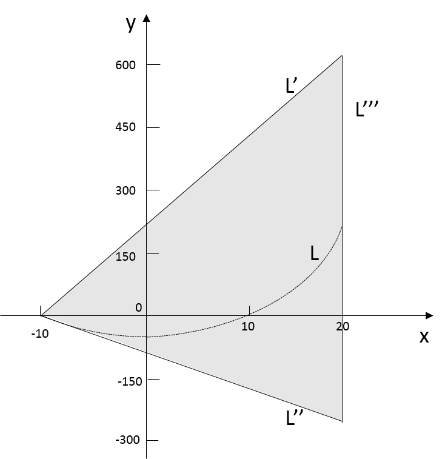
Hierarchical analysis of loops with relaxed abstract transformers

**Abstract:**

1. **Introduction**

基于抽象解释[1]的静态分析技术不运行程序，而是通过分析程序的源代码，来自动获取程序的语义，从而验证程序的正确性或者发现可能的程序错误。 抽象解释使用抽象语义来上近似程序的具体语义，从而在有限的时间和空间内，求得在任何程序点的程序可达状态集合的上近似。这种上近似包含了程序中不存在的行为，所以可能产生误报。提高分析精度，降低误报率是基于抽象解释的静态分析技术在实际应用所面临的主要挑战之一。

当前，基于抽象解释的分析的精度损失主要来源包括两个方面：一是受限于所选抽象域本身的表达能力，如常用的数值抽象域 Intervel(𝑥=[𝑎,𝑏]), Octagon(±𝑥±𝑦≤𝑐), Polyhedron(\Sum\_i a\_i x\_i<=𝑐)只能表示线性凸空间，对于非线性或者非凸空间无法精确描述。二是为了保证对循环迭代分析的收敛性，通常会引入加宽操作，此操作会aggressively增加循环头处的可行程序状态空间，以期迅速求得循环不变式，但一般也会导致了大量的精度损失。尤其，采用线性表达能力抽象域来分析含非线性循环不变式的程序，很容易导致变量值范围加宽到无穷边界，从而引入大量运行时错误的虚假报警。



void main()

{

int x=-10,y=0;

while(x<=20)

{

①x=x+1;

②y=y+x;//y=y+x\*x\*x;

}

}

(a) A program *sum* (b) Depiction of invariants

Fig. 1. A Motivating Example

考虑Figure 1中的Motivating例子，该程序通过一个循环计算了从-9到21之间所有整数的和，并将结果存储于变量y中。从程序具体执行的角度来看，程序点①处x和y总是满足二次等式关系y=(x-9)\*(x+10)/2, 其中-10<=x<=20，如Figure 1(b)中散点构成的抛物线段(L)。从这个二次关系不难得出y的值总是满足 -45<=y<=165,所以在程序点②处语句y=y+x;不存在整数溢出。但是基于现有的线性抽象域（包括区间、八边形或多面体抽象域等）和现有的加宽算子，在程序点①处得不到任何关于变量y的有意义不变式，即只能得到y属于[-oo,+oo]。从而，在程序点②y=y+x； the analysis会给出整数溢出的警报。但这显然是一个误报。 为了消除这类警告，很多工作设计了特定的非线性抽象域来描述程序中的非线性性质，但这类抽象域时间和空间复杂度高，一般是面向特定领域特征的，不具备通用性而且通常只能描述二次关系（而无法表达更高次关系）。在文献中的工作虽然能描述Motivation Example的y=(x-9)\*(x+10)/2不变式，但如果我们将y=y+x;语句改成y=y+x\*x\*x;则这些工作将不能表达新的更高次的多项式不变式，而且y的所有信息会在加宽过程中被损失掉，依然会产生整数溢出误报。

本文提出一种新的面向循环的层次化分析方法。该方法的主要思想是利用循环中的已计算出来的部分不变式（只涉及部分程序变量的不变式）对程序中的迁移函数进行Relaxing，使得分析relaxed之后的程序因为减少了不稳定因素从而能够得到更精确的分析结果，，从而提高循环的分析精度，减少误报。具体而言，本文首先提出了一种基于变量依赖关系的变量层次图，然后基于变量层次图构建程序的精化抽象版本序列。然后，通过对只涉及较低层次的变量的抽象程序进行分析，得到较低层次的变量间的不变式（称为部分不变式）。接下来，利用只涉及较低层次变量上的部分不变式，对涉及更高层次变量的精化抽象程序的迁移函数进行Relax（如使用低层次变量的变量范围替换迁移函数中低层次变量的出现），使得现有的基于抽象解释的加宽技术能够得到更加精确的循环不变式，从而减少抽象解释工具分析程序的误报。在此基础上，本文还提出了一些优化的分析策略来对Relaxing之后的程序进行分析，以得到更精确的分析结果。最后，基于上述技术途径，设计了一个基于抽象解释的程序分析框架，并实现了一个工具原型。初步实验结果表明，本文方法能够获得更高精度的不变式。

对于Motivation Example，基于本文的层次化分析方法，采用多面体抽象域进行分析即可得到循环头处关于x和y的线性不变式：{x<=20, -x<=10; -9x-y<=90; -21x+y<=210}。几何上，该不变式表示的区域见Figure 1(b)中灰色三角形区域（即超平面L’(-9x-y<=90 )、L’’(-21x+y<=210 )和L’’’(x<=20)之间的区域）。从Figure 1(b)可看出，基于线性抽象域虽然不能精确得推导出原程序的二次不变式（即几何上的抛物线），但可以通过多面体将它可靠得包住，得到一个有界的可靠上近似。此时再来检查语句y=y+x；是否导致整数溢出，从上述线性约束可以知道程序点②处 x的范围是[-9,21]，y的范围是 [-270,630]，从而，能够证明语句y=y+x；不会产生整数溢出，从而将此处的整数溢出误报消除。

