Haskell 语言的高阶特性及其应用*)

庞建民[1,3] 赵荣彩[1] 王怀民[2]

(信息工程大学信息工程学院 郑州450002)[1] (国防科技大学计算机学院 长沙410073)[2]

(英国 DURHAM 大学 Durham,UK,DH1 3LE)[3]

摘 要 Haskell 语言的高阶特性使笔者在开发软件时受益匪浅,但很遗憾,目前国内同行应用这一语言的人非常少。本文介绍 Haskell 语言的高阶特性,并通过几个与树相关的例子,阐述如何利用 Haskell 语言的高阶特性来编写功能强大但却简短漂亮的程序。

关键词 函数式语言,高阶函数,Haskell,多态,惰性计算

The Higher-order Features of Haskell and their Applications

PANG Jian-Min[1,3] ZHAO Rong-Cai[1] WANG Huai-Min[2]

(Institute of Information Engineering, University of Information Engineering, Zhengzhou450002)[1]

(School of Computer Science, National University of Defense Technology, Changsha410073)[2]

(University of Durham, Durham, DH1 3LE, UK)[3]

Abstract We get a lot of benefits from the higher-order features of Haskell when we develop software. But it is a pity that there are just few colleagues who know this language in China. In this paper we introduce the higher-order features of Haskell and explain how to design powerful but concise programs by using these features through several examples about trees.

Keywords Functional language, Higher-order function, Haskell, Polymorphism, Lazy computation

1 Haskell 语言简介

Haskell 是由国际 Haskell 委员会定义的"多态类型纯函数式惰性语言",其第一版 Haskell1.0诞生于1990年。目前发行的是 Haskell98,它包含了函数式语言发展至今的几乎所有功能。像大多数函数式语言一样,Haskell 是一个强类型语言,这意味着对于每个表达式,我们都能在编译时检查它们是否符合语言的类型规则,Haskell 既支持由基本类型构成的单态类型,如 Char 和 Int,也支持像记录和联合这样的用户自定义的结构类型,并且它还支持多态类型。Haskell 语言是一种基于 λ-演算模型的语言,它是无副作用的,并且支持惰性计算(Lazy computation)。目前国外已经有越来越多的程序设计爱好者使用该语言,用该语言实现的软件系统也越来越多,越来越大。

2 Haskell 的高阶特性

一个把函数作为自己参数的函数或者把函数作为结果传递的函数称为高阶函数。虽然在其它函数式语言中也有对高阶函数的支持,但像 Haskell 这样强大方便的支持并不多。在 Haskell 中,函数就像普通数值一样,可以作为参数传递,作为结果返回以及保持在数据结构中。通常的强制式语言几乎都不支持高阶函数,函数可以作为参数传递,但是在运行的时候创建新的函数是不可能的。

在数学中,我们已经熟悉了高阶函数的概念,例如,函数复合操作就可以被看作是一个高阶函数,因为它把两个函数进行复合从而得到一个新的函数。又如,微分和积分都是对函数的运算,因此也是高阶函数。事实上,在数学中也存在相应的计算模型,它可以让我们自由地利用高阶函数,构造出功能非常强大而且描述简洁的新函数,例如 λ-演算就是一种这样的模型。直接使用这种模型提供的功能来编写程序,是几代程序设计工作者的梦想。虽然 Lisp 也是基于 λ-演算模型的,但和数学上的使用方法有一定的区别,但 Haskell 的出现,使得直接使用 λ-演算的这种梦想成为了现实,但在梦想成真之际,由于缺乏必要的宣传,目前还有很多程序设计工作者不知道这一成果。为此,我们希望通过下面的一些应用实例,让读者了解 Haskell 语言的强大功能,从而在实践中加以应用,笔者在开发软件过程中利用这些特性已经受益匪浅。

3 高阶特性的应用

高阶函数是 λ-演算理论的精髓,是由 Lisp 首先介绍到程序语言这个世界的,在 Haskell 语言中,高阶函数的应用提升到了可与数学上的抽象级别相当的地步。也就是让函数对象成为程序语言当中所谓的"第一等公民"(first-class citizen)。我们所说程序语言当中的"第一等公民",指的是可以对这样的数据对象进行像整型那样的不受限制的操作。因此我们可以像对待整数和浮点型数据那样,把函数本身也作为某个函数的输入数据和输出数据。也就是说,我们的函数可以对函数本身进行操作。我们通过一个小例子来看看它的高阶应用。我们首先递归定义一个多态的二叉树类型:

```
data Tree a = Leaf{ value::a }
            |Node{left::Tree a, right::Tree a}
```

