第38卷 第5期
2011年5月
计 算 机 科 学
Computer Science
Vol. 38 No. 5
May 2011

逻辑语言剪枝算子的过程语义及其实现

李慧琪[1] 赵致琢[1,2]
(厦门大学计算机科学系 厦门 361005)[1] (仰恩大学计算机科学系 泉州 362014)[2]

摘 要 论述了逻辑程序设计中剪枝算子的作用及传统剪枝算子的过程性语义和说明性语义不一致问题;介绍了新型逻辑程序语言 Gödel 中的 commit 剪枝算子;通过引入一组定义描述其过程语义,并进一步阐述了剪枝算子和延迟计算规则之间的关系,讨论了 Gödel 语言的剪枝策略及控制机制,从而为逻辑程序语言的实现提供了依据。

关键词 逻辑程序设计,剪枝算子,延迟计算

中图法分类号 TP312 **文献标识码** A

Procedural Semantics and its Implementation of Pruning Operators in Logic Programming Language

LI Hui-qi[1] ZHAO Zhi-zhuo[1,2]
(Department of Computer Science, Xiamen University, Xiamen 361005, China)[1]
(Department of Computer Science, Yangen University, Quanzhou 362014, China)[2]

Abstract The use of pruning operators in logic programming is to reduce the search space of computations. The importance of pruning operator in logic programming was discussed. However, the implementation of traditional pruning operator may cause some semantic problems. We discussed the Gödel pruning operator, called the commit, which could be used to prune away parts of a search tree and can affect the completeness of the search procedure. In this perspective, we proposed the method to realize the control facility in logic programming language which could support the fully implementation of Gödel language.

Keywords Logic programming, Pruning operator, Delay computation

1 引言

自 Prolog 语言诞生以来,说明性程序设计范型的发展逐渐受到重视,对各种逻辑程序语言及操作语义模型等研究不断深入,但逻辑程序设计语言的效率问题始终是研究的重点之一。通常有3种途径提高其执行效率:第一,编译技术及其优化极大地提高了逻辑程序的效率,其中最具影响的是 Warren 提出的抽象机(Warren's Abstract Machine, WAM)[1];第二,借助并行计算提高执行速度,这一研究方向发展了并行逻辑语言 PARLOG, Concurrent Prolog 等;第三,引入剪枝算子,提供更灵活的控制机制,以有效地缩小求解搜索的范围。

剪枝算子的重要性体现在增强逻辑语言的表达能力和提高逻辑程序执行效率两个方面。在逻辑程序设计的发展过程中,绝大多数逻辑程序语言都引入了剪枝算子。例如,人们熟悉的 Prolog 语言的"截断"(cut)、并发逻辑语言中的"commit","once","conditionals"等[2]。为提高程序的效率,通常提倡在程序的子句定义中使用剪枝算子,但剪枝算子是过程性的而非说明性的。众多研究表明,剪枝算子带来了语义问题,这可能导致计算的不完备和(或)不正确,降低了程序的可读性和可靠性,同时给程序的分析和转换带来了困难[3]。逻辑程序设计语言需要怎样的剪枝算子,是否能够提供功能足够强且不会破坏程序说明性语义的剪枝算子,这些问题仍然是逻辑程序设计语言研究的一个重点。

本文将结合新型逻辑程序设计语言 Gödel 讨论剪枝算子的效率及其实现方式。首先指出 Prolog 语言提供的传统剪枝算子的不足,然后讨论 Gödel 语言提供的通用剪枝算子 commit,它扩展了并发逻辑程序语言的剪枝算子,支持不同风格的程序设计。语言设计者在继承 Prolog 的基础上发展其说明性语义,试图为程序提供有效的控制机制,尽可能缩小说明性语义和过程性语义之间的差距。

2 典型的剪枝算子

Prolog 是最具影响的逻辑程序设计语言,诸多逻辑语言是其扩展和增强。为此有必要先回顾 Prolog 中的剪枝算子[1]。

逻辑程序的计算过程是对 SLD-搜索树的完全遍历过程。但搜索树的某些分枝可能得不到解,存在失败分枝、无穷分枝或冗余分枝,因此需要在程序中对计算过程进行显式控制。为此,Prolog 中添加了非逻辑的控制成分 cut,称为"截断"(或直译为"割"),其语法形式是'!',可以出现在谓词的子句体或目标子句中。