论文第二节介绍本文方法的overview和总体框架；第三节重点介绍如何构建循环的变量依赖分层图，并基于该分层图对程序进行切片；第四节介绍基于程序部分不变式对语义方程进行Relaxing的过程，并介绍Relaxing的相关策略；第五节采用基于循环迭代次数和求通项公式的方式提高分析效率和部分程序的精度；第六节从理论上讨论方法的可靠性和精度分析；第七节介绍实验的设置和结果；第八节和第九节介绍相关工作并总结本文工作。

1. **Overview**

在实际程序中，我们很容易发现，在循环中变量之间存在层次化的依赖关系。比如，循环控制变量（如loop counter）除了依赖自己或某个输入，往往不依赖于循环中的其他变量；而很多变量单向依赖于循环控制变量。如果用抽象域来同时分析循环中所有的变量，在两次迭代时很可能因为不稳定因素太多（如很多变量值都在变化），加宽之后精度损失很大。尤其是对于诸如Motivating example含非线性行为的程序来说，如果用线性抽象域来同时分析所有的变量，很容易因为加宽导致分析不精确。

基于以上观察，考虑到循环中有些变量的值单向依赖于其他变量的值，因此本文首先会为循环中所涉及的变量构建一个层次式变量依赖图。然后，依据层次式变量依赖图，以一种迭代精化的方式，开展层次式程序分析，先分析底层变量之间的不变式，再利用底层变量间的不变式信息来分析高层变量之间的不变式，逐层分析，从底到高，依次类推。具体而言，在分析第i层变量时，可以利用涉及到该层（及以下层）的变量集合对原程序进行切片，得到程序的一个切片版本P\_i。接下来，对该切片版本P\_i进行分析，在分析过程中，利用已经获得的关于lower level变量的部分不变式对P\_i中向i层变量赋值并在有表达式中涉及lower level变量的语句迁移函数做relaxing，使得relaxed之后的向i层变量赋值语句的迁移函数中不再涉及lower level变量，从而消除了i层变量对lower level变量的直接数据流依赖。换而言之，在某种程度上，可以消除分析P\_i的两次不动点迭代间的因为依赖导致的不稳定因素（due to the dependency），从而在加宽时可以保留更多稳定的约束，最终得到更高精度的不变式。其中，一个key point是定义一个好的Relaxing算子。一个好的Relaxing算子应该是：在加宽之前将迭代过程中不稳定的因素尽量Relaxing成稳定因素，从而加宽过程中能保留更多稳定的约束，从而减少加宽带来的精度损失。最直观的想法是对于循环体中的部分语句的迁移函数中的不稳定表达式（即相邻两次迭代表达式的值会改变）或者抽象域不能有效的表达式（用Polyhedron分析平方等高次表达式），通过从已有的循环不变式中提取这些表达式的上近似表示，并用这些上近似表示的表达式替换语句中原有的表达式，以上过程即为对语义方程的Relaxing。但值得注意的是，Relaxing过程所依赖的循环不变式不需要是涉及所有变量的不变式，而只需要包含Relaxing的表达式中所包含的所有变量之间的可靠约束关系即可，本文称之为循环的部分不变式。

Relaxed Semantic Equations

Partial Loop Invariant

Input：a loop

Fixpoint solver

Relaxing

Abstract

Domain

Slice Versions

Invovling High-level varibles

Involving low-level

varaibles

HVDG

Fig. 2. Framework of our method

Figure 2给出了我们的方法的架构框图。从该框图中可以看出，本文生成循环不变式的方法主要包括以下四个过程：构建分层变量依赖图；基于变量层次的程序切片；基于部分不变式的语义Relaxing，以及基于抽象解释的不变式生成。这些过程将在后续章节详细介绍。

下面以Motivating example为分析对象，来illustrate我们的方法。对于Motivating example首先对循环体中所涉及的两个变量x,y之间的数据依赖和控制依赖进行分析，发现该程序中x只依赖于x自身的值，变量y依赖于x和 y的值。这样，可以把该循环涉及的两个变量x,y分成两个层次，变量x在第一层，而变量y在第二层，如果Figure 3(a)所示。接下来，首先利用第一层变量（即{x}）对程序进行切片，得到P\_0，如Figure3(a)所示，注意该程序中涉及变量x。因为P\_0只涉及最底层变量，无需relaxing。然后应用基于抽象解释的不变式生成技术来分析P\_0，可以得到变量x在程序点②处的部分不变式：x=[-9，21]。接下来，分析第二层变量，利用第一、二层变量（即{x,y}）对程序进行切片，得到P\_1，如Figure3(b)所示，注意这里P\_1即为原程序。然后，对P\_1中涉及第二层变量的赋值语句（即程序点②处的语句y=y+x;//y=y+x\*x\*x;）进行Relaxing。这里，我们使用之前求出关于第一层变量x的部分不变式，可以得到x的取值范围x\in[-9，21]。将程序点②后的赋值语句涉及低层变量x的右值表达式利用x的区间范围计算的结果来替换，得到Relaxing之后的迁移语义等价于Figure3(c)所示程序。注意，此时第二层变量y将只依赖于自身，不再数据依赖于x。对Figure3(c)所示程序应用多面体抽象域来分析，即可得到循环头处关于x和y的线性不变式：

{ -x+20>=0, x+10>=0, 9x+y+90>=0, 21x-y+210>=0}

因为程序变量只有两层，上面所得线性不变式即为本方法最终得到的不变式。

1. (b) (c)

Fig. 3. Sliced and relaxed versions of the original program

1. **分层变量依赖图**

本节介绍分层变量依赖图Hierarchal Variable Dependency Graph（HVDG）的定义与构建方法，以及基于HVDG的程序切片。在阐述过程中以Figure 4程序为例，介绍相关定义以及构建HVDG和程序切片的过程。

while(①t<1000)

