Monad 的一种自动生成技术*

吕江花+, 金成植

(吉林大学 计算机科学与技术学院,吉林 长春 130012)

An Automatic Generation Technique for Monad

LÜ Jiang-Hua+, JIN Cheng-Zhi

(College of Computer Science and Technology, Jilin University, Changchun 130012, China)

+ Corresponding author: Phn: 86-431-5166480 ext 101, E-mail: lvjianghua@sina.com

http://www.jlu.edu.cn

Received 2003-03-14; Accepted 2003-08-18

Lü JH, Jin CZ. An automatic generation technique for Monad. *Journal of Software*, 2003,14(12):1989~1995.
http://www.jos.org.cn/1000-9825/14/1989.htm

Abstract: The main focus in Monad-oriented programming is on a set of Monad definitions and a Monad can be defined in an MAP style or a BIND style. Given a Monad library, it can be used directly if it meets the style of the users' need, otherwise, a new style of Monad should be reconstructed, which is a fussy job and even more the new one may not always satisfy the axioms of a Monad definition. However if an automatic Monad generator is added to the library, users can use a Monad freely without asking whether a new style of Monad is needed or how to construct a Monad in order to satisfy the several axioms of the Monad definition, which not only benefits users but also extends the original Monad library. In this paper an automatic Monad generator from other style of Monad is designed and implemented in Haskell. The generating arithmetic is based on the identity relationship between two Monad definitions.

Key words: Monad; BIND Monad; MAP Monad; category; high-order function

摘 要: Monad 程序的核心是一组 Monad 定义.Monad 定义分为 MAP 型和 BIND 型.如果在 Monad 库中已有所需要的 Monad 定义型,则可以直接使用,而不需要重新构造;否则,需要重新构造.但如果在 Monad 程序设计环境中增加从一类 Monad 构造另一类 Monad 的自动生成器,那么既方便了用户也扩充了 1 倍原有的 Monad 库.鉴于这种思想,用支持 Monad 程序设计的高阶函数语言 Haskell 实现了一个 Monad 的自动生成系统.另外,用户构造 Monad 不仅要花费较多的时间,而且写出的 Monad 多态函数往往不满足 Monad 所需满足的几条公理,因此,从这方面也可以看出,从一种类型的 Monad 自动产生另一种类型的 Monad 的重要意义.

关键词: Monad;BIND 型 Monad;MAP 型 Monad;范畴;高阶函数

中图法分类号: TP301 **文献标识码**: A

Monad 思想在 20 世纪 60 年代起源于范畴论,自 Monad 思想诞生时起,近 30 年主要在范畴论理论自身中发

* Supported by the National Natural Science Foundation of China under Grant No.60073041 (国家自然科学基金)

第一作者简介: 吕江花(1975－),女,山东日照人,博士生,助教,主要研究领域为语义转换,反射技术.

展,未能真正地将它与计算机程序联系起来.1989 年,Moggi 发表了一篇题目为“Computational lambda calculus and monad”的论文[1],其中论述了基于范畴理论中 Monad 概念的计算的范畴语义.此后,很多人加入了 Monad 的研究行列,并取得了大量实质性的研究成果[2~6].

Monad 技术是近年来才兴起,并且具有重要理论价值和广泛引用前景的、孕育着新思想和新方法的崭新技术.Monad 技术已发展成大家所能接受的通用程序设计技术[7],并且已经证实 Monad 具有很强的描述能力.Wadler 等人研制的支持 Monad 式程序设计的高阶函数式语言 Haskell 及其解释器和编译器的问世,实际上已经把我们推向了 Monad 式程序设计的时代.

Monad 具有高度的抽象性、反射性[8]、重用性和模块性,又具有易于自动实现扩充和修改的特点[9],这些特点可望对软件体系结构和软件构件技术带来积极的影响.Monad 表示法具有简练性和易读性,这主要归功于它把表结构的 Comprehension 技术推广到了任意 Monad 上,这也是 Monad 技术的一大贡献.

