计算机科学 1996Vol. 23№. 6

纯函数式语言的I/O系统

袁华强 肖 倩 孙永强
(湘潭大学计算机科学系 湘潭 411105) (上海交通大学 上海 200030)

摘 要 Purely functional programming languages are restricted on the application scope to some extents because they connot deal with I/O very well. This paper discusses five purely functional I/O approaches. These I/O approaches show that I/O can be incorporated in purely functional languages without losing the advantage of purely functional languages.

关键词 Purely functional languages, I/O.

函数式程序设计是指程序完全由函数组成，其中都有一个主函数，主函数是根据其它函数来定义的，直到最低层函数成为函数式语言所提供的基本函数为止。

函数式程序设计的优点主要有：(1)不包含赋值语句；(2)不包含任何类型的副作用；(3)表达式可以在任何时间计算。由于人们可以自由地用变量的值来替代变量，或用变量来替代变量的值，所以说程序是"引用透明性"的，比起相应的传统程序来更富数学魅力。

没有赋值语句，没有副作用，使得函数式语言的指称语义简单清晰，数学特性良好，并易于推导证明，但同时也带来了另一个问题：函数式语言不能很好地处理I/O问题[1]。

怎样才能在函数式语言中加入I/O处理功能而又保持函数式语言的优点，一直是函数式语言学界关注的焦点。长期以来，人们在这方面做了许多工作，研究成果也比较多，这些研究成果主要分为两大类：基于流的方法和基于环境的方法。前者又分两种风格：流风格和后续(continuation)风格；基于环境的方法也有两种：环境传递方法和Monad方法。下面我们将分别予以介绍。

1 基于流处理的I/O模式

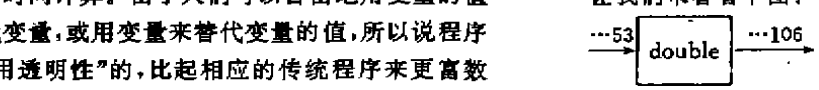
1.1 流风格

1.1.1 lazy流I/O 自文[1]在描述其纯函数式操作系统时采用流I/O模式后，流I/O模式便成为函数式程序设计语言处理I/O的最为流行的方法。流I/O模式的类型为：

type STR inp out = [inp]→[out]

类型STR可以看作是一个流转换器，其思想极其简单，流转换器将类型为inp的输入流转换为类型为out的输出流。将流转换器作用到输入流 $in_1 : in_2 : \cdots : in_n : \perp$ 上，将产生输出流 $out_1 : out_2 : \cdots : out_n : \perp$。表操作(:)应具有lazy语义，使得部分表 $(in_1 : in_2 : \cdots : in_n : \perp)$ 和 $(out_1 : out_2 : \cdots : out_n : \perp)$ 不会简单地等于⊥。

让我们来看看下图：



double :: [Int]→[Int]

double(x : xs) = (2 * x) : double xs

double将一个类型为整数的输入流转换为一个类型为整数的输出流，输出流中每一个值是输入流中相对应的值的2倍。让我们来看double在流I/O模式下是怎样执行的：

· 用户将double作为程序提交运行。

· 作为double隐含的变元，我们提供一个特殊的表达式keyboard，表示所有从键盘上输入并以其输入的顺序排列的数字流；当这个流的消费者(这里为double)要求下一个数字时，keyboard从键盘上获取下一个数字(若不是数字则等待)，将这个数字构成这个流的首部，挂起这个流的尾部调用。

· 执行最外归约，直至计算的形式为 n : e，其中n是一个完全计算好的数，将n在屏幕上显示出来，然后移至流的尾部e上，在e上我们可以进一步作用最外归约，必要时重复上述循环。

这种交互式I/O方法依赖于流的lazy计算，因此称这种方法为lazy流I/O方法。

1.1.2 对话I/O(Dialogue I/O)