{

②t=t+1;

③x=x+y+2;

④y=y+2\*x;

if(⑤z+y>100)

⑥z=z+y+1;

else

⑦z=x+u;

}

Fig. 4. Another example program

* 1. 变量依赖（Variable Dependency）

定义3.1[基于控制依赖的变量依赖] 称变量v1控制依赖于变量v2，if存在对v1的赋值语句出现在涉及v2的条件分支（包括循环条件）中。

例如： 在Figure 4程序中，程序点③处语句对x进行赋值，该赋值语句出现在程序点①的条件分支中，而该条件分支语句含有变量t，故变量x控制依赖于t。

定义 3.2[基于数据依赖的变量依赖] 称变量v1数据依赖于变量v2，if v2出现在对v1的赋值语句的右表达式中。

例如： 在Figure 4程序中，程序点③处语句对x进行赋值，且该赋值语句的右表达式中出现了变量x和y，故变量x数据依赖于x和y。

定义 3.3[变量直接依赖]. 对于任意两个变量x、y，若x直接控制依赖于y或者x直接数据依赖于y，则称x直接依赖于y。

定义 3.4[变量间接依赖].如果存在变量x0，x1，… ，xn-1（n>=2）使得对任意的i>=0,i<n，xi+1直接依赖于xi，则称xn间接依赖于x0。

例如：在Figure 4程序中，从程序点⑥的赋值语句可知变量z直接依赖于y，而从程序点④的赋值语句可知y直接依赖于x，由此可知变量z间接依赖于x。

定义 3.5[变量依赖] 若变量x直接或者间接依赖于y，则称变量x依赖于y。

* 1. 变量依赖图

3.2.1 相关定义

定义 3.7[变量集合依赖]变量集合A（直接）依赖于变量集合B，当且仅当存在变量xA，yB，x（直接）依赖于y。

例如：在Figure 4程序中，变量z依赖于变量t，故变量集合{z}依赖于变量集合{t}。

定义 3.8[变量直接依赖图]设循环中出现的所有变量构成的集合为Var。该循环的变量依赖图是a directed graph G = <N, E>，其中每个节点n\in N是变量集合Var的一个子集（记作Var\_n），节点n\_1到节点n\_2的有向边表示变量集合Var\_1直接依赖于变量集合Var\_2。

For the sake of simplicity，此处约定如果Var\_n依赖于Var\_n自身，不显示加入结点n到n的有向边，即变量依赖图中的节点上不存在自圈。本文的目标是为了推导不同变量之间的依赖关系，故这一约定不影响我们的结果。

定义 3.9[分层变量依赖图] 分层变量依赖图是由变量依赖图衍生出来的a undirected graph G = <N, E>, where如果结点n1对应的额变量集合V\_n1 直接依赖于 结点n2对应的变量集合V\_n2，则我们把结点n\_1放在结点n\_2的下一层，并添加一条n\_1到n\_2的无向边。换而言之，分层变量依赖图可以看成是由变量依赖图衍生出来的Hasse图。

3.2.2 构建变量依赖图

有了上述定义，接下来介绍如何构建循环中分层变量依赖图，总共分为以下四个步骤：

步骤一[直接依赖图构建]：若变量x直接依赖于y，则建立一条{x}到{y}的有向边。

例如：例子程序中根据变量之间直接依赖关系，可以得到Figure 5左一的变量直接依赖图。

Fig. 5. Variable Dependency Graph of example program

步骤二[变量合并]：若变量集合A依赖于变量集合B且B依赖于A，则将集合A和B中的所有变量合并成一个变量集合C，即C=AB，并将集合A和B与其他变量集合的关系转换成C和其他变量的关系。在图上的操作是将节点A和节点B合并成一个节点C，并分别将A和B的出边和入边转换成C的出边和入边。

例如：由变量集合{x}和{y}之间存在互相依赖关系，故可将二者进行合并，得到Figure 5中左二的变量依赖图。

步骤三[直接依赖边删除]：若变量集合A直接依赖于变量集合B，同时A也间接依赖于B，则可以移除A对B的直接依赖关系，即删除B到A的有向边。

例如：由于节点{z}通过{x,y}间接依赖于{t}，同时{z}又直接依赖于{t}，故可以删除{z}和{t}直接的直接依赖关系，从而得Figure 5右二的变量依赖图。

步骤四[构建Hasse图]：经过前三步处理后，再对该图进行自底向上的分层转换即可得到Hasse图，即如果节点A到B之间存在一条有向边，则从A到B绘制一条向上的线段，不存在依赖关系的节点自底向上置于同一层。

例如：节点{t}到{x,y}、{x,y}到{z}、{u}到{z}之间各有一条有向边，故将{t}置于{x,y}的下方，{x,y}置于{z}的下方，{u}置于{z}的下方；{t}和{u}、{x,y}都不存在依赖关系，根据自底向上的原则，{t}和{u}置于同一层。最终得到Faigure 7右所示的变量依赖Hasse图。

* 1. 变量分层

Hasse图中的节点按照变量集合的依赖关系从底向上排列，根据节点在Hasse图中的层次可以直接得到变量集合之间的层次关系，即如果Hasse图的高度为h，则表示循环中所有的变量可以分为h层；如果节点A出现在Hasse图的第i层（0<=i<h），则A中的所有变量为循环的第i层变量。

例如：根据Figure 5 右一中的变量依赖Hasse图可知，循环中所有变量可以分为3层，其中t、u是0层变量，x、y是1层变量，变量z为2层变量。

* 1. 基于变量分层的循环切片

设循环中变量共可分为n+1层(编号分别是从0到n)，则对于任意的i>=0,i<=n，基于第i层变量对循环体进行切片[13]，设从0层变量到i层变量中的所有变量构成的集合为S，设USE[S]表示语句S读取变量的集合，DEF[S]表示语句S所定义的变量的集合。

