目标独立的 Prolog 程序路径依赖分析语义*)

赵岭忠[1,2]　古天龙[2]　钱俊彦[2]

(西安电子科技大学电子工程学院　西安 710071)[1]　(桂林电子科技大学计算机与控制学院　桂林 541004)[2]

摘　要　在 Prolog 程序分析中,考虑程序的执行路径和非逻辑的 cut 操作可提高程序分析的精度。当前用于 Prolog 程序路径依赖分析的语义因依赖于程序执行的目标而不适合目标独立的程序分析。为此,本文采用了一种携带路径信息并允许 cut 操作的 Prolog 抽象语法,在此基础上给出了 Prolog 的操作语义和一种目标独立的标号树(LT)语义,并证明了 LT 语义相对于操作语义的正确性。LT 语义可作为目标独立的 Prolog 程序路径依赖分析的基础。

关键词　程序分析,Prolog 语义,目标独立,上下文信息,抽象解释

Goal-independent Semantics for Path Dependent Analysis of Prolog Programs

ZHAO Ling-Zhong[1,2]　GU Tian-Long[2]　QIAN Jun-Yan[2]

(Electronic Engineering School, Xidian University, Xi'an 710071)[1]

(School of Computer and Control, Guilin University of Electronic Technology, Guilin 541004)[2]

Abstract　Considering the execution path and cut operators of a Prolog program can improve the precision of program analysis. Known semantics for Prolog either makes use of limited amount of path information and hence leads to less precise analysis or is goal-dependent and therefore not suitable for goal-independent program analysis. This paper deals with the problems by proposing a goal-independent labeled tree semantics for Prolog with cut, which makes it possible to use call strings as context information to improve Prolog program analysis. This semantics is shown to be correct w. r. t the operational semantics presented in this paper.

Keywords　Program analysis, Prolog semantics, Goal-independence, Context information, Abstract interpretation

1　引言

程序分析是自动分析计算机程序行为的过程,其结果可作为编译器或部分求值器(partial evaluator)的输入用于程序的优化。目前程序分析常采用基于抽象解释的方法[7]。程序 P 的抽象解释可视为 P 在非标准语义域上的一次运行,其中非标准语义域刻画了分析人员所关注的程序特性 C。利用抽象解释对程序进行分析通常包括如下几个步骤:

1)构造能完整描述程序特性 C 的语义域 D 以及相应的语义 S,该语义被称作具体语义。具体语义可以是操作语义也可以是指称语义,但不要求该语义对任意程序是有限可计算的。

2)建立抽象语义域 $D^{\#}$ 以及基于 $D^{\#}$ 的抽象语义 $S^{\#}$,其中 $D^{\#}$ 是 D 的近似。通常 $D^{\#}$ 为有限域,以保证抽象语义的有效可计算性;否则必须为 $S^{\#}$ 设计能够保证语义计算收敛的语义运算符。

3)实现基于语义 S# 的程序分析器。

目前已存在多种分析 Prolog 程序的抽象解释框架[1~10,13~16,19,21,22],其中文[2,8,13,14,16,21,22]提出的语义能够处理 Prolog 程序的 cut 操作并考虑其深度优先的搜索规则,从而提高了程序分析的精度。除此之外,考虑程序中过程调用的上下文信息也可提高程序分析的精度,该方法已广泛应用于包含过程调用的命令式语言程序的分析中。常用的上下文信息有两种,一种是调用串(call strings),记录过程调用发生的轨迹;另一种是前提集(assumption set),记录调用发生时的程序状态[20]。在逻辑程序的分析中通常使用后者,如文[8,13]中的自顶向下语义隐式地使用部分解作为程序的上下文信息。在逻辑程序中目标的部分解是在该目标的 SLD-消解过程中任意中间步骤上获得的中间结果,可视为逻辑程序执行的中间状态。文[18,24]中的指称语义虽然利用了程序的调用路径信息,但信息量有限。L. Lu 首次把调用串作为上下文信息用于改善逻辑程序的分析[17]。本文把利用调用串信息的程序分析称为路径依赖分析。

逻辑程序分析可分为目标独立和目标依赖两种类型。目标依赖分析用于分析与特定目标相关的程序性质,通常基于程序的操作语义;目标独立分析则用于分析对任意目标均适用的程序性质,常基于目标独立的指称语义[21]。由于目标独立分析可在一次分析中提供任意目标的分析结果,因而尤其适用于需针对多个目标对同一程序进行优化的情况。

为了在抽象解释的框架内对具有 cut 操作的 Prolog 程序进行目标独立的分析,同时采用调用串改善分析的结果,需要一种能够完整描述 Prolog 程序执行过程中目标消解的路径信息并具有目标独立特性的 Prolog 具体语义。根据现有文献,没有任何一种现有 Prolog 语义可同时满足以上分析要求。为此本文采用了一种新的 Prolog 抽象语法,其中过程定义中每一个文字(literal)的入口和出口均由数字标号标明。

*)基金项目:国家自然科学基金(60563005,60663005)和广西青年科学基金(桂科青 0728093,0542036)。**赵岭忠**　博士生,研究方向:软件工程,形式化技术等;**古天龙**　教授,博士生导师,研究方向:实时混杂系统理论,协议验证,形式化方法等;**钱俊彦**　副教授,研究方向:模型检验,嵌入式实时系统。

在此基础上给出了一种 Prolog 操作语义。和传统操作语义相比，SLD-消解树上的每一个结点均由带标号的目标构成，其中标号记录了目标消解的路径信息。我们把这种采用了本文 Prolog 抽象语法的 SLD-消解树称为标号树。在标号树的基础上，给出了一种目标独立的标号树语义，并证明了该语义相对于本文操作语义的正确性。

2 相关工作

Prolog 的抽象解释可分为自顶向下和自底向上两种方式。自顶向下的抽象解释通常基于操作语义或指称语义，其传递信息的方向与 SLD-消解的方向一致；自底向上的抽象解释基于程序的不动点语义，其信息传递方式与计算最小不动点的过程相一致。本文给出的标号树语义可作为自底向上程序分析的基础，其中包含了完整的调用串信息，具有目标独立的特性，并能够处理 cut 操作符。同时具有以上特性的 Prolog 语义未见报道。

目前存在一些能够对 Prolog 控制规则和 cut 操作建模的操作语义和指称语义，其中包括 R. Barbuti 提出的 oracle 语义[2]，G. File 和 S. Rossi 给出的表式计算(tabled computations)语义[8]，B. L. Charlier 的基于序列(sequence-based)的语义[13]，以及 Levi 的不动点语义[16]等。但在利用路径信息提高程序分析的精度方面研究工作相对较少。文[8,13,14]中给出的用于自顶向下抽象解释的抽象语义均采用了前提集作为上下文信息。文[3,6]中用于自底向上抽象解释的传统语义没有利用任何路径信息，原因是在语义的计算中不存在任何对程序的执行路径进行编码的程序变换。

两种语义与本文工作密切相关。一种语义利用路径信息提高程序分析的精度，其中包括文[18]的指称语义和文[17]中的操作语义。L. Lu 提出的操作语义首次显式地利用调用串作为上下文信息来提高程序分析的精度。在该语义中程序状态由堆栈表示，堆栈元素是形如(δ,θ)的序偶，其中 δ 表示路径，θ 表示经由路径 δ 得到的替换(substitution)，即目标的部分解。程序的操作语义定义为所有初始状态的后继状态的集合。在操作语义的基础上 L. Lu 给出了一种用于程序分析的不动点语义。事实上，该不动点语义可视为操作语义的安全抽象。该语义的优势在于和现有逻辑程序的自顶向下语义和不动点语义相比，它包含了更多的路径信息，从而可以得到更精确的分析结果。该语义与现有逻辑程序语义的比较见文[17]。与本文语义相比，文[17]中语义的主要问题是其目标依赖性，因而不能用于目标独立的程序分析。其次，该语义针对通用的逻辑程序，其设计未考虑实际逻辑程序设计语言中为提高程序执行效率而存在的非逻辑操作，因而也不能直接用于含有 cut 操作的 Prolog 程序的分析。

除此之外，Spoto 等人提出的操作语义和树语义[21]以及 G. Levi 和 D. Micciancio 提出的不含非逻辑操作的 Prolog 程序语义[15]也与本文工作有密切的关系。本文借用与以上语义类似的方式获得了 LT 语义的目标独立性。所不同的是这些语义均不含程序过程调用的路径信息。

3 基础知识

假定读者熟悉基本的代数结构和 Prolog 语言。序列是元素的有序组合，其中允许存在重复元素。由集合 E 的元素构成的所有序列的集合记作 $Seq(E)$，其中非空序列的集合记作 $Seq^+(E)$。"::"表示两个序列的串接操作。空序列记作 ε。$\#\tilde{s}$ 表示序列 $\tilde{s}$ 的长度，即其包含元素的个数。

