第6卷第9期
1995年9月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 6, No. 9
Sep. 1995

PARLOG 跟踪调试器的设计实现技术*

马玉羚 温冬婵 王鼎兴

(清华大学计算机科学与技术系,北京 100084)

摘要 本文讨论了 PARLOG 交互式跟踪调试器的设计与实现技术. 作者改进了 L. Byrd 的盒式模型,使之可以描述 PARLOG 的顺序执行模型. PARLOG 顺序化执行模型使计算具有确定性,易于用户调试程序. 本文详细介绍了调试器实现中确定调试层次、将顺序化模型中的平坦化计算恢复为树型、实现不同层次的跟踪算法等方面的研究成果.

关键词 执行模型,调试器,平坦化计算,跟踪算法.

逻辑语言作为一种高级语言在人工智能领域中得到广泛应用. 随着人工智能领域对高速度大规模知识处理的要求和并行计算机的不断发展,对并行逻辑程序设计语言的研究日益受到人们的重视. 过去对并行逻辑语言的研究主要集中在如何提高其运行速度,相对地,对并行逻辑程序的编写、开发和调试环境研究较少. 为了便于用户理解逻辑程序的执行过程,减少用户检测程序的负担,开发逻辑语言的调试工具是十分必要的.

逻辑程序有说明性语义和操作性语义两方面. 因此,调试器在实现上相应地分为两类.

①程序跟踪器,基于操作性语义. 用户使用调试工具,根据程序的执行情况,由用户自己分析出程序错误所在. 目前,广泛使用的商用系统均采用这种方法. 如 Quintus Prolog[1]. 另外一些实验系统也采用这种方法[2,3].

②算法调试器,基于说明性语义. 调试系统向用户提出问题,询问有关被测程序的情况,分析用户的回答信息和程序的执行情况,指出错误的位置. 这种方法称为算法调试,最早由 Shapiro 在 1982 年提出[4]. 一些系统对 Shapiro 的分解——询问算法进行改进,形成各具特色的调试系统[5-7].

PARLOG 是一种并行逻辑语言[5],本文主要讨论了对 PARLOG 语言的交互式跟踪调试器(Interactive Tracing Debugger)的设计与实现工作. 由于 PARLOG 语言本身的并行性和语句选择的非确定性,使得跟踪纯并行的 PARLOG 程序变得十分复杂. 为了跟踪调试方便,也为了在单机上高效实现 PARLOG,笔者将 PARLOG 的并行操作语义映射到一个顺序化的操作模型上,从而实现了 PARLOG 顺序编译系统. 在此系统上,我们又完成了

* 本文 1993-11-22 收到,1994-04-28 定稿

作者**马玉羚**,女,1967 年生,1994 年硕士毕业于清华大学,主要研究领域为逻辑语言,调试器,软件集成环境. **温冬婵**,女,1946 年生,副教授,主要研究领域为智能语言,并行编译. **王鼎兴**,1937 年生,教授,主要研究领域为并行/分布处理技术与系统.

本文通讯联系人:温冬婵,北京 100084,清华大学计算机科学与技术系

PARLOG 跟踪调试器及其多窗口交互界面的工作.

L. Byrd 曾提出描述 Prolog 程序的盒式计算模型[8],这种模型能够清楚地表达程序中谓词的计算过程.本文对盒式计算模型进行修改,使之能够描述 PARLOG 语言的并行计算模型和顺序化后的顺序计算模型.而这顺序化后的计算模型正是实现跟踪调试器的基础.

本文第 1 节介绍 PARLOG 的并行计算模型,第 2 节引入顺序化的 PARLOG 计算模型,第 3 节详细地描述跟踪调试器的实现手段,最后给出结论.

1　PARLOG 并行计算模型

PARLOG 语言的关系分为单解关系和所有解关系,PARLOG 语言良好的并行性、执行的高效性以及其他区别于传统逻辑语言的特点均表现在单解关系中,全解关系仅是对单解关系的一个补充.故本文只讨论单解关系的计算模型和为其实现的跟踪调试器.

L. Byrd 提出的盒式模型如图 1 所示,清楚地表达了一个 Prolog 关系在执行过程中的种状态.目标关系由方框表示,4 个端口表示 4 种状态,Call 表示关系引用,Exit 表示执行成功,Redo 表示反向回溯,Fail 表示失败返回.下面首先对 PARLOG 单解关系做简单说明,然后给出盒式的并行计算模型.

