.h) 和 (f.g).h 显然等价的
- 仿照函数的复合我们可以合成 monadic function 的复合规则：

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Code

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

- 于是 f.(g.h) 和 (f.g).h 显然等价的
- 仿照函数的复合我们可以合成 monadic function 的复合规则：

Code

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

Monad law

- 第三条很类似于函数的复合
- 让我们回顾一下函数的复合

Code

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

- 于是 f.(g.h) 和 (f.g).h 显然等价的
- 仿照函数的复合我们可以合成 monadic function 的复合规则:

Code

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

- 到此我们就可以像操纵普通值和普通函数一样操纵 monadic value 和 monadic function

End.

End.

- 更加高级的东西：

End.

- 更加高级的东西:
  - Arrows.

End.

- 更加高级的东西：
    - Arrows.
    - Stream Fusion

End.

- 更加高级的东西：
  - Arrows.
  - Stream Fusion
  - Zipper

End.

- 更加高级的东西：
  - Arrows.
  - Stream Fusion
  - Zipper
  - .....

End.

- 更加高级的东西：
  - Arrows.
  - Stream Fusion
  - Zipper
  - .....
- 欢迎大家自行阅读相关资料

End.

- 更加高级的东西：
  - Arrows.
  - Stream Fusion
  - Zipper
  - .....
- 欢迎大家自行阅读相关资料
- 推荐作业：使用 haskell 实现一个平衡树

End.

End.

- 欢迎提问！