函数式 dfs（深度优先搜索）

2017, May 26 by Tesla Ice Zhang

这篇博客主要说说我很早以前整出来的函数式 dfs，老早就想写博客了但是现在才有空。本文应该是很简单的，因为我会先说一些预备知识，说不定你看完还能懂点 Haskell，多好。

消除恐惧

你需要消除你对那些概念的恐惧，而且就我个人来说，我觉得理解 `Monad` 是自函子范畴上的幺半群 和 在实际编程中使用 `>>=` 是没有关系的。

所以我觉得，就算你只学过 Java，也是可以看这篇文章的（不过你还需要消除对于 新语言 的恐惧）。

为什么我要说这个呢？因为我最早也是很恐惧 Haskell 这类语言的，但后来发现这些东西还是蛮好玩的。

虽然和编译器硬肛不是什么好受的事情，不过以前的 debug 经历也很不好受啊，而使用 Haskell 这种

> It compiles, it probably works

的语言，可能对你来说是一个好的新的尝试也说不定（？）。

~~我感觉我正在被一群 Rust 猿盯着~~

预备知识

符号约定

定义符号

```
(a -> b)
```

为类似这样的东西：参数类型为 a 返回类型为 b 的函数。

定义符号

```
(a -> b -> c)
```

为类似这样的东西：参数 2 个，类型为 a 和 b ，返回类型为 c 的函数。（其实这是柯理化过的，但是~~我懒得~~本文不会讲）

定义符号

```
fun :: a -> b
```

为类似这样的东西：函数 fun 接收一个 a 类型的参数，返回一个 b 类型的值。

定义符号

```
fun :: Functor f => f a -> f b
```

为类似这样的东西：函数 fun 接收一个 f a 类型的参数，返回一个 f b 类型的值。其中， f 是一个 Functor 。

我保证

你不会看到关于任何 如何定义一个 Functor 如何定义一个 Monad 为什么 List 是一个 Functor 之类的东西。

Functor

这里可以理解为 *可以进行 map 操作的东西*。在我的印象中，Java 等语言中一般都只有 List 等东西才有 map ， 而这里的 map 是广义的，即:

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

这个是什么意思呢？举个例子，比如对于 List ，它是一个 Functor ，然后我们可以通过 map 把一个 List<a> 变成一个 List<b> —— 不过你还需要一个函数 a -> b 。
也就是说，我们有:

```
map :: (a -> b) -> List a -> List b
```

这样看是不是好理解多了？我们接收一个 (a -> b) 的函数和一个 List<a> ，返回一个 List<b> 。
那么同理，把 List 抽象为一个 Functor 就可以得到最开始我给出的定义了:

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

我们再来想想 Kotlin 的 Nullable 类型（看懂下文不需要学 Kotlin 因为下一句就解释了这是什么意思）。 一个 Nullable 有两种可能的值，也就是 null 和 正常的值。
那么，我们可以有这样一个操作:

```
map :: (a -> b) -> Nullable a -> Nullable b
```

对一个 Nullable 进行 map ，如果它是正常值，就返回这个值传给第一个参数（那个函数）后的返回的结果。
如果它是 null ，那就继续返回 null 。
因此， Nullable 和 List 都是 Functor。
在 Haskell 中，我们可以把它简写为 <$> 。也就是

```
map toInt [1.1, 2.1, 3.1]
-- 等价于
toInt <$> [1.1, 2.1, 3.1]
```

当然这里的等价是忽略了运算符优先级的。可能两者在换来换去的时候需要加括号。
顺带一提， Nullable 在 Haskell 里面的等价物是 Maybe ，有值的叫 Just ， null 对应的是 Nothing 。

Alternative

这是另一个奇特的东西，它理解起来更简单。举个例子，我现在有五个 Nullable 。我要对他们挨个遍历找出第一个非 null 的。
那么 Nullable 作为一个 Alternative 就可以这样操作（这是很早以前就有的操作了）

```
a = getNullable
b = getNullable
c = getNullable
d = getNullable
```

```
e = getNullable

firstNotNull = a <|> b <|> c <|> d <|> e
```

看是不是很清晰 QAQ
最后一行这个表达式相当于是把一堆 `Alternative` 穿起来了，它最后的值也是 `Alternative` （因为如果全部都是 `null` 那最后也只能是 `null` ）。
要使用 `Alternative` ，请加上这个：

```
import Control.Applicative
```

dfs

那么，我们可以根据 `dfs` 的思路来把这两个函数式编程的概念融入这个算法中。
如果你不知道什么是 `dfs` 那就太惨啦。这里说的是一个很狭义的 `dfs` （就是走迷宫，初等 OI 内容，我校初中生每个都会做）， C++ 版本可以参考这些题目以及相关参考题解（因为这些代码都是我很早很早很早的时候写的了）：

