函数式语言编译实现技术的研究*

廖湖声

(北京工业大学计算机学院,北京 100044)

摘要 文章分析了编译实现函数式程序设计语言的主要技术,总结出函数式语言实现方法的两条设计准则:(1)简化函数调用处理的同时,减少被延迟求值参数所占用存储空间及其占用时间;(2)通过程序变换减少函数式程序的特殊语言现象,使其适应冯·诺依曼机的结构特征. 文中介绍了几种新的编译技术以及采用这些技术实现的一个函数式语言编译系统.

关键词 函数式程序设计语言,编译程序,延迟求值,部分求值.

编译实现方法的研究一直是函数式语言领域的重要研究课题. 函数式语言以函数为基本计算模型,允许使用高阶函数,使得程序执行中函数调用次数过多,参数传递频繁. 延迟求值方式的采用迫使系统必须使用内存单元来保存被推迟计算的实在参数表达式. 这种需求将长时间地占用大量存储空间,降低了系统处理能力. 部分参数化的存在更增加了及时释放无用单元的困难. 这些因素都限制了函数式程序执行效率的提高.

本文在分析函数式程序实现方法的基础上,总结出函数式语言实现方法的两条设计准则,介绍了我们提出的几种编译技术以及利用这些技术实现的一个函数式语言编译系统. 性能测试表明这些技术的使用有效地提高了函数式程序的执行效率.

1 函数式语言的编译实现

现有的函数式语言编译实现方法可分为两类. 一种是图归约方法(graph reduction)[1],程序执行中函数定义表达式被表示为图,通过图归约完成表达式的求值. 这种方法的弱点在于函数调用中按照函数定义生成图的需求直接影响了程序执行效率. 另一种方法是采用类似于SECD机的中间语言抽象机[2],采用一个求值环境来保存变量的值,函数和延迟求值的实在参数被表示为表达式及其求值环境组成的闭包(closure);函数式程序被变换成抽象机指令,程序执行按照抽象机指令进行求值环境的维护、参数传递和函数调用. 这种方法的缺点是生成与保存求值环境的开销对程序执行的时空效率影响很大. 两类方法的共同点是力图简化函数调用的处理,从而提高程序的执行速度.

* 本文1994-07-06收到,1994-09-19定稿

本研究项目得到北京市自然科学基金资助. 作者廖湖声,1954年生,副教授,主要研究领域为函数式程序设计,编译技术,自动程序设计.

本文通讯联系人:廖湖声,北京100044,北京工业大学计算机学院

在编译处理之前采用某种程序变换技术也是常见的方法.通过对程序的静态分析将程序变换成易于编译实现的形式,以求优化计算过程,简化编译设计,最终达到提高函数式程序执行效率的目的.然而,采用程序变换也会带来程序膨胀,可能抵消程序变换带来的好处.

通过研究函数式语言实现方法的实践,我们认为函数式语言实现方法的设计应当遵循以下两条准则:

(1) 实现模型的设计不仅应该尽量简化函数调用和延迟求值的处理,而且应该保证尽快地释放无用单元,以求从执行速度和存储空间的动态分配两个方面提高程序的执行效率.

(2) 使用程序变换的目的在于尽可能减少函数式语言的特殊语言现象,使得变换后的程序形式尽可能接近传统的过程型语言,从而适应冯·诺依曼机的结构特征.

参照以上两条准则,我们提出了几种编译技术并且应用于函数式语言FSL的编译设计和实现中.

2　一个函数式语言的编译系统

FSL语言是我们为了发展函数式语言在软件规格说明中的应用所设计的一种函数式语言.语言中提供了高阶函数、部分参数化、集合表示、等式函数定义、序列运算和流式输入输出处理等功能,其语法规则如表1.