非形式地说,Prolog 的截断机制用于静态地阻断回溯,剪除搜索树中尚未搜索的部分。一旦得到需要的解,就忽视该谓词的其他可选子句和子句中早期目标的可选解,通过删除

到稿日期:2010-06-08 返修日期:2010-09-09 本文受福建省自然科学基金(A030007),仰恩大学科学技术研究经费资助。
李慧琪 讲师,主要研究方向为逻辑程序设计;赵致琢 教授,主要研究方向为逻辑程序设计、分布式计算。

无用的搜索和回溯来提高计算过程的效率。

从说明性语义来看，cut 是一个特殊的 0 元谓词！/0，代表恒真的基原子，出现在子句体中任何位置都不会影响程序的说明性语义，但 cut 会影响程序的过程性语义。假设目标子句 G 为：

$? \text{-} A_1, \cdots, A_{k-1}, A_k, A_{k+1}, \cdots, A_m$

A_k 的谓词定义子句 C 为：

$A \leftarrow B_1, \cdots, B_i, !, B_{i+1}, \cdots, B_q$

若在计算过程中 A_k 和 A 合一成功，得到消解式 G'，则称 A_k 为截断点，控制在回溯时越过从截断点到'！'的所有子目标($A_k, B_1, \cdots, B_i$)，不再考虑和它们有关的其他规则，直接求 A_{k-1} 的下一个解。这意味着搜索树中以 C 的父目标子句为根的子树中尚未被搜索的部分均被剪除。

正确应用截断能剪除搜索树中不可能得到解的分支来有效缩小搜索空间，提高程序的效率，这称为"绿色截断"。但 cut 使用不当，有可能删除成功的分支，改变程序含义，破坏 SLD 归结法的完备性，并影响说明性语义和过程性语义的一致性，这称为"红色截断"。关于截断带来的副作用的讨论很多，主要集中在"截断"的顺序性导致的语义问题及其对搜索过程完备性的影响。

并发逻辑语言中的剪枝算子"commit"对 cut 作了改进，没有了和子句次序相关的顺序性带来的语义问题。但是，包含"commit"的逻辑程序在程序转换下是不封闭的，这也带来了其他的问题。Naish 在文献[9]中对各逻辑语言中定义的众多剪枝算子的优缺点作了详细说明，并提出了一个问题：是否能够提供功能足够强且不破坏程序的说明性语义的剪枝算子？Gödel 语言在这方面作出了改善，试图给出一种更具说明性的剪枝算子。

3 Gödel 语言的剪枝算子

Gödel 语言是继 Prolog 语言之后的新型通用逻辑程序设计语言，在 1990 年代中期由 J. W. Lloyd 等人设计开发[3]。与 Prolog 相比较，它充分吸取了高级程序语言发展的成分，如类型、多态、模块、延迟计算等，支持灵活的计算规则，允许动态调度控制机制。剪枝算子正是其控制机制的重要方面。

3.1 剪枝算子的非形式定义

首先给出剪枝算子的非形式定义。Gödel 语言中提供的剪枝算子称为"委托"或"提交"(commit)，其一般形式为{*formula*}_*n*。另有两种简化的剪枝算子形式：条形提交(bar commit)和单解提交(one-solution commit)[3]。

条形提交的语法形式为"|"，表示合取关系，其作用范围是语句体中出现在其左边的公式。条形提交只求出"|"辖域内公式的一个解，而搜索树中包含"|"的谓词定义的其它子句相关的分支均被剪除。例如，Del 子句定义如下：

$\text{Del}(x,[x|y],[y]) \leftarrow |.$

$\text{Del}(x,[y|z],[y|w]) \leftarrow x \neq y \,|\, \text{Del}(x,z,w).$

求解目标← Del(1,[1,2,3],y).，目标和 Del 的第一条子句头部匹配，得到 y=[2,3]，第二条子句不必进行推导，相应地在搜索树上的分支被删除。由于子句之间无顺序关系，Gödel 中剪枝没有 Prolog 中截断的顺序特征。