图1 盒式模型

PARLOG 单解关系采用确认选择非确定性操作语义,具有流式与并行,确认或并行.子句均为警卫 Horn 子句,形式为:

$$R(t1,t2,\cdots,tk)\leftarrow\langle\text{卫士合取}\rangle:\langle\text{体合取}\rangle$$

〈卫士合取〉和〈体合取〉为与并行操作.在搜索定义关系 R 的所有子句时,确认子句的操作是非确定性的或并行操作.执行一个单解关系调用时,首先搜索候选子句,搜索过程包括输入匹配和卫士执行.每个子句搜索的可能结果是:

- 候选子句:输入匹配及卫士执行都成功;
- 非候选子句:输入匹配及卫士执行失败;
- 挂起子句:输入匹配或卫士执行挂起,但均未失败.

因此,执行一个关系调用时也可能存在下列 3 种结果:

- 执行成功:至少存在一个候选子句,这时任选一个候选子句,并把调用关系归结为该子句的体合取调用.由于选择不同的子句将导致不同的执行,因而计算具有非确定性.
- 执行挂起:不存在候选子句,但存在挂起子句.这时忽略子句搜索过程中所有计算.
- 执行失败:所有子句都是非候选子句.

由于 PARLOG 的卫士条件必须是充分的,即当一个子句成为候选子句后,必须保证:

(1)确认选择这个子句必定能求出解来(程序有解情况下);

(2)或者选择任何子句都求不出解来(程序无解).

因此单解关系执行机制中不再有类似于 Prolog 的回溯.

上面的操作语义影射到盒式执行模型上,便转化为如图 2 所示的并行执行模型.

图中方框表示目标关系.所有方框之间是与关系.并行模型中,体部合取式中各个关系对应的方框间是与并行关系.4 个端口:Call 表示关系调用,Exit、Suspend 和 Fail 分别表示关系调用执行后的 3 种状态:成功、挂起和失败.

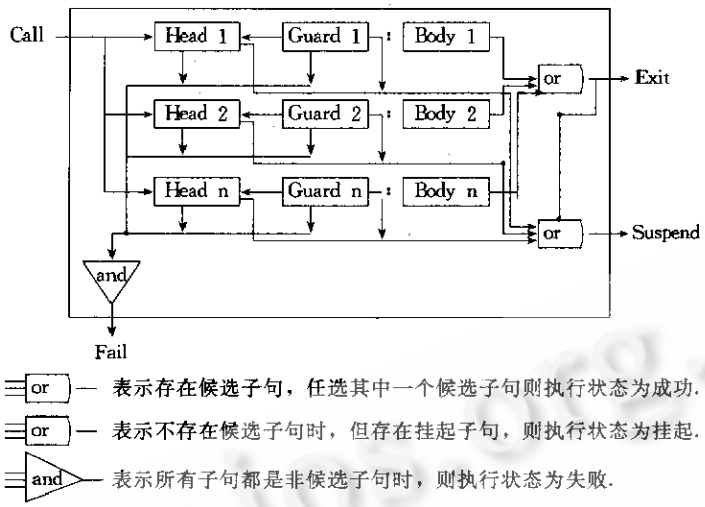

图2 PARLOG盒式并行执行模型

图 2 中子句 1 到子句 n 的输入匹配和卫士执行是并行执行的，表现了搜索子句的或并行操作. 从并行执行的模型可以看出，由于子句选择的或并行操作引起的不确定性，使程序的执行过程是不确定的，每次程序运行可能出现不同的问题解答，也就是说，有可能用户在运行程序之前不能预测程序的运行行为，这样用户调试程序具有一定的盲目性，因为一般情况下都是根据用户预测的运行行为同程序实际运行行为相比较，发现不同之处，从而去定位错误位置. 因此调试并行程序比较困难. 从便于调试 PARLOG 程序的目的出发，笔者又描述了 PARLOG 的顺序执行模型，以消除并行模型的不确定性.

2 PARLOG 顺序执行模型

PARLOG 语言具有流式与并行和子句选择的或并行操作语义，PARLOG 顺序执行模型将这种并行操作语义映射到一系列串行操作，从概念上没有改变 PARLOG 语言的并行语义，而仅从运算操作的次序上使其顺序化.

PARLOG 程序执行模型如图 3 所示.