对话I/O又叫同步流I/O[2]，实际上是lazy流I/O的变种。在对话I/O中，流转移器将一个请求流

转换成一个反应流，请求和反应是一一对应的。

一个对话是两个部分A，B之间的交互。A、B轮流向对方发出信号，每个部分都有一个包含交互历史信息的状态，A、B的状态分别叫α和β。每个部分都有自己的转换函数，其类型为：

f：：(state,input)→(state,output)

f从它前面的状态以及从另一部分接收的最后一个信号中计算新的状态和要向另一部分发送的信号。设A、B的转换函数分别为f_a、f_b，设A最初的状态为$\alpha_j, j\geq 0$，当建立B时，A发出最初的信号a_j给B，B的转换函数f_b就从最初状态β_0和从A接收来的信号a_l中计算其新状态β_1和要发送给A的信号b_0，这样就开始了A、B之间的对话。因此，转换函数f_a、f_b，初始状态α_j、β_0对每一部分分别定义了状态的序列和输出的序列。以下四个序列定义了对话的全部历史：

α：$(\alpha_0\ \alpha_1\ \alpha_2\cdots)$　a：$(a_0\ a_1\ a_2\cdots)$

β：$(\beta_0\ \beta_1\ \beta_2\cdots)$　b：$(b_0\ b_1\ b_2\cdots)$

由A初始化的A和B之间的对话是形式为(α a β b)的四元组，A和B之间的对话如下：

A	B
(initial)α_j	β_0(initial)
(send to B)$a_j\Rightarrow$	
	$(\beta_1, b_0)=f_b(\beta_0, a_j)$
	$\Leftarrow b_0$(send to A)
$(\alpha_{j+1}, a_{j+1})=f_a(\alpha_j, b_0)$	
(send to B)$a_{j+1}\Rightarrow$	
...	
$(\alpha_{j+k}, a_{j+k})=$ $f_a(\alpha_{j+k-1}, b_{k-1})$	
(send to B)$a_{j+k}\Rightarrow$	
	$(\beta_{k+1}, b_k)=f_b(\beta_k, a_{j+k})$
	$\Leftarrow b_k$(send to A)
...	

假设我们的对话的类型如下：

type Dialogue=STR Response Request

data Request=Get |Put Char

data Response=Got Char |Did

. Get：k：从键盘上输入字符n，然后执行k (Got,n)。

. (Put n)：k：将字符输出到打印机上，然后执行K Did。

下面就是用对话I/O写的echo程序：

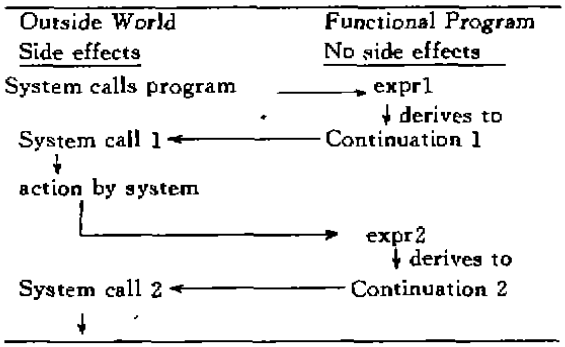
```
echo : : Dialogue
echo reps=Get : (\a-)
    if (a==eof)
    then[]
    else (Put a) : (echo (drop 2 reps))
    where Got a=reaps!! 1
```

\a→K是Haskell语言的记号，它表示一个λ表达式，即λa.k。

1.2 CPS I/O

后续是从中间值(中间值可能为空)到程序剩余部分的函数。因此，我们在定义函数的时候，不是直接返回函数的结果，而是将后续作为变元传递过去，作用在结果上。这样的程序设计风格常被叫做后续传递风格(Continuation passing style)或简称CPS。

假设中间值的类型为a，程序的剩余部分的类型为Answer，后续的类型则为a→Answer，那么怎样用CPS来处理I/O问题？让我们来看下图：



给定一个基于CPS的函数式I/O程序，假设程序从expr1开始运行，当程序希望执行一个I/O操作时，它发出一个信号给操作系统中断这个程序，程序剩余部分则保留下来。但是程序剩余部分的重新执行可能要依赖于这次I/O操作的结果，即上面提到过的中间值，中间值与程序剩余部分的映射则成为后续，系统执行完这次I/O操作之后，唤醒程序剩余部分，将后续作用到I/O的结果上，便得到一个新的程序，新程序从expr2处开始执行，重复上述过程，直至整个程序结束。