1 Monad 基本概念

范畴 C 是这样一个集合,对于任何 $A,B\in C$,都有 $f:A\to B$,称为 A 到 B 的射,并且定义了射的复合操作,它满足结合律;同时,对于任何对象 $A\in C$ 都有单位射 $id_A:A\to A$,使得对任意给出的射 $f:B\to D$ 都有 $f(id_B(b))=id_D(f(b))$.函子是范畴到范畴的一个映射:$F:C1\to C2$,如果 $C1=C2$,则称 F 为内函子.通常范畴表示类型的集合,即范畴的元素是一个类型.假设给定一个范畴 C 和其上的内函子 T,则可定义 T 上的 Monad,它分为 MAP 型和 BIND 型两大类.

MAP 型 Monad 是一个四元组 $M=(T,unit,map,join)$,其中 T 是某范畴上的函子,即一个类型转换器 $T:C1\to C2$,$unit^M$ 和 map^M 以及 $join^M$ 是如下基调的多态函数:

$unit^M:\alpha\to T(\alpha)$

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

$join^M:T(T(\alpha))\to T(\alpha)$

只要给出 T 的具体定义以及 $unit^M$,map^M 和 $join^M$ 的多态函数定义(称为 Monad 基本函数),就定义了一个 Monad.这些函数必须满足下面的条件(为了简单起见,有时省略上标 M):

$map\ id_\alpha=id_{T\alpha}$ (MR1)

$map\ (f\circ g)=(map\ f)\circ(map\ g)$ (MR2)

$join_\alpha\circ unit_{T\alpha}=id_{T\alpha}$ (MR3)

$join_\alpha\circ(map\ unit_\alpha)=id_{T\alpha}$ (MR4)

$join_\alpha\circ join_{T\alpha}=join_\alpha\circ(map\ join_\alpha)$ (MR5)

下面是 MAP 型 Monad 的一个实例:

$M=(ST,unit^M,map^M,join^M)$

$datatype\ ST(\alpha)=S\to(\alpha,s)$

$unit^M:\ \alpha\to ST(\alpha)$

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

$join^M:\ ST(ST(\alpha))\to ST(\alpha)$

$unit^M a=\lambda s.\ (a,s)$

$map^M\ fm'=\lambda s.$ **let** $(x,s0)=m's$ **in** $(fx,s0)$

$join^M\ m''=\lambda s.$ **let** $(m',s1)=m''s$ **in let** $(x,s0)=m's$ **in** $(x,s0)$

另一个 Monad 型,即 BIND 型 Monad 是一个三元组 $N=(T,unit,bind)$,其中 T 和 $unit$ 的含义与 MAP 型 Monad 情形基本相同;$bind$ 是具有如下基调的多态函数:

$$\_bind^N\_:T(\alpha)\to(\alpha\to T(\beta))\to T(\beta).$$

这里主要是把函数 map 和 $join$ 合并为一个函数 $bind$.为了醒目起见,通常用☆符号代替 $bind^N$.对于 BIND 型 Monad 来说,要求满足下面 3 条公理:

• LeftUnit: $unit\ a\☆\lambda x.(kx)=ka$ (BR1)

• RightUnit: $m\☆\lambda x.(unit\ x)=m$ (BR2)

• Associative: $(m\☆\lambda x.(kx))\☆(\lambda y.hy)=m\☆(\lambda x.(kx\☆(\lambda y.hy)))$ (BR3)

下面是 BIND 型 Monad 的一个例子:

$N=(ST,unit^N,bind^N)$

datatype $ST(\alpha)=S\rightarrow(\alpha,s)$

$unit^N:\alpha\rightarrow ST(\alpha)$

$unit^N:a=\lambda s.(a,s)$

$bind^N:ST(\alpha)\times(\alpha\rightarrow ST(\beta))\rightarrow ST(\beta)$

$m\ bind^N\ k=\lambda s.$ **let** $(a,s0)=ms$ **in** $ka\ s0$

Monad 的这两种定义形式的描述能力是等价的.bind 型中的 bind 函数的计算过程在 Map 型 Monad 中被拆成两个函数 map 和 join 分别表示,在不同的应用程序中,因为使用和理解方便而采用的方式也可能不同.例如,在类型构造等应用中使用 Map 型 Monad 描述比较方便、直接,而 bind 型 Monad 主要用于解释器的构造、语言的语义描述等应用中.