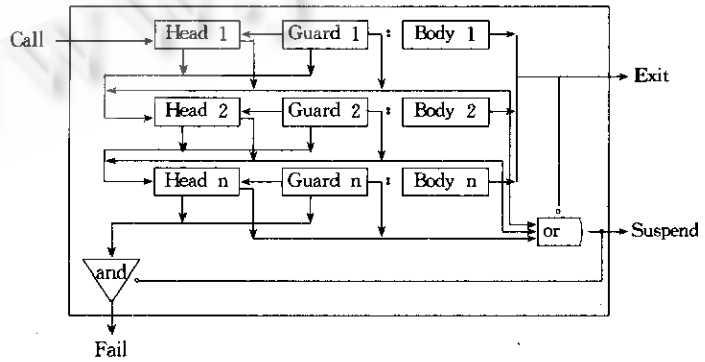

图3 PARLOG盒式顺序执行模型

在顺序模型中,由方框表示的目标间仍然是与关系,但是执行时不再并行,而是依次执行目标.在选择候选子句的过程中,也不再并发地对所有子句的头部进行输入匹配和卫士条件测试,而是顺序地执行各个子句的输入匹配和卫士条件测试,一旦某个子句被确认为候选子句,则立即确认选择该子句.经过这种处理后,程序执行过程中总是选择子句组的第一个候选子句,因而计算具有确定性.若测试完所有子句后没有候选子句,那么,若有挂起子句则执行结果为挂起,否则为失败.

3　跟踪调试器的实现

3.1　跟踪调试的层次

交互式跟踪调试器是让用户在交互的条件下,跟踪程序的执行过程,以便查找出程序错误.程序的执行过程是一个目标的求解过程,整个求解过程展现在用户面前是一棵与/或树,笔者改变与/或树的表达形式,用结点为方框的与树来代替与/或树,将执行的操作顺序和执行状态很好地表达出来.图 4 中的方框为上一节所描述的程序执行模型,用程序 fib.p 为例做一个说明.

```
程序:fib.p
    mode fib(?,^).
    fib(x,y)←x>1:SUB(x,1,x1),SUB(x,2,x2),
                 fib(x1,y1),fib(x2,y2),
                 ADD(y1,y2,y).
    fib(0,1).
    fib(1,1).
```

求解目标 fib(5,y)的过程如图 4 所示.不去看方框内部的情况,只看与树的形状,表达了程序执行的某一状态.方框内部表达子句搜索的过程.相应地,跟踪程序执行过程有两个层次:一是只关心子句选择的结果,即与树的形状;二是关心子句选择的过程,即深入到方框的内部.PARLOG 交互式跟踪调试器实现了这两个层次的跟踪.

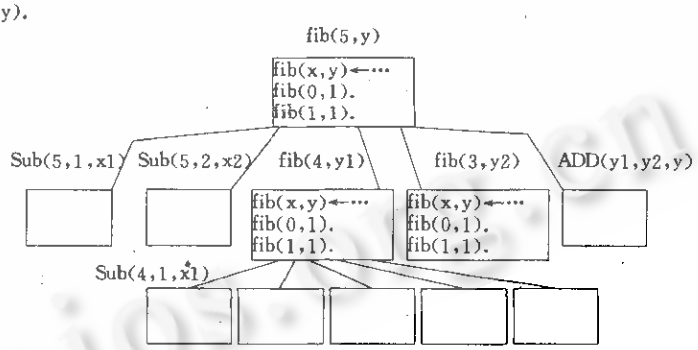

图4　程序fib.p的结点为方框的与树

3.2　平坦化计算向树型结构的恢复

PARLOG 单解关系采用警卫 Horn 子句,选择子句时只在众多的候选子句中任选一个,由于卫士条件的充分性使得在程序有解的情况下,选择该子句一定能求出解来,或者在程序无解情况下,选择其他子句同样求不出解.这样在程序的运行过程中是没有回溯的,于是在顺序化的单解执行系统中采用了受限的深度优先不带回溯的进程调度算法,需要说明的是,PARLOG 程序定义的关系在执行系统中我们称之为进程.由于没有回溯,实现时将树型的执行模型平坦化,形成一条线性的进程链,即与树退化成一条遍历与树的线性序列.这种做法使 PARLOG 单解执行系统实现起来简单,而且执行效率高,但这样却丧失了树型的

结构，不能将执行过程简单地照原样显示给用户，使用户很难理解，因为用户头脑里是程序的树型结构. 所以，调试器的设计必须将线性进程链恢复为树型结构.