为了简化语义操作的设计，采用了一种抽象的 Prolog 程序语法[1]。其基本思想是把 Prolog 程序视为约束逻辑程序设计(CLP)的一个实例[12]。Spoto 和 Levi 在 DT 语义的设计中应用了类似的思想[21,22]。所不同的是，本文语法用标号标明了程序中每个子句的程序点(program points)。由于二元谓词表达式 $p(x,y)$ 可表示为 $\exists z\,(z=(x,y)\wedge p(z))$，不失一般性，Prolog 程序中的所有谓词均假定为一元谓词。子句具有以下形式：$p(x):-G_1$ or$\cdots$or G_n，其中 $n\geqslant 1$，$G_i(i=1,\cdots,n)$被称为过程 $p(x)$ 的第 i 个定义并由下面文法中的非终结符 $GOAL$ 产生：

$$GOAL::={}^{l}c^{l}\mid{}^{l}\,p(x)^{l}\mid\text{exists }x.\,GOAL\mid\text{cut}(GOAL)^{l}\mid\text{cut}_{d}(GOAL)^{l}\mid GOAL\text{ and }G\mid H\text{ and }GOAL\mid{}^{l}GOAL^{l}$$

$$G::=c^{l}\mid p(x)^{l}\mid GOAL^{l}\quad H::={}^{l}c\mid{}^{l}p(x)\mid{}^{l}GOAL$$

其中 l 是程序的标号，c 是一个基本约束(定义见下文)，x 是一个程序变量，$\text{cut}_d(G)$中的 d 是一个非负整数，表示 cut 操作(在目标的 SLD-消解树中)的作用域。$\text{cut}(G)$中的 cut 操作具有无限作用域，被称为开放 cut；而 $\text{cut}_d(G)$中的 cut 则具有有限的作用域 d，被称为闭合 cut。闭合的 cut 不直接出现在 Prolog 程序中，而被用于操作语义和标号树语义的描述(见第 4 和第 5 节)。在引入标号树的概念后再进一步讨论 cut 的作用域。在以上语法中由非终结符 $GOAL$ 导出的表达式称为标号目标，简称目标；而由非终结符 G 和 H 导出的表达式则称为部分目标。所有可能目标的集合记作 $\mathbb{LG}$。所有程序标号的集合记作 L，并规定 L 中包含一个特殊的不能出现在任意 Prolog 程序中，仅起辅助作用的虚拟标号 e。本文中 L 为集合 $Z^+\cup\{e\}$，其中 Z^+ 为正整数集。

对任意谓词 p，表达式 $p(x)$被称作过程调用。如果目标 G 包含过程调用，则称 G 是发散的(divergent)，记作 $\text{div}(G)$。不发散的目标被称为是收敛的。

定义 3.1[11,21] 基本约束域是一个格$\langle\mathcal{B},\leqslant,\vee,\wedge,\text{true},\text{false}\rangle$，且满足以下条件：

1)对任意变量 x 和 z，$\mathcal{B}$ 中包含元素 $\delta_{x,z}$，表示变量 x 和 z 相等；

2)存在 $\mathcal{B}$ 上的一元运算符 $\exists x$，表示隐藏一个约束中所有与变量 x 相关的信息后所得约束。

使用运算符 $\exists x$ 是避免 Prolog 目标消解过程中重命名问题的一个简单途径。如：把约束 b 中的变量 x 重命名为 z 可通过表达式 $\exists x(\delta_{x,z}\wedge b)$来实现。对基本约束域进行扩展可得可见性约束集的概念。

定义 3.2 可见性约束集 $\mathcal{O}$ 是包含集合 $\mathcal{B}$ 且满足以下条件的最小集合：

1)如果 $S\subseteq\mathcal{O}$，则$\sqcap S$ 和$\sqcup S$ 均属于 $\mathcal{O}$；

2)如果 $o\in\mathcal{O}$，则 o 的非$-o\in\mathcal{O}$。

可见性约束 o 为真，当且仅当 $o\in\mathcal{B}$ 且 $o\neq$false，或者 $o=\sqcap S$ 且 S 中的所有约束均为真，或者 $o=\sqcup S$ 且 S 中存在一个约束为真，或者 $o=-o'$ 且 o' 不为真。可见性约束可视为基本约束的集合，可见性约束 o 的真反映了构成该约束的各个基本约束的可满足性，其中，基本约束的可满足性与一阶逻辑中命题的可满足性概念一致。例如，约束$\sqcap\{-(x=2),(x=4)\}$为真表明$(x=2)$不可满足而$(x=4)$可满足。下文中用 $o_1\sqcap\cdots\sqcap o_n$ 表示$\sqcap\{o_1,\cdots,o_n\}$，用 $o_1\sqcup\cdots\sqcup o_n$ 表示$\sqcup\{o_1,\cdots,o_n\}$，用 $\text{true}_{\mathcal{O}}$和 $\text{false}_{\mathcal{O}}$分别表示$\sqcap\{\}$和$\sqcup\{\}$。

在约束集 $\mathcal{O}$ 上定义了以下运算：

• 一元运算 ∞obs：用于把基本约束转化为可见性约束。为了方便，常把 $b\infty obs$ 写成 b，从上下文可判断其为基本约束还是可见性约束。

• 二元运算 $\cdot$：用于利用基本约束对可见性约束进一步实例化，定义为：$b'\cdot(b\infty obs)=(b'\wedge b)\infty obs$；$b'\cdot\sqcap S=\sqcap\{b'\cdot o|o\in S\}$；$b'\cdot\sqcup S=\sqcup\{b'\cdot o|o\in S\}$；$b'\cdot(-o)=-(b'\cdot o)$。

• 一元运算 $\exists xo$：把约束 o 中的变量 x 转化为约束变量，其定义如下：$\exists x(b\infty obs)=(\exists xb)\infty obs$；$\exists x(\sqcap S)=\sqcap\{\exists xo|o\in S\}$；$\exists x(\sqcup S)=\sqcup\{\exists xo|o\in S\}$；$\exists x(-o)=-(\exists xo)$。

标准 Prolog 可直观地转化为本文的抽象文法。例如：Prolog 程序 P′：

{$p(x):-q(x),r(x).\ q(x):-x=4,!.\ q(x):-x=5.\ r(x):-x=5,q(x).$}

可转化为 P：{ $p(x):-^1q(x)^2$ and $r(x)^3.\ q(x):-\mathrm{cut}(^4(x=4)^5)^6$ or$^7(x=5)^8.\ r(x):-^9(x=5)^{10}$ and $q(x)^{11}.$ }。

定义 3.3 已知 $G\in LG$，G 的入口标号 $entry(G)$ 和出口标号 $exit(G)$ 分别定义如下：

$entry(c^l)=entry(^lp(x)^{l'})=entry(^l(Goal^{l'})=l$；

$entry(exists\ x.\ (Goal))=entry(Goal)$；$entry(cut(Gaal)^l)=entry(cut_d(Goal)^l)=entry(Goal)$；

$entry(Goal\ \text{and}\ G)=entry(Goal)$；$entry(H\ \text{and}\ Goal)=entry(H^{entry(Goal)})$；

$exit(^lc^{l'})=exit(^lp(x)^{l'})=exit(^lGoal^{l'})=l'$；

$exit(exists\ x.(Goal))=exit(Goal)$；$exit(cut(Goal)^l)=exit(cut_d(Goal)^l)=l$；

$exit(Goal\ \text{and}\ G)=exit(^{exit(Goal)}G)$；$exit(H\ \text{and}\ Goal)=exit(Goal)$。

定义 3.4 已知 $G\in LG$，函数 $con_{LG}(G)$ 表示 G 中第一个过程调用前所有约束的合取，定义如下：

$con_{LG}(^lc^{l'})=c$，$con_{LG}(^lp(x)^{l'})=true$；$con_{LG}(exists\ x.G)=\exists x con_{LG}(G)$；

$con_{LG}(cut(G)^l)=con_{LG}(cut_d(G)^l)=con_{LG}(G)$；$con_{LG}(^lGoal^{l'})=con_{LG}(Goal)$；

$$con_{LG}(Goal\ and\ G)=\begin{cases}con_{LG}(Goal) & \text{if } div(Goal)\\ con_{LG}(Goal)\wedge con_{LG}(^{exit(Goal)}G) & \text{if } not\ div(G_1)\end{cases}$$

$$con_{LG}(H\ and\ Coal)=\begin{cases}con_{LG}(H^{entry(Goal)}) & \text{if } div(H)\\ con_{LG}(H^{entry(Goal)})\wedge con_{LG}(Goal) & \text{if } not\ div(H)\end{cases}$$

下文中，称目标 G 是矛盾的，当且仅当 $con_{LG}(G)=\mathrm{false}$。

4 操作语义

在本文的操作语义中，任意目标在程序 P 中的 SLD-消解树(SLD-resolution tree)均被形式化地表示为标号树。标号树的结点由标号目标构成。该语义可视为标准 Prolog 操作语义的标号版本，即在目标消解过程中普通目标被标号目标所代替。标号树的集合 LT 定义为：$LT=\{(G,\varepsilon)\mid G\in LG\}\cup\{(G,\tilde{t})\mid G\in LG,\ \mathrm{div}(G)\text{且}\tilde{t}\in Seq(LT)\}$。

LT 上的包含关系 $\subseteq$ 定义如下：

1) $(G,\varepsilon)\subseteq(G,\varepsilon)$；

2) $(G,\varepsilon)\subseteq(G,\tilde{t})$；

3) $(G,\tilde{t}_1)\subseteq(G,\tilde{t}_2)$ if $\tilde{t}_1\subseteq\tilde{t}_2$；

4) $(\tilde{t}_1::\tilde{t}_2)\subseteq(\tilde{t}'_1::\tilde{t}'_2)$ if $\tilde{t}_1\subseteq\tilde{t}'_1$ and $\tilde{t}_2\subseteq\tilde{t}'_2$。

可以证明该关系是LT 上的偏序关系。如果 $\tilde{t}_1\subseteq\tilde{t}_2$，则 $\tilde{t}_1$ 和 $\tilde{t}_2$ 由相同数目的标号树构成，即 $\#\tilde{t}_1=\#\tilde{t}_2$。