2 面向 MONAD 语言——MOL

顾名思义,面向 Monad 语言是基于 Monad 机制的一种语言.目前 Monad 机制仅限于函数式语言,而且还没有纯粹意义上的使用型 Monad 语言,但有支持 Monad 程序设计的语言,这就是由 Wadler 等人研制的 Haskell 语言.目前已有在网上提供的 Haskell 解释器和编译器,因此,实际上我们已经进入了面向 Monad 的程序设计时代.

下面是一个 MOL 语言程序的小实例:

```
monad M
    dataType ST x=S→(x,S)
    unit a=λs.(a,s)
    map fm=λs.[(fa,s1)|(a,s1)←ms]^Id
    join m=λs.[(a,s2)|(m1,s1)←ms,(a,s2)←m1 s1]^Id
end
data Exp=Var Ix|Const Val|Plus Exp Exp
data Com=Asgn Ix Exp|Seq Com Com| If Exp Com Com
data Prog=Program Com Exp
func fetch::ST S   (定义部分省略)
func exp::Exp→ST(Val)
    exp(Var i)=[v|v←fetch i]^ST
    exp(Const v)=[v]^ST
    exp(Plus e1 e2)=[v1+v2|v1←exp e1,v2←exp e2]^ST
func com::Com→ST()   (定义部分省略)
func prog::Prog→Val
    prog(Prog c e)=init() [v|()←com c, v←exp e]^ST
objexp   (省略)
```

3 MAP 型到 BIND 型 Monad 的自动生成技术

MAP 型 Monad 包含 4 个部分:类型构造子(函子)部分、UNIT 多态函数部分、MAP 多态函数部分和 JOIN 多态函数部分.BIND 型 Monad 则包含 3 个部分:类型构造子(函子)部分、UNIT 多态函数部分和 BIND 多态函数部分.其中很重要的一点是,多态函数必须满足一些公理:MAP 型 Monad 要满足公理(MR1)~(MR5),而 BIND

型 Monad 则要满足公理(BR1)~(BR3).

我们用 MAP-Monad 表示一个 MAP 型 Monad,用 BIND-Monad 表示一个 BIND 型 Monad.因为一个 Monad 既不是表达式,也不是一个函数,因此,不可能设想被产生的 Monad 与原 Monad 如何等价,但也不能不加任何条件.这种条件首先是针对某个相同函子 T 而言的,其次 unit 多态函数应是等价的,另外,MAP-Monad 中的 map 以及 join 函数和 BIND-Monad 中的 bind 函数,对任意给出的 $m\in T(\alpha),k\in\alpha\to T(\beta)$,都应满足下面关系式($\alpha$和$\beta$是任意类型):

$$m\ bind\ k=join(map\ km). \tag{1}$$

注意,bind 函数的基调与 map 基调、join 基调彼此不相同,因此,单个函数没有可比性.而等式(1)的左右表达式的值都属于 $T(\beta)$域,因此是可比的.其中($m\ bind\ k$)的计算过程是:首先从 $m\in T(\alpha)$取出α型部分的值(设其为 a),然后再把它传给函数 k,即计算 $k(a)$,其结果是某个值 $m1\in T(\beta)$.而对于等式(1)右部($join(map\ km)$)来说,首先要计算($map\ km$),因为 $m\in T(\alpha),k\in(\alpha\to T(\beta))$,故有$(map\ km)\in T(T(\beta))$,进而有 $join(map\ km)\in T(\beta)$.

