# 基于 Monad 的纯函数式程序设计*

袁华强 孙永强

(上海交通大学计算机科学与工程系 上海 200030)

**摘要** Philip Wadler 在探讨用 Monad 构造纯函数式程序时,介绍了一个简单的词法分析程序的构造过程.本文进一步研究了这种方法,并用这种方法构造出一个能进行复杂的 layout 分析、词法分析与语法分析的纯函数式分析程序.

**关键词** 函数式程序设计,纯函数式语言,Monad,分析技术.

Monad 是范畴论中一个重要的概念,E. Moggi 在文献[1,2]中提出用 Monad 来构造计算的语义,其主要思想是将一个类型的值的对象与计算的对象区别开来,并把一个类型的程序的指称语义看作是该类型计算的对象的元素.

P. Wadler 在文献[3~6]中采用了 E. Moggi 的思想,把 Monad 作为构造函数式语言的工具,并构造出诸如错误处理、状态、I/O 等非纯函数式语言的特征,引起了人们的广泛关注.其原因在于:以往人们在纯函数式语言中加入非纯函数式语言的特征时,不得不对纯函数式语言自身作大量的修改,从而很难避免失去纯函数式语言的某些优点,这个问题始终困扰着函数式程序的学术界.使用 Monad 则不然,Monad 是一个抽象数据类型和模块化的结构,我们只需将不同的非纯函数式语言的特征定义成相应的 Monad,并只对纯函数式语言作少量的局部修改,甚至不作任何修改,就可以在纯函数式语言中构造出非纯函数式语言的特征,这被认为可能是函数式语言中的突破.

词法与语法分析技术在编译理论中占有很重要的位置,P. Wadler 在文献[5]中介绍了一个比较简单的分析程序的构造过程.本文将利用这种方法构造出一个能进行比较复杂的 layout、词法分析与语法分析的纯函数式分析程序.本文所采用的方法是将分析程序直接构造成函数,比较大的分析程序是由其组成部分通过 Monad 提供的 2 个函数 unit 与 * 逐级组合起来的,因而显示了这种方法的优点,即模块化易于理解,便于修改.

## 1 基本概念

**定义 1.1.**[2] 范畴 $\zeta$ 上的 Monad 是这样的三元组 $(T,\eta,\mu)$,其中 $T:\zeta\to\zeta$ 是一个函子,$\eta:Id_\zeta\to T$ 和 $\mu:T^2\to T$ 是自然变换,并使得下列图可交换.

---

* 作者**袁华强**,1966 年生,博士生,主要研究领域为新型语言及其支撑环境.**孙永强**,1931 年生,教授,博士导师,主要研究领域为计算机软件,计算理论.

本文通讯联系人:孙永强,上海 200030,上海交通大学计算机科学与工程系

本文 1995-11-03 收到修改稿

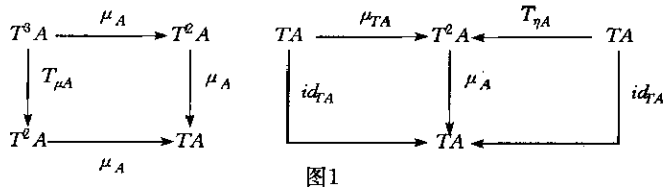

图1

**定义** 1.2.[5] 一个 Monad 是这样的三元组($M$,*unit*,$*$),其中 $M$ 是类型构造子,*unit*,$*$ 是分别具有如下类型的函数:

$unit :: \alpha \to M\alpha$

$* \quad :: M\alpha \to (\alpha \to M\beta) \to M\beta$

并且满足下列 3 个条件:

(1) $(unit\ v) * k = kv$

(2) $m * unit = m$

(3) $(m * k) * w = m * \lambda v.\ (kv * w)$

其中 $v$ 不在 $w$ 中自由出现.

定义 1.1 与 1.2 是等价的,详细说明见文献[5]. 本文中的 Monad 采用了定义 1.2 的形式. 定义 1.2 中的第 3 个条件与文献[5]的 $m * (\lambda a.\ n * \lambda b.\ o) = (m * \lambda a.\ n) * \lambda b.\ o$ 是等价的,其中 $a$ 不在 $o$ 中自由出现,由于 $a$ 在左边的作用域包括 $o$,在右边的作用域不包括 $o$,所以文献[5]中的这个条件写成如下形式更合适:

$$m * \lambda a.\ (n * \lambda b.\ o) = (m * \lambda a.\ n) * \lambda b.\ o$$

在纯函数式语言中没有错误处理的功能,下面我们给出一个能进行错误处理的 Monad,并用它在纯函数式语言中构造出错误处理功能.