表1　FSL语言的主要语法规则

语法域:		
expr	∈Exp	程序表达式
s_expr	∈Sexp	简单表达式
b_expr	∈Bexp	基本表达式
decl	∈Decl	定义表达式
var	∈Var	变元表达式
x	∈Ide	标识符
c	∈Bas	常数
vars	∈Vars	变元表达式表
exprs	∈Exps	程序表达式表
qual	∈Qual	集合条件式
抽象语法:		
expr	::=s_expr|FUN var … var . expr| s_expr WHERE {decl AND … AND decl}| s_expr WHEREREC {decl AND … AND decl}	
s_expr	::=b_expr|IF s_expr THEN s_expr ELSE s_expr| s_expr OP2 s_expr|OP1 s_expr| {s_expr : qual … qual }|s_expr OP2…	
b_expr	::=x|c|(expr)|x s_expr…s_expr|{exprs}	
decl	::=var=expr|x var … var=expr	
var	::=x|x…|[vars]|[vars]…	
vars	::=var|var|var, vars|ε	
exprs	::=s_expr;s_expr|s_expr, exprs|ε	
qual	::=s_expr|s_expr→var	
注释:		
OP2	表示双目运算符	
OP1	表示单目运算符	

以下程序例是计算所有素数的FSL程序:

```
sieve [2…]
whererec
    sieve (p,x) = p, sieve {n: x→n; n%p != 0}
```

其中[2…]表示从2开始的自然数序列，{n: x→n; n%p!=0}表示 x 中所有不能被 p 整除的整数.

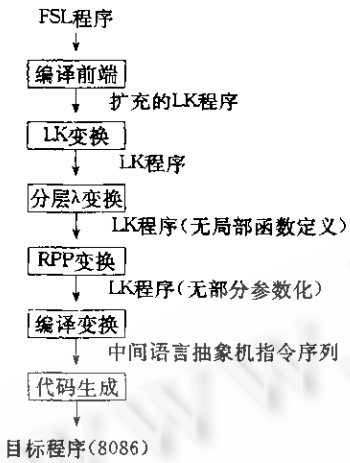

图1 FSL编译系统

FSL 语言的编译系统是一个多级程序变换系统(见图1). 系统的主要特点在于:

1. 通过程序变换逐级地消除函数式程序的特殊语言现象. 变换后的程序中没有局部函数定义，也不存在部分参数化的函数调用. 变换后的函数式程序和过程型程序的不同之处只有高阶函数和延迟求值方式.

2. 在编译变换中对实在参数采用了部分求值，减少了被延迟求值的参数，并且使求值环境中的无用单元及时得到释放，尽可能地减少了保留延迟求值参数所占用的存储空间和时间.

3. 由于部分参数化的消除和实在参数部分求值的采用，使得中间语言抽象机的改进成为可能. 和传统的 SECD 机相比，新的中间语言抽象机的求值环境均采用了连续的存储空间来实现，从而实现了求值环境参数的常数时间存取，明显地提高了程序的执行效率.

在编译系统中，编译前端程序作为第一级处理，用于源程序的词法分析和语法分析，生成以 S 表达式形式表示的中间语言.

在生成抽象机指令之前，引入了三级程序变换，以求减少程序中的特殊语言现象. LK 变换程序将 FSL 语言中等式函数定义、参数模式匹配、集合表示和序列运算等特殊计算变换成等效的 λ 表达式，生成以 S 表达式为表示形式的 LK 程序. 第三级的分层 λ 变换程序分析程序中可能出现的重复计算，通过 λ 抽象引入新的函数定义，消除重复计算，并且将所有局部函数变换成全局函数. 第四级的 RPP 程序变换程序用于消除程序中的部分参数化，根据对程序中函数调用的分析和变换，使得所有函数调用的实在参数和虚拟参数的个数相同，为简化抽象机设计创造条件.

经过多级变换的程序被编译变换程序变换为抽象机指令序列. 处理中对实在参数采用了部分求值技术，优化实现了延迟求值方式. 最后，代码生成程序将各个抽象机指令依次变换为相应的8086汇编指令，形成目标程序.

以下各节介绍系统中采用的编译技术.

3 分层 λ 变换

分层 λ 变换是一种程序变换技术，用于消除高阶函数和部分参数化应用中出现的重复计算[3]. 所谓部分参数化就是在函数调用中使用少于虚拟参数的实在参数. 例如，如果有函数式程序：

```
func1 x y=(x+3) * y
```

```
func2=func1 12
```

函数func2定义为函数func1的部分参数化函数调用.使用中,例中的加法计算将重复出现在函数func2的每次调用中.通过引入一个新的函数func3,可将该程序变换为

```
func1 x =func3(x+3)
func3 n y=n * y
func2=func1 12
```

使得通过函数调用func1 12的求值能够得到func3 15,从而消除了重复计算.

