Warren抽象机(WAM)的数据结构和解释实现†

苗占禄 刘椿年 钟宁燕
(北京工业大学计算机学院，北京，100022)

摘 要 用C语言严格描述了Warren抽象机(WAM)的数据结构，并给出WAM指令代码的具体解释实现及算术表达式的高效处理.

关键词 逻辑程序设计，Warren抽象机(WAM)，Prolog编译
分类号 TP 391.2

Prolog 语言是一种逻辑型的程序设计语言，它的高效实现技术经历了漫长的研究过程. 1983年，Warren总结了先前工作，提出Warren抽象机(WAM[1~3])，经过十多年的改进和实践，目前，WAM 已成为国际上 Prolog 标准的技术.

我们已设计开发出基于WAM的优化的全 Prolog 编译系统[4]. 本文属于系统的后端代码解释部分. 我们给出用 C 语言严格表达的 WAM 的数据结构和解释算法，给出重要的非逻辑成分 ASSERT，RETRACT 的实现及算术表达式的高效处理，最后以一个实例加以说明.

1 WAM 的抽象数据结构及其 C 语言描述

1.1 源程序

论域：PROGRAM(Prolog 程序集)；LIT(句节)； TERM(项)
ATOM(谓词 / 函数符)； ARITY(元数)； CLAUSE(子句)

1.2 内存

论域：MEMORY(内存，MEMORY = DATAAREA+CODEAREA)

1.3 数据区 (DATAAREA)

包括存放复合数据结构的 HEAP，环境与运行栈 STACK，变元寄存器 AREGS. 各分区之间的 < 序定义为：HEAP < STACK< AREGS. 分区内部的单元之间的 < 序即单元的排列顺序.

论域：PO (Prolog 对象)

收稿日期：1998—01—22
† 国家自然科学基金、国家"八六三"高技术发展资助项目

函数：+，−：DATAAREA → DATAAREA

val　：DATAAREA → PO + MEMORY

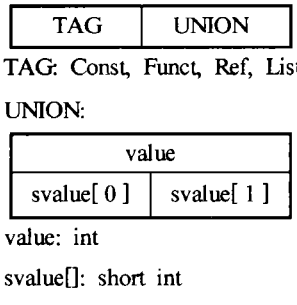
type　：PO → {Ref—变量, Const—常数, List—表, Struct结构, Funct—函数}

ref　：PO → ATOM（常数：Const的值）+ ATOM(ARITY(函数符及其元数) + DATAAREA(数据区的地址)

说明：type和ref合起来描述一个对象：

① 常数 c:(Const, c); ② 函数符及其元数 f/n:(Funct, (f, n)); ③ 变量及其例化：(Ref, l)(注：自由变量表达为val(l)=(Ref, l)); ④ 表：(List, l)（l为头元素地址，l+ 为尾元素地址); ⑤结构：(STRUCT, l):(l = (Funct, (函数符, 元数)), l + i 为第 i 个变元的地址).

1.3.1 数据区构成及分区关系(见图1)



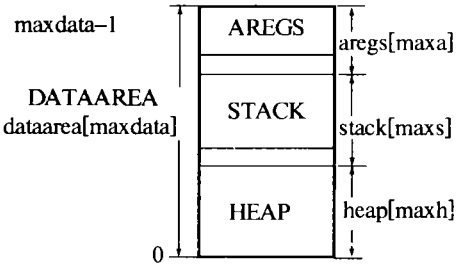
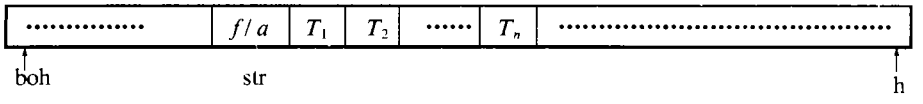