单解提交形式为{*formula*}，作用范围是"{"和"}"内所包含的公式。这一方式对搜索树进行剪枝，只寻求作用范围内的一个解，其他可选的正确答案将被剪除。例如，谓词 Perm(x,y)表示列表 y 是列表 x 中元素的一种排列组合，目标子句采用单解剪枝形式：←{Perm([1,2,3],x)}.，则只得到一个解 x=[1,2,3]就返回，将其余可能的 5 种排列形式的解剪除。

commit 的一般形式为{*formula* }_*n*，其中整数标号 n 指示 commit 的作用范围。若"{"和"}"内所包含的公式成功，其他有相同标号 n 的语句被删除。一般形式的 commit 剪枝能力强，可适用于元程序的构造。条形提交和单解提交都可以转换为一般形式的 commit。

定义包含 commit 的语句和目标的语法形式：

Body→[*cFormula*(f)] '|' [*cFormula*($f1$)]

　　|*cFormula*(f)

cFormula(0)→(*cFormula*(f))

　　|{*cFormula*(f)}_*Label*

cFormula(2)→*cFormula*(f) & *cFormula*($f1$)

cFormula(f)→*Formula*(f)

其中，$f \leqslant 1$，$f1 \leqslant 2$，[…]中的项出现 0 或 1 次。

3.2 剪枝算子的过程语义

由于逻辑程序的计算实质上是 SLD 树的搜索算法，为了定义剪枝算子的过程语义，先引入带 commit 的逻辑程序及其求解目标的搜索树的定义，这些定义是在 Lloyd 给出的定义[4]基础上针对公式中出现 commit 的扩充。

定义 1 令 P 为程序，G 为目标子句。$P \cup \{G\}$ 的搜索树是一棵以 G 为根结点的有穷树，内部结点标识为子目标，搜索树中的结点的每个子结点对应于一步扩展中创建的子结点。

搜索树的一个分枝对应一个推导。对应于成功推导的分枝称为成功分枝，它是从树根到一个空子句□的一条路径。

计算规则是从目标子句到其中一个原子的映射(即从目标子句中选择子目标的规则)。搜索树是按计算规则通过一系列的扩展(extension step)和剪枝创建的。不同计算规则产生的搜索树在大小和形状上不同。搜索树的初始状态仅包含目标 G，若树中有非空叶子结点，则对这一结点进行扩展。此结点称为当前目标，在扩展中将当前目标的儿子结点加入搜索树。剪枝则从搜索树中删除一棵子树。Gödel 搜索树的扩展步骤有 4 种不同方式：扩充(expansion)、替换(replacement)、消去(elimination)和分裂(splitting)步骤[3]。对于含有 commit 的程序只有第一种扩展方式：选中当前目标体中的原子公式，用该原子定义的谓词进行扩展(即 SLD-消解的推导)。结点 G 的一步扩展将为从 G 中选中的文字匹配的每一条子句添加一组子结点。

定义 2 设←W 是程序 P 对应的搜索树的当前目标，扩展步骤定义如下：

扩充步骤(expansion step)：要求选择的子公式是原子 $R(s_1, \cdots, s_n)$，$n \geqslant 0$。设 $R(t_{11}, \cdots, t_{1n}) \leftarrow B_1, \cdots, R(t_{k1}, \cdots, t_{kn}) \leftarrow B_k$ 是 P 中关于 R 的定义的所有语句(经过标准化，保证这些语句不含有变元或搜索树中已经出现过的 commit 标号)。这个步骤添加儿子结点 $\leftarrow W_1, \cdots, \leftarrow W_k$，其中，$W_i$ 是 W 用 B_i' 替换原子 $R(s_1, \cdots, s_n)$ 得到的，而 B_i' 是 B_i 增加顶层合取项 $s_1 = t_{i1} \,\&\, \cdots \,\&\, s_n = t_{in}$ 得到的，$i \in \{1, \cdots, k\}$。