*)本课题得到欧盟项目 TYPE 资助(项目编号 types project 29001)。庞建民 副教授,英国 Durham 大学博士生,主要研究方向:计算机辅助推理与软件理论。赵荣彩 教授,博士生导师,主要研究方向:分布式计算与软件理论。王怀民 教授,博士生导师,主要研究方向:分布式计算与软件理论。

这里，a 称为类型变元，它可以被实例化为任意类型。

比如，我们要给一棵整型二叉树上的每个叶节点的值加1，可以定义如下的函数 *add1*：

```
add1:: Tree Int→Tree Int
add1(Leaf x)= Leaf(x+1)
add1(Node l r)=Node(add1 l) (add1 r)
```

又比如，我们要给一棵整型二叉树上的每个叶节点的值翻倍，可以定义如下的函数 *times2*。

```
times2::Tree Int→Tree Int
times2(Leaf x)=Leaf (2 * x)
times2(Node l r)=Node (times2 l) (times2 r)
```

仔细分析上面的两个例子，不难看出，我们可以利用高阶函数把上面的两个例子推广到一个任意的对叶子节点的运算。这就是我们下面要定义的高阶函数 *mapTree* 的功能。高阶函数的一个重要的应用就是捕捉公共模板，提供更强的通用性，从而达到更高级的软件重用。

3.1 高阶函数 mapTree

下面定义的 *mapTree* 函数把作为参数的函数 f 应用到一棵作为参数的二叉树的所有叶节点上，结果还是一棵二叉树。

```
mapTree ::(a→b)→Tree a→Tree b
mapTree f(Leaf x)=Leaf(f x)
mapTree f( Node l r)=Node (mapTree f l)(mapTree f r)
```

有了高阶函数 *mapTree*，前面介绍的函数 *add1* 和 *times2* 就可以通过它来构造：

```
add1=mapTree (\x→x+1)
times2=mapTree (\x→2 * x)
```

这里 $\backslash x \rightarrow x+1$ 和 $\backslash x \rightarrow 2 * x$ 是 *Haskell* 中的无名函数，它们分别对应 λ-演算中的函数 $\lambda x.\, x+1$ 和 $\lambda x.\, 2 * x$。可见，通过这个高阶函数，我们获得了在普通数学中都难以应用的抽象，因为在数学中我们对可重用性的使用不如计算机科学中用得多。

事实上，我们可以把 *mapTree* 看作如下类型：

```
mapTree::(a→b)→(Tree a→Tree b)
```

这看上去像是将一种 a 到 b 的函数变换为 *Tree a* 到 *Tree b* 的函数。

3.2 高阶函数 foldTree

下面考虑求树中有多少个叶节点。我们用多态函数 leafsize 来计算这个值：

```
leafsize ::Tree a→Int
leafsize (Leaf_) =1
leafsize (Node l r)= leafsize l+leafsize r
```

不难看出，我们可以用 *leafsize* 求出任意类型的二叉树的叶节点个数，并且定义非常简单。

我们再考虑求树中有多少个分枝节点。我们用多态函数 *branchsize* 来计算这个值：

```
branchsize::Tree a→Int
branchsize(Leaf_)=0
branchsize(Node l r)=1+branchsize l+branchsize r
```

不难看出，我们可以用 *branchsize* 求出任意类型的二叉树的分枝节点个数。

仔细分析上述两个函数，我们发现，它们存在着一些共性，那就是，对叶节点有一种一元函数的处理方，对分枝节点有另外一种二元函数的处理方法。我们可用如下的高阶函数 *foldTree* 来刻画上述抽象：

```
foldTree ::(a→b)
          →( b→b→b)
          →Tree a
          →b
foldTree lf nf (Leaf x)=lf x
foldTree lf nf (Node l r)=nf(foldTree lf nf l) (foldTree lf nf r)
```

有了高阶函数 *foldTree*，上面的函数 *leafsize* 和 *branchsize* 就可分别定义为：

```
leafsize=foldTree(\_→1)(\x y→x+y)
branchsize =foldTree (\_→0)(\x y →1+x+y)
```

由此可见，有了 *foldTree* 之后，像 *leafsize* 和 *branchsize* 这样的函数的定义是多么的方便。这是任何一种强制式语言所无法比拟的。

3.3 函数作为结果

上述的 *mapTree* 和 *foldTree* 通常被看作是把函数作为参数的例子，事实上，正像3.1节中所讨论的那样，它们也可被看作是把函数作为结果的例子。不过，在直觉上把函数作为结果的情况也很多。

例如：下面的函数 *selectFun* 将根据实际的布尔值来决定返回求平方值函数还是求立方值函数。

```
selectFun::Bool→(Int→Int)
selectFun True=(\x→x * x)
selectFun False =(\x→x * x * x)
```