图1　数据区分区示意图

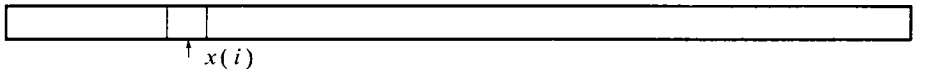
1.3.2 数据区单元构成：UNIT

1.3.3 HEAP栈(存放复合数据结构—表和函数)



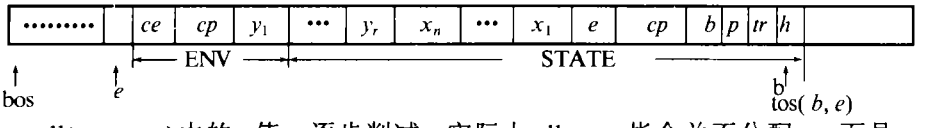
h为HEAP的顶(初值为boh)；boh为HEAP的底；str为HEAP的结构指针.

1.3.4 AREGS(调用变元，DATAAREA的一种)



$x(i)$是调用的第 i 个变元在AREGS的地址，简记为 $x\,i$.

1.3.5 STACK (存放环境和回溯点)



r: call(q, m, r)中的 r 值，逐步削减. 实际上allocate指令并不分配 y，而是put指令对它们进行逐步的分配和例化；

n: 谓词的变元数，即要拷贝到STACK栈中的调用变元个数；

e: 当前状态的的环境起点，由各个定义子句的allocate指令产生；

b: 当前状态的回溯点，由第一个定义子句的try指令建立.

bos为STACK的底，是tos(b, e)，b, e 的初值，通常设为0；

tos(b, e)为STACK的顶. 是一个表达式,每次引用时，都要先进行比较计算.

ENV: 环境

ce: 存放返回点的环境的起点; cp: 存放返回点的下一调用的指令地址; *y*: 存放 e 的子句中所含的第 *i* 个变量.

STATE: 状态

h: 存放回溯点的 HEAP 栈顶; tr: 存放回溯点的 TRAIL 栈顶; *p*: 存放回溯点的程序指针, 其初值由系统的前端模块得到; *b*: 存放回溯点的回溯点指针, 回溯时, 若 *b*=bos, 则程序终止; *cp*: 存放回溯点的下一调用的指令地址; e: 存放回溯点的环境的起点; x_i: 存放回溯点关键地址 l 的第 *i* 个调用变元.

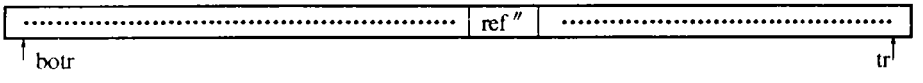
1.3.6 C 语言实现

```
typedef enum{Const, Funct, Ref, List, Struct}TAG;
typedef union{int value; int svalue[2];}UNION;
typedef struct
{TAG tag;  UNION value;  }UNIT;
UNIT dataarea[maxdata], *heap, *stack, *aregs;
heap=new UNIT[maxh];
heap=dataarea;
stack=new UNIT[maxs];    //STACK 栈的起始地址为 heap[maxh]的地址
stack=&dataarea[maxh];
aregs=new UNIT[maxa]; // AREGS 的起始地址为 stack[maxs]的地址
aregs=&dataarea[maxs+maxh];
int  trail[maxtrail];  //定义尾栈
int  pdl[maxpdl] ;   //定义下推表
```

1.4 尾栈(TRAIL 存放在回溯时需恢复为未约束状态的变量地址)

函数: +、−: TRAIL → TRAIL

ref″: TRAIL → DATAAREA(刚被例化的变量的地址)

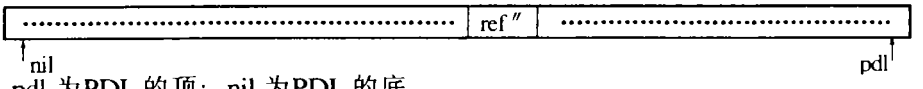


botr 为TRAIL 的底; tr 为TRAIL 的顶.

1.5 下推表 PDL(复合项的合一所使用的数据结构)

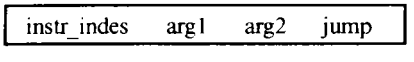
函数: +, −: PDL → PDL

ref′: PDL → DATAAREA



pdl 为PDL 的顶; nil 为PDL 的底.