替换步骤(replacement step)：添加儿子结点 $\leftarrow W_1\theta$，其中

W_1 是 W 用不含有 commit 的公式 V_1 替换选中的子公式 V 得到的，而 $\forall(V\theta\leftrightarrow V_1\theta)$ 是 P 的 completion 的逻辑结果的规范形式，θ 为替换。

消去步骤(elimination step)：要求选择的子公式形如 $\exists x_1\cdots\exists x_n V$。这个步骤添加儿子结点 $\leftarrow W_1$，其中 W_1 是 W 用 V 替换选中的子公式得到的。

分裂步骤(splitting step)：要求选择的子公式形如 $V_1\vee V_2$。这个步骤添加儿子结点 $\leftarrow W_1$ 和 $\leftarrow W_2$，其中 W_i 是 W 用 V_i 替换选中的子公式得到的，$i\in\{1,2\}$。

下面引入概念，说明何时进行剪枝以及如何选择被剪枝的子树。

定义 3 令 T 为搜索树，G_0 是 T 中的非叶子结点，G_1 是 G_0 的子结点，称 G_1 是 G_0 的 l-子结点(*l-child*)，如果：

1)G_0 包含一个标识为 l 的 commit，且选中的子公式在 commit 的作用域中，或者

2)G_1 是 G_0 经过一步扩展导出的，且扩展步骤使用的子句包含标识为 l 的 commit。

定义 4 令 T 为搜索树，称搜索树 T' 是 T 经一步剪枝(pruning step)得到的，如果满足下列条件：

1)T 的结点 G_0 有不同的 l-子结点 G_1 和 G_2，且 T 中结点 G_2' 处于 G_2 的同一层或下层，G_2' 不包含标识为 l 的 commit；

2)T' 是 T 删除以 G_1 为根的子树之后得到的，则 G_1 称为截断点(cut node)，二元式 $\langle G_2,G_2'\rangle$ 称为剪枝依据(explanation)。

以下面的程序片断为例说明上述定义。

例 1 考虑子句如下：

$P(x)\leftarrow\{Q(x)\}$_1.

$P(x)\leftarrow\{R(x)\}$_1.

$Q(A)$.

$Q(B)$.

$R(C)$.

$R(D)$.

求解目标 $\leftarrow P(w)$.。根据子句定义的顺序，在扩展步骤先后选择 P 和 Q 的第一条子句，得到推导形式 $P(w)\Rightarrow\{Q(w)\}$_1 $\Rightarrow\square(w=A)$，只得到目标的一个解 $(w=A)$，其余的解 $(w=B)$，$(w=C)$ 和 $(w=D)$ 均被删除。初始搜索树仅包含一个目标结点，执行所有可能的扩展步骤，可以得到搜索树，如图 1 所示(选中的子目标加下划线表示)。

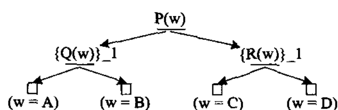

图 1 程序 1 对应的搜索树

结点 $\{Q(w)\}$_1 包含标号为 1 的 commit 且选中的原子 $Q(w)$ 在 commit 的作用域内，根据定义 3 的条件 2，结点 $\langle\square(w=A)\rangle$ 是结点 $\langle\{Q(w)\}$_1$\rangle$ 的 1-子结点。又根据定义 3 的条件 1，结点 $\langle\{Q(w)\}$_1$\rangle$ 是结点 $\langle\{P(w)\}\rangle$ 的 1-子结点，因为前者是后者用带标号为 1 的 commit 的子句经过一步扩展导出的。

对这棵搜索树可以进行一组剪枝操作：第一步，对叶子结点 $\langle\square(w=B)\rangle$ 进行剪枝 Pruning1。此时，$\langle\square(w=B)\rangle$ 为截断点，二元式 $\langle\square(w=A),\square(w=A)\rangle$ 为剪枝依据；第二步，剪枝操作 Pruning2 对结点 $\langle\{R(w)\}$_1$\rangle$ 进行，其中 $\langle\{R(w)\}$_1$\rangle$ 为截断点，二元式 $\langle\{Q(w)\}$_1$,\square(w=A)\rangle$ 为剪枝依据。经过剪枝之后得到的搜索树如图 2 所示(搜索树中的虚线表示该分枝被删除)。

