第 7 卷第 7 期
1996 年 7 月

软 件 学 报
JOURNAL OF SOFTWARE

Vol. 7, No. 7
Jul. 1996

CLP 系统中推理机与约束求解器的协调技术

张秀珍　刘椿年

(北京工业大学计算机系　北京　100022)

摘要　本文讨论了我们自行开发的 BPU－CLP(R)系统中推理机与约束求解器的协调技术.协调主要发生在回溯机制中.本文描述的方法妥善地解决了在存储优化的情况下推理机与约束求解器的协调问题.

关键词　PROLOG, CLP, CLP(R),存储优化技术.

J. Jaffar 和 J. －L. Lassez 1987 年提出的约束逻辑程序设计 CLP(constraint logic programming)方案[1]定义了一类基于规则的约束程序设计语言,每种计算域指派定义了一种 CLP 语言.CLP(R)即该方案的一个实例,其计算域 R 是实数上的无解释算子构成的代数结构.

我们对 CLP 的操作模型做一粗略描述.设 P 是一个 CLP 程序,G 是目标,G 中有原子公式和约束,约束是已解的或延迟的.从 G 到另一个目标 G_1 的推导可以定义如下:子目标选择策略从 G 中选择了延迟约束 δ,并且约束求解器确定 δ 与现有已解约束相容(无矛盾),则 G_1 的已解约束集中增加了 δ,延迟约束集中减少了 δ,原子公式序列不变;或者,子目标选择策略从 G 中选择了原子公式 A,并且 P 中有 A 的定义子句 R,则 G_1 的延迟约束集中加入了 R 的头部与 A 的对应自变量之间产生的等式及 R 体内的约束,原子公式序列中加入了 R 体内的原子公式,已解约束集不变.一个 CLP 程序和目标执行的过程就是一个不断地消解目标中剩余的原子公式和求解延迟约束的过程.初始目标中,所有约束都是被延迟的.在推导的任何一步,如果新产生的约束与已解约束集中的约束都相容,就认为其可解并加入已解约束集中,计算向前推进一步;否则就认为约束不可解,进行回溯.这样得到的成功推导序列最终目标中的约束称为解约束,且构成程序的输出.可见,CLP 方案采用类似 PROLOG 的消解原理,用本意论域上的约束可解性代替了 Herbrand 全域上的语法合一,将对原子公式的消解化为对一组等式约束的求解.

CLP 系统的基本结构是向纯 PROLOG 推理机中嵌入约束求解器,考虑到回溯机制,特

---

* 本文研究得到国家自然科学基金和国家 863 高科技项目基金资助.作者张秀珍,女,1969 年生,助教,主要研究领域为人工智能,逻辑程序设计.刘椿年,1944 年生,教授,主要研究领域为人工智能,知识工程,软件技术.

本文通讯联系人:刘椿年,北京 100022,北京工业大学计算机系

本文 1995－04－28 收到修改稿

别是存储优化(尾递归优化、确定末端结点优化等),这一嵌入绝非易事.而从系统效率方面来考虑,约束求解算法固然是保证 CLP 系统效率的一个关键,但不容忽视的是,纯 PROLOG 程序是 CLP 程序的一个严格子集,任何实用的 CLP 解释系统都必须保证对纯 PROLOG 程序能够以纯 PROLOG 解释器的速度运行.本文所讨论的推理机与约束求解器的协调技术就是针对以上 2 方面的问题提出的高效解决方法.本文提出的方法已成功地运用在 BPU—CLP(R)系统中.笔者相信,文中提出的方案对于实用的 CLP 系统的研制具有普遍的指导意义.

1 CLP 系统的总体结构及控制机制

为了得到高效的 CLP 系统,必须保证:只有当程序中的约束必须调用约束求解器才能解时,才应引起那部分开销,作为 CLP 程序子集的纯 PROLOG 程序能以与在纯 PROLOG 解释器下几乎相同的速度被解释执行.为此,BPU—CLP(R)解释器采用如图 1 所示的总体结构.