为此，引入下面的数据结构来记录程序运行过程中的层次变化.

```
typedef struct pathnodetype {
        int state;          /* 程序执行状态 */
        int deep;           /* 归约深度 */
        int clsnum;         /* 归约子句号 */
        char *name;         /* 进程名称 */
        int tag;            /* 标志，为扩展功能时准备 */
        struct pathnodetype *parent;     /* 指向父进程 */
        struct pathnodetype *child;      /* 指向第一个子进程 */
        struct pathnodetype *brother;    /* 指向兄弟进程中下一进程 */
        struct pathnodetype *next;       /* 指向进程链中的下一个进程 */
        process *point;                  /* 指向该进程对应的进程结构实体 */
        }Pathnodetype;
```

数据结构 pathnodetype 不仅能记录层次变化，还可以记录进程执行的状态、进程在与树中的深度、确认选择的子句号等信息. 这一数据结构是整个跟踪调试器实现的基础. 这里，以图 4 中进程 fib(4,y1)所对应的 pathnodetype 为例来说明它的含义.

```
state=1;          /* fib (4,y) 执行成功 */
deep=2;           /* fib (4,y) 在与树中的第二层 */
clusum=1;         /* 第一个子句被确认选择 */
name="fib";       /* 进程名为 "fib" */
tag=0;            /* 暂无用 */
parent=&fib(5,y);     /* 父进程 fib (5, y) */
child=&SUB(4,1,x1); /* 第一个子进程 SUB (4, 1, X1) */
brother=&fib(3,y2);   /* 兄弟进程中下一个为 fib(3, y2) */
next=&SUB(4,1,x1); /* 进程链中的下一个进程为 SUB(4,1,x1) */
point=(func *) fib;   /* 进程结构实体为 fib */
```

进程 fib(4,y1)所对应的 pathnodetype 的前 4 项记录了该进程的各种状态信息，后 5 项指针将该进程同其父进程、子进程、兄弟进程以及将在其后运行的进程联系起来，具有层次信息. 把所有进程的 pathnodetype 联接起来，构成一棵与树，恢复了程序运行的树型结构.

3.3 实现断点的算法

跟踪调试分为两个层次，一个是跟踪到与树方框结点的外部，即断点为方框结点称与树级；另一个是跟踪到与树方框结点的内部，即跟踪到子句确认选择的过程. 本小节所讨论的断点是指将程序暂停在用户指定的关系运行之前，即运行暂停在方框结点的外面.

PARLOG 程序的运行是在执行系统的进程调度策略控制下进行的，修改进程调度策略，在适当位置加入调试控制，可以达到程序执行受人为控制的目的. 进程调度策略的简单框架和加入调试控制的位置如图 5 所示.

修改后的进程调度策略算法如下:

STEP1 若所有目标进程均已计算完毕,则程序执行结束,进入调试命令接受状态,报告程序运行结束状态.

STEP2 若有目标进程未计算,根据选择点,确定下一个运算进程.

STEP3 根据调试器接收的用户命令操作:

(1)若是单步命令,则暂停运行,显示默认信息,进入调试命令接收状态.

(2)若下一个运算进程是用户指定的断点,则暂停运行,显示默认信息,进入调试命令接收状态.

(3)若用户指定了归约次数,则归约计数器加1,如果归约计数器的值等于指定归约次数,则暂停运行,显示默认信息,进入调试命令接收状态.

(4)若用户指定了起始子句号,则修改子句号寄存器,使子句的确认选择从用户指定的子句开始.

STEP4 根据选择点调用当前进程.

SETP5 依据进程执行的结果修改工作空间和环境空间.

上述算法中提到的单步、断点和归约次数等均以方框为单位.

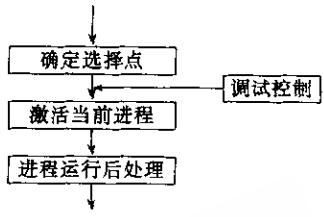

图5 调试控制的位置

3.4 子句级实现跟踪调试

子句级是实现跟踪调试的第二个层次,按照盒式顺序执行模型的操作顺序,跟踪到确认选择子句的过程.

