HASKELL中的惰性求值(LAZY EVALUATION)

本文主要是对Haskell中的惰性求值(Lazy Evaluation)的一点个人的理解，在编写程序中可以一定程度上节省计算机程序的空间占用。

Update: 2018.03.30

这里写的比较早，可能会有一些缺漏，更新的总结请参考hs-tutorial

Introduction

函数式编程打交道比较多的就是函数(Function)了，而函数的参数调用也是一个比较有意思的问题，惰性求值(Lazy evaluation)是Haskell的一个特性，本文主要写一写自己的看法，希望对大家有一些启发

Haskell中，所有运算几乎都可以看作是给定表达式给定参数来求值。所以Haskell解释器对于某一个语句所做的便是表达式求值(Expression evaluation)了。有时候我们告诉解释器的参数是一些复合表达式，拿一个给给定参数增加一的函数来说:

```
increment :: Int -> Int
increment n = n + 1
```

如果我们计算`increment (2*5)`，Haskell会如何计算？一般的我们有两种求法

```
increment (2*5)
increment (10)
then 10 + 1
then get 11
```

以及另外一种求法

```
increment (2*5)
then (2*5) + 1
then 10 + 1
then get 11
```

Haskell的一个很重的特性就是不管我们的求值顺序如何，我们最终得到的结果都是一致的。

Evaluation Strategies

这里有一个概念叫做 `reducible expression` 或者也可以简写为 `redex`，我个人的理解这个reduce并不代表一定要去把函数调用的表达式逻辑的精简而是某种意义上的化简，把它解释成计算机能够理解的语言。比如说，某一个乘积运算

```
mult :: (Int, Int) -> Int
mult (x, y) = x * y
```

类似的，我们在计算 `mult (1+2) (2+3)` 时，这个表达式就包含着3个 `redex`，分别是 `1+2`， `2+3` 和 `mult (x, y)`，我们根据之前的想法，可以知道这个表达式有三个求值步骤，分别是

```
mult (3, 2+3)
mult (1+2, 5)
(1+2) * (2+3)
```

当对这个表达式进行求值时，就有了两种想法，一种叫做 `innermost evaluation`

```
mult (1+2, 2+3)
mult (3, 2+3)
mult (3, 5)
then 3 * 5
then get 15
```

另外一种就叫做 `outermost evaluation`

```
mult (1+2, 2+3)
then (1+2) * (2+3)
then 3 * (2+3)
then 3 * 5
then get 15
```

乍一看可能看不出区别，但是仔细想想这两种求值策略(Evaluation strategy)的不同。第一种的 `innermost evaluation` 中，在函数调用的时候，所有的参数的值都是必须确定的，因为在计算的时候，最后一个调用的过程是 `mult`，我们必须保证在函数调用时，数对(number pair)必须是两个整形(Int)组成的数对(number pair)，我们称这种情况下，参数的调用是 `arguments passed by value`。第二种过程中，我们在调用 `mult` 函数时，是可以暂时不必考虑我们的参数的值的，我们在调用 `mult` 函数时，是不必检查我们的类型的，这种情况下，类型检查发生在函数调用后，我们称这种情况下，参数的调用是 `arguments passed by name`。但是在Haskell中也有很多内置的函数即使在使用 `outermost` 时，也一定要类型检查，比如说 `+`， `-`， `*`， `\`，称为 `strict` 模式。

Lambda Expressions

函数编程语言中，有一个比较有趣的概念是 `curried function`，这一想法来自于 `lambda calculus`，这里贴一个Wiki的链接curring，lambda calculus。对于 `lambda calculus` 应该足够另外开一篇文章了，这里仅仅对 `lambda expression` 进行讨论。

举个栗子，定义函数 $f(x,y) = x*x + y*y$

```
f :: Num a => a -> a -> a
f x y = x*x + y*y
```

但是仔细想想，一个函数好像并不一定要有一个名字，所以我们可以构建一个没有名字的映射 $(x,y) \mapsto x*x + y*y$ 这样的计算也是可以被接受的 $((x,y) \mapsto x*x + y*y)(1,2) = 1*1 + 2*2 = 5$

```
f = \x y -> x*x + y*y
```

甚至再大胆一些，多参数函数与一个参数的函数好像也不需要区分了，这便是 `curried function` 了。我们可以让多个参数的函数变成这样 $x \mapsto (y \mapsto x*x + y*y)$ 这个函数只有一个参数x，并把x映射成接受一个参数y的高阶函数(high order function)。这样的转换也叫做柯里化(curring)。

先前的 `mult` 函数也可以这样定义了

```
mult :: Int -> Int -> Int
mult x = \y -> x*y
```

现在再来看 `mult (1+2) (2+3)` 的行为

```
mult (1+2) (2+3)
mult 3 (2+3)
(\y -> 3*y) (2+3)
(\y -> 3*y) (5)
then get 15
```

这里Haskell的惰性运算(Lazy evaluation)就体现了出来，它只在需要的时候去检查，运算参数。这也就是为什么在Haskell中存在无限长的列表(List)的原因之一。

Strict application

Haskell中默认的是惰性运算(Lazy evaluation)，但是也提供了一种方式($!)，强迫函数在调用前检查它的参数，拿 `foldl` 来说。一种 `foldl` 可以被定义为

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl step zero (x:xs) = foldl step (step zero x) xs
foldl _ zero []        = zero
```

我们利用它来计算一个列表的和

```
foldl (+) 0 [1, 2, 3]
foldl (+) 0 (1:2:3:[])
foldl (+) (0+1) (2:3:[])
foldl (+) ((0+1)+2) (3:[])
foldl (+) (((0+1)+2)+3) []
then get 6
```

利用 `strict application` 我们可以定义新的 `foldl` 函数

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' step zero (x:xs) = ((foldl step) $! (step zero x)) xs
foldl' _ zero           = zero
```

同样利用它来计算列表 `[1..3]` 的和

```
foldl' (+) 0 [1, 2, 3]
foldl' (+) 0 (1:2:3:[])
foldl' (+) (0+1) (2:3:[])
foldl' (+) 1 (2:3:[])
foldl' (+) (1+2) (3:[])
foldl' (+) 3 (3:[])
foldl' (+) (3+3) []
foldl' (+) 6 []
then get 6
```

所以，在恰当时候使用 `strict application` 对减少软件资源占用有很大的帮助，这里主要是对 `Lazy evaluation` 的自己的一些看法，结合一些简单的栗子，对代码性能的一点分析。Learn you a Haskell for great good!

Reference

(1) Programming in Haskell 2nd Edition (Graham Hutton)

(2) 让我们来谈谈Lambda演算 王盛颐