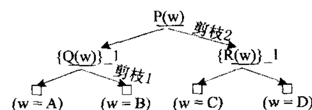

图 2 经过剪枝的的搜索树

最后，给出计算结果的定义。

定义 5 令 P 为程序，G 为目标子句。$P\cup\{G\}$ 的计算结果由 G 中的自由变元经置换得到，该置换是一组连续置换的复合，其中每个置换与搜索树从根结点到空目标叶子结点的路径上的各结点相关。

3.3 剪枝算子的分析

Gödel 语言的设计者仅仅描述了正确剪枝操作集合，而没有指定何时、如何执行剪枝步骤，甚至没有明确剪枝操作是可选还是强制的，这给语言的实现和程序含义的理解带来困难。为了给出剪枝算子的实现依据，必须考虑下列 4 个约束条件：

(1)剪枝操作是强制进行的，不是可选的；

(2)剪枝步骤的执行顺序必须是确定的；

(3)剪枝步骤必须是一致的；

(4)剪枝操作应当是充分的。

更进一步考虑剪枝操作应当满足的单调性[8]：令 CA 为搜索树到计算结果集合的映射，若 T_{i+1} 由 T_i 经扩展或剪枝得到，有 $CA(T_i)\subseteq CA(T_{i+1})$ 成立，则称 CA 是单调性的。若不考虑剪枝算子，CA 显然具有单调性。若 T_{i+1} 由 T_i 剪枝得到，$CA(T_{i+1})$ 不包含被剪枝的子树的解，为了保证计算结果的单调性，有必要对计算结果的定义作出修正，为此引入新的定义如下。

定义 6 令 T 为搜索树，l 为剪枝的标号。如果 T 中以 G 为根的子树包含的结点中不出现标号 l 且叶子结点不能继续扩展，则称结点 G 是 l-无关的(l-impotent)。

定义 7 令 T 为搜索树，如果某个分枝中的每个 l-子结点 G 的兄弟结点都是 l-无关的，则称该分枝为持久分枝(persistent)。

定义 8 令 P 为程序，G 为目标子句。$P\cup\{G\}$ 的计算结果由 G 中的自由变元经置换得到，该置换是一组连续置换的复合，其中每个置换与搜索树从根结点到空目标叶子结点的一条持久分枝中的各结点相关。

下面给出的例子说明了正确的剪枝顺序以及剪枝的充分程度。

例 2 考虑子句如下：

$P(x)\leftarrow\{Q(x)\}$_1 & $\{R(x)\}$_2.

$P(x)\leftarrow\{S(x)\}$_1.

$P(x)\leftarrow\{T(x)\}$_2.

$Q(A)$.

$R(A)$.

$S(B)$.

$T(C)$.

求解目标 $\leftarrow P(w)$.，生成的搜索树如图 3 所示。

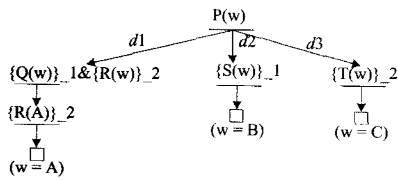

图3 程序2对应的搜索树

为了叙述方便，用 d_1，d_2，d_3 表示从搜索树根结点出发的3个分支对应的推导。对搜索树进行的剪枝操作有两步：(P1)根据 d_1 删除 d_3，剪除以〈{$Q(w)$}_1 & {$R(w)$}_2，□($w=A$)〉为依据的截断点〈{$T(w)$}_2〉；(P2)根据 d_2 删除 d_1，剪除以〈{$S(w)$}_1，□($w=B$)〉为依据的截断点〈{$Q(w)$}_1 & {$R(w)$}_2〉。经过剪枝得到的搜索树只有一个分支，对应推导 d_2：$P(w)$⇒{$S(w)$}_1⇒□($w=B$)。但这存在问题：第二步剪枝P2和P1提交的解不一致。事实上，推导过程 $P(w)$⇒{$Q(w)$}_1 & {$R(w)$}_2⇒{$R(A)$}_2⇒□($w=A$)得到的解是对{$R(w)$}_2的提交，也是对{$Q(w)$}_1的提交。因此，一旦确定了剪枝依据，应当进行充分的剪枝：(P1)根据 d_1 删除 d_3；(P2)根据 d_1 删除 d_2。