标号树中结点的高度定义如下：

1)根结点的高度为 0；

2)如果一个结点 N 的高度为 h，则 N 的直接后继结点的高度为 $h+1$。

标号树 T 的高度定义为 T 中所有结点高度的最大值。

利用标号树中结点高度的概念可解释开放 cut 和闭合 cut 在标号树中的作用域。开放 cut 操作与标准 Prolog 中的 cut 作用类似，即如果某个结点 N 执行了一个开放的 cut 操作且该操作是在 N 的一个前趋结点 N' 中引入，那么该 cut 将“剪除”标号树中连接 N' 和 N 的路径右侧的所有候选路径。而闭合 cut 则不然，假定其作用域为 d，执行该 cut 的结点 N 的高度为 h，且高度为 $(h-d)$ 的 N 的前趋结点为 N''，那么该闭合 *cut* 只“剪除”标号树中连接 N'' 和 N 的路径右侧的候选路径。下面的例子说明了开放 cut 和闭合 cut 的作用域。

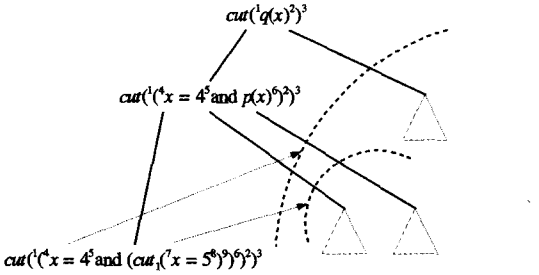

图 1 开放 cut 及闭合 cut 的作用域

例 4.1 在图 1 所示的标号树中，虚线标明了开放 cut 或闭合 cut 的作用域。cut 及其相应的作用域由带箭头的实线连接。图中三角形表示子标号树。

已知程序 P，t 被称为 P-导向的标号树当且仅当 t 中任意结点与其直接后继结点符合由 P 中过程定义规定的消解关系。以第 3 节中的程序 P 为例，标号树 $(^lp(x)^l,(^l(^1q(x)^2\ \text{and}\ r(x)^3)^l,\varepsilon))$ 符合子句 $c=\{p(x):-^1q(x)^2\ \text{and}\ r(x)^3.\}$ 规定的消解关系，而 $(^lp(x)^l,(^l(^1q(x)^2)^l,\varepsilon))$ 则不符合。为了形式化地给出 P-导向标号树的定义，需要定义两个辅助函数。$\iota(Goal,P,i)$ 是把目标 $Goal$ 中最左边的过程调用用程序 P 中该过程的第 i 个定义进行替换所得的目标，在标号树中 $\iota(Goal,P,i)$ 是 $Goal$ 的第 i 个直接后继。如果 not $div(Goal)$，则 $\iota(Goal,P,i)$ 无定义。$choices(G,P)$ 是目标 G 在程序 P 中执行最左边的过程调用 $p()$ 时候选分支的个数，即过程 $p()$ 定义的个数。

定义 4.2 已知程序 P，映射 $\iota(Goal,P,i)$ 定义为

$\iota(^lGoal^{l'},P,i)=^l(\iota(Goal,P,i))^{l'}$；$\iota(exists\ x.(Goal),P,i)=exists\ x.\iota(Goal,P,i)$；

$\iota(cut(Goal)^l,P,i)=cut(\iota(Goal,P,i))^l$；$\iota(cut_d(Goal)^l,P,i)=cut_{d+1}(\iota(Goal,P,i))^l$；

$$\iota(Goal\ \text{and}\ G,P,i)=\begin{cases}\tau_{LG}(Goal)\ \text{and}\ \iota(G,P,i) & \text{if } not\ div(Goal)\\ \iota(Goal,P,i)\ \text{and}\ G & \text{if } div(Goal)\end{cases}$$

$$\iota(H\ \text{and}\ Goal,P,i)=\begin{cases}\tau_{LG}(H)\ \text{and}\ \iota(Goal,P,i) & \text{if } not\ div(H)\\ \iota(H,P,i)\ \text{and}\ Goal & \text{if } div(H)\end{cases}$$

$$\iota(^lp(x),P,i)=\begin{cases}^l(G_i) & \text{if } x=y;\\ ^lexists\ y.(^c\delta_{x,y}\ \text{and}\ G_i) & \text{if } x\neq y.\end{cases}$$

$$\iota(p(x)^l,P,i)=\begin{cases}(G_i)^l & \text{if } x=y;\\ exists\ y.\ (\delta_{x,y}\ \text{and}\ G_i)^l & \text{if } x\neq y.\end{cases}$$

$$\iota({}^lp(x)^{l'},P,i)=$$

$$\begin{cases}{}^l(G_i)^{l'} & \text{if } x=y;\\ {}^l exists\ y.\ (\delta_{x,y}\ \text{and}\ G_i)^{l'} & \text{if } x\neq y \text{ and } not(l=l'=e);\\ exists\ y.\ (\delta_{x,y}\ \text{and}\ G_i) & \text{if } x\neq y \text{ and } l=l'=e.\end{cases}$$

$\iota({}^lGoal)={}^l(\iota(Goal,P,i))$；$\iota(Goal^l)=(\iota(Goal,P,i))^l$；

其中，程序 P 中过程 $p(\ )$ 的定义为：$p(y):-G_1$ or …or G_n. 且定义中所有的 cut() 结构均由 $cut_l()$ 结构替代，函数 $\tau_{LG}(G)$ 的功能是删除目标 G 中的 cut 操作，其定义如下：

$\tau_{LG}({}^lc^{l'})={}^lc^{l'}$；$\tau_{LG}({}^lp(x)^{l'})={}^lp(x)^{l'}$；$\tau_{LG}({}^lGoal^{l'})={}^l(\tau_{LG}(Goal))^{l'}$

$\tau_{LG}(exists\ x.\ (Goal))=exists\ x.\ \tau_{LG}(Goal)$；$\tau_{LG}(ct(Goal)^l)=\tau_{LG}(cut_d(Goal)^l)=\tau_{LG}(Goal))^l$；

$\tau_{LG}(Goal\ \text{and}\ G)=\tau_{LG}(Goal)\ \text{and}\ \tau_{LG}(G)$；$\tau_{LG}(H\ \text{and}\ Goal)=\tau_{LG}(H)\ \text{and}\ \tau_{LG}(Goal)$；

$\tau_{LG}({}^lc)={}^lc$；$\tau_{LG}({}^lp(x))={}^lp(x)$；$\tau_{LG}({}^lGoal)={}^l(\tau_{LG}(Goal))$；

$\tau_{LG}(c^l)=c^l$；$\tau_{LG}(p(x)^l)=p(x)^l$；$\tau_{LG}(Goal^l)=\tau_{LG}(Goal)^l$。

注意：在 $\iota(Goal,\ P,\ i)$ 的定义中，没有采用传统 Prolog 消解过程中使用的更名操作，而是利用操作符 $\exists x$ 和 $\delta_{x,z}$ 约束达到同样的效果。在 $\iota(Goal\ \text{and}\ G,\ P,\ i)$ 的定义中，删除 *Goal* 中 cut 操作的原因是，如果 not div(*Goal*)，则按照 Prolog 语义规则 *Goal* 中的 cut 操作已经在结点“*Goal* and *G*”中执行，不能再出现在其直接后继中。

定义 4.3 已知程序 P 和目标 G，函数 $choices(G,P)$ 的定义如下：

$choices({}^lc^{l'},P)=0$；$choices({}^lp(x)^{l'},P)=n$；

$choices({}^lGoal^{l'},P)=choices(exists\ x.\ (Goal),P)$；$=choices(cut(Goal)^l,P)=choices(cut_d(Goal)^l,P)=choices(Goal,P)$；

$choices(Goal\ \text{and}\ G,P)=$

$$\begin{cases}choices(^{exit(Goal)}G,P) & \text{if not div}(Goal)\\ choices(Goal,P) & \text{if div}(Goal)\end{cases}$$

$choices(H\ \text{and}\ Goal,P)=$

$$\begin{cases}choices(Goal,P) & \text{if not div}(H)\\ choices(H^{entry(Goal)},P) & \text{if div}(H).\end{cases}$$

其中，程序 P 中过程 $p(\ )$ 的定义为：$p(y):-G_1$ or …or G_n。

定义 4.4 已知程序 P，P-导向的标号树定义如下：

1）任给 $G\in LG$，(G,ε) 是 P-导向的；

2）$(G,t_1,\cdots,t_n)$ 是 P-导向的当且仅当 $n\leqslant choices(G,P)$，$t_1,\cdots,t_n$ 均为 P-导向的标号树且 t_i 的根结点 G_i 满足 $G_i=\iota(G,P,i)$。

如果把以上定义中的条件 $n\leqslant choices(G,P)$ 替换为 $n=choices(G,P)$，则得到完全 P-导向树的定义。

下面定义扩展函数 $expand_P:LT\mapsto LT$。直观地讲，$expand_P(T)$ 是对标号树 T 进行一次 SLD-消解所得的标号树，消解过程使用最左文字优先的文字选择规则和深度优先的消解树搜索规则并考虑程序中 cut 操作的影响。为了描述 cut 操作对消解树的影响，需定义描述标号树中 cut 操作执行条件的映射 cuts()。任意 cut 操作的执行条件均由集合 $\mathcal{B}\cup(\mathcal{B}\times N)$ 的一个子集表示，其中基本约束 $b\in\mathcal{B}$ 描述开放 cut 的执行条件，$(b,d)\in\mathcal{B}\times N$ 描述闭合 cut 的执行条件及其作用域。首先定义以下基本函数：