然而,这个变换仅仅在部分参数化func1 12存在时才是必要的;否则,只会带来不必要的程序膨胀.所谓分层λ变换就是首先统计程序中部分参数化的信息,然后仅仅对使用部分参数化的函数进行以上程序变换,从而消除了重复计算并且避免了不必要的程序膨胀.这是本方法区别于其他λ抽象方法的主要特点.分层λ抽象算法的核心部分如表2.

表2　分层λ抽象算法的核心部分

函数定义说明:

$u \in 2Dec$　　$d \in Dec$　　$d ::= x == e$

部分参数化信息:

$w \in 2N$

分层λ抽象规则:

$M: Exp \to 2^N \to (Exp \times 2^{Dec})$

$M[b]w = b$

$M[x]w = x$

$M[e_0, e_1]w = (\langle e_0'\ e_1' \rangle, u_0 \cup u_1)$
　where $\langle e_0', u_0 \rangle = M[e_0]w\{\}$
　　$\langle e_1', u_1 \rangle = M[e_1]w\{\}$

$M[$lambda $(x_1 \cdots x_n)\ e]w =$
　if $m \geqslant n$ or $w = \{\}$ then
　　$(($lambda $(x_1 \cdots x_n)\ e'), u)$
　　where $\langle e', u \rangle = M[e]w'$
　else $(($lambda $(x_1 \cdots x_m)\ e'), u \cup u'')$
　　where $\langle e', u \rangle = M[e'']w'$
　where $m = \min(w)$
　　$w' = \{l - m \mid l \leftarrow w, l \rangle m\}$
　　$\langle e'', u'' \rangle = Lamlf[$lambda $(x_{m+1} \cdots x_n)\ e]$

$M[$letrec $e_0(x_1, e_1) \cdots (x_n, e_n)]w =$
　$\langle e_0'', u'' \cup u_0' \cup \cdots \cup u_n' \rangle$
　where $e =$ letrec $e_0(x_1, e_1) \cdots (x_n, e_n)$
　　$\langle e_0'', u'' \rangle = Lamlf[$letrec $e_0'\ (x_1, e_1') \cdots (x_n, e_n')$
　　$\langle e_i', u_i' \rangle = M[e_i]Getp[e]x_i$ for $i = 0 \cdots n$

注释:

min(w)	用于求最小值
{e|c}	表示满足条件c的表达式e
$\langle e', u \rangle = Lamlf[e]$	用于完成表达式e的λ变换,返回变换后的表达式e'和变换中生成的函数定义集u
$w = Genp[e]x$	用于求函数x在表达式e中的实参个数集合

求素数程序例在经过编译前端和LK变换得到以下形式:

```
(letrec
   (sieve (from 2))
   (sieve lambda(v1)
      (let (cons p (sieve (filter x (lambda(n) (neq (rem n p) 0)))))
         (p hd v1) (x tl v1) )))
```

经过分层λ抽象变换成:

```
(letrec
   (sieve (from 2))
   (sieve lambda(v1) (f1 (hd v1) (tl v1)))
   (f1 lambda(p x) (cons p (sieve (filter x (f2 p)))))
   (f2 lambda(p n) (neq (rem n p ) 0)) )
```

在变换过程中，所有的局部函数被变换成全局函数，减轻了后续程序变换和编译处理的负担.

4 RPP 变换

RPP 变换是一种程序变换技术，用于消除程序中的部分参数化[4]. 如上所述，部分参数化为函数式语言提供了一种特殊的程序结构抽象功能，可以用于消除重复计算. 然而，部分参数化的存在增加了设计中间语言抽象机的困难.

对于图归约方法，部分参数化的存在迫使每次函数调用时必须检查参数栈上有无足够的实在参数；如果实在参数比虚拟参数少，则将现有的实在参数与相应的虚拟参数结合后，以图的形式生成一个函数，作为函数调用结果返回.

在基于求值环境的编译实现中，部分参数化的存在直接影响求值环境的结构设计. 例如，在以下程序中

```
prefix 0 c=c
prefix n c=λx. prefix (n−1) [x|c]
quad=prefix 4
exp1=quad [1]
exp2=qurd [2]
```

函数 exp1和 exp2是利用部分参数化的函数调用所生成的，执行中表示为表达式及其求值环境组成的闭包：

```
exp1⇒quad [1]
    ⇒prefix 4 [1]
    ⇒λx. prefix (n−1) [x|c] with n=4 and c=[1]
exp2⇒λx. prefix (n−1) [x|c] with n=4 and c=[2]
```

在求值环境中，参数 n 的值是在函数 quad 的计算中产生的，属于 quad 的求值环境；在函数 exp1和 exp2的求值环境中共享了 quad 的求值环境. 由于各求值环境的这种关系，迫使设计者只能采用链表来实现求值环境，而无法直接利用参数栈或某种数组结构.