3.4 与延迟计算结合的剪枝操作

由上面的讨论可以看出，如何选择扩展原子决定了搜索树的伸展方式，因此剪枝操作的实现与计算规则相关。逻辑程序的反驳-消解过程依赖于计算规则和搜索策略，其中计算规则是从目标子句到其中一个原子的映射，也就是从给定子句体中挑选子目标的函数，而搜索策略是从程序中选取匹配子句的策略。标准的计算规则采用最左文字优先原则，即总是选取最左子目标，但这种简单的做法易于导致计算陷入无穷循环。不同的计算规则得到的搜索树在大小和形状上可以差别很大。但可以证明，给定程序 P 和目标 G 的所有搜索树或者有无穷多个成功分支，或者有相同个数的有穷多个成功分支[5]。因此，Gödel语言中采用更加灵活的计算规则，进一步提高了搜索树的剪枝效率。

Gödel语言的计算规则引入延迟计算，可选择的扩展文字并不限制一定是最左文字，而是根据相应谓词的延迟声明中指定的条件，判断参数是否已实例化来决定子公式需要延迟调用或是能够进行扩展。若没有出现延迟声明，Gödel将沿用缺省的最左文字优先计算规则，这比Prolog采用的计算规则更为灵活。由于Prolog总是选择最左子目标进行扩展，子目标的书写顺序将直接影响搜索树的扩展，规则中子公式顺序安排不当将易于导致程序陷入无穷运算。Gödel避免了这一点，从而可提高计算效率。

延迟声明的语法形式为：

DELAY *Atom* UNTIL *Cond*.

其中，*Atom* 为原子(即谓词)，*Cond* 的文法规则如下：

```
Cond→Cond1 | Cond1 & AndSeq
     | Cond1 ∨ OrSeq
Cond1→NONVAR(Variable)
     | GROUND(Variable)
     | TRUE
     | (Cond)
AndSeq→Cond1 | Cond1 & AndSeq
OrSeq→Cond1 | Cond1 ∨ OrSeq
```

下面以排序程序为例说明延迟声明的使用。为了创建一个朴素的排序程序，首先引入Sorted谓词如下：

```
PREDICATE Sorted:List(integer).
DELAY
  Sorted([ ]) UNTIL TRUE;
  Sorted([ _ ])UNTIL TRUE;
  Sorted([ x,[ y | _ ]]
    UNTIL GROUND(x) & GROUND(y).
Sorted([ ]).
Sorted([ _ ]).
Sorted([ x,y|z ])←x≤y & Sorted([ y|z]).
```

可以看出，谓词sorted根据递归定义给出：空列表和长度为1的列表是有序的；若列表中第1个元素小于等于第2个元素，且从第2个元素开始的子列表有序，则该列表有序。在这个例子中，采取3种不同延迟模式：

1)若列表参数为空，则列表有序，延迟条件声明为TRUE，表示无须延迟；

2)若列表参数的长度为1，那么无论该列表元素取值如何，均已有序，延迟条件声明为TRUE，表示无须延迟；

3)若参数列表的长度大于1，则计算将延迟至条件满足时进行，这里延迟条件是列表中的头两个元素已约束为确定的值。

利用谓词Sorted和Permutation定义Sort模块如下：

```
MODULE Sort.
IMPORT Lists.
PREDICATE Sort:List(integer) * List(integer).
Sort(x,y)←Sorted(y) & Permutation(x,y).
    ——Sort(a,b) 表示列表 a 经排序产生列表 b
```