假设有下面的 MAP-Monad:

$M=(T,unit,map,Join)$

- $T\alpha=\tau[\alpha]$
- $unit^M::\alpha\to T(\alpha)\quad unit^M\ a=UnitBody^M[a]$
- $map^M::(\alpha\to\beta)\to(T(\alpha)\to T(\beta))\quad map^M\ fm=MapBody^M[f,m]$
- $join^M::T((T(\alpha))\to T(\alpha)\quad join^M\ m=JoinBody^M[m]$

并考虑从 MAP 型 Monad 自动生成 BIND 型 Monad 的问题.具体来说,要从给定 MAP-Monad 自动构造出下面的形式的 BIND-Monad,其条件是类型构造器 T 相同,并且自动构造出的基本函数 $unit^N$ 和 $bind^N$ 要满足等式(1),同时还要满足公理(BR1)~(BR3).

- $N=(T,unit,bind)$
- $T\alpha=\tau[\alpha]$
- $unit^N::\alpha\to T(\alpha)\quad unit^N\ a=UnitBody^N[a]$
- $bind^N::T(\alpha)\to(\alpha\to T(\beta))\to T(\beta)\quad m\ bind^N\ k=bindBody^N[k,m]$

基本函数定义包括两部分:一部分是函数的基调定义,另一部分是函数方程的定义.$unit^N$ 定义式的构造是比较简单的事情,即只要定义

$$unit^N\ b=UnitBody^M[b/a]$$

即可,其中 a 是 $unit^M$ 的参数.当然,最节省的办法是如下定义:

$$unit^N\ a=UnitBody^M[a].$$

这样,主要问题是 $bind^N$ 定义式的构造问题.这里需要 $MapBody^M[f,m]$和 $JoinBody^M[m]$相互合作定义一个 $bindBody^N[k,m]$.

因为被构造的 BIND-Monad 必须满足等式(1),因此,我们将从等式(1)出发考虑我们的问题.关键是等式(1)的右部表达式不能作为 bind 函数定义式中的右部表达式,因为其中没有 join 和 map 的函数定义.因此,必须独立地给出构造 $BindBody^N[k,m]$的方法.这里值得注意的是,两类 Monad 的基本函数具有不相同的基调,这是自动生成过程中的最大障碍之一.事实上,从式(1)可以导出函数体之间的如下等式关系:

$$BindBody^N[m',k']=JoinBody^M[(MapBody^M[k\ `/f,m'/m])/m] \tag{2-1}$$

$$BindBody^N[m',k']=(\lambda m.JoinBody^M[m])(MapBody^M[k\ `/f,m'/m]) \tag{2-2}$$

上面几个等式都是等价的,只是表示方法不同而已,它们表示 $BindBody^N$ 可以用 $JoinBody^M$ 和 $MapBody^M$ 来定义.我们可以根据需要选择其中某一等式.

下面是从 MAP-Monad 的基本函数定义到 BIND-Monad 基本函数定义的生成规则,其中 $ToBind【E】$ 表示从 E 自动生成的 BIND 型表达式,而 E 则表示 MAP 型表达式;$E[a]$表示 E 中含 a,$E[E'/a]$表示将 E 中 a 替换为 E'.下面首先是函数定义式的构造规则:

- $unit^N::\alpha\to T\alpha$

$unit^N\ a=ToBind\ 【\ UnitBody^M[a]\ 】$

- $bind^N::T\alpha\rightarrow(\alpha\rightarrow T\beta)\rightarrow T\beta$

  $m\ bind^N\ k=(\lambda m.ToBind\ 【\ JoinBody^M[m]\ 】)(ToBind\ 【\ MapBody^M[k/f,m/m]\ 】)$

其次是表达式的转换规则:

$ToBind\ 【\ v\ 】=v$

$ToBind\ 【\ E1+E2\ 】=ToBind\ 【\ E1\ 】+ ToBind\ 【\ E2\ 】$

$ToBind\ 【\ E1\ E2\ 】=ToBind\ 【\ E1\ 】\ ToBind\ 【\ E2\ 】$

$ToBind\ 【\ \textbf{let}\ p=E1\ \textbf{in}\ E2\ 】=\textbf{let}\ p=ToBind\ 【\ E1\ 】\ \ \textbf{in}\ ToBind\ 【\ E2\ 】$

$ToBind\ 【\ (E0\rightarrow E1,E2)\ 】=(ToBind\ 【\ E0\ 】\rightarrow ToBind\ 【\ E1\ 】,ToBind\ 【\ E2\ 】)$

$ToBind\ 【\ unit\ E\ 】=unit\ ToBind\ 【\ E\ 】$

$ToBind\ 【\ map\ E1\ E2\ 】=ToBind\ 【\ E2\ 】\ bind(\lambda a.\ unit(ToBind\ 【\ E1\ 】\ a))$

$ToBind\ 【\ join\ E\ 】=ToBind\ 【\ E\ 】\ bind(\lambda m.\ m)$

$ToBind\ 【\ [E|Quali]^M\ 】=[ToBind\ 【\ E\ 】\ |ToBindQ\ 【\ Quali\ 】\ ]^M$

$ToBindQ\ 【\ \Lambda\ 】=\Lambda$

$ToBindQ\ 【\ x\leftarrow E\ 】\ x\leftarrow ToBind\ 【\ E\ 】$

$ToBindQ\ 【\ Quali,Quali\ 】=ToBindQ\ 【\ Quali\ 】,ToBindQ\ 【\ Quali\ 】$

考虑在前面曾经定义过的 Monad 例子 $M=(ST,unit^M,map^M,join^M)$,主要看从 Map 函数体和 Join 函数体出发构造 bind 函数体的过程.对于 M,有

$$UnitBody^M[a]=\lambda s.\ (a,s),$$

$$MapBody^M[f,m]=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ (fx,s0),$$

$$JoinBody^M[m]=\lambda s.\ \textbf{let}\ (m',s')=ms\ \textbf{in let}\ (x'',s'')=m's'\ \textbf{in}\ (x'',s'').$$

我们应该得到的是如下的两个函数体:

$$UnitBody^N[a]=\lambda s.(a,s),$$

$$BindBody^N[k,m]=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ kx\ s0.$$

其中,$UnitBody^N[a]$部分是显而易见的,而 $BindBody^N[k,m]$部分,则可使用(2)和 $ToBind\ 【\ E\ 】$ 构造规则导出来,其具体推导过程如下所述:

$$\begin{aligned}BindBody^N[k,m]&\Rightarrow(\lambda m.\ ToBind\ 【\ JoinBody^M[m]\ 】)\ (ToBind\ 【\ MapBody^M[k/f,\ m/m]\ 】)\\&\Rightarrow\lambda s.\ \textbf{let}\ (m',s')=(\textbf{let}\ (x,s0)=ms\ \textbf{in}\ (\ kx,s0))\ \textbf{in let}\ (x'',s'')=m's'\ \textbf{in}\ (x'',s'')\\&\Rightarrow\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in let}\ (m',s')=(\ kx,s0)\ \textbf{in let}\ (x'',s'')=m's'\ \textbf{in}\ (x'',s'')\\&\Rightarrow\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ kx\ s0.\end{aligned}$$

显然,得到了所要得到的结果.注意,其中经过了优化.