定义 4.5 在集合 $\mathcal{B}\cup(\mathcal{B}\times N)$ 上定义如下运算：

$b\cdot\{\langle c_1,d_1\rangle,\cdots,\langle c_n,d_n\rangle,c'_1,\cdots,c'_m\}=\{\langle b\wedge c_1,d_1\rangle,\cdots,\langle b\wedge c_n,d_n\rangle,b\wedge c'_1,\cdots,b\wedge c'_m\}$；

$\exists x\{\langle c_1,d_1\rangle,\cdots,\langle c_n,d_n\rangle,c'_1,\cdots,c'_m\}=\{\langle\exists xc_1,d_1\rangle,\cdots,\langle\exists xc_n,d_n\rangle,\exists xc'_1,\cdots,\exists xc'_m\}$；

$\{\langle c_1,d_1\rangle,\cdots,\langle c_n,d_n\rangle,c'_1,\cdots,c'_m\}\infty obs=\{\langle c_1\infty obs,d_1\rangle,\cdots,\langle c_n\infty obs,d_n\rangle,c'_1\infty obs,\cdots,c'_m\infty obs\}$；

$\sqcup\{\langle c_1,d_1\rangle,\cdots,\langle c_n,d_n\rangle,c'_1,\cdots,c'_m\}=(\vee_i c_i)\vee(\vee_i c'_i)$；

$\zeta(\{\langle c_1,d_1\rangle,\cdots,\langle c_n,d_n\rangle,c'_1,\cdots,c'_m\}=\cup_{d_i>1}\{\langle c_i,d_i-1\rangle\}\cup\{c'_1,\cdots,c'_m\}$.

定义 4.6 已知目标 G，与之相关联的 cut 操作执行条件集 $cuts(G)$ 定义为：

$cuts({}^lc^{l'})=cuts({}^lp(x)^{l'})=\varnothing$；$cuts({}^lGoal^{l'})=cuts(Goal)$；$cuts(exists\ x.\ (Goal))=\exists xcuts(Goal)$；

$cuts(cut(Goal)^l)=$

$$\begin{cases}cuts(Goal)\cup\{con_{LG}(Goal)\} & \text{if not div}(G)\\ cuts(Goal) & \text{if div}(G)\end{cases}$$

$cuts(Goal\ \text{and}\ G)=$

$$\begin{cases}cuts(Goal)\cup\{(con_{LG}(Goal)\cdot cuts(^{exit(Goal)}G)\} & \text{if not div}(Goal)\\ cuts(Goal) & \text{if div}(Goal)\end{cases}$$

$cuts(cut_d(Goal)^l)=$

$$\begin{cases}cuts(Goal)\cup\{\langle con_{LG}(G),d\rangle\} & \text{if not div}(G)\ \text{and}\ d>0\\ cuts(Goal) & \text{if div}(G)\end{cases}$$

$cuts(H\ and\ Goal)=$

$$\begin{cases}cuts(H^{entry(Goal)})\cup(con_{LG}(H^{entry(Goal)})\cdot cuts(Goal)\} & \text{if not div}(H)\\ cuts(H^{entry(Goal)}) & \text{if div}(H)\end{cases}$$

映射 $cuts(G)$ 可扩展为标号树上的映射，即：

$cuts(G,\varepsilon)=cuts(G)$；$cuts(G,\tilde{t})=cuts(G)\cup\zeta(cuts(\tilde{t}))$；$cuts(\tilde{t}_1::\tilde{t}_2)=cuts(\tilde{t}_1)\cup cuts(\tilde{t}_2)$。

直观地，为了完整地搜集与一个标号树 t 相关联的 cut 执行条件，必须搜索标号树的所有结点。$cuts(t)$ 只关心那些能够对 t 之外的标号树构成影响的 cut 操作。为此必须关注那些影响域不超过树的根结点的 cut 操作，这些 cut 的执行条件不能作为 $cuts(t)$ 的元素。该功能由函数 ζ 来完成。例如，$cuts(p(x),\ cut_1(x=4))=\varnothing$，而 $cuts(p(x),\ cut_2(x=4))=\{\langle x=4,1\rangle\}$。现在可给出 $expand_P(T)$ 的定义。

定义 4.7 已知程序 P 和标号树 T，$expand_P(T)$ 由以下扩展规则定义：

规则 1 $expand_P(G,\varepsilon)=$

$$\begin{cases}(G,\varepsilon) & \text{if } con_{LG}(G)=\text{false 或 not div}(G) \quad \text{（情况 1）}\\ (G,(\iota(G,P,l),\varepsilon)) & \text{otherwise} \quad \text{（情况 2）}\end{cases}$$

规则 2 $expand_P(G,t_1::\cdots::t_n)=$

$$\begin{cases}(G,t_1::\cdots::t_{n-1}::\text{expand}_P(t_n)) & \text{if}(t_n\ \text{可扩展}) \quad \text{（情况 3）}\\ (G,t_1::\cdots::t_n::(\iota(G,P,n+1),\varepsilon)) & \text{if}(t_n\ \text{不可扩展},\\ \quad \sqcup cuts(t_n)=\text{fales 且 } choices(G,p)>n) & \text{（情况 4）}\\ (C,t_1::\cdots::t_n) & \text{otherwise} \quad \text{（情况 5）}\end{cases}$$

注意：标号树 t_n 不可扩展当且仅当 t_n 的每一个叶子结点 G 或 $con_{LG}(G)=$ false 或 not div(G)。

利用 $expand_P()$ 对标号树 (G,ε) 进行连续扩展的过程对应目标 G 在程序 P 中的消解过程，扩展过程遵循 Prolog 的深度优先搜索规则。规则 1 对叶子结点进行扩展，显然符合深度优先的要求。规则 2（情况 3）在标号树序列 $(t_1,\cdots,t_n)$ 中选

择最右子树 t_n 进行扩展也遵循了深度优先的搜索规则，原因是$(t_1,\cdots,t_n)$中的任意标号树均由规则 1(情况 2)或者规则 2(情况 4)产生，而产生 t_i 的前提是要求$(t_1,\cdots,t_{i-1})$中任意标号树均不可扩展且 t_i 不受$(t_1,\cdots,t_{i-1})$结点中 cut 操作的影响；因此在标号树序列$(t_1,\cdots,t_n)$中 t_n 是唯一可能继续扩展的子树。所以选择最右子树 t_n 进行扩展与 SLD-消解过程一致。

定义 4.8 目标 G 在程序 P 中的操作语义 $\mathcal{O}_{G,P}$ 定义为：$\mathcal{O}_{G,P}=\mathrm{lub}_{i\geqslant 0}\,\mathcal{O}_{G,P,i}$，其中 $\mathcal{O}_{G,P,0}=(G,\varepsilon)$，$\mathcal{O}_{G,P,i+1}=\mathrm{expang}_P(\mathcal{O}_{G,P,i})$。

5 标号树语义

操作语义的目标依赖性使其不能用于目标独立的 Prolog 程序分析。本节引入的标号树(LT)语义是一种能够以自顶上下和自底向上两种方式进行构造的指称语义。两种构造方法可得到相互等价的目标独立 Prolog 语义。其中自顶上下的构造过程与目标依赖的操作语义直接相关，被用于证明 LT 语义相对于第 4 节中操作语义的正确性。

定义 5.1 在集合 $\mathrm{Seq}^+(\mathrm{LT})$上定义如下操作：

(1)给定目标 G 满足 not div(G)，实例化操作$\textcircled{@}^{\mathrm{LT}}$定义为：$G\textcircled{@}^{\mathrm{LT}}(\tilde{t}_1::\tilde{t}_2)=G\textcircled{@}^{\mathrm{LT}}(\tilde{t}_1)::G\textcircled{@}^{\mathrm{LT}}(\tilde{t}_2)$；$G\textcircled{@}^{\mathrm{LT}}(G',\varepsilon)=(AND(G,G'),\varepsilon)$；$G\textcircled{@}^{\mathrm{LT}}(G',\tilde{t})=(AND(G,G'),\tau_{LG}(G)\textcircled{@}^{\mathrm{LT}}\tilde{t})$，

其中 $AND(G_1,G_2)$ 的定义如下：

$$AND(G_1,G_2)=\begin{cases}G'_1 \text{ and } G_2 & \text{if } exit(G_1)=entry(G_2) \text{ and } G_1=G_1'^{exit(G_1)}\\ & \text{or } exit(G_1)=e \text{ and } G_1=G_1'^{exit(G_1)};\\ \text{undefined} & \text{otherwise.}\end{cases}$$

(2)已知 $T_2\in\mathrm{LT}$ 且其根结点为 G_2，标号树的积操作$\otimes^{\mathrm{LT}}$定义为：$(\tilde{t}_i::\tilde{t}_2)\otimes^{\mathrm{LT}}T_2=\tilde{t}_1\otimes^{\mathrm{LT}}T_2::\tilde{t}_2\otimes^{\mathrm{LT}}T_2$，其中：$(G_1,\tilde{t}_1)\otimes^{\mathrm{LT}}T_2=(AND(G_1,G_2),\tilde{t}_1\otimes^{\mathrm{LT}}T_2)$，$(G_1,\varepsilon)\otimes^{\mathrm{LT}}T_2=$

$$\begin{cases}(AND(G_1,G_2),\varepsilon) & \text{if div}(G_1)\\ G_1\textcircled{@}^{\mathrm{LT}}T_2 & \text{if not } div(G_1)\end{cases}$$

