函数式语言中的赋值语句*

石跃祥[1] 袁华强[1] 孙永强[2] 陈 静[1]

[1](湘潭大学计算机科学系 湘潭 411105)

[2](上海交通大学计算机科学与工程系 上海 200030)

摘要 文章探讨了怎样在纯函数式语言中加入赋值操作,而又不丧失引用透明性特征的问题,给出了这些操作的指称语义,并用这些赋值操作定义了一个简单的命令式语言的解释程序.

关键词 纯函数式语言,赋值语句,Monads.

中图法分类号 TP312

怎样在纯函数式语言中加入赋值语句,而又不丧失引用透明性特征,一直是函数式语言学界关注的焦点.人们在这方面提出了一系列的方法,这些方法大都是基于类型系统的,而且类型系统十分复杂,没有被人们广泛接受.文献[1]运用Monad方法[1~3],将赋值语句加入到纯函数式语言中,避免了人们以往采用的复杂的类型系统.但是,文献[1]提出了一个异步的I/O操作 *performIO*,这个操作与引用透明性是相冲突的.本文针对这一缺陷,采用状态转换器以及对状态进行参数化的方法进行改进,使得在纯函数式语言中加入赋值语句,而又不丧失引用透明性特征,并给出了这些赋值操作的指称语义,用这些赋值操作定义了一个简单的命令式语言的解释程序.

1 状态转换器

定义 1.1. 状态转换器ST是这样的函数,它作用在一个类型为 s 的初始状态上,返回一个类型为 a 的值和一个类型仍为 s 的结束状态.

这样,状态转换器ST的类型可定义为: type $ST\ s\ a$ = state s -> (a, state s)

最简单的状态转换器unitST,它仅仅传递值,丝毫不影响状态:

$unitST :: a -> ST\ s\ a$

$unitST\ x = \backslash s -> (x, s)$,其中$\backslash s ->$是一个 λ 表达式.

状态转换器能够顺序组合,以构成更大的状态转换器,这个工作由bindST来完成:

$bindST :: ST\ s\ a -> (a -> ST\ s\ b) -> ST\ s\ b$

$m\ `bindST'\ k = \backslash s ->$ let $(x, s') = m\ s$

in $k\ x\ s'$

定理 1.1[3]. $(ST, unitST, bindST)$构成了一个Monad.

2 对状态进行参数化

在文献[1]中,对赋值语句的处理采用了文献[4]的可赋值引用类型 $Ref\ a$,其中 $IO\ a$ 是一种Monad:

$newVar :: a -> IO(Ref\ a)$

$assignVar :: Ref\ a -> a -> IO\ ()$

---

* 作者**石跃祥**,1966年生,讲师,主要研究领域为软件工程.**袁华强**,1966年生,博士,副教授,主要研究领域为函数式语言,软件工程.**孙永强**,1931年生,教授,博士生导师,主要研究领域为并行理论,函数式语言.**陈静**,女,1968年生,工程师,主要研究领域为函数式语言.

本文通讯联系人:袁华强,湘潭411105,湘潭大学计算机科学系

本文1997-07-21收到原稿,1998-04-14收到修改稿

deRefVar::*Ref a*->*IO a*

文献[1]提出了一个异步的I/O操作 *performIO*,这个操作与引用透明性是相冲突的.

performIO::*IO a*->*a*

performIO m=case (*m newWorld*) of

MkIORes r w'->*r*

让我们来看下面的例子:

let *v*=*performIO*(*newVar true*)

in *performIO*(*deRefVar v*).

这样做将蕴涵着一个错误,因为 *v* 是由 *newVar true* 生成的,执行 *performIO* 之后,*v* 被释放到了外部世界,该程序不能控制在读操作 *deRefVar v* 之前,有其他操作对 *v* 进行了修改,程序的结果将依赖于计算的顺序,引用透明性特征因此丧失.