例 1.1:

```
data Ex α=Raise Exception|Return α
type Exception=String
unit::α→Exα
unit a=Return a
*::Mα→(α→Mβ)→Mβ
m*k=case m of
      Raise e→Raise e
      Return a→k a
```

在类型构造子 $Ex$ 中,若一个值是正确的,则返回这个值,若是错误的,则指明是什么样的错误. 在 $m * k$ 中,首先计算 $m$,若是错误的则报错,若是正确的则将 $k$ 作用到这个正确的值上. 比如,在进行除法运算时,若除数为 0,则会出现错误,用 Monad 写除法运算的纯函数式程序如下:

```
eval(Div t u)=eval t * λa.(eval u * λb.if b=0
                    then Raise "divided by zero" else unit(a÷b)
```

下面定义一个基于表的状态 Monad,本文将用它来构造纯函数式分析程序.

**定义** 1.3.

```
type Mα=State→[(α→State)]
```

$type\ State=[\delta]$

$unit::\alpha\to M\alpha$

$unit\ a=\lambda x.[(a,x)]$

$*::M\alpha\to(\alpha\to M\beta)\to M\beta$

$m*k=\lambda x.[(b,z)|(a,y)\leftarrow m\ x;(b,z)\leftarrow k\ a\ y]$

定义 1.3 中的$(M,unit,*)$构成了一个 Monad,下面我们将验证 $unit$,$*$ 满足定义 1.2 中的 3 个条件.

$(1)(unit\ v)*k=\lambda x.[(b,z)|(a,y)\leftarrow(unit\ v)x;(b,z)\leftarrow k\ a\ y]$

$=\lambda x.[(b,z)|(b,z)\leftarrow k\ v\ x]$

$=\lambda x.k\ v\ x=k\ v$

$(2)m*unit=\lambda x.[(b,z)|(a,y)\leftarrow m\ x;(b,z)\leftarrow unit\ a\ y]$

$=\lambda x.[(b,z)|(a,y)\leftarrow m\ x;(b,z)\leftarrow[(a,y)]]$

$=\lambda x.[(a,y)|(a,y)\leftarrow m\ x]=\lambda x.m\ x=m$

$(3)(m*k)*w=\lambda x.[(b,z)|(a,y)\leftarrow(m*k)x;(b,z)\leftarrow w\ a\ y]$

$=\lambda x.[(b,z)|(a,y)\leftarrow[(b1,z1)|(a1,y1)\leftarrow m\ x;(b1,z1)\leftarrow k\ a1\ y1];(b,z)\leftarrow w\ a\ y]$

$m*\lambda v.(kv*w)=\lambda x.[(b,z)|(a1,y1)\leftarrow m\ x;(b,z)\leftarrow(\lambda v.(kv\ *\ w))\ a1\ y1]$

$=\lambda x.[(b,z)|(a1,y1)\leftarrow m\ x;(b,z)\leftarrow[(b1,z1)|(a,y)\leftarrow k\ a1\ y1;(b1,z1)\leftarrow w\ a\ y]]$

因而$(m*k)*w=m*\lambda v.(kv*w)$.

所以$(M,unit,*)$构成了一个 Monad.

## 2　基本分析程序

P. Wadler 在文献[5]中定义了 $zero$,$item$,$\odot$,$\triangle$,$letter$,$digit$,$lit$,$iterate$,$number$ 等函数:$zero$ 返回一张空表;$item$ 返回输入串的第 1 个字符,输入串为空时返回 1 个空表,如 $item$ "$monad$"=[('$m$',"$onad$")];$m\odot n$ 将分析程序 $m$,$n$ 的分析结果连接起来;$m\triangle p$ 判断 $m$ 分析出来的值是否满足条件 $p$,若满足则返回这个值,不满足则失败;$letter=item\triangle$is$Letter$ (is$Letter$ 判定一个字符是否为字母);$digit=(item\triangle$is$Digit)*\lambda a.unit(ord\ a-ord\ '0')$ (is$Digit$ 判断一个字符是否为数字);$lit\ c=item\triangle(\lambda a.a=c)$;$iterate\ m=(m*\lambda a.iterate\ m*\lambda x.unit(a:x))\odot unit\ []$;例如 $iterate\ (lit\ 'a')$ "$aab$"=[("$aa$","$b$"),("$a$","$ab$"),("","$aab$")]. 下面我们将定义其它一些基本函数,通过定义 1.3 给出的 Monad,我们可以将它们组合成一个比较复杂的分析程序.