程序的过程性语义解释如下：当 x 和 y 的值至少有一个已确定时，SlowSort才能计算。若 y 为约束的，则Sorted无须延迟。若调用结果为成功，则当 x 为约束的时，Permutation验证 x 是否为 y 的一个排列；若 x 为非约束的，则 x 的值可以取 y 的所有排列；若 y 为非约束的而 x 为约束的，则Sorted将被延迟，先调用Permutation将 y 实例化为 x 的一个排列，但不必完成整个排列过程，只要求得排列的前两个元素即可。若这两个元素已表明 y 不是有序的，那么计算不必继续；若前两个元素有序，则Sorted可进行递归调用，Permutation继续产生排列的其它元素。一旦Sorted在某一点失败，Permutation将回溯，开始计算另一排列可能。与Prolog的对应朴素排序的计算过程比较可知，利用延迟计算效率显著提高。

为了简化实现，Gödel语言对DELAY声明作出限制：同一个谓词的一组延迟声明中出现的任意两对原子，都不能有共同的实例值，从而避免了一个延迟条件满足但另一个延迟条件不满足的矛盾情况发生。延迟声明头部出现的原子必须是Contraint-free(指类型限制为integer，…，set(τ))，且不能包含重复变量，这使得合一时不必考虑和特定类型相关的等词，从而简化了实现。

4 剪枝策略及控制机制的实现

逻辑程序的计算是通过搜索树完成的，搜索树则通过对初始搜索树运用一系列扩展和剪枝操作步骤生成。因此，计算可以看成搜索树的序列 T_0，T_1，…，其中 T_0 为初始目标 G，T_{i+1} 由 T_i 经扩展或剪枝得到。每个扩展步骤包括两次选择：第一次选择是关于选取搜索树中的某个结点进行扩展；第二次选择由计算规则决定从当前目标结点中选取某一子公式进行扩展。

(下转第148页)

DsStream。算法既保持了基于密度的聚类算法可以发现任意形状的聚类和对噪声数据不敏感的优点，又能够动态地调整全局参数，故算法是参数不敏感的。算法应用了滑动窗口技术对数据进行动态更新，并提供在线维护策略，从而能够在保障聚类结果尽可能精确的基础上有效地控制内存的消耗。

参考文献

[1] Barbar D. Requirements for Clustering Data Streams[J]. SIGKDD Explorations, 2003, 3(2): 23-27

[2] Guha S, Mishra N, Motwani R, et al. Clustering Data Stream[C]// FOCS 2000. 2000: 359-366

[3] O'Callaghan L, Mishra N, Meyerson A, et al. Streaming-Data algorithms for high-quality clustering[C]// ICDE Conf. 2002: 685-704

[4] Aggarwal C C, Han J, Wang J, et al. A framework for clustering envolving data streams[C]// VLDB. 2003: 81-92

[5] Zhang T, Ramakrishnan R, Livny M. BIRCH: An Efficient Data Clustering Method for Very Large Databases[C]// ACM SIGMOD Conference. 1996: 103-113

[6] Cao Feng, Ester M, Qian Weining, et al. Density-based clustering over an evolving data stream with noise[C]// 2006 SIAM Conference on Data Mining. 2006: 326-337

[7] Ester M, Kriegel H P, Sander J, et al. A density-based algorithm for discovering clusters in large spatial databased with noise[C]// Proc of KDD. 1996

[8] Chang Jianlong, Cao Feng, Zhou Aoying. Evolving datastreams clustering based on the sliding window[J]. Journal of Software, 2007, 17(4): 905-918

[9] Udommanetanakit K, Rakthanmanono T, Waiyamai K. E-Stream: Evolution-based Technique for Stream Clustering[C]// Proceedings of ADMA. Berlin Heidelberg: Springer-Verlag, 2007: 605-615

（上接第 126 页）

对于不存在 commit 的逻辑程序而言，做出这两次选择已经足够。对于出现了剪枝算子的程序，则不仅同样需要做出这两次选择，而且剪枝算子还引入了计算过程中新的不确定性：给定搜索树 T_i，首先需要选择下一步操作是扩展还是剪枝。在选定剪枝操作后，还需要决定对哪棵子树进行剪枝，即选择剪枝的依据并选择截断点。