在CPS I/O中，可执行的类型是这样的代数类型，对每一种可表示的I/O行为有相应的类型构造子：

data CPS = INPUT (Char→CPS) | OUTPUT Char CPS |Done

CPS程序的意义能够容易地给出：

. INPUT k：从键盘输入n，然后执行k n。

. OUTPUT n p：将字符n输出到打印机上，然后执行p。

. Done：立即终止。

下面我们来看一个使用CPS I/O的程序：

```
echo : : CPS→CPS
echo c=INPUT(\a→)
    if (a==eof)
    then c
```

else OUPTUT a(echo c))

由于在 echo 完成之后我们可能想做更多的 I/O，所以我们必须为 echo 提供一个后续 c，来表示当 echo 完成时还要做什么。

2 基于环境的 I/O 模式

顾名思义，基于环境的 I/O 模式是在一个特定的对象即环境(environment)上来直接进行 I/O 操作的、环境表示世界的状态。在基于环境的 I/O 模式中，对环境的处理分为隐式、显式两种方法；前者主要有 Monad I/O 方法，后者主要有 Clean I/O 方法。

2.1 Monad I/O

Monad I/O 方法是基于隐式环境的方法，环境构成了机器的状态，在环境上的操作有一个特定的类型 IO a，环境仅能出现在 IO 类型里。IO a 表示这样的动作：执行某些 I/O 操作，返回类型为 a 的值。IO Monad 实际上是一个抽象数据类型，因此称 Monad I/O 中的环境为隐式环境。例如，给定两个 I/O 操作：

```
getcIO : : IO char
putcIO : : char→IO()
```

getcIO 从标准输入设备上读取一个字符，putcIO 将一个字符写到标准输出上，getcIO 返回一个类型为字符的值，而 putcIO 则不返回任何值。这两个 I/O 操作都会导致环境发生变化，但是由于 IO Monad 是一个抽象数据类型，环境仅能出现在 IO 类型中，所以环境的任何变化对用户都是不可见的。

I/O 操作是怎样执行的呢？答案就是提供一个顶层的标识符 mainIO，其意义在于将整个程序的值看作是一个单一的 I/O 动作，叫 mainIO，通过执行这个动作来执行程序。譬如：

```
mainIO : : IO()
mainIO=putcIO"!"
```

以上定义的函数在屏幕上打印惊叹号"!"，这只是一个简单的 I/O 动作，怎样将一些简单 I/O 动作组合成一个比较大的 I/O 动作呢？这里我们引入两个组合子：

```
unitIO : : a→IO a
bindIO : : IO a→(a→IO b)→IO b
```

unitIO x 表示仅返回 x 的 I/O 动作。'bindIO' 是 Haskell 语言的记号，表示 bindIO 为中缀操作。m 'bindIO' k，其中 m : : IO a，k : : a→IO b，其意义为：先执行 m，它产生类型为 a 的值 x，再执行 k x，它产生类型为 b 的值 y。(IO，unitIO，bindIO)构成了一个 Monad。

下面再来看另外两个组合子：

```
doneIO : : IO()
seqIO : : IO a→IO b→IO b
```

done IO 什么 I/O 动作也不做，m 'seqIO' k，其中 m : : IO a，k : : IO b，首先执行 m，然后再执行 k，返回由 k 产生的值，doneIO、seqIO 可以由 unitIO、bindIO 定义出来：

```
doneIO=unitIO()
m 'seqIO' k=m 'bindIO' \a→k
```

下面我们来看如何用 doneIO、seqIO 定义 1.1.2，1.2 节的 echo 程序的：

```
echo : : IO()
echo=getcIO 'bindIO' \a→
    if(a = = =eof)then
        doneIO
    else putcIO a 'seqIO' echo
```

2.2 Clean I/O

Clean I/O[5]是基于显式环境的 I/O 方法。一个处理 I/O 的程序是这样的函数：给定初始环境，产生一个新的环境，在这个新环境里所有相继发生的变化都是由这个 I/O 程序引起的。程序能改变世界的状态，并通过对环境有访存权的函数获取有关信息，这样的函数由两个动作组成：立即改变世界的状态；当改变世界的状态反馈回来时，生成环境的新实例。每一个新 I/O 操作作用到前面操作所产生的环境结果上，我们来看下面的例子：

```
echo : : * World→ * World
echo world
  |c=eof=world2
    =echo world2
      where
      (c,world1)= : getchar world
      world2= : putchar c world1
```