$sat::(\alpha\to bool)\to M\alpha$

$sat\ p=item\triangle p$

$seq::M\alpha\to M\beta\to M(\alpha,\beta)$

$seq\ p1\ p2=p1*\lambda a.(p2*\lambda b.unit(a,b))$

$map::(\alpha\to\beta)\to M\alpha\to M\beta$

$map\ f\ p=p*\lambda a.unit(fa)$

*iterance*::$M\alpha \to M[\alpha]$

*iterance p*=*map cons* (*seq p* (*iterate p*))

*word*::$M$[*char*]

*word*=*iterance*(*letter*)

*string*::$[\alpha] \to M[\alpha]$

*string* []=[]

*string* (*x*:*xs*)=*map cons* (*seq* (*lit x*) (*string xs*))

*seq*1::$M\alpha \to M\beta \to M\alpha$

*seq*2::$M\alpha \to M\beta \to M\beta$

*seq*1 *p*1 *p*2=*map fst* (*seq p*1 *p*2)

*seq*2 *p*1 *p*2=*map snd* (*seq p*1 *p*2)

其中 $fst(x,y)=x, snd(x,y)=y$.

*sat p* 返回满足条件 *p* 的输入串的第1个字符;*seq p*1 *p*2 先执行 *p*1,再执行 *p*2,按顺序返回它们分析出来的值对$(a,b)$;*map f p* 把 *f* 作用到 *p* 分析出来的值上;*iterance* 与文献[5]的 *iterate* 的区别在于 *iterance* 不返回空串,如 *iterance* (*lit* 'a') "*aab*"=[("*aa*","*b*"),("*a*","*ab*")];*string* 判断一个给定的字符串是否在输入串的首部,如 *string* "begin""begin end"=[("begin","end")].

例 2.1:给定算术四则混合运算的 BNF 语法如下:

*expn*::=*expn*+*term*|*expn*−*term*|*term*

*term*::=*term* * *factor*|*term*÷*factor*|*factor*

*factor*::=$digit^+$|(*expn*)

我们可以得到如下用 Monad 构造的四则混合运算的纯函数式分析程序:

*expn*=(*map plus* (*seq expn* (*seq*2 (*lit* '+') *term*)))⊙
(*map minus* (*seq expn* (*seq*2 (*lit* '+') *term*)))⊙*term*

*term*=(*map times* (*seq term* (*seq*2 (*lit* '*') *factor*)))⊙
(*map divide* (*seq term* (*seq*2 (*lit* '÷') *factor*)))⊙*factor*

*factor*=(*map Num number*)⊙(*seq*2 (*lit* '(') (*seq*1 *expn* (*lit* ')')))

其中 $plus(x,y)$=add $x$ $y$, $minus(x,y)$=sub $x$ $y$,

$times(x,y)$=mul $x$ $y$, $divide(x,y)$=div $x$ $y$.

例如:*expn* "5+(3−1)÷2"
=[(Add (*Num* 5) (Div (Sub (*Num* 3) (*Num* 1)) (*Num* 2))),""]

## 3 Layout 分析程序

**定义** 3.1. 一个语法类的对象的任何符号直接在其第1个符号的下面或右边,则称该语法类符合 layout 规则.

许多函数式语言如 Miranda,Haskell 等都采用了 layout 规则,这样使得在 where,let,of 等关键字后面的大括号与分号可以省略,而不会导致二义性.

例 3.1:给定下列用 layout 规则写的程序:

```
f x y=mul a b
        where
              a=add x y
              b=sub x y
result=mul (f 3 7) 5
```

显然,$x$,$y$ 是全局定义,$a$,$b$ 是函数 $f$ 中的局部定义. 文献[5]中的分析程序无法对这样形式的函数式程序进行分析,本文的解决办法是对输入的每一个字符加上它所在的行与列组成的坐标,这样便得到了如下类型:

$$type\ pos\ d=(d,(num,num))$$

本文讨论的 Monad 中的 state 的类型均为[$pos\ d$].