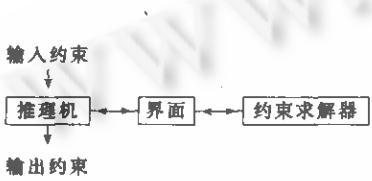

图1 BPU—CLP(R)解释器总体结构

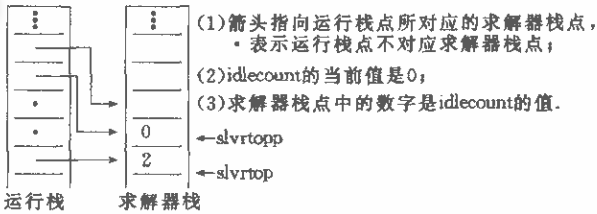

图2 系统控制机制

解释器主要由推理机、界面和约束求解器 3 部分构成.推理机除了控制推导外还必须处理约束.推理机求解能够用标准合一机制求解的约束,合一的结果即约束求解的结果,可解约束形成类似 PROLOG 的变量约束(Binding).对于不能用合一求解的约束,推理机调用界面.界面通过对算术表达式化简等手段可能得到"$X=15.3$"这样的约束并直接形成变量约束$\{15.3/X\}$,在其他情况下,界面调用约束求解器,而只将约束求解的结果返回给推理机.推理机根据约束求解的结果决定是继续推进还是回溯.不难看出,这样的总体结构保证了纯 PROLOG 程序的效率.

下面来看如何实现回溯机制.从概念上讲,PROLOG 解释器只需一个运行栈就可以控制推理的进行:合一成功后进栈,保存合一现场,回溯时出栈,恢复到相应现场.但是,由于 CLP(R)系统的模块化设计,约束求解的任务已分配到推理机和约束求解器中,因此,约束求解失败时,这 2 部分必须协调工作才能完成回溯的任务.因此,必须增加新的数据结构和控制机制来协调推理机和约束求解器(简称求解器)的工作.

由于只有推理机选择点才能产生真正的推理回溯点,求解器的选择点仅是为推理机选择点服务的,它只能依附于推理机选择点而存在.因此,与推理机运行栈相对应,我们设立求解器栈来保存与推理机选择点对应的求解器状态的改变,以便以后推理机回溯时求解器也恢复到相应状态.这样,只有遇到原子公式子目标时才可能生成推理机选择点.如果对该原子公式子目标进行消解(即对原子公式与定义子句头部对应自变量间产生的一组等式求解)的过程中,以及对下一原子公式子目标进行消解前的原约束求解过程中,涉及到求解器,则

生成相应的求解器选择点,对约束求解器状态所作的所有修改均记录其中.

因此,运行栈点和求解器栈点并非一一对应,并且可能有多个运行栈元素不对应求解器栈点,所以我们设置全程变量 *idlecount* 来协调推理机和求解器的工作. *idlecount* 的语义是从求解器栈顶对应的运行栈元素开始,到运行栈顶为止,没有对应求解器栈点的运行栈元素的个数. 因此,"*idlecount*=0"的语义是该结点(运行栈顶元素)有对应的求解器栈元素. *idlecount* 的初值为 0,它的值只可能在对谓词消解的过程中被修改. 如果在对谓词的消解过程中,以及对下一个原子公式子目标消解之前没有调用约束求解器,则 *idlecount* 的值加 1,否则生成相应的求解器选择点,并将 *idlecount* 的当前值保存在求解器选择点中,重置 *idlecount* 的值为 0.