3.3.1 *Uncurring* 在3.1节中我们已经看到，*mapTree* 的类型可以被看成是下列两种情况中的任何一种：

(1) $(a \rightarrow b) \rightarrow Tree\ a \rightarrow Tree\ b$

(2) $(a \rightarrow b) \rightarrow (Tree\ a \rightarrow Tree\ b)$

我们可以定义一个和 *mapTree* 相关的函数 *uncurriedmapTree* 如下：

```
uncurriedmapTree::((a→b),Tree a)→Tree b
uncurriedmapTree (f,(Leaf x))=Leaf (f x)
uncurriedmapTree (f,( Node l r))= Node (uncurriedmapTree (f,l)) (uncurriedmapTree (f,r))
```

注意，它与 *mapTree* 的唯一区别在于 *uncurriedmapTree* 必须同时取一对参数。事实上，我们可以借助于 *mapTree* 来定义 *uncurriedmapTree*，即：

```
uncurriedmapTree (a,b)=mapTree a b
```

不难看出，对任意类型为 $a \rightarrow b \rightarrow c$ 的函数，用同样的方式可以得到一个相应的类型为 $(a,b) \rightarrow c$ 的函数。进一步，我们可以定义一个高阶函数 *uncurry2* 来完成上述转换：

```
uncurry2::(a→b→c)→(a,b)→c
uncurry2 f (x,y) = f x y
或者：
uncurry2 f =\(x,y)→f x y
借助于高阶函数 uncurry2我们可以定义 uncurriedmapTree 为：
uncurriedmapTree = uncurry2 mapTree
```

3.3.2 *Currying* 类似地，每一个类型为 $(a,b) \rightarrow c$ 的函数都可以被转换为类型为 $a \rightarrow b \rightarrow c$ 的相应函数。这种转换被称为 *currying*，用于纪念著名的类型论专家 *H. B. Curry*（*Haskell* 也是以他的名字命名的）。下面的高阶函数 *curry2* 完成上述转换：

```
curry2::((a,b)→c)→(a→b→c)
curry2 g x y = g (x,y)
或者：
curry2 g =\x y→g(x,y)
```

借助于高阶函数 *curry2*，我们可以定义 *mapTree* 为：

```
mapTree = curry2 uncurriedmapTree
```

上面讨论的是对两个参数的情况，对任意 n 个参数，也存在类似的 *curryn* 和 *uncurryn*。例如，对3个参数的情况，*curry3* 将把一个类型为 $(a,b,c) \rightarrow d$ 的函数转换成一个类型为 $a \rightarrow b \rightarrow c \rightarrow d$ 的函数。相反的转换可由 *uncurry3* 完成。

3.4 搜索算法

下面我们将介绍如何实现树上的搜索算法，为此，我们需要定义高阶函数 map 以及引入 Haskell 的列表内涵特性，另

（下转第198页）

理较为消极，若能在死锁检测发起前就积极防止冗余检测的发生(即解决多重启动问题)，将更有效地降低伪死锁率而提高算法性能。所以，我们进一步的研究将重点关注如何解决多重启动问题。

参 考 文 献

1 Choudhary A N. Cost of Distributed Deadlock Detection: A Performance Study. In: Proc. the 6th Intl. Conf. on Data Engineering, 1990. 174～181

2 Rubio J M M, Lipez P, Dutao J. FC3D: Flow Control-based Distributed Deadlock Detection Mechanism for True Fully Adaptive Routing in Wormhole Network. IEEE Trans. on Parallel and Distributed Systems, Aug. 2003, 14(8): 765～779

3 Ekmecic I, Tartalja T, Milutinovic V. EM³: A contribution to taxonomy of heterogeneous computing systems. IEEE Computers, Dec. 1995. 68～70

4 Knapp E. Deadlock Detection in Distributed Databases. ACM Computing Surveys, Dec. 1987, 19(4): 79～100

5 Cao J, Zhou J Y, Zhu W W, Chen D X, Lu J. A Mobile Agent Enabled Approach for Distributed Deadlock Detection. In: Proc. Of GCC 2004, Wu Han, 2004

6 Wu J. Distributed System Design. CRC Press LLC, 1999

7 Chandy K M, Misra J. A distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems. In: Proc. Of ACM SIGACT-SIGOPS Symp. on Principles of Dist. Computer, Ottawa, Canada, Aug. 1982

8 Chandy K M, Misra J, Haas L M. Distributed Deadlock Detection. ACM Trans. on Computer Systems, May 1983, 1: 114～156

9 Singhal M. Deadlock Detection in Distributed Systems. IEEE Computer, 1989, 22(11): 37～48

10 Obermarck R. Distributed Deadlock Detection Algorithm. ACM Trans. On Database Systems, 1982, 7: 187～208

(上接第168页)