（1）对于所有出现在L中的赋值语句sa，如果DEF[sa]包含于S，则保留L中的sa语句，否则删除该语句。

（2）对于所有出现在L中的所有条件语句sc，如果USE[s1]包含于S，则保留L中sc语句。当条件语句sc分别对应的真分支或假分支中不包含任何赋值语句时，则在相应分支中插入skip；语句。

经过上述两步操作即可得到基于前（低于）i层变量进行切片得到的子循环体L[i]。当i=n时，0到n层变量包含了循环体L中出现的所有变量，故此时切片后得到的程序为L本身。

根据以上规则，对于例子程序可得到程序序列L[0]、L[1]和L[2]，即Figure 6从左至右三段程序。

Fig. 6. Loop sequence after slicing by variable hierarchy

while(①t<1000)

{

②t=t+1;

}

while(①t<1000)

{

②t=t+1;

③x=x+y+2;

④y=y+2\*x;

if(⑤z+y>100)

⑥z=z+y+1;

else

⑦z=x+u;

}

while(①t<1000)

{

②t=t+1;

③x=x+y+2;

④y=y+2\*x;

if(⑤z+y>100)

⑥skip;

else

⑦skip;

}

1. **基于部分程序不变式的迁移函数Relaxing**

Relaxing的策略是本文方法能否提高精度的关键，但定义Relaxing的策略可以有很多。不同的策略对同一个程序提高精度的效果可能不一样，而同一个Relaxing策略对于不同的程序、不同的抽象域和不同的迭代方式其效果可能都不一样，即很难有一个策略能适用于所有的情形。本节首先介绍Relaxing的定义及其可靠性保证[14]，再介绍了一些通用的Relaxing策略和针对特定抽象域和迭代方式的策略。

* 1. Relaxing的定义及其可靠性

在语义方程中，有两类节点，一类是赋值语句如V <- exp, 一类是条件语句exp 0，（其中表示<,>,=等运算符）。在语义方程的Relaxing时，我们不改变左值变量V和常量0的值，主要是为了对exp做Relaxing，下面首先介绍对exp的可靠Relaxing。

定义4.1[表达式的部分序关系][14]

当且仅当 。

上述定义中要求对任意的environment ，都满足表达式的值存在包含关系，但在实际的循环迭代过程中，变量在具体语义下的environment总是限定在一个固定的循环不变式中，为此我们定义在固定环境下的表达式部分序关系。

定义4.2[基于不变式的表达式序关系]

当且仅当 。

定义4.3[迁移函数的Relaxing]

设，则将迁移函数 和 0 分别替换成 和 0，称为抽象环境的迁移函数Relaxing。

在语义方程的Relaxing中，通常是我们在切片得到的子程序分析得到的部分不变式，它是基于抽象域表示的抽象环境的集合，设抽象域到具体域的具体化算子为，则以下定理保证了Relaxing的可靠性。

定理4.1 [可靠性定理] [14]设，则

,

(。

* 1. Relaxing策略

设基于HVDHG Slicing后得到的循环序列为*L0，L1,…,Ln*，且当前已经通过不动点求解（Relaxed）*Li-1*（*1=<i<=n*）得到部分循环不变式，接下来我们将对*Li*进行Relaxing得到*Li’*。为了更好得描述Relaxing策略， 首先假设在给定的抽象域D#中定义了一个投影操作projection，该操作的功能即为：将一个抽象环境中的每个变量的值都投影成一个区间范围。在保证可靠性的前提下，Relaxing算子可以有多种多样的定义方式，如何定义一个好的Relaxing策略是提升精度的关键。

4.2.1 All variables relaxing strategy

该策略的主要思想是：对高level的切片程序中新增的所有赋值语句和条件语句中出现的低level的变量全部做relaxing。

具体而言，首先将低层切片程序分析获得的抽象循环不变式Inv投影到变量的各个维度，即可得到所有低层变量的一个区间范围，然后对于当前程序的赋值语句的右值表达式和条件语句的左侧表达式中出现的所有低层变量都用其区间值进行Relaxing，从而得到一个新的语义方程，即为Relaxed语义方程。

这种策略虽然具有很强的可操作性，却有一个明显的弱点，即如果投影后得变量的值范围是top，此时如果再对当前的程序做All variables relaxing，则包含该变量的赋值语句的左值变量的不变式直接也会变成top，条件语句也变成了没有任何限制，即等同于brandom(True or False)语义一样。这样不但无法提高分析精度，反而可能因丢失部分变量之间的约束关系而降低分析精度。

例如：设循环的初始值满足y==z，求解*Li-1*得到的关于x的不变式为，*Li*相对于*Li-1*新增的两条语句为y=y+x;z=z+x;此时如果不对x做Relaxing，虽然不能得到y和z的有界值范围，但依然能得到y=z的约束关系；如果对x做Relaxing，反而连y=z的约束关系也不能得到。

为了解决这个问题，对该策略稍作调整，得到Bounded variables relaxing：只对程序中有具体的上界或者下界的变量做relaxing，对于包含无穷上界和下界的变量不做relaxing，这样可以避免单个无上下界变量的Relaxing带来整个表达式的无上下界。

此外，在本策略中，基于单个变量的值范围做Relaxing最易想到的，当然也可以基于复合表达式的值范围做Relaxing。

例如：设求解*Li-1*得到的关于x的不变式为，*Li*相对于*Li-1*新增的语句为{z=z+x+y；}如果先将投影成变量x和y的区间范围会得到x，y都是top，再对{z=z+x+y；}进行Relaxing要么不对x,y做替换，要么替换了只能得到z也为top；而如果将表达式x+y作为整体求得其值范再Relaxing，即可得到更加简洁的语句，即{z=z+10；}。

