为什么函数程序设计如此重要

John Hughes

Department of Computing Science
Chalmers University of Technology
412 96 Goteborg Sweden
rjmh@cs.chalmers.se

Abstract

随着软件变得越来越复杂，使软件具有良好的结构也显得越来越重要。结构良好的软件易编写、易调试，并且提供了一组可以重用的模块，从而降低未来的编程费用。传统的语言在将问题如何模块化方面有概念上的局限性。函数程序设计语言越过了这些局限性。在这篇论文中，我们将提出函数程序设计的两个非常有助于模块化的特性：高阶函数和惰性计算。我们将以表和树、数值算法、alpha-beta启发算法（人工智能中用于博弈的算法）作为例子。因为模块化是程序设计成功的关键，函数程序设计语言对现实世界是至关重要的。

Contents

1 简介 3

2 与结构程序设计的对比 4

3 将函数粘合在一起 4

1 简介

本论文试图向"现实世界"证明函数程序设计是至关重要的，并且帮助函数程序设计员充分探讨函数程序设计的优点。

函数程序设计意指每个程序均为一个函数。主程序本身也是一个函数，它将程序的输入作为函数的自变量，程序的输出作为函数的结果。一般地，主程序是由其他函数定义的，而这些函数也是由另外一些函数来定义的，直到最底层由程序设计语言提供的原始函数。所有这些函数都非常类似于数学上的函数，在本文中函数的定义将使用普通的等式。我们将沿用 Turner 设计的语言 Miranda(TM)[Tur85] 的表示法，但是，即使没有函数程序语言背景的读者也是容易理解的（Mirande是 Resecrch Softwre Ltd. 的商标）。

函数程序设计的特点和优点通常可以大致概括如下。函数程序不包含赋值语句，所以，一个变量一旦给定一个值，其值将永远不变。更一般地说，函数程序绝对没有副作用。一个函数的调用只计算其值，没有任何其他作用。这一特点可以排除程序错误的一个主要渊源，而且使得程序的执行顺序无关紧要–因为没有副作用可以改变一个表达式的值，表达式可以在任何时候被计算。因此，程序员不需要担忧控制流的编写。因为表达式可以在任何时候被计算，我们可以自由地将变量用其值代替，或者将值代换为变量，也就是说，程序是"引用透明的"（referential transparent）。这种自由度使得函数程序比传统的程序在数学上更容易处理。

但是，如果函数程序设计外行并不认为这些是优点，我们也不应该吃惊。这些优点讲了许多函数程序设计不是什么（没有赋值语句，没有副作用，没有控制流），但没有说它是什么。对于那些更喜欢物质利益的人们，这些"优点"能非常令人信服。

函数程序设计宣称其具有极大的物质利益–函数程序设计员的生产效率比一般程序员高一个数量级，因为函数程序比普通程序短一个数量级。为什么是这样呢？唯一似乎可信的理由是普通程序的 90% 由赋值语句构成，而这些赋值语句在函数程序中可以全部省去。这显然是荒唐的。如果省略赋值语句能带来这么大的利益，FORTRAN程序员不会编写了20年的程序。通过省略某些特征而使得一个语言表达力更强，无论这些特征如何糟，这在逻辑上是不可能的。

即使一个函数程序设计员也会对这些所谓的优点不满，因为这些对于挖掘函数程序语言的能力毫无帮助。我们不可能编写一个特别缺乏赋值语句，或者特别引用透明的程序。如果没有程序质量的尺度，我们也便没有奋斗的目标。

显然，这样刻画函数程序设计是不恰当的。我们必须既能解释函数程序设计的表达力，又能给函数程序设计员清楚地指出应该追求的目标。

2 与结构程序设计的对比

对比函数程序设计与结构程序设计是很有帮助的。在过去，结构程序设计的特征与优点大致概括如下：结构化程序不包含goto语句。结构程序中的程序块没有多入口和多出口。结构程序在数学上比非结构程序更容易处理。结构程序设计的这些优点非常类似于我们前面讨论的函数程序的优点。这些优点基本上是否定语句，已经陷入了有关"基本goto"的无意义讨论之中。

作为事后诸葛亮，显然，结构程序设计的这些性质并没有点到问题的核心。结构程序与非结构程序最重要的区别在于结构程序是由模块化的方式设计的。模块化设计带来了生产率的极大提高。首先，小模块容易快速编写。其次，通用模块可以被重用，因此加快了随后的开发过程。第三，模块可以被单独测试，减少了调试的时间。

Goto语句的不存在及其效应与模块化没有多少关系，它仅对"小规模程序设计"有帮助，而模块化有助于"大规模程序设计"。因此，尽管程序员需要做更多的工作，程序员仍然可以享受使用 FORTRAN 或者汇编语言进行结构化程序设计带来的好处。