1.6 代码区的数据结构及 C 语言实现



1.6.1 存储单元构成 CUNIT

instr_index: 指令索引代号　(short int)　// 初值由系统前端部分得到.

arg1: 参数 1　(int)　　// 当为 CALL 指令时, arg1 表示变元数

arg2: 参数 2 (int)　　// 当为 CALL 指令时, arg2 表示有用变量数

jump: 跳转地址 (int)　// WAM 标准指令中, 除 CALL 指令外,jump 均空.

1.6.2 C 语言实现

```
typedef struct
    {short int instr_index;  int arg1;   int arg2;  int jump;} CUNIT;
CUNIT code[maxcode];
```

2　解释器的实现

1. 系统初始化

A. 程序指针 p: 问题子句代码的第一条指令; B. 初始环境: 即问题子句的环境、系统运行前先进栈

b = tos(b, e) = bos　　ce= e;　cp =1

y_i 为问题子句中所有的变量, 初始置为自由的

2. 解释程序实现(见图 2. 结构流图和图 3. C 语言描述)

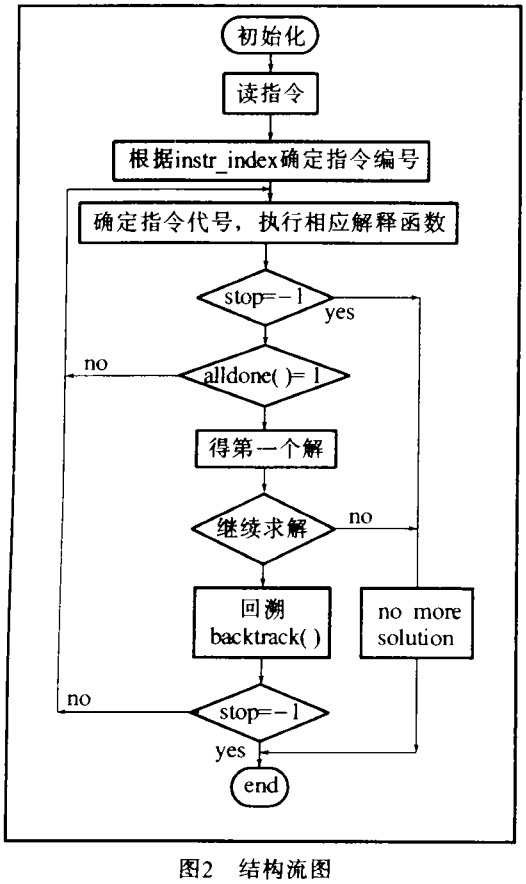


图2　结构流图

```
interpret ( )
 {
 initialize ( ); //初始化
 数据结构定义;
 得到程序指针p值; //从系统的前端部分得到
 do
  {
  do
   {
   p=instr_index;
   switch(code[ p].instr_index)
    {
     case put_value:
      put_value(code[ p].arg1,code[p].arg2)
      break;
        ...
      case:...
       }
      if(stop==-1)
        break;
      }while(!alldone( ) );
   flag=next_solution( );
   if(flag=='y'&&stop!=-1)
     backtrack( );
   else
     break;
  }while(stop!=-1)
 }
 }

}
```

图3　C语言描述

3 算术表达式的高效处理

对 Prolog 编译系统来说，算术表达式的处理在很多时候是影响系统效率的关键因素. 在我们研制的系统中，对算术表达式进行了高效处理，下面举例说明其实现过程.

对于一个任意的算术表达式，如 3 + 2，语法分析将它转换为前缀形式：+ 3 2. 一般的编译方案是将其看作一种复合结构，生成指令:

put_structure +2 x_1
unify_constant 3
unify_constant 2

HEAP 栈中的存放如图 4. 可看到表达式的结构保存在栈中，占用存储空间，当程序引用此表达式时，在 HEAP 中遍历计算. 显然，这种方法大大降低了程序运行速度.

Struct	2	
Funct	+	2
Const	3	
Const	2	

图4 表达式3+2存储格式

现在我们采用新的编译方法，生成一条直接计算的指令:

plus 3 2 x_1 // 3 加 2，把结果存放在变元寄存器 x_1中.