假设每一程序行开头的空格都是由 *tab* 键形成,同一程序行的不同字符串之间的空格看作是程序的一部分,这样我们只处理由"\t","\n"形成的空格.

```
position::(num,num)→[char]→[pos char]
position (r,c) []=[]
position (r,c) (x:xs)=(x,(r,c)):position (r,tab c) xs, xs="\t"
                     =(x,(r,c)):position (r+1,0) xs, xs="\n"
                     =(x,(r,c)):position (r,c+1) xs, otherwise
tab c=((c div 8)+1)*8
```

有时我们对所分析字符的位置并不关心,因而有必要去掉有关位置的信息,我们对文献[5]的 *item* 的定义作如下修改即可达到这样的目的:

```
item::M char
item []=[]
item (x:xs)=[(a,xs)] where (a,(r,c))=x
```

经过 *position* 对输入字符串进行预处理后,由于每个字符都带有其位置,所有关于 layout 的信息不受任何影响,因此 layout 分析程序定义如下:

```
layout::M char→M char
layout p=λx.[(v,drop(#(takewhile (behind (hd x)) x)) x)|
            (v,[])←p(takewhile(behind (hd x)) x)]
            where behind (a,(r,c)) (b,(r',c'))=r'≥r∧c'≥c
```

其中 # 计算一个表的长度,*takewhile p xs* 把表 *xs* 中从第 1 个元素起连续满足 *p* 的元素找出来组成一个新表,*drop n xs* 去掉表 *xs* 中前 *n* 个元素.

## 4　建立实用的分析程序

(1)词法分析

词法分析的基本功能是把输入字符串分解成系统能够识别的符号,本文的处理方式是对每个经词法分析程序分析出来的符号加上一个标志,如(*Ident*,"*add*"),(*Lpar*,"(")分别表示"*add*"是一个标识符,"("是一个左括号.

*type token*=(*tag*,[*char*])

*data tag*=*Ident*|*Number*|*Keyword*|*Tab*|*Return*|*Lpar*|*Rpar*|*Eq*

其中 *Ident* 表示标识符,*Number* 表示整数,*Keyword* 表示 Where 等关键字,*Tab*,*Return*,*Lpar*,*Rpar*,*Eq* 分别表示"\t","\n","(",")","=".

下面定义的函数 *addtag* 的功能是给每个值加上给定的标志:

*addtag*::*M*[*char*]→*tag*→*M*[*pos token*]

*addtag p t*=λ*x*.[(((*t*,*xs*),(*r*,*c*)),*out*)|(*xs*,*out*)←*p x*]

where (_,(*r*,*c*))=*hd x*

例如,*addtag* (*String* "*where*") *Keyword* 将返回值((*Keyword*,"*where*"),(*r*,*c*)),其中(*r*,*c*)是输入串中"*where*"第 1 次出现时"*w*"位置.

这样便可以建立如下的词法分析程序:

*lex*::[(*M*[*char*],*tag*)]→*M*[*pos token*]

*lex*=*iterate* (*foldr op zero*)

where *op* (*p*,*t*) *xs*=(*addtag p t*)⊙*xs*

其中 *foldr f a* [*x*1,*x*2,…,*xn*]=*f x*1 (*f x*2 (…(*f xn a*)…))

*lexical*::*M*[*pos token*]

*lexical*=*lex* [(*iterance* (*foldr* (⊙ (*lit* "\t")) *zero*),*Tab*),

(*iterance* (*foldr* (⊙ (*lit* "\n")) *zero*),*Return*),

(*word*,*Ident*),(*number*,*Number*),(*string* "(",*Lpar*),

(*string* ")",*Rpar*),(*string* "=",*Eq*)]

(2)语法分析

语法分析是在词法分析的基础上构成 1 棵语法分析树. 在语法分析过程中,一个符号的标志比这符号自身更重要,下面定义的 *kind t* 能识别带标志 *t* 的符号,而不管这符号是什么,一旦语法分析完成以后,其标志就成为多余,最后 *kind t* 返回被分析的符号,其标志就被丢弃.