在以往的 *IO a* 类型中,状态隐含在 *IO* 类型,外界不能直接对状态进行操作.为了改正文献[1]的错误,我们将 *IO* 类型中的状态显式地表示出来,改进为 *ST s a*,使得状态成为 *ST* 的一个参数,我们称之为对状态进行参数化.与文献[4]一样,我们将状态处理为引用变量的地址到值的映射的集合,将类型 *Ref a* 改进为 *MutVar s a*,它表示引用变量是从类型为 *s* 的状态中分配而来,并包含有类型为 *a* 的值.因此,文献[1]的引用变量的3个基本操作可改进为

newVar::*a*->*ST s* (*MutVar s a*)

readVar::*MutVar s a*->*ST s a*

writeVar::*MutVar s a*->*a*->*ST s* ()

这样,通过状态转换器和对状态进行参数化,我们就可以定义我们的异步I/O操作 *performIO*:

performIO::∀ *a*.(∀ *s*.*ST s a*)->*a*

这不是一个Hindley-Milner类型,因为量词不全在顶端.为什么这个类型会防止文献[1]中的错误发生呢?我们还是来看下面这个例子:

let *v*=*performIO*(*newVar true*)

in *performIO*(*readVar v*)

我们先来看 *performIO*(*readVar v*),引用变量 *v* 在状态转换器(*readVar v*)中,*readVar v* 的类型依赖于 *v* 的类型,因此,这个类型推导将包含下列形式的判断:

{...,*v*:*MutVar s Bool*}⊢*readVar v*:*ST s Bool*

为了执行 *performIO*(*readVar v*),*readVar v* 的类型应该是∀ *s*.*ST s Bool*,但此时 *s* 并不是自由变元,因此,*readVar* 的类型不是∀ *s*.*ST s Bool*,与 *performIO* 所要求的类型不匹配.

再来考虑 *v* 的定义:*v*=*performIO*(*newVar true*),*newVar true* 具有类型 *ST s* (*MutVar s Bool*),因而可推广为∀ *s*.*ST s* (*MutVar s Bool*),但这依然与 *performIO* 的类型不匹配.考虑将 *performIO* 的类型∀ *a*.(∀ *s*.*ST s a*)->*a* 中的 *a* 用 *MutVar s Bool* 例化后的类型:

performIO::∀ *s'*.*ST s'* (*MutVar s Bool*)->*MutVar s Bool*

当 *a* 用 *MutVar s Bool* 例化时,我们必须将 *performIO* 类型的约束变元改名,这样 *newVar true* 的类型∀ *s*.*ST s* (*MutVar s Bool*)与∀ *s'*.*ST s'* (*MutVar s Bool*)不匹配.

综上所述,通过状态转换器和对状态进行参数化,我们将 *performIO* 的类型定义为∀ *a*.(∀ *s*.*ST s a*)->*a*,这样就完全避免了文献[1]中的错误,保证了引用透明性不丧失.

3 指称语义

状态操作很容易加到纯函数式语言的标准语义中.首先,我们定义扩展了状态转换器的纯函数式语言的语法:

e::=*x*|*k*|*e*$_1$*e*$_2$|*x*->*e*|let *x*=*e*$_1$ in *e*$_2$

$k::=\ldots\mid unitST\mid bindST\ newVar\mid readVar\mid writeVar\mid performIO$

其中 x 表示变量,k 表示像 $unitST$,$bindST$ 这样的内部函数.

以下是这个简单的纯函数式语言的指称语义:

ε $\llbracket Expr\rrbracket:Env->Val$

ε $\llbracket k\rrbracket\rho=\beta\llbracket k\rrbracket$

ε $\llbracket x\rrbracket\rho=\rho x$

ε $\llbracket e_1e_2\rrbracket\rho=(\varepsilon\llbracket e_1\rrbracket\rho)(\varepsilon\llbracket e_2\rrbracket\rho)$

ε $\llbracket\backslash x->e\rrbracket\rho=\lambda v.(\varepsilon\llbracket e\rrbracket(\rho[x->v]))$

ε $\llbracket$let $x=e_1$ in $e_2\rrbracket\rho=\varepsilon\llbracket e_2\rrbracket(fix(\backslash\rho'->(\rho[x->\varepsilon\llbracket e_1\rrbracket\rho'])))$

β $\llbracket performIO\ e\rrbracket=VperformIO(\varepsilon\llbracket e\rrbracket\rho)$

β $\llbracket e_1$ '$bindIO$' $e_2\rrbracket=VbindST(\varepsilon\llbracket e_1\rrbracket\rho)(\varepsilon\llbracket e_2\rrbracket\rho)$

β $\llbracket unitST\ e\rrbracket=VunitST(\varepsilon\llbracket e\rrbracket\rho)$

β $\llbracket newVar\ e\rrbracket=VnewVar(\varepsilon\llbracket e\rrbracket\rho)$

β $\llbracket readVar\ v\rrbracket=VreadVar(\varepsilon\llbracket v\rrbracket\rho)$

β $\llbracket writeVar\ v\ e\rrbracket=VwriteVar(\varepsilon\llbracket v\rrbracket\rho)(\varepsilon\llbracket e\rrbracket\rho)$

我们使用 Env 表示环境的论域,Val 表示值的论域:

$Env=\prod_\tau(var_\tau->D_\tau)$

$Val=\bigcup_\tau D_\tau$

环境将类型 τ 的变量映射为论域 D_τ 的值,并且值的论域就是所有 D_τ 的并集.