外我们也给出一个常用的高阶函数 *filter*。

3.4.1 *高阶函数 map* 这是一个与 *mapTree* 类似的高阶函数，但它作用在列表而不是二叉树上。它将一个作为参数的函数应用于一个列表中的每个元素，从而产生另一个列表。该函数的定义如下：

```
map::(a→b)→[a]→[b]
map f [ ]=[ ]
map f (a:as) = f a:map f as
```

例如：*map* (\x→x+1)[1,2,3]的结果为[2,3,4]。

换句话说，*map* 将一个函数的定义域扩展到相应的列表上。

3.4.2 *高阶函数 filter* *filter* 的功能是将列表中不满足过滤条件的元素过滤掉，这里，过滤条件是一个谓词(即值域是布尔类型的函数)，因此，它也是高阶函数。

```
filter::(a→Bool)→[a]→[a]
filter p[ ]=[ ]
filter p (a:as)| p a=a:filter p as
               | otherwise = filter p as
```

例如，*filter* (\x→x>3)[2,3,4,5]得到[4,5]。

3.4.3 *列表内涵特性* 在数学中，我们已经知道如何利用集合的内涵特性来描述集合的定义。例如我们可以用{$x|x \in N, x>10$}来表示大于10的自然数的集合，而不必逐个列出各个元素。Haskell 语言的列表内涵特性，可以像集合论中通过内涵的方法表示集合那样，让我们通过内涵的方法来表示列表，例如上面的表示大于10的自然数的集合，在 Hascall 中可以用列表[x|x←[0..],x>10]来表示。这里 x←[0..],x>10起到了内涵的作用。利用这种机制可以很方便地在现有列表的基础上定义新的列表。

一个列表内涵通常由三部分组成，分别是：

(1)产生器(generator)，即源列表集，也就是一个元素的选择范围，它是一个已知的列表。

(2)过滤器(filter)，规定了元素的选择条件，即如何从一个已知的列表中选择元素，选择哪些元素。

(3)转换(transformation)，即如何将从源列表集中选择的元素转换为希望生成的新列表的元素，可以理解为新列表中元素的表达形式。

例如，在列表内涵表达式[(x,*True*)|x←[1 .. 100],*even* x,x>15]中：x←[1..100]称为产生器，*even* x,x>15称为过滤器，(x,*True*)则称为新列表的转换。

借助于列表内涵，我们可以给出 *map* 和 *filter* 的更为简洁的定义如下：

```
map f xs=[f x|x←xs]
filter p xs=[x|x←xs,p x]
```

3.4.4 *树及其搜索* 前面我们只是局限于二叉树来讨论相应的应用，事实上，我们很容易定义一棵普通的树并且实现其上的搜索算法。下面定义的 *Gtree* 就是一棵普通的多态类型树：

```
data Gtree a=Gtree {root_item::a
                   ,children ::[Gtree a]}
```

利用上面定义的高阶函数 *map* 以及列表内涵特性，我们只用下面的两行，就可以给出深度优先搜索函数 *depth_first* 的类型说明及其实现。

```
depth_first::Gtree a→[a]
depth_first(Gtree n cs)
=n:[m|df←map depth_first cs,m←df]
```

比较一下在强制式语言中与此相关的概念和算法的定义，我们不难看出这样的实现是多么的简洁和优美！

结束语 本文简要阐述了 Haskell 语言及其高阶特性，通过一些简单的例子揭示了如何利用其高阶特性来编写程序模版。

综合上面的论述可以看出，使用高阶函数有以下优点：(1)定义简洁；(2)定义容易理解；(3)定义容易修改；(4)代码可重用性强。

本文中的程序，均已在 Haskell 的解释器 Hugs98(versionNov2003)和其编译器 GHC6.2.1上调试编译通过，真心希望能有更多的国内同行来使用该语言，编写出简洁漂亮的程序。

参 考 文 献

1 Pang J, Callaghan P, Luo Z. An approach to verification of domain properties based on LF. TYPES 2002 Workshop. Netherlands

2 Callaghan P C, Luo Z, Pang J. Object languages in a type-theoretic meta-framework. Workshop of Proof Transformation and Presentation and Proof Complexities (PTP'01), Italia, 2001

3 Callaghan P C. Functional Programming. Unpublished lecture notes. UK, 2003

4 Thompson S. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999. ISBN 0-201-34275-8

5 Bird R. Introduction to Functional Programming using Haskell, 2nd edition. Prentice Hall Press, 1998, ISBN: 0-13-484346-0

6 Jones S P. Haskell 98 Language and Libraries. Cambridge University Press, 2003, Hardback, ISBN: 0521826144

7 Structuring Depth First Search Algorithms in Haskell. In: King D, Launchbury J, eds. Proc. ACM Principles of Programming Languages, San Francisco, 1995

8 Haskell 98 report. http://www.haskell.org