现在人们普遍认为模块化设计是程序设计成功的关键，程序设计语言如 Modula-II[Wir82]，Ada[oD80] 和标准 ML 都包含了特别用于改善模块化设计的特征。然而，有一点经常被忽视了。当我们编写一个模块化程序以解决一个问题时，通常将一个问题分解为一些子问题，然后解决这些子问题，并将这些解结合起来。分解问题的方法直接依赖于将这些解粘合在一起的方法。因此，欲在概念上增强模块化问题的能力，我们必须在程序设计语言中提供新的粘合方法。复杂的作用域规则和分别编译只对书写细节有帮助，而并没有提供分解问题的新概念工具。

让我们返回函数程序设计。我们将在下面论证函数程序设计提供了两个新的、非常重要的粘合方法。我们将给出许多可以用新方法模块化的，而且大大简化的程序例子。这是函数程序设计的能力的关键——它能极大地改进模块化。这也是函数程序设计员应该争取的目标——设计简短和更通用的模块，然后用我们即将描述的新方法将其粘合在一起。

3 将函数粘合在一起

第一种粘合方法可以将简单的函数粘合在一起形成更复杂的函数。我们将用一个简单的表处理问题——把表中的元素加在一起--来解释之。我们定义表如下：

```
listof  X :: = nil | cons X (listof  X)
```

其意义为：一个 X（X 是任意的）上的表或者是 nil，即一个不含任何元素的表；或者是 X 的一个元素及 X 上的一个表的 cons。一个 cons 表示其第一个元

素为 X，其第二个及其后的元素为 X 上的表的元素。X 在这里表示任意类型，例如，如果 X 是"整数"，那么，以上的定义表示一个整数表或者是空，或者是一个整数和另一个整数表的 cons。我们将按照惯例，将一个表的元素列在一对方括弧内表示之，而不显式地使用 nil 和 cons。这只是为了表示的方便。例如:

```
[]      ⇒ nil
[1]     ⇒ cons 1 nil
[1,2,3] ⇒ cons 1 (cons 2 (cons 3 nil))
```

我们可以用一个递归函数 sum 将一个表的元素加在一起。sum 需要定义在两种参数上：一个空表(nil)，以及一个 cons。因为空表的和为 0，我们定义:

```
sum nil = 0
```

因为一个cons的和是其第一个元素与另一个表之和相加。故有:

```
sum (cons num list) = num + sum list
```

检查以上定义，我们发现在计算和时只有带筐的部分是特定的:

```
          +---+
sum nil = | 0 |
          +---+

                            +---+
sum (cons num list) = num   | + | sum list
                            +---+
```

这表示 sum 的计算可以通过把一个一般递归模式与带框部分粘合起来模块化。这种递归模式通常称为归约（reduce），因此 sum 可以表示为:

```
sum = reduce add 0
```

为方便起见，我们给 reduce 传递了一个二元函数 add 而非一个运算符号。add 的定义如下:

```
add x y = x + y
```

函数 reduce 的定义可以将 sum 参数化， 由下列方程式导出:

```
(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)
```

这里我们使用了括号 (reduece f x) 以表示其代表 sum。通常括号可以省略，所以 ((reduce f x) l) 可写作 (reduce f x l)。像 reduce 这样的三元函数只作用于两个参数时可视为一元函数。一般地，一个n元函数作用于 m(<n) 个参数可视为 n−m 元函数。今后我们将沿用此习惯。

将 sum 如此模块化后，我们可以重用一些函数。其中最有趣的部分是 reduce，可用于编写求表中元素积的函数:

```
product = reduce multiply 1
```

还可用于测试一个布尔表中是否存在元素为 true:

```
anytrue = redeuce or false
```

或者一个布尔表的所有元素全为 true:

```
alltrue = redeuce and true
```

一种理解 (reduce f a) 的方法是将表中的所有 cons 用 f 代替，所有的 nil 用 a 代替。以 [1,2,3] 为例，因为它表示:

```
cons 1 (cons 2 (cons 3 nil))
```

所以 (reduece add 0) 将上表转换为:

```
add 1 (add 2 (add 3 0)) = 6
```

同样，(reduce multiply 1) 将上表转换为:

```
multiply 1 (multiply 2 (multiply 3 1)) = 6
```

现在我们可以明显地看出 (reduce cons nil) 恰恰复制一个表。因为两个表的连接可以通过将第一个表的元素 cons 到第二个表上获得，所以:

```
append a b = reduce cons b a
```