我们引入了两个新的类型构造子 ST,$MutVar$,为了给出它们的意义,语义函数必须提供它们的结构:

$D_{ST\ a}=State\ s->(D_a->State\ s)$

$D_{MutVar\ s\ a}=N_\perp$

$State\ s=(N->Val)_\perp$

状态是从地址(地址由自然数表示)到值的有限部分函数,我们用$\perp$表示未定义状态,这样我们就可以给出变量操作的指称语义:

$(VperformIO\ m)\sigma=x$ where $(x,\sigma')=m\ \sigma$

$(VbindST\ m\ k)\sigma=k\ x\ \sigma'$ where $(x,\sigma')=m\ \sigma$

$(VunitST\ v)\ \sigma=(v,\sigma)$

$VnewVar\ v\ \sigma=(\perp,\perp)$ if $\sigma=\perp$

$(p,\sigma[p->v])$ otherwise

$VreadVar\ p\ \sigma=(\perp,\perp)$ if $p\notin dom$

$(\sigma p,\sigma)$ otherwise

$VwriteVar\ p\ v\ \sigma=(\perp,\perp)$ if $p\notin dom$

$((),\sigma[p->v])$ otherwise

我们来看一个例子,这是一个简单的命令式语言的解释程序,$Assign\ Var\ Exp$ 表示将 Exp 的值赋给变量 Var,$Read\ Var$ 表示一个读操作,它从键盘读取一个数据赋给变量 $Vatr$,为简单起见,假设所有的输入已预先存入一个表 input 中,$Write\ Exp$ 是一个输出操作,While Exp[Com]是通常的循环语句.

$Data\ Com=Assign\ Var\ Exp\mid Read\ Var\mid Write\ Exp\mid$While Exp [Com]

$type\ Var=Char$

$data\ Exp=\ldots$

$interpret::[Com]->[Int]->[Int]$

$interpret\ cs\ input=performIO(newVar\ input$ '$bindST$' $\backslash inp->command\ cs\ inp)$

$command::[Com]->MutVar\ s\ [Int]->ST\ s\ [Int]$

```
command cs inp=obey cs
where
  obey::[Com]->ST s [Int]
  obey (Assign v e : cs)=eval e 'bindST' \val->
                         writeVar v val 'bindST' \_->
                         obey cs
  obey (Read v:cs)=readVar v 'bindST' \(x:xs)->
                   writeVar v x 'bindST' \_->
                   writeVar inp xs 'bindST' \_->
                   obey cs
  obey (Write e:cs)=eval e 'bindST' \out->
                    obey cs 'bindST' \outs->
                    unitST (out:outs)
  obey (While e bs:cs)=eval e 'bindST' \val->
                       if val= =0
                       then obey cs
                       else obey(bs++While e bs:cs)
```

参考文献

1 Jones SL Peyton, Wadler PL. Imperative functional programming. In: Hughes ed. Proceedings of the 20th ACM Symposium on Principles of Programming Languages. New York: ACM Press, 1993. 71～84

2 Wadler PL. Comprehending Monads. Mathematical Structures in Computer Science, 1992,2(4):461～493

3 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:上海交通大学,1996
(Yuan Hua-qiang. The research and implementation of a pure functional I/O system[Ph.D. Thesis]. Shanghai: Shanghai Jiaotong University, 1996)

4 Swarup V, Reddy US, Ireland E. Assignments for applicative languages. In: Hughes ed. Proceedings of Functional Programming Languages and Computer Architecture. Heidelberg: Springer-Verlag, 1991. 192～214

Assignments for Pure Functional Languages

SHI Yue-xiang[1] YUAN Hua-qiang[1] SUN Yong-qiang[2] CHEN Jing[1]

[1](*Department of Computer Science Xiangtan University Xiangtan* 411105)

[2](*Department of Computer Science and Engineering Shanghai Jiaotong University Shanghai* 200030)

Abstract In this paper, the authors show that assignments can be incorporated into pure functional languages without loss of referential transparency. And the denotational semantics of these assignment operations are given. Using these assignment operations, the authors define an interpreter of a simple imperative language.

Key words Pure functional languages, assignments, Monads.