(3)已知 $\tilde{t}\in\mathrm{Seq}^+(\mathrm{LT})$，$update(\tilde{t},l,l')$用于标号树路径信息的更新，定义如下：

$update((G,\varepsilon),l,l')=(upd(G,l,l'),\varepsilon)$；$update((G,\tilde{t}),l,l')=(upd(G,l,l'),update(\tilde{t},l,l'))$；

$update(\tilde{t}_1::\tilde{t}_2,l,l')=update(\tilde{t}_1,l,l')::update(\tilde{t}_2,l,l')$；

其中 $upd(G,l,l')$ 定义为：$upd(G,l,l')=\begin{cases}lG' & \text{if } G={}^{e}G'^{e};\\ {}^{l}G' & \text{otherwise.}\end{cases}$

(4)操作符 $\exists_x^{\mathrm{LT}}$ 定义为：$\exists_x^{\mathrm{LT}}(\tilde{t}_1::\tilde{t}_2)=\exists_x^{\mathrm{LT}}(\tilde{t}_1)::\exists_x^{\mathrm{LT}}(\tilde{t}_2)$，其中

$\exists_x^{\mathrm{LT}}(G,\varepsilon)=(exists\ x.\ G,\varepsilon)$；$\exists_x^{\mathrm{LT}}(G,\tilde{t})=(exists\ x.\ G,\exists_x^{\mathrm{LT}}(\tilde{t}))$。

(5)cut 运算符($!_l$ 和$!_l^d$)及 uncut 运算符$\downarrow^{\mathrm{LT}}$定义为：

$!_l(G,\varepsilon)=(cut(G)^l,\varepsilon)$；$!_l(G,\tilde{t})=cut(G)^l$，$!_l(\tilde{t})$；$!_l(\tilde{t}_1::\tilde{t}_2)=!_l(\tilde{t}_1)::!_l(\tilde{t}_2)$

$!_l^d(G,\varepsilon)=(cut_d(G)^l,\varepsilon)$；$!_l^d(G,\tilde{t})=cut_d(G)^l$，$!_l^{d+1}(\tilde{t})$；$!_l^d(\tilde{t}_1::\tilde{t}_2)=!_l^d(\tilde{t}_1)::!_l^d(\tilde{t}_2)$

和

$\downarrow^{LT}(\tilde{t})=\downarrow_0^{\mathrm{LT}}(\tilde{t})$；$\downarrow_i^{\mathrm{LT}}(G,\varepsilon)=(\downarrow_i G,\varepsilon)$；$\downarrow_i^{\mathrm{LT}}(G,\tilde{t})=(\downarrow_i G,\downarrow_{i+1}^{\mathrm{LT}}\tilde{t})$；$\downarrow_i^{\mathrm{LT}}(\tilde{t}_1::\tilde{t}_2)=\downarrow_i^{\mathrm{LT}}\tilde{t}_1::\downarrow_i^{\mathrm{LT}}\tilde{t}_2$

其中 $\downarrow_i(\tilde{t})$把 $\tilde{t}$ 中的开放 cut 均转化为闭合的 cut，定义如下：

$\downarrow_i({}^{l}c^{l'})={}^{l}c^{l'}$；$\downarrow_i({}^{l}p(x)^{l'})={}^{l}p(x)^{l'}$；$\downarrow_i({}^{l}Goal^{l'})={}^{l}\downarrow_i(Goal)^{l'}$；

$\downarrow_i(exists\ x.\ (Goal))=esists\ x.\ \downarrow_i(Goal)$；

$\downarrow_i(cut(Goal)^l)=cut_i(\downarrow_i Goal)^l$；　$\downarrow_i(cut_d(Goal)^l)=cut_d(\downarrow_i Goal)^l$；

$\downarrow_i(Goal \text{ and } G)=\downarrow_i Goal \text{ and } \downarrow_i G$；$\downarrow_i(H \text{ and } Goal)=\downarrow_i H \text{ and } \downarrow_i Goal$；

$\downarrow_i({}^{l}c)={}^{l}c$；$\downarrow_i({}^{l}p(x))={}^{l}p(x)$；$\downarrow_i({}^{l}Goal)={}^{l}(\downarrow_i Goal)$；

$\downarrow_i(c^l)=c^l$；$\downarrow_i(p(x)^l)=p(x)^l$；$\downarrow_i(Goal^l)=(\downarrow_i Goal)^l$.

(6)求和操作符$\oplus^{\mathrm{LT}}$定义为：$(G,\tilde{t}_1)\oplus^{\mathrm{LT}}(G,\tilde{t}_2)=(G,\tilde{t}_1::\tilde{t}_2)$。

(7)扩展操作 ϕ_{LT}^{G}，根结点交换操作 $\varphi_{\mathrm{LT}}^{G}$和替换操作 $\tilde{t}[x/\alpha]$分别定义如下：

$\phi_{\mathrm{LT}}^{G}(\mathrm{T})=(\mathrm{G},\mathrm{T})$；

$\varphi_{\mathrm{LT}}^{G}(G',\varepsilon)=(G,\varepsilon)$；$\varphi_{\mathrm{LT}}^{G}(G',\tilde{t})=(G,\tilde{t})$；

$({}^{l}p(\alpha)^{l'},\varepsilon)[x/\alpha]=({}^{l}p(x)^{l'},\varepsilon)$；

$({}^{l}p(\alpha)^{l'},\exists_x^{\mathrm{LT}}(({}^{l}\delta_{x,\alpha}^{l'},\varepsilon)\otimes^{\mathrm{LT}}\tilde{t}))[x/\alpha]=({}^{l}p(x)^{l'},\tilde{t})$；

$({}^{l}p(\alpha)^{l'},\exists_y^{\mathrm{LT}}(({}^{l}\delta_{y,\alpha}^{l'},\varepsilon)\otimes^{\mathrm{LT}}\tilde{t}))[x/\alpha]=({}^{l}p(x)^{l'},\exists_y^{\mathrm{LT}}(({}^{l}\delta_{y,x}^{l'},\varepsilon)\otimes^{\mathrm{LT}}\tilde{t})$。

以上定义的语义操作的功能解释如下。如果标号树 T 表示目标G'的消解树且 not div(G)，那么 $G\textcircled{@}^{\mathrm{LT}}T$ 就表示目标 G'' and G'的消解树，其中 G'' 是删除 G 的出口标号后所得部分目标。这里要求 $entry(G')=exist(G)$ 或 $exist(G)=e$。后一种情形是必须的，因而我们常需要利用$\textcircled{@}^{\mathrm{LT}}$模拟变量更名的效果，在此情况下 $G={}^{e}\delta_{x,y}^{e}$。直观地讲，$\tilde{t}\otimes^{\mathrm{LT}}T$ 是把标号树 T 的根结点与 $\tilde{t}$ 中的每一个收敛结点“拼接”并更新中的每一个目标所得的标号树，即，假定 T 表示目标 G 的消解树，$t_1,\cdots,t_n$ 分别表示目标 $G_1,\cdots,G_n$ 的消解树，那么$(t_1::\cdots::t_n)\otimes^{\mathrm{LT}}T$ 则表示目标 G_1 and G'，…，G_n and G'的消解树，其中 G' 是删除 G 的入口标号后所得部分目标。可见标号树的积操作是实例化操作的推广。假定 T 表示目标 G 的消解树，则$!_l(T)$和$!_l^d(T)$分别表示目标 $cut(G)^l$ 和 $cut_d(G)^l$ 的消解树。$\downarrow_i^{\mathrm{LT}}(\tilde{t})$用于把 $\tilde{t}$ 中的开放 cut 通过指定适当的作用域而转化为闭合 cut。最后，求和操作符$\oplus^{\mathrm{LT}}$合并两个根结点相同的标号树；扩展操作 $\phi_{\mathrm{LT}}^{G}(T)$则通过为树 T 赋予一个父结点 G 而使 T 中所有结点的高度均增加 1；根结点交换运算 $\varphi_{\mathrm{LT}}^{G}(T)$则把树 T 的根结点替换为 G；而替换操作 $T[x/\alpha]$则利用约束表达式模拟把 T 中出现的变量 α 更名为另一变量 x，即：如果 T 表示目标${}^{l}p(\alpha)^{l}$的消解树，那么 $T[x/\alpha]$则表示目标${}^{l}p(x)^{l}$的消解树。

下面给出标号树解释的定义。

定义 5.2 标号树解释 I 是一个从程序的谓词符号集合到标号树集合LT的映射，且对于程序中的任意谓词符号 p，$I(p)$描述了目标${}^{e}p(\alpha)^{e}$在该程序中的所有可能行为，其中 α 是一个不允许出现在任何 Prolog 程序子句中的变量。标号树解释的集合记作LTI。

标号树集合上的序关系“$\subseteq$”可扩展到标号解释的集合，即：$I_1\subseteq I_2$ 当且仅当对任意谓词 p，$I_1(p)\subseteq I_2(p)$。如果对任意谓词 p，$I(p)$均为完全 P-导向标号树，则称标号树解释 I 为

完全 P-导向的。

定义 5.3 已知程序 P，目标 G 在完全 P-导向标号树解释 I 中的指称由以下表达式给出：

$J_P^{LT}[\![{}^l c^{l'}]\!]I=({}^l c^{l'},\varepsilon)$；$J_P^{LT}[\![{}^l p(x)^{l'}]\!]I=update(I(p)[x/\alpha],l,l')$；