但当 3, 2 是由引用得到时，比如生成指令：plus x_3 x_2 x_1，即变元寄存器 x_3、x_2 中恰好分别存放常数 3 和 2，而系统中实际生成的指令为：plus 3 2 1，这样，我们就不能区分 3, 2 是如何得到的，如果 3, 2 常数存放在 x_4, x_5中，则用同样方法解释时，会产生错误.这时，采用生成多指令方法，即用多个指令区分：

plus0 C R x
plus1 R C x
plus2 R R x
plus3 C C x

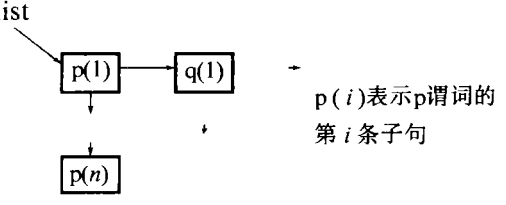
注：C 为常数； R 为引用常数 ($R < 0$ 时，常数存放在 STACK 中)

采用这种方法，用 N 皇后问题来测试，程序的总体运行速度提高了20 %.

4 非逻辑成分 Assert,Retract 的实现

4.1 主要辅助数据结构

a. 链表 list



每个结点保存各子句的信息：谓词名，变元数，子句指令序列的起始地址，头部指令序列长度. 编译过程中生成链表，执行 Assert, Retract 时，根据链表以修改代码区.

b. record 表

记录插入或删除的无条件子句中谓词. 例如，出现 assert (p (X))，而其它子句调用了 $p(X)$，如 e: - $p(X)$, …. 编译产生 100: call n_1, n_2, p，则在 record 表中记录 [100，p]，

从而将那些有可能被 retract．assert 引起入口地址变化的代码位置记录下来．将来一旦发生插入删除，只须根据 record 记录，直接去找要改的地方，不须遍历整个代码区．

4.2 assert

用法：$q(\cdots)$: -…, assert(p (X)), …．

含义：是在 p 子句前部插入一条事实子句，X 是基项．

编译生成指令

```
        : :
    put_seq   X
    call_assert   p, 变元数
        : :
```

解释： call_assert (p, 变元数)指令做如下工作：

a．在代码区追加代码

生成 try 类指令

for (i =1; i <= 变元数; i++)

{ 调用 get_seq (i, i) 函数 }

生成 proceed 指令

b. 修改子句链表 list 和有关的 call, execute 指令的入口地址

c. get_seq(X_1, X_2)函数

根据 X_1 单元类型的不同生成相应的 WAM 指令

4.3 retract

用法：$q(\cdots)$: -…, retract (p (X)), …．

含义：retract 带回溯，删除第一个匹配到的子句，并继续往下执行．回溯到此处，再匹配下一条子句．匹配不成功则回溯，X 可以是自由变量．

编译生成指令

```
        : :
    put_seq   X
    call  n1, n2, retractp
        : :
retractp:   try_me_else   n   retractp+1
        del   p
        proceed
```

解释：del()函数

删除所有可以匹配的子句信息，若无可匹配子句则回潮．

5 一个实例

例： append

append ([], Y, Y).

append ([H | T], Y, [H | Z]): -

append (T, Y, Z).

?-append ([1, 2], [3, 4], L).

生成的 WAM 指令序列及其在运行时的存储区内容见图 5.