下面讨论求解器栈的进栈时机. 运行栈是在对原子公式消解成功后进栈,求解器栈进栈可能发生在运行栈进栈前(对谓词消解过程中调用求解器),也可能发生在运行栈进栈后(谓词消解过程中未调用求解器,但对原约束求解时调用求解器). 我们的策略是:通过指向求解器栈的顶指针 *slvrtop* 和辅助指针 *slvrtopp* 是否合拢来标识此次对求解器的调用是否需要生成求解器栈点(求解器栈进栈). 每次调用求解器时检查此 2 个指针,只有 2 个指针合拢时才生成求解器栈点,并将这 2 个指针分开(*slvrtop* 的值加 1),如果 *slvrtop*>*slvrtopp*,则不生成新的求解器栈点;另一方面,只有消解原子公式时才合拢这 2 个指针. 在这样的策略下,对于原子公式子目标,如果对谓词的消解过程中第 1 次调用求解器,由于此时 2 个指针合拢了,则生成新的求解器栈点,并分开 2 个指针,这样,对下一原子公式子目标消解之前的所有求解器调用(包括对原约束的求解)都不会生成新的求解器栈点;而对于原约束子目标,只有调用求解器求解并且 *slvrtop*=*slvrtopp* 时才生成求解器栈点. 这只能是在运行栈顶子目标消解过程中没调用求解器,紧接着对原约束求解第 1 次调用求解器的情况下. 所以应生成新的求解器栈点,使之与运行栈顶元素对应(如图 2 所示).

求解器栈进栈所做的主要工作包括:

生成求解器栈点(下一求解器栈点生成前求解器状态的改变记录在该栈点中);

idlecount 当前值保存在求解器栈点中;

idlecount 重置为 0.

这里需要解释一下 *idlecount* 当前值的概念. 由前面对于求解器栈进栈时机的讨论我们可以看出,求解器栈进栈可能发生在运行栈进栈前或进栈后. 在第 1 种情况下,*idlecount* 的值反映了运行栈到栈顶为止不涉及求解器的栈元素个数,而对于后者,由于运行栈进栈,造成了不涉及求解器的运行栈元素个数加 1 的假象,*idlecount* 的值错误地加 1,因此我们有如下的作法:只要在谓词消解过程中未涉及求解器,不管下面做什么,就将 *idlecount* 的值加 1,而如果紧接着又遇到源程序中的原约束求解器必被调用且 2 个指针合拢,从而生成求解器栈点,这时需将 *idlecount* 的值减 1 并将其记入该栈点,以抵消刚才 *idlecount* 盲目加 1 的效果.

这样的控制机制满足如下的 2 个不变式:

(1) *idlecount*>0 当且仅当 *slvrtop*=*slvrtopp*;

(2) *idlecount*=0 当且仅当 *slvrtop*>*slvrtopp*;

它们构成处理回溯的依据. 每当约束求解失败(这包括对原子公式消解失败和对原约束的求解失败 2 种情况)时,就进行回溯:

```
if (idlecount>0)     /* 对原子公式进行消解且不涉及求解器 */
    idlecount = idlecount - 1;
else { /* 对定义子句体内的原约束进行求解或对原子公式消解中涉及到求解器 */
      求解器栈出栈;
      idlecount 重置为求解器栈顶元素保存的值;
}
运行栈出栈; /* slvrtop 被修改 */
if(idlecount > 0)
    slvrtopp = slvrtop;
else
slvrtopp = slvrtop - 1;
```

2 PROLOG 系统存储优化技术概述

第 1 节所描述的推理机的控制机制在存储优化的情况下需要做深入的考虑和细微的修改.在有关 CLP 实现技术的文献[2]中没有这一方面的论述.本节先概述 PROLOG 存储优化技术[3],下节讨论如何在 CLP 系统中实现这些优化,主要问题仍是推理机与约束求解器的协调.

与通常允许递归过程的语言一样,PROLOG 在运行时也要维持一个运行栈.当前子目标与定义子句合一成功后进栈,接着求解该定义子句体内的子目标,当所有这些子目标解完以后,返回到引入该定义子句的子句去求解它的下 1 个子目标,但与其它递归语言不同的是,返回时不能出栈.这样,运行栈极易溢出.存储优化技术的核心思想是找到运行栈提前出栈的时机,常用的有确定末端结点优化和尾递归优化这 2 种优化技术,而且这 2 种优化技术必须配合使用.

2.1 确定末端结点优化

确定末端结点优化发生在对当前子目标消解成功后返回时.我们先来看几个有关的定义.若运行栈元素中的定义子句是子目标的最后 1 条定义子句,则该运行栈元素称为确定结点,否则称为回溯点.若运行栈元素是栈顶,且所存放的子目标的定义子句体为空,则称为末端结点.确定末端结点就是确定的末端结点.