$J_P^{LT}[\![exists\ x.\ (Goal)]\!]I=\exists_x^{LT} J_P^{LT}[Goal]I)$；$J_P^{LT}[\![cut\ (Goal)^l]\!]I=!^l(J_P^{LT}[\![Goal]\!]I,)$；

$J_P^{LT}[\![cut_d(Goal)^l]\!]I=!_d^l(J_P^{LT}[\![Goal]\!]I)$；$J_P^{LT}[\![{}^l Goal^{l'}]\!]I=update(J_P^{LT}[\![Goal]\!]I,l,l')$；

$J_P^{LT}[\![Goal\ \text{and}\ G]\!]I=J_P^{LT}[\![Goal]\!]I\otimes^{LT}J_P^{LT}[\![^{exit(Goal)}G]\!]I$；

$J_P^{LT}[\![H\ \text{and}\ Goal]\!]I=J_P^{LT}[\![H^{entry(Goal)}]\!]I\otimes^{LT}J_P^{LT}[\![Goal]\!]I$。

定义 5.4 已知程序 P，目标 G 在完全 P-导向标号树解释 I 中的扩展标号树由以下表达式给出：

$D_P^{LT}[\![{}^l c^{l'}]\!]I=({}^l c^{l_1},\varepsilon)$；$D_P^{LT}[\![exists\ x.\ (Goal)]\!]I=\exists_x^{LT}D_P^{LT}[Goal]I)$；

$D_P^{LT}[\![cut(Goal)^l]\!]I=!^l(D_P^{LT}[\![Goal]\!]I)$；$D_P^{LT}[\![cut_d(Goal)^l]\!]I=!_d^l(D_P^{LT}[\![Goal]\!]I)$；

$D_P^{LT}[\![{}^l Goal^{l'}]\!]I=update(D_P^{LT}[\![Goal]\!]I,l,l')$；

$D_P^{LT}[\![Goal\ \text{and}\ G]\!]I=D_P^{LT}[\![Goal]\!]I\otimes^{LT}D_P^{LT}[\![^{exit(Goal)}G]\!]I$；

$D_P^{LT}[\![H\ \text{and}\ Goal]\!]I=D_P^{LT}[\![H^{entry(Goal)}]\!]I\otimes^{LT}D_P^{LT}[\![Goal]\!]I$；

$$D_P^{LT}[\![{}^l p(x)^{l'}]\!]I=\begin{cases}update(t,l,l') & \text{if } x\neq y,\\ update(t',l,l') & \text{if } x=y,\end{cases}$$

其中，程序 P 中过程 $p()$ 的定义为：$p(y):-G_1$ or $\cdots$or G_n.，

$t=\phi_{LT}^{{}^e p(x)^e}\exists_y^{LT}({}^e\delta_{y,x}^e,\varepsilon)\otimes\perp^{LT}\downarrow^{LT}(\phi_{LT}^{{}^e p(y)^e}J_P^{LT}[\![G_1]\!]I\oplus^{LT}\cdots\oplus^{LT}\phi_{LT}^{{}^e p(y)^e}J_P^{LT}[\![G_n]\!]I))$，

$t'=\downarrow^{LT}(\phi_{LT}^{{}^e p(x)^e}J_P^{LT}[\![G_1]\!]I\oplus^{LT}\cdots\oplus^{LT}\phi_{LT}^{{}^e p(x)^e}J_P^{LT}[\![G_n]\!]I))$。

可见：$D_P^{LT}[\![G]\!]I$ 与 $J_P^{LT}[\![G]\!]I$ 的区别在于过程调用的指称是否由定义该过程的子句体的指称替代。容易证明：如果标号树解释 I 是完全 P-导向的，则 $J_P^{LT}[\![G]\!]I$ 和 $D_P^{LT}[\![G]\!]I$ 均为完全 P-导向的标号树。

定义 5.5 已知程序 P，标号树 t 相对于完全 P-导向标号树解释 I 的展开操作定义为：

$(G,\varepsilon)\bowtie^P I=D_P^{LT}[G]I$；$(G,\tilde{t})\bowtie^P I=(G,\tilde{t}\bowtie^P I)$；$(\tilde{t}_1::\tilde{t}_2)\bowtie^P I=(\tilde{t}\bowtie^P I)::(\tilde{t}_2\bowtie^P I)$。

该定义可扩展到标号树解释，即：对任意谓词 p，$(I_1\bowtie^P I_2)(p)=I_1(p)\bowtie^P I_2$。

在以上定义的基础上可定义程序 P 的直承算子(immediate consequence operator) T_P^{LT} 如下：

$T_P^{LT}(I)=I_0^{LT}\bowtie^P I$，其中对任意谓词 p，$I_0^{LT}(p)=({}^e p(\alpha)^e,\varepsilon)$。

根据标号树解释展开方式的不同，可定义以下两个不同的完全 P-导向标号树解释序列：

1) $\{I_i\}_{i\geqslant 0}$，其中 $I_0=I_0^{LT}$，$I_{i+1}=I_0^{LT}\bowtie^P I_i=T_P^{LT}(I_i)$ 以及

2) $\{J_i\}_{i\geqslant 0}$，其中 $J_0=I_0^{LT}$，$J_{i+1}=J_i\bowtie^P I_0^{LT}$。

由 $\bowtie^P$ 的定义知：序列 $\{I_i\}_{i\geqslant 0}$ 相对于包含关系升序排列，即：

命题 5.6 已知两个完全 P-导向标号树解释 I_1 和 I_2 且 $I_1\subseteq I_2$，则 $T_P^{LT}(I_1)\subseteq T_P^{LT}(I_2)$。

证明：根据定义 5.1 中基本语义运算的单调性(见附录)，易证该命题成立。

可以证明序列 $\{J_i\}_{i\geqslant 0}$ 也是相对于包含关系的升序排列。

由格理论可知，任意偏序集 P 均可扩展为完备格 L，同时定义在集合 P 上的单调函数 $f:P\mapsto P$ 也可相应地扩展为 L 上的连续函数 $f':L\mapsto L$。本文把完备格 L 称为偏序集 P 的完备化扩展。因此，在 $(\mathbb{LTI},\subseteq)$ 的完备化扩展上，标号树解释序列 $\{I_i\}_{i\geqslant 0}$ 和 $\{J_i\}_{i\geqslant 0}$ 分别存在上确界 $lub_i I_i$ 和 $lub_i J_i$；T_P^{LT} 可扩展为连续函数，下面仍把该连续函数记作 T_P^{LT}。根据 Tarski 不动点定理[23]，T_P^{LT} 存在最小不动点 $lfp(T_P^{LT})$。由 $\{I_i\}_{i\geqslant 0}$ 的定义可知：$I_i=(T_P^{LT})^i(I_0^{LT})$，从而 $lub_i I_i=lfp(T_P^{LT})$。

根据以上分析有以下定义：

定义 5.7 在 $(\mathbb{LTI},\subseteq)$ 的完备化扩展上可定义：$F_P^{bu}=\text{lub}_i I_i=\text{lfp}(T_P^{LT})$ 和 $F_P^{td}=\text{lub}_i J_i$。

F_P^{bu} 是程序 P 的自底向上展开语义，而 F_P^{td} 则是 P 自顶向下的展开语义。除此之外，给定目标 G 可以定义

$K_0^G=(G,\varepsilon)$，$K_{i+1}^G=K_i^G\bowtie^P I_0^{LT}$

并称 $\text{lub}_i K_i^G$ 为目标 G 在程序 P 中的目标依赖语义。对于以上语义有以下结论：

定理 5.8 已知程序 P 和目标 $G\in\mathbb{LG}$，有以下关系成立：

(1) $F_P^{bu}=F_P^{td}$；

(2) $\text{lub}_i K_i^G=J_P^{LT}[\![G]\!]F_P^{bu}=J_P^{LT}[\![G]\!]F_P^{td}$。

以上定理表明自底向上和自顶向下的语义构造方式是等价的，即对于程序 P 中的任意目标 G 通过二者可得到相同的指称。因此可定义程序 P 的标号树语义 F_P^{LT} 为：

$F_P^{LT}=F_P^{bu}=F_P^{td}$。

例 5.9 已知 Prolog 程序 P：{ $p(x):-({}^1(x=4)^2$ and $p(x)^3)$ or ${}^4q(x)^5$. $q(x):-(\text{cut}^6(x=3)^7)^8$ or${}^9(x=5)^{10}$. }，其标号树语义如图 2 所示，其中 $F_P^{LT}(p)=T_1$，$F_P^{LT}(q)=T_2$。T_3 是由该语义得到的目标 ${}^l p(x)^{l'}$ 的语义 $J_P^{LT}[\![{}^l p(x)^{l'}]\!]F_P^{LT}$。为了方便表述，直接对目标中的有关变量进行了更名。

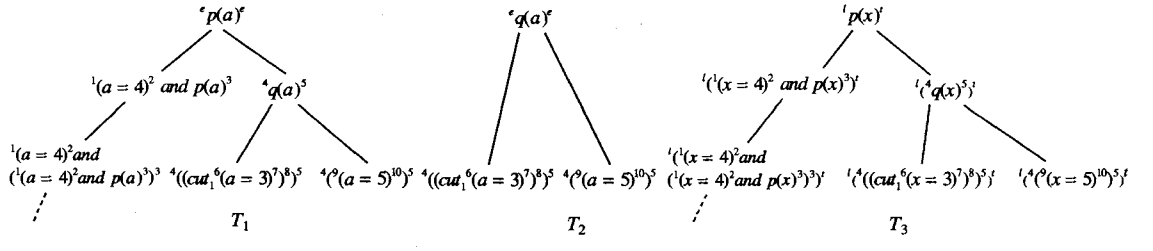

图 2 程序 P 及目标 ${}^l p(x)^{l'}$ 的标号树语义

由图 2 可见，标号树中任意结点(目标)均包含了始于根结点的完整路径信息。比如：T_3 中目标 ${}^l({}^4({}^9(x=5)^{10})^5)^{l'}$ 说明在目标 ${}^l p(x)^{l'}$ 的消解过程中，经由路径 l-4-9-10-5-l' 得到了该目标的一个解 $x=5$；而 ${}^l({}^1(x=4)^2$ and $p(x)^3)^{l'}$ 则说明