4 BIND 型到 MAP 型 Monad 的自动生成技术

考虑从 BIND-Monad 出发构造相应 MAP-Monad 的问题.这里的实质性问题是,如何从给定 $BindBody[m,k]$ 出发构造出 $MapBody[f,m]$和 $JoinBody[m]$的问题.首先利用关系式(1)可以导出下面关系式:

$$map^M\ fm=m\ bind^N\ \lambda a.\ unit(fa), \tag{3}$$

$$join^M\ m=m\ bind^N\ (\lambda m'.\ m'). \tag{4}$$

但是,式(3)和式(4)的右部并不能直接作为 $MapBody[f,m]$和 $JoinBody[m]$,因为在这两个函数体中都不能出现 $bind^N$ 函数名.关系式(3)和式(4)为我们提供了导出 $MapBody[f,m]$和 $JoinBody[m]$的理论依据,不难看出,我们要从一个 $BindBody[m,k]$函数体出发导出两个函数体.具体可以导出下面的关系:

$$MapBody[f,m']=ToMap\ 〚\ BindBody[m'/m,\ \lambda a.\ unit(fa)/k]\ 〛, \tag{5}$$

$$JoinBody[m']=ToMap\ 〚\ BindBody[m'/m,\ (\lambda m'.\ m')/k]\ 〛. \tag{6}$$

其中,$ToMap\ 〚E〛$ 表示构造出的 MAP 型表达式,E 则表示 BIND 型表达式.在转换时主要用到了以下规则:

$$ToMap\llbracket E1\ bind\ E2\rrbracket=join(map(ToMap\llbracket E2\rrbracket)(ToMap\llbracket E1\rrbracket)).$$

其他部分和 $ToBind【E】$情形基本类同,在此不再赘述.综上所述,从给定 BIND-Monad 出发构造 MAP-Monad 的方法如下(其中函子即类型构造子 T 取相同的定义):

- $unit^M::\alpha\to T(\alpha)$
  $unit^M\ a=ToMap\llbracket UnitBody^N[a]\rrbracket$
- $map^M::(\alpha\to\beta)\to T(\alpha)\to T(\beta)$
  $map^M\ fm=ToMap\llbracket BindBody[m/m,\lambda a.unit(fa)/k]\rrbracket$
- $join^M::T(T(\alpha))\to T(\alpha)$
  $join^M\ m=ToMap\llbracket BindBody[m/m,(\lambda m'.m')/k]\rrbracket$

作为例子,还是要考虑前面曾经讨论过的 Monad,其中 bind 函数的定义如下:

$$m\ bind\ k=\lambda s.\ \textbf{let}\ (x,s0)=ms\quad \textbf{in}\ kx\ s0.$$

据此和函数的构造规则,可以求出 Map 和 Join 函数的体部分:

$$MapBody[f,m]=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ unit(fx)\ s0,$$
$$JoinBody[m]=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ x\ s0.$$

于是,最后可以得到如下函数定义式:

$$map\ fm=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ \text{unit}\ (fx)\ s0,$$
$$join\ m=\lambda s.\ \textbf{let}\ (x,s0)=ms\ \textbf{in}\ x\ s0.$$

5 验证 Monad 条件

每个 Monad 必须满足相应的条件,MAP-Monad 要满足条件(MR1)~(MR5),BIND-Monad 则要满足条件(BR1)~(BR3).由于篇幅所限,我们将只证明,从 MA-Monad 出发构造出来的 BIND-Monad 满足条件(BR1)~(BR3).也就是说,假定 MAP-Monad 满足条件(MR1)~(MR5),并证明构造出来的 BIND-Monad 满足条件(BR1)~(BR3).我们容易导出下面一个辅助等式:

$$(map^M\ f)\circ unit^M=unit^M\circ f. \tag{MR6}$$

首先,考察条件(BR1),其具体推导过程如下:

$$\begin{aligned} unit\ a\ \ ☆\ \ \lambda x.(kx)&=BindBody[unit\ a,k] && \text{根据☆的定义式}\\ &=join(map\ \lambda x.\ (kx)(unit\ a)) && \text{根据 BindBody 的定义}\\ &=join(unit(ka)) && \text{根据(MR6)}\\ &=ka && \text{根据(MR3)} \end{aligned}$$

因此,根据(BR1)的定义,BIND-Monad 满足(BR1).