* World 为 Clean I/O 系统中环境的类型，函数 echo 是环境 World 上的递归函数，它仅通过预先定义的函数 getchar 从环境中取回一个字符，使用预先定义的函数 putchar 将这个字符打印在屏幕上，如果取回一个"eof"，则 echo 的递归调用结束，getchar 的类型为 * World→(a，* World)，putchar 的类型为 * World→ * World。

显式环境传递的 I/O 方法可以从环境变量 world 的变化中看出来，程序的初始环境为 world，经过对环境有访存权的函数 getchar 作用后，环境变为 world1，并将 world1 传递给另一个对环境有访存权的函数 putchar，putchar 作用以后环境成为 world2，环境这一系列变化都是 I/O 操作的结果。

参考文献

[1] P. Henderson, Purely functional operating systems, In J. Darlington et al. eds., Functional Programming and its Applications, Cambridge University Press, 1982

[2] JT O'Donnell, Dialogue: a basis for constructing programming envirnoments, In Proc ACM Symposium on Language Issues in Programming Environments, Seattle, ACM, 1985

[3] E Moggi, Notions of computation and monads, Information and Computation, 93, 1991

[4] SL Peyton Jones & PL Wadler, Imperative functional programming, In Proc. 20th ACM Symp. on Principles of Programming Languages. Charleston, 1993

[5] PM Achten & MJ Plasmeijer, The ins and outs of Clean I/O, J. of Functional Programming 5(1), 1995

[6] 袁华强，函数式 I/O 系统的研究与实现：一种 Monad 方法[博士学位论文]. 上海交通大学, 1996

（上接第 5 页）

D. Moon[20]提出了用散列表技术的查找结构，并将其应用于 Flavor 系统中。在散列表中要预先填充所有可能的分派情况，而在多方法环境下，各种分派情况的总数是组合爆炸的（复杂度为 $O(e^n)$[21]）。

G. Kicales[6]也提出了类似的方法应用于 CLOS 系统中。作为改进，P. H. Dussud[21]提出使用动态缓冲技术，动态缓冲中只保存被调用的方法。这种技术是非常量时间的，也没有解决组合爆炸的问题。

C. Lecluse 等[22]提出了结构子类化（sturctural subtyping）的概念。他们要求定义辅助的方法以满足仅仅通过子类型关系就可确定 MSA 方法。

W. G. Mugridge 等[23]定义参数序列 $(t_1, \cdots, t_n)$ 的覆盖（cover）为集合 $\{(S_1, \cdots, S_n) \mid S_1 \leqslant t_1, \cdots, S_n \leqslant t_n\}$，一个方法的可应用性定义为方法参数的覆盖与实际调用参数的覆盖的交集（可以认为集合越小的方法越具体）。对于一个类属函数调用，可通过比较相应覆盖的交集来确定 MSA 方法。这种解决方式应用于 Kea 中，当两个方法无法确定优先顺序时就声明调用是二义的。

总的说来，目前的研究工作都着重于提高各自算法所得到的目标代码的时间和空间性能。这些算法各有优劣之处，其中一部分已经应用于实际的多方法 OO 语言中。

我们认为，未来在多方法分派研究中的工作主要有几个方面：①对目前算法的改进和完善；②提出新的综合性能更佳的算法；③通过理论分析和实践检验，综合评价各种算法；④对算法进行综合。

对算法进行综合也有不同的方法。一方面，我们可以在编译系统中设置编译开关，由程序人员根据实际情况在时间与空间性能之间进行权衡和选择。另一方面，也是更重要的一方面，我们可以让编译系统检查分析程序的结构，自动选择最佳算法，甚至组合各种算法。

由于多方法语言强大功能所具有的吸引力，这方面的研究工作将会进一步深入下去。尽管存在着相当大的困难，我们仍然可以期望，在不久的将来，多方法语言将获得普遍的接受和应用。（参考文献共 22 篇略）