故All Variables Relaxing策略可以考虑先对表达式中的低层变量和高层变量进行分离（加/减法可分离，但乘除法不好分离时，仍采用代入区间结果的方式），分离后得到的只含低层变量的表达式可以求其整体的值范围，再基于表达式整体的值范围做Relaxing。

4.2.2 Templated relaxing strategy

上述All Variable Relaxing策略虽然对抽象域和不动点迭代方式都没有任何限制，具有较好的通用性，但对于不同的抽象域和不动点迭代方法却不一定有明显的精度提升，甚至可能将低精度。为了更好的提高Relaxing后的分析精度，本节介绍一些在给定抽象域（如多面体或者区间抽象域）和不动点求解方式（如带加宽的Kleene迭代，或者将在下一节中介绍的基于通项公式的方法）下的基于模版的Relaxing策略。

1. Linearization Relaxing of single assignment

本策略是为了提高基于Polyhedron抽象域的传统抽象解释（即基于加宽的Kleene迭代求不动点）分析精度。该策略的主要思想是：首先将表达式Relaxing成带区间仿射形式[14][15](Interval affine)

其中是常量区间

例如: 设求解*Li-1*得到的关于y和z的不变式为y=[c,d],z=[e,f]，*Li*相对于*Li-1*新增的语句为{x=y\*y+z；}，基于区间仿射的Relaxing策略只需要替换一次变量y的值，即可以得到：x=[c,d]\*y+z;

此时若采用Liqian实现的Interval Polyhedron Domain[16]即已经可得到变量之间的带区间系数的线性约束。若依然采用经典的Polyhedron，可进一步将区间仿射形式线性化成如下通用模式[14]

（其中是具体常量，通常取区间的中点）

例如：对于上例中已经得到的区间仿射语句x=[c,d]\*y+z；可进一步进行线性化得到Polyhedron能有效表示的语句：x=(c+d)\*y/2+z+[m,n];其中[m,n]=[(c-d)/2,[d-c]/2][c,d]，从而可以得到变量x，y，z之间的线性不等式约束。

（2）多语句关联Relaxing策略

该策略同样适用于提高基于Polyhedron抽象域的传统抽象解释分析精度。本节以两条赋值语句为例，阐述本策略的主要思想：首先将两条赋值语句Relaxing成带区间常量的单变量赋值语句：

;

; （其中、为区间常量，、为具体常量）

Case 1. 如果，则基于Polyhedron抽象域可以得到和之间的线性约束。

例如：在本文的Motivation Example中，基于本Relaxing策略可以得到两条赋值语句：

x=x+1;

y=y+[-9,21];

又已知循环的初始值x=-10，y=0，即可在循环头得到Polyhedron抽象域表示的循环不变式：

{-x+20>=0; x+10>=0; 9x+y+90>=0; 21x-y+210>=0}

Case 2. 如果，此时和之间存在指数关系，无法直接用Polyhedron表示，则可尝试基于已经获得的部分不变式，将二者变成相等，即如下通用形式：

;

; （其中、为区间常量， or ）

例如：对本文的Motivation Example中，将x的初始值修改为0，将语句x=x+1；修改为x=2\*x+1;其余都不变，即如Figure 7所示。

void main()

{

int x=0,y=0;

while(x<=20)

{

①x=2\*x+1;

②y=y+x;

}

}

Fig. 7. Modified motivation example

第一遍分析只含x的程序，依然可以得到循环头，即程序点①处，程序点②处。此时基于上述策略将①和②处语句Relaxing成：

①x=x+[1,21];

②y=y+[1,41];

再基于Polyhedron抽象域和带加宽的Kleene迭代策略可以得到循环入口点的x、y之间的如下线性约束关系,进一步可得变量x和y的有穷边界：

{x>=0;-x+20>=0; -x+21y>=0; 41x-y>=0}。

当Case 1和Case 2的条件都不能满足时，则可以考虑每次单独分析每条语句的不变式，即套用接下来的Relaxing 模版。

（3）单变量区间系数赋值语句

在程序中，如果循环体中的赋值语句是可逆的，低层变量都是有界的，这时可以将除左值变量以外的其余变量Relaxing成区间，从而得到一个单变量区间系数的赋值语句形式：

;（其中和为区间常量）

在该赋值语句形式中，如果不是 1、-1、0等常数时，则和循环迭代次数总是存在指数关系，此时基于Polyhedron和Kleene迭代策略在通常情况下已经不能得到有界的值范围，故本文试图用基于通项公式求解的方式快速获得变量的值范围，具体过程将在下一节中展开。本节接下来只讨论如何获得单变量区间系数的赋值语句形式。

Tec 1. 基于Relaxing的多变量互相依赖时的单变量化

通过抽象和Relaxing的方式并不能将所有类型的语句都化成;的形式，当*Li*相对于*Li-1*新增多条语句对不同变量赋值，而这些变量在这些语句中又相互依赖，此时无法直接将这些赋值语句转换成单变量形式，如Figure 8中所示程序（输出非波拉切数列）中情况，x和y与循环迭代次数

int i=0, x=0,y=1;

while(i<10)

{

i=i+1;

①x=x+y;

②y=x+y;

//print x,y;

}

Fig. 8. 斐波那契程序

存在指数关系，x和y是同1层变量，语句①和②会同时加入到切片程序*L1*中，由于x和y互相依赖，故利用现有的Relaxing策略无法对①和②做Relaxing。考虑更加一般的情形，以两条赋值语句为例,经过符号代入[17]，将赋值语句右侧表达式中出现的变量都转换为上遍迭代返回的符号变量，即可得到如下一般形式：