在实践中，我们注意到只要没有部分参数化，就能够利用栈和数组来实现求值环境，从而简化参数的存取. 另一方面，通过程序变换能够使所有函数调用的实在参数个数和虚拟参数相同. 例如，对于函数

```
func = λx1 x2 x3 x4 x5. e
```

如果程序中存在函数调用 func s t 和 func a b c d，则将函数定义变换成

```
func = λx1 x2. λx3 x4. λx5. e
```

就能够使每次函数调用都具有足够的实在参数.

实现这种程序变换的主要困难在于需要找到所有部分参数化的函数调用，而函数调用可能是间接产生的，可能是通过参数传递或函数调用生成的函数完成的. 因此，需要对可能出现的所有间接函数调用进行静态分析. 为此，我们设计了间接函数调用的分析算法，实现了 RPP 程序变换. RPP 变换算法的核心部分见表3.

表3　RPP 程序变换算法的核心部分

部分参数化信息:
$w \in 2^N$
函数变换规则:
T:Exp→2^N→Exp
T[b]w=b
T[x]w=x
T[e_0 e_1]w=(e_0' e_1')
T[lambda ($x_1 \cdots x_n$) e]w=
if m≥n then
(lambda ($x_1 \cdots x_n$) e)
else
(lambda ($x_1 \cdots x_m$) T[]w')
where m=min(w)
w'={l−m|l←w,l>m }
RPP 规则:
R:Exp→Exp
R[b]=b
R[x]=x
R[e_0 e_1]=(e_0 e_1)
R[lambda ($x_1 \cdots x_n$) e]=(lambda ($x_1 \cdots x_n$) e)
R[letrec $e_0(x_1.e_1) \cdots (x_n.e_n)$]=(letrec $e_0(x_1.e_1') \cdots (x_n.e_n')$
where e=(letrec $e_0(x_1.e_1) \cdots (x_n.e_n)$)
e_i'=T[e_i]Getp[e]x_i for i=1…n
注释:
min(w)　用于计算最小值
Getp[e]x　用于计算函数 x 在表达式 e 中部分参数化的参数个数集合

求素数的程序例在经过 RPP 程序变换后得到以下形式:

```
(letrec
   (sieve (from 2))
   (sieve lambda(v1) (f1 (hd v1) (tl v1)))
   (f1 lambda(p x) (cons p (sieve (filter x (f2 p)))))
   (f2 lambda(p) (lambda(n) (neq (rem n p) 0))) )
```

其中,函数 f2的定义形式被改变.

5　实在参数的部分求值

在编译变换阶段,采用了一种新的实现方法来完成延迟求值.在函数式程序的执行中,所有数据结构都依赖于动态存储空间分配;而且在函数调用时为了保存被延迟求值的实在参数,还需要长时间地占用大量存储空间.如果处理中不能及时释放无用单元,将迫使程序反复调用无用单元收集程序,影响程序执行效率和系统处理能力.

针对函数式程序的这种特点,在程序执行中函数调用之前对实在参数采用了部分求值.这种参数部分求值的作用在于:(1)考虑到延迟求值的主要作用是处理无限数据结构,对诸如整数四则运算等不可能造成死循环的参数表达式直接进行求值;(2)对于一般的函数调用表达式,对其实在参数进行部分求值后,将该函数调用关系保存在一种专用数据结构中.这种方法的好处在于(1)对实在参数尽可能地采用了按值调用,提高了执行速度,减少了内存消耗;(2)实在参数的保存仅仅涉及该计算有关的数据,释放了当前求值环境中与实在参数

计算无关的数据所占用的存储单元，尽可能地减少了延迟求值所占用的存储空间以及占用的时间，同时为抽象机的改进创造了条件.

这种处理方法可以看作是图归约方法和求值环境方法的结合. 参数部分求值生成的数据结构组成了一个图. 在程序执行过程中被推迟计算的实在参数是以这种图的形式保存在求值环境中. 这时的图归约就是对实在参数的求值. 程序执行得益于程序运行环境的改善.