除跟踪调试层次降低外,子句级的跟踪算法还解决另一个顺序化系统存在的问题,即非确定性语义映射到确定操作上带来的问题.在纯并行系统中,子句选择是并发进行的,最终选中那个候选子句是不确定的,而在顺序系统中总是按固定的调度策略选择子句,因此,一个有错的程序在纯并行系统中运行可能出错,而在顺序化的系统中却可能永远运行正确.举一例子说明,如果程序中对关系 A 用两个子句定义:

$$A \leftarrow G1 : B1 \tag{1}$$

$$A \leftarrow G2 : B2 \tag{2}$$

假定子句(1)是正确的,而子句(2)的体部 $B2$ 有错,再假定某种例化条件下,两子句的输入匹配和卫士测试都成功,即子句(1)和(2)都是候选子句,在顺序系统中总是选择子句(1),那么程序运行结果正确.而在并行系统中却可能选择子句(2),那么程序运行出错.总之,在顺序系统中,由于程序运行的确定性,使得存在一些子句永远不能被执行,而检查这些子句的正确性便由调试器来完成,也就是人为地干预子句的确认选择,从而得到一棵新的与树.

实现时的具体作法是:设置标志寄存器 singlestep2 来表示是否为子句级跟踪,1 表示"是",0 表示"否".设置起始子句号寄存器 clause－num,表示从用户指定的子句开始向后作确认选择,算法如下:

```
1. n=clause－num;
2. do {
        对第 n 个子句进行输入匹配和卫士测试;
```

```
case 测试结果状态 {
    成功:if singlestep2=1 进入调试命令接收状态;
        return            子句体部和执行状态;
    挂起:if singlestep2=1 进入调试命令接收状态;
                          设置挂起标志;
    失败:if singlestep2=1 进入调试命令接收状态;
                    }
n=n+1;
} while (n<=最大子句号),
```

3.5 跟踪调试的环境

本文实现的是PARLOG交互式跟踪调试器.为了达到良好的交互性,利用Sun工作站提供的Sunview窗口系统,实现了一个多窗口跟踪调试环境.整个环境分为3个子窗口,一个是编辑子窗口,用于显示和修改PARLOG源文件;另一个是调试命令子窗口,列出调试器提供的所有命令;第3个是程序运行状态显示子窗口,用于显示用户所关心的执行情况.

由于3个子窗口同时出现在屏幕上,所以调试器的交互性很好.

4 结 论

笔者采用了改进的L. Byrd盒式模型来描述顺序执行过程,设计实现了PARLOG交互式跟踪调试器.在实现过程中考虑了PARLOG语言的特殊性和用户的要求,整个调试系统有如下特点:

- 有两个层次的跟踪调试,一层为与树级,另一层为子句级.
- 调试PARLOG程序不需要修改源程序,即不需要象Qintus Prolog那样增加"SPY"操作到源程序,整个调试工作在调试环境中完成.
- 环境的交互性好,信息显示清晰.
- 系统经多家用户使用,效果良好,达到了实用要求.

参考文献

1 陈世福,潘金贵等. Quintus Prolog. 重庆:科学技术文献出版社重庆分社,1989.

2 Tamura H, Aiso H. Logic programming debugger using control flow specification. Proc. of the Logic Programming Conference'88, 1988.

3 Plummer D. Coda: an extended debugger for PROLOG. Univ. of Texas Austin, Proc. of the Logic Programming International Conference, 1988.

4 Shapiro E. Algorithmic program debugging. MIT Press, 1982.

5 Takahashi H, Shibayama E. PRESET—a debugging environment for prolog. Proc. of the Logic Programming Conference'85, 1985.

6 Lichtenstein Y, Shapiro E. Abstract algorithmic debugging. Proc. of the Logic Programming International Conference, 1988.

7 Takeuchi K. Algorithmic debugging of GHC programs and its implementation in GHC. Tech. Report TR—85, Institute of New Generation Computing Technology.

8 Byrd L. Understanding the control flow of PROLOG programs. In: Tarnlund S ed. Proceedings of the Logic Programming Workshop, 1980. 127—138.

DESIGN AND IMPLEMENTATION TECHNIQUES FOR A PARLOG TRACING DEBUGGER

Ma Yuling　Wen Dongchan　Wang Dingxing

(*Department of Computer Science and Technology, Tsinghua University, Beijing* 100084)

Abstract　The paper discusses the design and implementation techniques for a PARLOG interactive tracing debugger. The authors modify the L. Byrd's Box model and make it capable to illustrate the sequential execution model of PARLOG. In sequen tial execution model, it is easy for programmer to debug program because the computation is determinism. The paper introduces three aspects of the research in debugger implementation in detail: define the debugging level; change flat computation in sequential model to tree — shape computation; complete tracing algorithms in different level.

Key words　Execution model, debugger, flat computation, tracing algorithms.