; （是区间常量）

; （是区间常量）

其中和是上一遍循环迭代返回的值，和是本次循环迭代后返回的值，它们是同一变量在相邻两次迭代的不同出现。

令，，，则上式可以可靠得抽象成

; （是区间常量）

; （是区间常量）

从上式可以看出，对抽象后的程序经过一次迭代以后和的区间值范围总是相等的。此时再设的初始值为，的初始值为，令，并将和的初始值都抽象成，则可知，在循环入口点，抽象后的循环中变量和的区间值范围始终相等，即。由此我们再对上式进行等价转换成：

; （是区间常量）

; （是区间常量）

但是需要注意的是上式中和前的的区间系数不能简单相加，这是因为变量和已经被抽象成了区间变量，而区间又不严格满足分配律（设X、Y、Z为区间，则X(Y+Z) ⊆ XY+XZ），直接合并系数会导致结果不可靠。为了得到可靠的抽象，我们设，在上式中将和进一步抽象成，此时和前的系数是一样的，可以做可靠合并(当然也可以根据和的上下界关系进一步分情况讨论得到更加精确的，此处就不再展开)，即得到

; （是区间常量）

; （是区间常量）

通过上述过程，可以将两个变量互相依赖的语句转换成满足;（其中和为区间常量）的一般形式。对于2个以上变量相互依赖的处理方法也是类似。

例如：Figure 8循环中的语句①和②通过符号带入得到

x’=x+y;

y’=x+2y;

经过一步抽象得到

x’=x+[1,2]y;

y’=x+[1,2]y;

此时再将x和y的初始值都抽象为[0，1]，可知总是满足x=y，则可进一步抽象，并合并系数可得到：

x’=[2,3] x;

y’=[2,3] y;

即转换成了通用的形式。

Tec 2. 基于符号传播的单变量区间系数语句合并

在单变量区间系数赋值语句;中要求左值表达式中的和右值表达式中的是相邻两遍循环中在同一点的不同时刻的值，即对的赋值语句在循环体中只出现一次。故对于循环体中对于的多次赋值，需要通过符号抽象和传播[17][18][19]将其转换成一条赋值语句。现假设循环关于的单变量区间系数赋值语句经过Relaxing后有且只有两次出现，即其他语句都没有对进行赋值，在Figure 9的程序片段中省去了这些语句。对于这两次出现，要么是串行关系，要么是并行关系。

如果是串行关系，如Figure 9 左侧程序所示。通过符号传播技术，可将对于的这两条语句替换成skip；语句，并在程序点①处插入关于变量的赋值语句

如果是并行关系（即对变量的两次赋值出现在if-else分支中，如果if或else中只有一个分支对赋值，则在另一个分支中插入=；的赋值语句），如Figure 右侧程序所示。同样通过符号传播技术，可将对于的这两条语句替换成skip；语句，并在程序点①处插入关于变量的赋值语句

通过不断迭代的使用符号传播技术对同一个变量的两次串行或者并行赋值进行合并，最终可以

while()

{

if(\*)

；//、是区间常量

else

; //、是区间常量

//①

}

while()

{

；//、是区间常量

…

; //、是区间常量

//①

}

Fig. 9. Relation of two assignments in loops

将程序中对的任意次赋值语句都转化成满足要求的对变量的单次赋值。

1. **基于循环迭代次数的通项公式法(Formula Method)求区间不变式**

基于Kleene迭代和加宽的方式求解不动点是抽象解释中最常用的技术，虽然能够保证迭代过程的收敛，却也是最易损失分析结果精度。为此，本文对单变量区间系数等特定形式的赋值语句，可以不采用加宽的方式进行迭代，而是通过求通项公式的方式快速获得变量的区间值范围。

* 1. 单变量区间系数语句的通用求解公式

在通常情况下，如果循环不变式是多项式形式的有穷次迭代，很多情况下可以通过多面体抽象域进行可靠近似。但如果循环不变式是指数形式的，则很难再通过多面体抽象域+Kleene迭代的方式获得多面体近似，从而无法得到变量的有穷边界。为此，本文接下来将采用基于通项公式求解的方式直接获得变量的值范围。

为了方便表述，本文在程序中引入一个循环计数变量n，进入循环前插装n=0,循环体尾部插装语句n=n+1，由于n可能不是0层变量，故本文强制将循环计数变量作为0层变量。本文基于模板的 Relaxing策略在很多情况下能够将赋值语句Relaxing成单变量区间系数赋值语句形式：

设x在循环执行前的初始值为x0和，xn表示第n次迭代后x的区间值范围，则通过不断代入可得到：

（公式一）

在公式一中，为了求得的值需要大量的区间加法和乘法运算，故可根据a的正负号进一步对该公式进行化解。

当时，对任意的，，故可以由公式一简化得到如下通项公式

（公式二）

其中的计算方式如下：

当时，对任意的，和不存在固定的大小关系，的取值时正时负，此时无法继续通过上述公式二来求解描述。为了简化公式一得到一个新的通项公式，本文将以损失一部分精度获得一个上近似的结果。即将公式一Relaxing成如下形式：

, 其中

合并后可得到如下通项公式：

其中的计算方式如下：

例如，在Figure 10程序中，变量y的取值和x的取值存在指数型关系。经过对变量的层次依赖图关系进行分析后可知x为0层变量，同时将插入的循环计数变量n也作为0层变量，y为1层变量。通过对0层变量分析可得循环头的不变式为x=[0,10]，n=[0,5]基于这个不变式，此时再对语句y=2\*y+x；进行Relaxing，可得到y=2\*y+[2,12];此时再基于传统的多面体抽象域+Kleene迭代方式得不到y的上界。故我们可以套用公式二，可直接得到y的取值范围是[1,224]。

x=0;

y=1;

n=0;

while(x<=10)

{

x=x+2;

y=2\*y+x;

n=n+1;