6 中间语言抽象机

如上所述，RPP 程序变换的引入使得编译设计者第一次能够在不考虑部分参数化的情况下，为函数式语言设计实现模型. 在 FSL 语言的中间语言抽象机设计中，我们利用程序中无部分参数化的特点，结合实在参数的部分求值方式，实现了求值环境元素的常数时间存取，并且改进了存储空间的动态分配方法.

这种中间语言抽象机是一种改进的 SECD 机，由5个部分组成：(1)参数栈，用于传递实在参数和保存计算的中间结果. (2)动态求值环境，用于保存约束变元的约束值. (3)静态求值环境，用于保存当前处理函数的自由变元的约束值. (4)程序计数器，用于指示当前指令. (5)运行栈，用于函数调用时保存程序计数器和静态求值环境指针.

这种抽象机和传统的 SECD 机的主要区别在于使用了两种求值环境. 由于在函数调用时对实在参数采用了部分求值，使得保存约束变元的存储单元仅用于函数定义表达式的求值过程中. 因此，对约束变元可以采用的栈式存储分配，从而自动释放了与变元约束值相关的无用单元. 函数定义中自由变元的约束值属于函数定义表达式的环境. 因此，其生存期取决于该函数的生存期. 由于使用 RPP 变换确定了每次函数调用的参数个数，使得在编译时能够计算出自由变元的个数，因此在生成函数时可以为其自由变元分配一个连续的存储空间. 这两种求值环境的实现使存取求值环境元素的计算量从 O(n)降低到常数(n 是元素个数).

因此，提高函数式程序执行速度的关键措施是采用 RPP 程序变换和实在参数部分求值使得设计高效率的中间语言抽象机成为可能.

7 性能测试

我们选用以下几个程序对 FSL 语言的各种功能进行了测试，并且用 C 语言完成相同算法的程序实现. 在同一硬件环境下(486/33)，程序执行速度(单位：秒)的比较如下：

	C 程序	FSL 程序	速度比
ack(3,8)	8.62	21.64	2.51
hanoi(12)	1.98	2.69	1.36
prime(400)	0.33	1.53	4.64
qsort(100)	0.16	0.28	1.75
queen(8)	0.61	2.20	3.61
tak(24,16,8)	8.06	22.63	2.81

以上测试采用的程序都比较小，在两种语言的程序执行环境中不可比因素比较多. 但是，这种结果仍然在一定程度上可以说明，我们采用的编译技术是有效的，利用这些技术实

现的函数式程序达到了较高的执行效率.

8 结束语

实践证明，改进延迟求值方式的实现技术和采用多级程序变换是提高函数式程序的执行效率和发展函数式语言实现技术的主要方法. 本文介绍的几种程序变换技术都是在分析函数式程序结构的基础上进行的. 通过静态分析获取必要的程序结构与数据结构信息，根据这些信息将程序变换成易于编译实现的形式. 这种方法已经成为改进函数式语言编译技术的主要手段.

函数式语言编译技术的发展为函数式程序设计的应用创造了必要的条件. 函数式程序设计已经应用于部分求值技术和软件规格说明等领域的研究中. 这些应用研究也将为函数式语言及其实现技术提出新的要求.

参考文献

1 Jones S L P. The implementation of functional programs. London: Prentice Hall, 1987.

2 Henderson P. Functional programming: application and implementation. London: Prentice Hall, 1980.

3 廖湖声. 函数式程序的分层λ抽象. 计算机学报，1989，12(12)：892－890.

4 Liao H S. Removing partial parametrization for efficient implementation of functional languages. Computer Languages, 1992，17(4)：241－250.

ON THE IMPLEMENTATION OF COMPILERS FOR FUNCTIONAL LANGUAGES

Liao Husheng

(*Computer Institute, Beijing Polytechnic University, Beijing* 100044)

Abstract This paper gives the main techniques for implementing compiler for functional languages and proposes two rules for the design of the compiler's implementation: (1) Saving the memory space for suspended arguments and to release them as early as possible, in addition to simplify function call; (2) Reducing the characteristic structure in functional programs by some transformations. Some techniques and their application in a compiler for a functional language are also presented.

Key words Functional programming language, compiler, lazy evaluation, partial evaluation.