例如,

```
append [1,2] [3,4] = redeuce cons [3,4] [1,2]
                   = (reduece cons [3,4]) (cons 1 (cons 2 nil))
                   = cons 1 (cons 2 [3,4])
(□cons □□cons□[3□4] □□nil)
                   = [1,2,3,4]
```

将表中所有元素加倍的函数可以定义为:

```
doubleall = reduce doubleandcons nil
```

其中 doubleandcons num list = cons (2*num) list

函数 doubleandcons 可以进一步模块化，第一步为:

```
doubleandcons = fandcons double
fandcons f el list = cons (f el) list
```

其中 double n = 2*n

然后,

```
fandcons f = cons . f
```

这里"."（标准函数合成运算符）定义为:

```
(f . g) h = f (g h)
```

将fandcons作用于一些参数，我们可以看出fandcons的定义是正确的:

```
fandcons f el = (cons . f) el
              = cons (f el)
```

所以 fandcons f el list = cons (f el) list

最后的结果是:

```
doubleall = reduce (cons . double) nil
```

进一步模块化， 我们有:

```
doubleall = map double
map f = reduce (cons . f) nil
```

其中 map 将函数 f 应用于表的所有元素。map是又一个常用函数。

我们还可以把距阵的所有元素相加，其中矩阵表示为一个由表构成的表。其定义为:

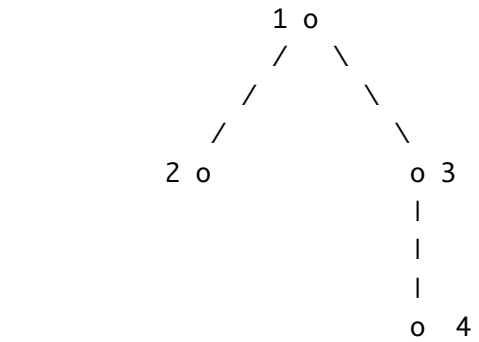
```
summatrix = sum . map sum
```

在这里，map sum 计算每一行的和， 而最左边的 sum 将这些和相加，从而得到整个距阵之和。

这些例子应该足以使读者相信，模块化带来的益处是极大的。通过把一个简单函数 sum 模块化为一个"高阶函数"和一些简单参数的组合，我们得到一个可以用之轻易地编写许多表上函数的构件（reduce）。我们不应该停留在表上的函数。作为另一个例子，考虑如下定义的数据类型有序标号树:

```
treeof X ::= node X (listof (treeof X))
```

此定义说，X 上的树是一个标有 X 上元素的节点和一列子树，每个子树都是 X 上的树。例如，下图所示的树:



可以表示为:

```
node 1
   (cons (node 2 nil)
      (cons (node 3
            (cons (node 4 nil) nil))
         nil))
```

我们将直接定义一个类似于 reduce 的函数 redtree，而非由一个例子抽象出来。回想 reduce 带有两个参数，一个用以代替 cons，一个用以代替 nil。因为树是由 node，cons 和 nil 构成的，所以 redtree 必须带三个参数，以代替以上三个部分。因为树和表具有不同的类型，我们需要在树和表上各定义一个函数。为此，我们定义:

```
redtree f g a (node label subtrees) =
      f label (redtree' f g a subtrees)
redtree' f g a (cons subtree rest) =
      g (redtree f g a subtree) (redtree' f g a rest)
redtree' f g a nil = a
```

通过粘合 redtree 和其他函数，我们可以定义许多有趣的函数。例如，对于一个标签为数的树，所有标签之和可由下列函数求得:

```
sumtree = redtree add add 0
```

以先前的树为例，应用 sumtree 得到:

```
add 1
   (add (add 2 0)
      (add (add 3
            (add (add 4 0) 0))
         0))
= 10
```

一棵树上所有标签构成的表可由下式求得:

```
labels = redtree cons append nil
```

将其应用于先前的树得到如下的表:

```
cons 1
   (append (cons 2 nil)
      (append (cons 3
            (append (cons 4 nil) nil))
            nil))
= [1,2,3,4]
```

最后，我们可以定义一个类似于 map 的函数，它将一个函数作用于树上的所有标签:

```
maptree f = redtree (node . f) cons nil
```

所有这些结果的取得是因为那些在常规语言中不可拆分的函数在函数程序语言中可以表示为一些部分的组合: 一个通用的高阶函数和一些特殊函数。这些高阶函数一旦定义好，许多运算的定义变得非常容易。对于每一个数据类型，应该定义相应的高阶函数处理之。这样不仅使得数据结构的处理变得容易，而且也使得数据类型表示的细节知识局部化。与常规语言的最好类比是语言的可扩充性——函数程序设计语言好像在任何需要的时候都可以扩充以新的控制结构。