}

Fig. 10. Example program with exponential invariants

* 1. 基于通项公式的循环迭代次数求解问题

在循环中插入循环计数变量n后，基于抽象解释对低层变量和迭代次数n同时分析得到的循环不变式，在有些情况下可以直接能够推导出n的上界，如Figure 10程序中可基于第一次分析的不变式得到n的上界为5。但如果无法通过不变式直接推导出n的上界时，可以先假设n是一个符号常量，代入到通项公式求解得到其余变量关于n的函数，再将关于n的函数代入到循环条件中，尝试推导出n的取值上界。接下来以Figure 11中的程序为例介绍推导循n的上界的过程。

x=0;y=0;n=0;

while((y-x)<=100)

{

x=x+2;

if(\*)

y=2\*y+1;

else

y=3\*y+2;

n=n+1;

}

Fig. 11. Computing loop counter by formula method

通过对语义方程进行Relaxing可以得到循环体语句：x=x+2;y=[2,3]y+[1,2];分别代入公式可以获得；再代入到循环条件y-x<=100中得到]，由区间运算规则可知n必须满足通过求解该不等式可知n≤6。从而可以得到循环头x和y的有穷值范围分别是x=[0,12],y=[0,728]。

* 1. 基于通项公式求解和多面体+Kleene迭代方式的结合

基于通项公式的方法只能获得变量的区间值范围，但得不到变量之间的约束关系，为此本文可以基于通项公式法获得的值范围，进一步对迁移函数进行Relaxing，最后再基于多面体+Kleene迭代的方式获得多面体不变式，并将多面体不变式与区间范围做Meet，从而得到更加精确的不变式。

例如Figure 11中的例子，在已经得到x=[0,12],y=[0,728]的基础上，基于多面体模板的Relaxing策略，进一步将y=[2,3]y+[1,2]；Relaxing成y=y+[1,2]y+[1,2]=y+[1,1458];再基于多面体+Kleene的迭代方式，可以得到y和x的线性关系2y>=x,y<=729x，再与区间结果做Meet可以得到最终不变式：

2y>=x,y<=729x，x>=0,x<=12,y<=728

1. **Discussion**

给定循环的初始状态，根据循环体建立的精确迁移函数，则在具体语义下，基于Kleene不动点迭代定理[10]可知，循环体迭代的最小不动点满足：

where (1)

但在实际计算过程中，这种在具体语义下的无限迭代策略效率很低，并且需要大量的存储空间来保存中间结果，为此抽象解释通过一个可靠的抽象化算子,将具体语义下的初始状态和迁移函数分别抽象到基于抽象域表示的抽象状态和抽象迁移函数，从而得到经典抽象解释理论下的可靠不动点迭代过程：

where (2)

在抽象语义下，在Kleene迭代过程中可能产生无穷递增链导致迭代不终止，故为了保证终止性，以及加速Kleene 不动点迭代的收敛速度（同样适用只含有穷递增链的抽象域），抽象解释框架提供了“加宽”（widening）算子。接下来得到含加宽操作的不动点迭代结果记为：

where (3)

为了保证收敛，加宽导致的精度损失可能很大。

在抽象解释框架下，由不动点迭代理论[10][11]可知（1）（2）（3）满足以下关系：

其中是与抽象化算子相对应的具体化算子

我们的目标是基于抽象解释框架在保证分析结果可靠的情况下，通过对迁移语义方程进行Relaxing来提高分析的精度。

在抽象解释中有一个well-know的事实，即加宽算子本身不具有单调性[11]。一方面加宽算子关于第一参数不单调，因此即使有 ，也未必有。例如：设，则显然满足，但。事实上，文[11] 证明了加宽算子不能关于第一参数单调，否则该算子在程序分析中将没有实际意义（例如，设计某加宽算子使其结果恒等于，虽然该算子满足单调性，但是基于该算子的分析得不到任何有意义的结果）；另一方面，加宽算子关于第二参数也不单调，即就算有 ，也未必成立，例如，设在多面体抽象域中, ，。显然满足，但。换而言之，对迁移函数Relaxing 比不Relaxing会得到更不精确的中间结果，但是加宽后可能反而得到更精确的结果。

基于上述观察，我们定义了一个Relaxing算子分别作用在抽象语义下的初始状态和迁移函数上，从而得到如下基于Relaxing的Kleene迭代结果：

where

Relaxing后得到的迁移函数F#满足，由于关于初始值和输入函数单调，又由于算子的单调性可得到：

结合（1）（2）（3）满足的关系，可得到 ，即经过可靠Relaxing后得到的循环不变式是程序具体语义下的不变式的上近似，即结果是可靠的。

在保证Relaxing可靠的前提下，虽然迁移函数存在关系，但由于加宽算子的非单性，和之间的关系反而是不确定。正是由于二者之间关系的非确定性，才使得通过对迁移函数进 行Relaxing来提高分析的精度成为可能，而接下来问题的关键就在于如何设计一个好的Relaxing算子，使得在保证可靠的前提下，尽量减少加宽带来的精度损失。

1. **实验及结果分析**

本文实现所采用的抽象解释分析工具是Interproc[20]，使用的抽象域主要是Apron抽象域库下的Interval[2]和Polyhedron[4]。Our experiments were taken on a virtual machine (using VMware Workstation), with a guest OS of Ubuntu 14.04(2GB Memory), host OS of Windows 8, and a 3.2 GHz quad-core Intel(R) Core(TM) i7-5500U host CPU.