根据 PROLOG 的操作模型,对当前子目标消解成功后,运行栈进栈,保存当前状态,而如果该结点是末端结点必定执行返回操作且必定返回到栈顶,而只要它又是确定结点,则它所保存的连接信息在回溯时就是无用的,因此无需保存,这样,在把该结点的内容拷贝出来后,即可让其出栈,这就是所谓确定末端结点优化(实际上,当用于消解当前子目标的定义子句为最后一条定义子句且体为空时,该子句头部与当前子目标合一成功后根本无需进栈,直接考虑下一子目标即可).

设某运行栈元素所含子目标的定义子句为:A:- $B1$, ..., Bn.

当 $B1$,...,Bn 均已解完时,必返回该结点.另一方面,可用归纳法证明:若 $B1$,...,Bn 所有解均已求出且一直执行确定末端结点优化,则此时该结点必在栈顶,只要又是一确定结点(即上述定义子句为最后 1 条定义子句),则对该结点可象上面一样进行出栈操作.

在确定末端结点优化条件下,由于返回后定义子句中的变量不会再被直接引用,因此它

的部分变量的例化信息无需保存,可以释放.

2.2 尾递归优化

尾递归优化的判断发生在合一前.考虑合一前的现场,若

(1) 当前子目标是当前子句最后 1 个子目标;

(2) 用于消解当前子目标的定义子句是当前子目标的最后 1 条定义子句;

(3) 当前子目标左边的子目标均已得到所有的解,且一直执行确定末端结点优化,则称当前子目标为确定的尾调用.

对于确定的尾调用,首先我们看到,在当前子目标与定义子句头部合一成功后,运行栈不需记录合一现场信息.因为由(2),回溯时不可能再找到下一条定义子句,因此可以省去进运行栈的操作.其次,(1)表明合一成功后,当前子句中的变量不再被直接引用,因此当前子句中的部分变量的例化信息也不必保存,可以释放.另外,(3)给出了一个极为快捷的判断条件.所有这些优化可能性称为尾递归优化.因为尾调用通常是递归的,故此得名,实际上无需一定是递归调用.

3 CLP 系统中存储优化算法的设计与实现

CLP 程序中,谓词定义子句体内既有原子公式,又有原约束.由于原约束在程序执行过程中不可能产生真正的回溯点,所以,CLP 系统中的尾递归优化和确定末端结点优化仍针对原子公式子目标进行,而且定义和判断条件不变.在具体实现上,由于运行栈在确定末端结点优化下提前出栈、在尾递归优化下免于进栈,因此,必须修改 *idlecount*、*slvrtop* 和 *slvrtopp* 3 个全局控制变量以满足 2 个控制不变式,使推理机和求解器协调工作,这就是设计优化技术下求解器栈操作的核心思想.

确定末端结点优化条件下,如果 *idlecount*>0,则说明从与求解器栈顶对应的运行栈元素开始,到运行栈顶,有不涉及求解器的栈点,因此,将 *idlecount* 的值减 1,运行栈出栈即可.需要特别注意的是,如果 *idlecount*=1,那么,除了上述操作外,还需调整指针 *slvrtopp*.这是因为优化前的状态满足不变式(1),由于 *idlecount* 的值优化后变为 0,所以必须调整 *slvrtopp* 以满足不变式(2).否则,*idlecount*=0,说明运行栈顶元素对应求解器栈顶.这时,如果运行栈出栈,那么与之对应的求解器选择点应与运行栈次栈顶对应.有 2 种情况:如果次栈顶没有对应的求解器栈点,则只需将其保存的 *idlecount* 的值减 1 即可;否则,该求解器栈点也必须取消掉,但是,与推理机选择点不同,其中的求解器状态的改变信息必须归到上 1 个求解器选择点中保存起来,也就是说合并这 2 个求解器栈点,并调整指针 *slvrtopp* 以满足不变式(2).确定末端结点优化可以概括为:

```
if(idlecount > 0) { /* 运行栈不对应求解器栈顶 */
    idlecount = idlecount - 1;
    if(idlecount == 0) /* 运行栈次栈顶有相应求解器栈元素,*/
      slvrtopp = slvrtop - 1; /* 打开合拢的 slvrtop 和 slvrtopp 指针 */
}
else { /* 运行栈顶对应求解器栈顶 */
    记求解器栈顶元素保存的 idlecount 值为 idle;
    if( idle > 0 ) /* 运行栈次栈顶与求解器无关,*/
```

```
        idle 的值减 1; /* 将求解器栈顶元素与之对应 */
      else { /* 运行栈次栈顶有对应的求解器栈点,合并求解器栈顶和次栈顶 */
        将求解器栈顶元素保存的状态修改信息合并到次栈顶元素中;
        求解器栈出栈; /* slvrtop--; */
        slvrtopp--;
      }
    }
    运行栈顶元素出栈;
```