值得注意的是，这些选择的具体实现可以有不同的方式。通常，在逻辑程序语言的实现中，采用积极的剪枝策略，这意味着在 commit 的范围内尽可能地进行搜索树的修剪，以达到提高程序效率的目的。这里，给出的积极剪枝操作实现策略如下[8]：

(C0) 若允许剪枝，则选择剪枝操作，否则选择扩展操作；

若(C0) 为扩展步骤，则

(C1) 选择最新建立的非空结点进行扩展(深度优先)；

(C2) 根据计算规则，选择结点中的最左非延迟子公式进行扩展；

若(C0) 为剪枝步骤，则

(C3) 选择剪枝的依据 J；

(C4) 选择 J 的所有截断点。

图 4 给出了 Gödel 语言控制机制的简要示意图，略去了各子过程的实现算法。该算法的思想已用于一个 Gödel 语言编译器雏形的研发，其中细节如包含动态类型的合一处理、延迟检查仍在进一步优化中[10]。

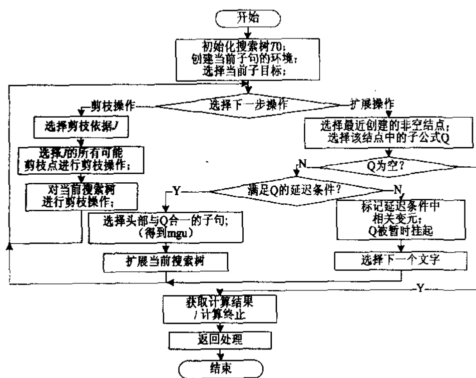

图 4　Gödel 语言控制机制示意图

结束语　逻辑程序设计有别于命令式程序设计，其特点在于计算存在不确定性和一致匹配的计算方式。其中，不确定性分为“不知不确定性”和“无关不确定性”。前者指的是通过回溯进行计算求解，后者可以通过剪枝操作去除无关的搜索空间。剪枝操作对提高逻辑程序的执行效率非常重要。在逻辑语言的发展中，Gödel 改进了传统逻辑语言中的语义问题，设计了一个更具说明性的剪枝算子，提供了 3 种不同的 commit 形式以满足不同的需要。剪枝算子和谓词的延迟声明结合更具有实用价值，事实上就是已知了谓词中参数的使用模式，能够进行更有效的剪枝。本文引入了带 commit 的逻辑程序及其求解目标的搜索树的定义，对剪枝操作的执行剪枝步骤作出了规定，为剪枝算子的实现提供了依据，并结合带有延迟声明的计算规则集合，讨论了 Gödel 的控制机制，为语言的编译实现提供了参考。

参考文献

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

[2] Saraswat V A. Concurrent Constraint Programming Languages [M]. MIT Press, 1993

[3] Hill P M, Lloyd J W. The Gödel programming Language[M]. London: MIT Press, 1994

[4] Lloyd J W. Foundation of Logic Programming (2 ed) [M]. Springer-Verlag, 1987

[5] Maluszynski N. Logic, Programming and Prolog (2 ed) [M]. John Wiley& Sonds Ltd, 1995

[6] Börger E. Logic+Control Revisited: and Abstract Interpreter for Gödel programs[M]. Levi G. editor. Advance in Logic Programming Theory. Oxford Univ. Press, 1994

[7] Jeffery D. Expressive Type Systems for Logic Programming Language[D]. 2002

[8] Brogi A, Buarino C. Pruning the Search Space of Logic Programs [J]. Lecture Notes in Computer Science, 1996, 1050: 35-49

[9] Lee Naish. Pruning in logic programming [R]. 95/16. Melbourne, Australia: Department of Computer Science, University of Melbourne, 1995

[10] Li Hui-qi, Zhao Zhi-zhuo. A Polymorphic Type System in Logic Programming[C]// Proceedings of 2008 3rd International Conference on Intelligent System and Knowledge Engineering (ISKE2008). 2008: 125-130