- openjudge 1818 以及 我的题解 （不过这个似乎是 bfs 解）
- openjudge 1792 以及 我的题解 （不过这个似乎是 bfs 解）

找了半天没找到 dfs 的。。。呃。。。我上个专门讲的：

- 我好歹还是找了一会

题目

这还是一道 <4kyu> 的题目，先上原题：

- Escape The Mines

语言支持 `Haskell JavaScript Python Ruby` 。

我们选 Haskell。

可以看到数据模型：

```
type XY = (Int, Int)
data Move = U | D | R | L deriving (Eq, Show)
```

关于题面，英文的就自己看了，这是我稍微改了下的谷歌翻译版本：

> 一个可怜的矿工被困在一个矿井里，你必须帮助他离开！
>
> 只不过这个矿是黑暗的，所以你必须告诉他去哪里。
>
> 在这个 kata 中，您将必须实现一个解决方法（map，miner，exit），例如：['up', 'down' ， 'right', 'left']。有4个可能的动作，向
>
> map是布尔值的二维数组，表示正方形。False 代表墙壁，True 代表可以
> 它被列为一列列。所有列将始终是相同的大小，尽管不一定与行大小相同（
> 地图永远不会包含任何环，所以总是只有一个可能的路径。地图可能包含死
>
> miner 是起始矿工的位置，作为由两个自然数值 x 和 y 组成的对象。例

`exit` 是出口的位置，与矿工的格式相同。

请注意，矿工不能走出地图，因为它是隧道。

举例：

下面是例子：

```
let map = [[True, False],
           [True, True]]

solve map (0,0) (1,1)
-- Should return [R, D]
```

思路

首先， dfs 到某一个点时的决策，有以下几个因素。

1. 这个点是否是目标点（如果是，就表示成功了）
2. 这个点是否已经被走过（判重）
3. 这个点能不能走（有些地方是不能走的，所以才叫迷宫啊）
4. 这个点是否超出迷宫范围（不能走出去了）

上面是几个特殊情况（会导致 `dfs` 递归函数直接返回），剩下的时候，对周围四个点分别进行递归调用。
由于这道题要求我们记录路径，因此还得把路径存在参数里面。
总体来说，考虑到：

1. 起点终点 +2
2. 地图 +1
3. 走过的点 +1
4. 目前的路线 +1

总共需要五个参数。
那么 dfs 函数原型就出来了：

```
dfs :: [[Bool]] -> [XY] -> XY -> XY -> Maybe [Move]
dfs m v s e = undefined
```

这里体现出了刚才的思想——返回的是一个 Maybe ，因此我们可以对它进行 <$> 和 <|> 。
然后实现那些返回的条件以及搜索的过程就好了。举个例子，判断目标：

```
dfs m v s e
  |s == e = Just []
```

判重：

```
dfs m v s e
  |elem s v = Nothing
```

其中， v 是判重用的序列。
剩下的请读者自己思考哦。
以及最后的对四个边的遍历，我这里就写两条边吧：

```
dfs m v s e
  |otherwise = ((R :) <$> dfs m (s : v) (x + 1, y) e)
          <|> ((D :) <$> dfs m (s : v) (x, y + 1) e)
```

看到了吗，这里就体现出了对 Functor 和 Alternative 的使用。

首先，把当前行走的方向加进『路径』中，然后递归，看是否走得通。要是走通了，就把剩下的路径加上这一步的路径。

等成功后回溯到开头，就可以把这整条路径从末尾给『收』起来（回溯）。

部分概念在 Kotlin 中的等价物

Haskell	举例	Kotlin	举例
<$>	func <$> value	?.run	value?.run(func)
<|>	val1 <|> val2	?:	val1 ?: val2

所以说 Kotlin 还是很函数式的（逃

最终的代码

为了防止你们直接复制过去提交破坏游戏规则，我不直接给（傲娇

这里贴个链接，根据我的推断，应该是必须提交并通过该题才能看到这个链接的内容的：

- 我的题解

各位大佬可以过来给个 best practise 或者 clever 啥的（逃

毕竟比题解区那些辣鸡超长的代码不知道高到哪里去了。

作业

如果读者想检验自己是否真的看懂了本文，试试这道题，支持 `Haskell` `JavaScript` 。

- Determining if a graph has a solution

反正都是 <4kyu> 的大水题（逃

真的作业

如果读者觉得看完本文之后的收货不够多，就看看这道题吧。

它是完全为我在上面说的题解量身定制的题目，不过需要稍作改动。

更新

发知乎啦

Tweet this

Top

创建一个 issue 以申请评论

Create an issue to apply for commentary

协议/License