在目标$^t p(x)^t$的消解过程通过路径 l-1-2 引发了对过程 $p(x)$的调用，此时 $x=4$。以上路径正是程序分析过程所需要的调用串信息。给定标号树 T，通过设计适当抽象函数，容易获取该信息。因此，标号树语义可以作为 Prolog 路径依赖分析的基础。

需指出，以上给出的程序语义，无论是 F^{φ}_{P} 还是 $F^{\varphi}_{\wp}$，均是目标独立的，即根据定理 5.8(2)，程序 P 中任意目标 $G\in LG$ 的语义 $\mathrm{lub}_i K^{\varphi}_i$ 均可由程序语义 F^{φ}_{P} 或 $F^{\varphi}_{\wp}$ 获得。这意味着如果能够证明标号树语义相对于本文操作语义的正确性，则以两种方式构造的程序语义均可作为目标独立 Prolog 程序分析的基础。特别地，通过计算语义算子 T^{LT}_{P} 的不动点可以获得程序的标号树语义，该算子可用于 Prolog 程序的自底向上分析。在抽象解释的框架进行程序分析时，仅需构造相应语义操作的抽象操作并证明其正确性或安全性。该内容将另文介绍。下面说明标号树语义的正确性。

6 标号树语义相对于操作语义的正确性

为了获得目标独立性，LT 语义中包含了按照 Prolog 规则不能访问的某些结点。即在完全 P-导向的标号树中包含了目标所有可能的消解路径，而其中某些路径可能由于无限路径的出现或者 cut 操作的执行而不能被 Prolog 解释器访问到。因而，为了证明标号树语义相对于本文操作语义的正确性，需要从中删除那些不可访问的结点。为此，我们首先引入 D-标号树(Decorated Labeled Tree)的概念。其主要思想是向标号树的每一个结点中增加一个可见性约束，表示树中结点如何受到发散的叶子结点(即包含发散目标的叶子结点)和 cut 操作的影响。从而可见性约束可用于判断哪些结点可访问，哪些结点不可访问。该技术被称为“控制编译”[1,17]。

例 6.1 以例 5.9 中的标号树 T_3 为例说明如何使用可见性约束描述 Prolog 控制规则和 cut 操作对结点可访问性的影响。标号树 T_3 的最左路径是无限路径，根据程序可知该路径仅当($x=4$)可满足时才可能出现。于是为了使 Prolog 解释器能够访问根结点的右孩子结点，即：使该右孩子结点可见，必然要求($x=4$)不可满足。类似地，结点$^t(^4(^9(x=5)^{10})^5)^t$(下文忽略其中标号简记为“$x=5$”)可见的必要条件是结点 cut($x=3$)中的 cut 操作不执行。否则结点 $x=5$ 将因为处于该 cut 操作的作用域内而被“剪除”。由于仅当($x=3$)可满足时该 cut 操作才可执行，故要求该前提不成立。显然，一个结点可见仅当其父结点可见，于是结点 $x=5$ 可见的完整条件是($x=4$)和($x=3$)均不可满足。该条件可描述为要求可见性约束$-(x=4)\sqcap-(x=3)$为真，即仅当$-(x=4)$和$-(x=3)$均为真时该结点可见，于是与结点 $x=5$ 相关联的可见性约束为$-(x=4)\sqcap-(x=3)$。按照 Prolog 深度优先的搜索规则标号树最左路径上的所有节点必然可见，因而其可见性约束均为 $true_{\emptyset}$。

可见，计算与标号树结点相关联的可见性约束需考虑发散结点和 cut 操作两方面的因素。形式化地，D-标号树的集合DLT 可定义如下：

$\mathrm{DLT}=\{(o+G,\varepsilon)\mid o\in\mathcal{O},G\in LG\}\cup\{(o+G,\tilde{t})\mid o\in\mathcal{O},\ G\in LG,\ \mathrm{div}(G)\text{且}\tilde{t}\in Seq(\mathrm{DLT})\}$。

转化函数：$\alpha_{\mathrm{DLT}}Seq^{+}(\mathrm{LT})\mapsto Seq^{+}(\mathrm{DLT})$定义如下：

$\alpha_{\mathrm{DLT}}(G,\varepsilon)=(\mathrm{true}_{\emptyset}+G,\varepsilon)$；

$\alpha_{\mathrm{DLT}}(G,\tilde{t})=(true_{\emptyset}+G,\alpha_{DLT}(\tilde{t}))$；

$\alpha_{\mathrm{DLT}}(\tilde{t}_1::\tilde{t}_2)=\alpha_{\mathrm{DLT}}(\tilde{t}_1)::-\sqcup\beta^{\mathrm{LT}}(\tilde{t}_1)\odot^{\mathrm{DLT}}\alpha_{\mathrm{DLT}}(\tilde{t}_2)$。

其中 $\beta^{\mathrm{LT}}(\tilde{t}_1)$是 $\tilde{t}_1$ 的阻塞条件，即 $\tilde{t}_1$ 中存在发散的叶子结点或 $\tilde{t}_1$ 执行了影响域超出其自身的 cut 操作的条件，定义如下：

$\beta^{\mathrm{LT}}(G,\varepsilon)=$

$$\begin{cases} cuts(G)\infty obs\cup\{con_{LG}(G)\infty obs\} & \text{if } \mathrm{div}(G)\\ cuts(G)\infty obs & \text{if not } \mathrm{div}(G)\end{cases}$$

$\beta^{\mathrm{LT}}(G,\tilde{t})=cuts(G)\infty obs\cup\zeta(\beta^{\mathrm{LT}}(\tilde{t}))$；$\beta^{\mathrm{LT}}(\tilde{t}_1::\tilde{t}_2)=\beta^{\mathrm{LT}}(\tilde{t}_1)\cup(-\sqcup\beta^{\mathrm{LT}}(\tilde{t}_1)\times\beta^{\mathrm{LT}}(\tilde{t}_2))$；

$\odot^{\mathrm{DLT}}$用于向 D-标号树的结点中增加可见性约束，定义如下：

$o\odot^{\mathrm{DLT}}(o_1+G,\varepsilon)=(o\sqcap o_1+G,\varepsilon)$；

$o\odot^{\mathrm{DLT}}(o_1+G,\tilde{t})=(o\sqcap o_1+G,o\odot^{\mathrm{DLT}}\tilde{t})$；

$o\odot^{\mathrm{DLT}}(\tilde{t}_1::\tilde{t}_2)=(o\odot^{\mathrm{DLT}}\tilde{t}_1::o\odot^{\mathrm{DLT}}\tilde{t}_2)$。

由 α_{DLT}的定义可知，任给 $T\in\mathrm{LT}$，$\alpha_{\mathrm{DLT}}(T)$最左路径上结点的可见性约束均为 $true_{\emptyset}$。在一个标号树序列 $\tilde{t}_1::\tilde{t}_2$ 中，$\tilde{t}_2$ 中的结点可见仅当 $\tilde{t}_1$ 不阻塞，即 $\tilde{t}_1$ 既不发散也没有执行一个影响域超过 $\tilde{t}_1$ 而使 $\tilde{t}_2$ 被“剪除”的 cut。

例 6.2 已知第 3 节中的程序 P，图 3(a)中的 P-导向标号树 T 可以转化为图 3(b)中的 D-标号树。由图可知 T 的右孩子结点可访问仅当$\exists x(\delta_{x,a}\wedge x=4)$不为真。

既然通过结点的可见性约束可判定该结点是否可访问，因此可定义一个从 $\mathrm{Seq}^{+}(\mathrm{DLT})$到 $\mathrm{Seq}^{+}(\mathrm{LT})$的映射 α_{LT}，删除 D-标号树中不能被访问的结点。$\alpha_{\mathrm{LT}}:\mathrm{Seq}^{+}(\mathrm{DLT})\mapsto\mathrm{Seq}^{+}(\mathrm{LT})$的定义如下：

$\alpha_{\mathrm{LT}}(o+G,\tilde{t})=$

$$\begin{cases}\varepsilon & \text{if } (o \text{ 不为真});\\ (G,\varepsilon) & \text{if } (o \text{ 为真且 } con_{LG}(G)=\mathrm{false});\\ (G,\alpha_{\mathrm{LT}}(\tilde{t})) & \text{if}(o \text{ 为真且 } con_{LG}(G)\neq\mathrm{false}).\end{cases}$$

$\alpha_{\mathrm{LT}}(\tilde{t}_1::\tilde{t}_2)=\alpha_{\mathrm{LT}}(\tilde{t}_1)::\alpha_{\mathrm{LT}}(\tilde{t}_2)$.