*kind*::*tag*→*M*[*char*]

*kind t*=*map snd* (*sat* ((=*t*)∘*fst*))

其中∘表示函数复合,。是 Monad 的 * 的特殊情形,这是因为。的类型为 $\alpha\to(\alpha\to\beta)\to\beta$,* 的类型为 $M\alpha\to(\alpha\to M\beta)\to M\beta$,这时只要将 Monad 定义成如下形式即可:

*type Mα*=*α*

*unit a*=*a*

*m* * *k*=*k*∘*m*

下面给出例 3.1 中这类程序的 BNF 语法:

*prog*::=*defn*$^*$

*defn*::=*var*$^+$"="*body*

*body*::=*expr*["*where*"*defn*$^+$]

*expr*::=*expr prim*|*prim*

*prim*::=*var*|*num*|"("*expr*")"

如果一个程序是合式的,语法分析程序将产生一个类型为 *script* 的分析树,*script* 的定义如下:

*script*::=*Script* [*def*]

*def*::=*Def var* [*var*] *expn*

*expn*::=*Var var*|*Num num*|*expn expn*|*expn* Where[*def*]

其中 *var*=[*char*]

在上述 BNF 语法中,存在一个如何消去 *expr*::=*expr prim* 中的左递归问题,按照文献[5]中左递归消去办法,便可得到如下语法分析程序:

```
prog,defn,body,expr,prim::M script
prog=map Script (iterate defn)
defn=map defnFN (seq (iterance (kind Ident))
                     (seq2 (kind Eq) (layout body)))
body=map bodyFN (seq expr (seq2 (kind Keyword)
                     ((iterance defn) ⊙ unit [])))
expr=seq prim (closure prim)
prim=(map Var (kind Ident))⊙(map numFN (kind Number))
     ⊙(seq2 (kind Lpar) (seq1 expr (kind Rpar)))
```

其中 *defnFN* (*f*:*xs*,*e*)=*Def f xs*,*numFN xs*=*Num xs*,

*bodyFN* (*e*,[])=*e*,*bodyFN* (*e*,*d*:*ds*)=*e* where *d*:*ds*,

*closure prim*=λ*x*.(*seq* (*prim x*) (*closure prim*))⊙*unit x*

(3)建立实用的分析程序

不难看出,语法分析程序没有对"\t","\n"进行处理,"\t","\n"并不是 BNF 语法中所固有的,因此,在进行语法分析之前应该将"\t","\n"去掉,这样丝毫不影响语法分析树的产生,下面定义的 *eliminate* 函数的作用就是去掉经词法分析后带有 *Tab* 或 *Return* 的符号.

```
space::[char]→bool
space xs=(xs=Tab)∨(xs=Return)
eliminate::[pos token]→[pos token]
eliminate=filter(space∘fst∘fst)
```

其中 *filter p xs* 把表 *xs* 中所有满足 *p* 的元素按原来的顺序组成一个新表.

在定义完整的分析程序之前,再定义 2 个辅助函数.

```
block1::[char]→M[pos token]→[pos token]
block1 xs m=let [(a,x)]=m(position (0,0) xs) in a
block2::[pos token]→Mscript→script
block2 xs m=let [(a,xs)]=m xs in a
```

以上我们定义了 4 个重要的函数:位置函数 *position*,词法分析函数 *lexical*,删除空格函数 *eliminate* 以及语法分析函数 *prog*,为简便起见,假设分析总是成功的,这样,完整的分析程序定义如下:

```
parser::[char]→script
```

*parser xs=block2 (eliminate (block1 xs lexical)) prog*

## 参考文献


1 Moggi E. Computational Lamda-calculus and Monads. In: Symposium on Logic in Computer Science, California, IEEE, 1989. 14～23.

2 Moggi E. Notions of computation and Monads. Information and Computation, 1991,**93**:55～92.

3 Wadler P. Comprehending Monads. Mathematical Structures in Computer Science, 1992,**2**:461～493.

4 Wadler P. The essence of functional programming. In: ACM Symposium on Principles of Programming Languages, New Mexico, ACM, 1992. 1～14.

5 Wadler P. Monads and functional programming. In: Broy M ed. Program Design Calculi, Proceeding of the Marktoberdof Summer School, Springer Verlag, 1993. 1～32.

6 Peyton Jones P L, Wadler P. Imperative functional programming. In: ACM Symposium on Principles of Programming Language, South Carolina, ACM, 1993. 71～84.


# PURE FUNCTIONAL PROGRAMMING BASED ON MONADS


Yuan Huaqiang Sun Yongqiang

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



**Abstract** Philip Wadler introduced the constructing course of a simple lexical parser when he studied how to structure pure functional programs by a monadic approach. This paper studies the approach further. A pure functional parser which can process complicated layout analyses, lexical analyses and syntax analyses is given according to the monadic approach.



**Key words** Functional programming, pure functional language, Monad, parsing technique.