其次,证实 BIND-Monad 满足(BR2)条件,推导过程如下:

$$\begin{aligned} m\ \ ☆\ \ \lambda x.(unit\ x)&=BindBody[m,\ \lambda x.(unit\ x)] && \text{根据☆的定义式}\\ &=join(map\ \lambda x.(unit\ x)m) && \text{根据 BindBody 的定义}\\ &=join(map\ unit\ m) && \text{根据}\eta\text{变换}\\ &=m && \text{根据(MR4)} \end{aligned}$$

最后,要证实满足(BR3)条件.以下 k 和 h 均属于空间$(\alpha\to T\beta)$,因此可以假设 $k=\lambda a.\ unit\ (fa)$,$h=\lambda b.\ unit(gb)$.下面分别是(BR3)的左部和右部的推导过程:

$$\begin{aligned} (m☆k)☆h&=BindBody[BindBody[m,k],h]\\ &=join(map\ h\ join(map\ km))\\ &=join(map(\lambda b.\ unit(gb))join(map(\lambda a.\ unit(fa))m))\\ &=map\ g(map\ fm)\\ &=map(g\circ f)m \end{aligned}$$

$$
\begin{aligned}
m\☆\lambda a.(ka\☆h) &= BindBody[\lambda a.BindBody[ka,h],m] \\
&= join(map(\lambda a.\ join(map\ h(ka)))m) \\
&= join(map(\lambda a.\ join(map\ (\lambda b.\ unit(gb))(unit(fa)))m) \\
&= join(map(\lambda a.\ (map\ g(unit(fa)))m) \\
&= join(map(\lambda a.\ unit(g(fa)))m) \\
&= map(g \circ f)m
\end{aligned}
$$

显然,(BR3)的左部和右部相等,因此,被构造的 BIND-Monad 满足(BR3)条件.

6 结束语

Monad 程序设计环境最核心的部分是 Monad 库和函数库,而 Monad 有 MAP 型和 BIND 型两大类,而且每种 Monad 都需要满足几条公理,在人工书写 Monad 时,有时往往不满足所规定的公理,因此这里所提供的异类 Monad 之间的转换技术是非常实际而有用的技术.在转换技术中最重要的原则是使被产生的 Monad 满足相应的公理.

References:

[1] Moggi E. Computational lambda-calculus and monads. Technical Report, ECS-LFCS-88-66, Department of Computer Science, Edinburgh University, 1988.

[2] Wadler P. Comprehending Monads. Mathematical Structures in Computer Science, 1992,2(3):461~493.

[3] Walder P. Monad and composable continuations. Lisp and Symbolic Computation, Special Issue on Continuations, 1994,7(1): 39~56.

[4] Benton N, Walder P. Linear logic, Monads and lambda calculus. In: Proceedings of the 11th IEEE Symposium on Logic in Computer Science. New Brunswick: IEEE Computer Society Press, 1996. 420~431.

[5] Yuan Q. Research on Monad theory and its application [Ph.D. Thesis]. Changchun: Jilin University, 2000 (in Chinese with English abstract).

[6] Yuan Q, Jin CZ. Transforming Monad representing to OO program. Journal of Computer Research and Development, 2000,37(6): 668~671 (in Chinese with English abstract).

[7] Walder P. The essence of functional programming (invited talk). In: Sethi R, ed. Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages. Albuquerque, ACM press, 1992. 1~14.

[8] Filinski A. Representation Monads. In: Wing JM, ed. Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Oregon: ACM Press, 1994. 446~457.

[9] Hook J, Kieburtz R, Shear T. Generating programs by reflection. Technical Report, 93-01, OGICSE, 1993.

附中文参考文献:

[5] 袁琦.Monad 理论及其应用研究[博士学位论文].长春:吉林大学,2000.

[6] 袁琦,金成植.Monad 的面向对象程序的自动生成.计算机研究与发展,2000,37(6):668~671.