根据 α_{LT}的定义，任意 D-标号树序列 $\tilde{t}\in\mathrm{Seq}^{+}(\mathrm{DLT})$中的矛盾目标仅可作为 $\alpha_{\mathrm{LT}}(\tilde{t})$的叶子结点。任意矛盾目标的后继结点均可根据定义删除。

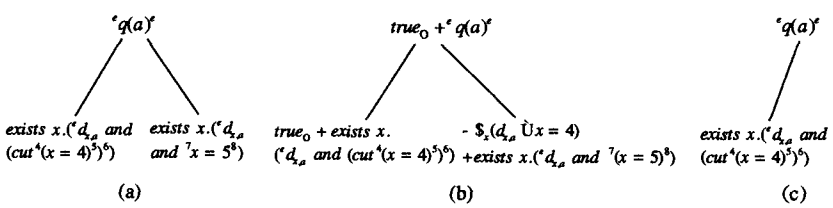

图 3 (a)标号树 T；(b) T 对应的 D-标号树；(c)由 T 中可见结点构成的标号树 $\alpha_{\mathrm{obs}}(T)$

于是可定义从 $Seq^{+}(LT)$到其自身的映射：$\alpha_{\mathrm{obs}}(\tilde{t})=\alpha_{LT}(\alpha_{DLT}(\tilde{t}))$，其功能是删除 $\tilde{t}$ 中不可见的结点，并保留最少数目的矛盾结点。以例 6.2 中标号树 T 为例，T 中不包含矛盾结

(下转第 297 页)

片 ATOM-I。实际芯片的性能评测结果表明：该嵌入式协处理器对多媒体应用具有显著的加速效果，并且整个芯片的平均功耗仅有 0.91W。因此，本文研究并设计的基于 TTA 结构的嵌入式协处理器较好地满足了数字信号处理的需求，能够完成多媒体应用领域的关键算法加速。

参 考 文 献

1 Corbal J, et al. DLP+TLP processors for the next generation of media workloads. In: Proc. 7th Intl Symp on HPCA, 2001
2 TMS320C64 CPU and Instruction Set Reference Guide, Texas Instruments, Inc, USA, 2000
3 Corporaal H, Janssen J, Arnold M. Computation in the Context of Transport Triggered Architectures. International Journal of Parallel Programming, August 2000, 28(4): 401～427
4 13 Dr Cichon G. Introduction into Synchronous Transfer Architecture (STA). Feb. 2005
5 岳虹，沈立，戴葵，等. 基于 TTA 的嵌入式 ASIP 设计. 研究与发展，2006，43(4)：752～758
6 Hoogerbrugge J, Corporaal H. Transport-triggering vs operation-triggering. In: Compiler Construction Conference CC-94, 1994
7 Yue Hong, Lai Ming-Che, Dai Kui, et al. Design of a Configurable Embedded Processor Architecture for DSP Functions. In: Proceedings. 11th International Conference on, July 2005, (2): 27～31
8 Kapasi U J, Dally W J, Rixner S. Efficient Conditional Operations for Data-parallel Architectures. In: Proc. Intl Symp on Microarchitecture, Dec. 2000. 159～170
9 Jayasena N, Erez M, Ahn Jung Ho, et al. Stream Register Files with Indexed Access. In: Tenth International Symposium on High Performance Computer Architecture, Madrid, Spain, February 2004
10 Wu Nan, Wen Mei, Li Haiyan, et al. A Stream Architecture Supporting Multiple Stream Execution Models. LNCS 3740. Sep. 2005. 143～156
11 Lopez-Lagunas A, Chai S M. Compiler Manipulation of Stream Descriptors for Data Access Optimization. In: Proceedings of the International Conference Workshops on Parallel Processing, Jan. 2006. 337～344
12 http://www.gaisler.com/cms4_5_3/index.php?option=com_content&task=view&id=115&Itemid=103
13 Pitkanen T, Rantanen T, Cilio A, et al. Hardware Cost Estimation for Application-Specific Processor Design. SAMOS 2005, LNCS 3553, 2005. 212～221

(上接第 252 页)

点；由于 $\exists x(\delta_{x,a} \wedge x=4)$ 为真，因此根结点的右孩子结点不可见，故 $\alpha_{dbs}(T)$ 为图 3(c)所示的标号树。

定理 6.3 已知程序 P 和目标 G，在 LT 的完备化扩展上以下关系成立：

$\bigcup_{i\geqslant 0}\mathcal{O}_{G,P,i}=\alpha_{dbs}(\bigcup_{i\geqslant 0}K_i^G)=\alpha_{dbs}(J_P^{LT}[\![G]\!]F_P^{LT})$。

以上定理表明，LT 语义包含了比本文操作语义更多的信息，通过删除 LT 语义中那些实际上不能被访问但对于获得语义的目标独立性所必须的结点，可以得到与操作语义等价的信息。该定理可视为 LT 语义相对于操作语义的正确性结论。因此 LT 语义可作为目标独立的 Prolog 程序路径依赖分析的基础。

结束语 在抽象解释的框架内进行目标独立的 Prolog 程序路径依赖分析，需要建立能完整描述程序执行路径的目标独立语义。现有 Prolog 语义不能满足该分析的要求。本文给出了一种以标号树序列为语义域、目标独立的标号树语义。由于该语义可利用开放和闭合 cut 为 cut 操作建模，且每一个标号树结点均携带了完整的执行路径信息，因而可用于具有 cut 操作的 Prolog 程序目标独立分析，并利用调用串提高程序分析的精度。我们证明了该语义相对于本文操作语义的正确性。将来的工作包括根据不同需求开发基于该语义的抽象解释工具，并研究其在 Prolog 编译器和部分求值器中的应用。

参 考 文 献

1 Barbuti R, Codish M, Giacobazzi R, et al. Modeling Prolog control. Journal of Logic and Computation, 1993, 3: 579～603
2 Barbuti R, Codish M, Giacobazzi R, et al. Oracle Semantics for Prolog. In: Kirchner H, Levi G, eds. Algebraic and Logic Programming, Proceedings of the Third International Conference, Lecture Notes in Computer Science, vol 632, 1992. 100～114
3 Barbuti R, Giacobazzi R, Levi G. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. ACM Transactions on Programming Languages and Systems, 1993, 15(1): 133～181
4 Bossi A, Bugliesi M, Fabris M. Fixpoint Semantics for Prolog. In: Warren D S, ed. Proceedings of the Tenth International Conference on Logic Programming, Cambridge, MA: MIP Prese, 1993. 374～389
5 Börger E. A Logical Operational Semantics for Full Prolog. In: Börger E, Büning H K, Richter M M, eds. CSL'89. Third Workshop on Computer Science Logic, Lecture Notes in Computer Science, vol 440, 1990. 36～64
6 Codish M, Dams D, Yardeni E. Bottom-up Abstract Interpretation of Logic Programs. Journal of Theoretical Computer Science, 1994, 124: 93～125
7 Cousot P, Cousot R. Abstract Interpretation and Applications to Logic Programs. Journal of Logic Programming, 1992, 13 (23): 103～179
8 Filé G, Rossi S. Static Analysis of Prolog with Cut. LPAR, 1993. 134～145
9 Fitting M. A Deterministic Prolog Fixpoint Semantics. Journal of Logic Programming, 1985, 2(2): 111～118
10 Gabbrielli M, Levi G, Meo M C. Resultants Semantics for Prolog. Journal of Logic and Computation, 1996, 6(4): 491～521
11 Henkin L, Monk J D, Tarski A. Cylindric Algebras (Parts I and II). North-Holland, Amsterdam, 1971
12 Jaffar J, Maher M J. Constraint Logic Programming: A survey. Journal of Logic Programming, 1994. 19～20, 503～581
13 Charlier B L, Rossi S, Van Hentenryck P. An abstract Interpretation Framework Which Accurately Handles Prolog Search Rule and the Cut. In: Bruynooghe M, ed. Proceedings of the 1994 International Symposium on Logic Programming, Cambridge, MA, MIT Press, 1994. 157～171
14 Le Charlier B, Rossi S, Van Hentenryck P. Sequence-based Abstract Interpretation of Prolog. Theory and Practice of Logic Programming, 2002, 2(1): 25～84
15 Levi G, Micciancio D. Analysis of Pure Prolog Programs. In: Sessa M I, Ed. Proceedings GULP-PRODE '95, 1995. 521～532
16 Levi G, Spoto F. Accurate Analysis of Prolog with Cut. In: Lucio P, Martelli M, Navarro M, eds. Proceedings APPIA-GULP-PRODE'96, 1996. 481～492
17 Lu L. Path Dependent Analysis of Logic Programs. In: Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '02), 2002. 63～74
18 Marriott K, SØndergaard H, Jones N D. Denotational abstract interpretation of logic programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, 16(3): 607～648
19 Mellish C. Abstract Interpretation of Prolog Programs. In: Abramsky S, Hankin C, eds. Abstract Interpretation of Declarative Languages, 1987. 181～198
20 Nielson F, Nielson H R, Hankin C. Principles of Program Analysis. Springer-Verlag Heidelberg, 1999
21 Spoto F. Operational and Goal-independent Denotational Semantics for Prolog with Cut. The Journal of Logic Programming, 2000, 42: 1～46
22 Spoto F, Levi G. Abstract Interpretation of Prolog Programs. In: Haeberer A M, ed. Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology, AMAST'98, Lecture Notes in Computer Science, vol 1548. Amazonia, Manaus, Brazil, January, Springer, Berlin, 1999. 455～470
23 Tarski A. A Lattice Theoretical Fixpoint Theorem and its Applications. Pacific Journal of Mathematics, 1995, 5: 285～310
24 Winsborough H. Path-dependent Reachability Analysis for Multiple Specialization. In: Proceedings of the North American Conference on Logic Programming, 1989. 133～153