尾递归优化条件下,对原子公式消解成功后运行栈不进栈,所以即使对原子公式的消解过程中调用求解器,也不应生成新的求解器选择点,而将对求解器所作的修改累计到现有的求解器选择点中.从3个全局控制变量看,在尾递归条件下,不应合拢 *slvrtop* 和 *slvrtopp* 指针,对谓词消解成功后,*idlecount* 的值不增加.因此,只有尾递归优化不成立时,才有下面有关求解器栈的操作,而在尾递归优化成立时,这些求解器栈操作均可免去.

```
if(当前子目标是原子公式) { /* 在对谓词消解前 */
    判断是否确定的尾调用,将尾递归优化标志保存在 tro 中;
    if(tro == 0) { /* 尾递归优化条件不成立 */
      slvrtopp = slvrtop;
    }
    if(调用求解器 ∧ slvrtopp == slvrtop) { /* 在谓词消解过程中 */
      生成新的求解器栈栈点;
      slvrtop = slvrtop + 1;
    }
}
...
if(tro == 0) { /* 对谓词消解成功后 */
      运行栈进栈;
      if(slvrtopp == slvrtop) /* 未涉及求解器 */
        idlecount 的值加 1;
    }
```

4 结束语

本文所述存储优化情况下推理机与约束求解器的协调技术已成功地应用于BPU—CLP(R)系统的实现中.在有关CLP实现的文献[2]中未见到有关这一问题的讨论.

对于一个完整的逻辑程序设计系统来说,除了深度优先加回溯的基本控制机制外,还有附加的控制机制(Cut)以及数据库的动态维护谓词 asserta,assertz 和 retract 等.由于对它们的解释均涉及对运行栈正常状态的修改,所以,在实现过程中要时刻保持2个控制不变式以及全局控制变量的语义,才能使推理机和约束求解器协调工作.关于它们的具体实现,在此不再赘述.读者只要明白了上述的控制机制,它们的实现技术是可以想见的.

参考文献

1 Jaffar J, Lassez J L. Constraint logic programming. In: Proc. of the 14th ACM Symposium on Principles of Programming Languages, Munich, 1987. 111~119.

2 Jaffar J, Michaylov S, Stuckey P J *et al*. The CLP(R) language and system. Yorktown Heights, NY: IBM Research Report RC16262(#72336), 1990.

3 刘慕年,曹德和. PROLOG 语言,它的应用与实现. 北京:科学出版社,1990.

4 张秀珍. 一个 CLP(R)系统的研制及其部分演绎的研究[硕士论文]. 北京工业大学,1994.

SYNCHRONIZATION OF THE INFERENCE ENGINE WITH THE CONSTRAINT SOLVER IN A CLP SYSTEM

Zhang Xiuzhen　Liu Chunnian

(*Department of Computer Science　Beijing Polytechnic University　Beijing*　100022)

Abstract　This paper discusses the synchronization of the inference engine with the constraint solver in the implementation of the BPU－CLP(R) system. Synchronization is needed in backtracking. This method can solve the synchronization problem properly even in the presence of storage optimization in the system.

Key words　PROLOG, CLP, CLP(R), storage optimization techniques.