WAM指令代码

```
ap_1: try_me_else   ap2 3
      get_constant  [], x1
      get_variable  x4, x2
      get_value     x4 x3
      proceed
ap_2: trust_me      3
      get_list      x1
      unify_variable x5
      unify_variable x6, x2
      get_list      x3
      unify_local_variable x4
      unify_vatiable x7
      put_value     x5, x1
      put_value     x6, x2
      put_value     x7, x3
      execute  ap_1  3
start: put_list     x4
      unify_const 2
      unify_const nil
      put_list      x1
      unify_const   1
      unify_value   x4
      put_list      x5
      unify_const   4
      unify_const   nil
      put_list      x2
      unify_const   3
      unify_value   x5
      put_value     y1, x3
      call          ap_1 3 1
end: proceed
```

存储区内容

HEAP

0		
1	List	2
2	Const	2
3	Cosnt	nil
4	List	5
5	Const	1
6	List	2
7	List	8
8	Const	4
9	Const	nil
10	List	11
11	Const	3
12	List	8
13	List	14
14	Const	1
15	List	16
16	Const	2
17	List	11
18		

STACK

101		
102	nil	
103	nil	
104	List	14
105	Ref	17
106	List	11
107	Const	nil
108	101(e)	
109	end(p)	
110	101(b)	
111	ap_2(p)	
112	18(tr)	
113	3(h)	

AREGS

1	Const	nil
2	List	11
3	Ref	17
4	List	11
5	Const	nil
6	List	11
7	Ref	17

TRAIL

104
15
17

注：左边代码与实际生成代码有所不同

代码指令参数为x_1, x_2, x_3, …实际为1, 2, 3, …

代码指令参数为y_1, y_2, y_3, …实际为−1, −2, −3, …

例：put_value x_5 实际为put_value 5

put_value y_1 实际为put_value −1, 3

图5 append实例

6 结束语

我们设计开发的基于 WAM 的优化的全 Prolog 编译系统的特点是：

1. 进行了高度的编译优化，系统效率较高. 在对 WAM 指令进行解释的情况下，程序的运行效率是 Prolog 解释系统 BPU-Prolog[5] 7～8 倍，是国际上某些未进行优化的 Prolog 编译系统(如荷兰阿姆斯特丹大学的 SWI-Prolog 系统)的 3～4 倍. 若将 WAM 指令集翻译为汇编语言，运行速度还可有数倍的提高.

2. 这是一个全 Prolog 编译系统，已实现 Read, Write 等预定义谓词，支持 CUT, NOT, ASSERT, RETRACT 等各种非逻辑成分，且能对算术表达式进行高效处理.

3. 系统易于扩充到约束逻辑程序设计 (CLP)，例如算术表达式已在内部转化为前缀形

式，为CLP方向上的扩充作了一定的准备. 本文给出WAM数据结构和C语言解释实现. 系统的运行效率为我们先前研制的Prolog解释系统的7～8倍，是国际上SWI-Prolog编译系统的3～4倍. 当然，若把WAM指令代码用汇编语言实现，其运行效率必将再有数倍的提高.

参 考 文 献

1 Warren D H D. An Abstract PROLOG Instruction Set. Technical Note 309, SRI International, 1983

2 Egon Borger, Dean Rosenzweig. WAM -Definition and Compiler Correctness, chapter 2 of Logic Programming (C Beierle and L Plumer Ed): Formal Methods and Practical Applications. Elsevier Science B V. 1995. 21～90

3 Hassan Ait-Kaci. Warren's Abstract Machine: A Tutorial Reconstruction, The MIT Press, 1991

4 钟宁燕，刘椿年，苗占禄. 基于WAM的Prolog编译优化. 北京工业大学学报，1998, 24(4): 109～111

5 刘椿年，曹德和. Prolog语言，它的应用与实现. 北京：科学出版社，1990

6 Wielemaker J. SWI-Prolog 1.6: Reference Manual. Univ of Amsterdam, 1992

Data Structure and Interpretation of Warren Abstract Machine

Miao Zhanlu　Liu Chunnian　Zhong Ningyan

(Computer Institute, Beijing Polytechnic University, Beijing, 100022)

Abstract The data structure of Warren Abstract Machine(WAM) is discussed. Interpretive implementation of the instruction repertoire of WAM and a new strategy to deal with arithmetic expressions efficiently are given. It is the back-end module of an optimizing Prolog compiler system which we developed, based on WAM. The system execution speed is 7～8 times as fast as the Prolog interpreter we developed previously, and 3～4 times the speed of SWI-Prolog compiler system.

Keywords logic programming, Warren Abstract Machine (WAM), prolog compiler