本文共对38个程序进行试验，其中35个程序主要来源是THANHVU NGUYEN and DEEPAK KAPUR等人在[21][22]中所使用的（Non-linear Algorithmic）NLA的Benchmark，在此基础上Sankaranarayanan 等人在[23]中还使用mul2作为测试用例，Rahul Sharma在[24]中使用了prod，petter作为测试用例。这些程序经常作为测试用例集出现在非线性不变式生成的工作中[25-29]，它们的规模都在100行代码以内，都实现了相对独立的数学功能，且包含非线性循环不变式。在这些程序中，很多函数都含有符号输入，为了保证循环终止，我们给定了一组固定的参数传递给这些函数，其中 5个程序包含无穷循环，为了将程序行为限定在Polyhedron能表示的范围内，本文对最大迭代次数进行有界限定。对这些程序分别采用不带Relaxing的传统抽象解释分析和带Relaxing的结合多种不动点迭代方式的结果对比如下表所示：

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | Desr | #V | #D | #H | Standard AI | | Relaxing&AI | | P |
| #B | T/ms | #B | T/ms |
| mul2 | product | 2 | 1p | 2 | 2 | 8 | 2 | 16 | = |
| prod | product | 3 | e | 3 | 0 | 28 | 3 | 84 |  |
| cohendiv | division | 6 | e | 3 | 4 | 28 | 6 | 96 |  |
| mannadiv | division | 3 | 1p | 2 | 3 | 16 | 3 | 36 | = |
| divbin | division | 3 | e | 3 | 0 | 36 | 3 | 128 |  |
| hard | division | 4 | e | 2 | 4 | 28 | 4 | 120 | > |
| wensley | division | 4 | e | 2 | 4 | 36 | 4 | 96 | > |
| dijkstra | square root | 4 | e | 2 | 0 | 40 | 4 | 284 |  |
| sqrt | square root | 3 | 2p | 2 | 0 | 12 | 3 | 32 |  |
| z3sqrt | square root | 3 | e | 1 | 2 | 24 | 2 | 44 | = |
| freire1 | square root | 2 | 2p | 1 | 0 | 8 | 2 | 8 |  |
| freire2 | cubic root | 3 | 3p | 1 | 0 | 16 | 3 | 36 |  |
| euclidex1 | extended gcd | 9 | e | 3 | 0 | 96 | 4 | 420 |  |
| euclidex2 | extended gcd | 6 | e | 2 | 0 | 36 | 0 | 72 | < |
| euclidex3 | extended gcd | 11 | e | 4 | 0 | 96 | 3 | 312 |  |
| fermat | divisor | 3 | 2p | 1 | 0 | 36 | 0 | 112 | < |
| fermat2 | divisor | 3 | 2p | 1 | 2 | 28 | 3 | 84 |  |
| knuth | divisor | 5 | e | 1 | 0 | 124 | 5 | 300 |  |
| lcm | lcm | 4 | e | 2 | 0 | 28 | 0 | 46 | < |
| lcm2 | lcm | 4 | e | 2 | 0 | 32 | 0 | 68 | < |
| illinois | protocol | 4 | 3p | 3 | 0 | 116 | 2 | 176 |  |
| berkeley | protocol | 4 | 3p | 3 | 0 | 60 | 3 | 188 |  |
| firefly | protocol | 4 | 3p | 3 | 0 | 76 | 2 | 140 |  |
| mesi | protocol | 4 | 3p | 3 | 0 | 52 | 3 | 220 |  |
| moesi | protocol | 5 | 3p | 3 | 0 | 72 | 4 | 136 |  |
| prod4br | product | 4 | e | 3 | 0 | 168 | 4 | 244 |  |
| readers\_writers | simulation | 3 | 1p | 2 | 0 | 36 | 0 | 76 | = |
| cohencu | cubic sum | 4 | 3p | 4 | 2 | 16 | 4 | 76 |  |
| petter | power sum | 2 | 6p | 2 | 1 | 8 | 2 | 16 |  |
| ps1 | power sum | 3 | 1p | 2 | 3 | 8 | 3 | 20 | = |
| ps2 | power sum | 3 | 2p | 3 | 0 | 8 | 3 | 36 |  |
| ps3 | power sum | 3 | 3p | 3 | 0 | 12 | 3 | 40 |  |
| ps4 | power sum | 3 | 4p | 3 | 0 | 12 | 3 | 40 |  |
| ps5 | power sum | 3 | 5p | 3 | 0 | 12 | 3 | 44 |  |
| ps6 | power sum | 3 | 6p | 3 | 0 | 12 | 3 | 44 |  |
| geo1 | geo series | 3 | e | 2 | 0 | 16 | 3 | 60 |  |
| geo2 | geo series | 3 | e | 2 | 0 | 20 | 3 | 64 |  |
| geo3 | geo series | 3 | e | 2 | 0 | 24 | 3 | 68 |  |
| Total |  | 146 |  | 89 | 27 | 1484 | 105 | 4082 |  |

Tab. 1. Experimental results

program列表示程序的名称；#V列表示程序中变量的个数；#D列表示程序精确不变式关于循环迭代次数的最高Degree，p表示Polynomial，其前面的常数表示多项式的最高次数，e表示Exponential，默认指数类型比多项式的Degree高；#H表示对变量的分层Hierarchy数量；Standard AI列是使用经典的抽象解释（基于Polyhedron和带加宽的Kleene迭代方式）分析的结果；Relaxing&AI列使用了本文设计的Relaxing策略并结合Kleene迭代和通项公式求解的方法分析得到的结果；#B列表示分析结果中循环头的不变式中有界变量的个数；T/ms列表示分析时间，单位是ms，P列表示循环不变式精度比较。

在本文的框架中，基于0层变量，通过采用Interval、Polyhedron抽象域，并结合Kleene迭代、通项公式等求解不动点的方式，以获得更加精确的不变式信息，为接下来的Relaxing提供支持，从而能够提升循环中高层变量不变式精度，也即变量分层是本文框架的基础。从上表结果的#Hierarchy列可以看出，总计38个程序中，有32个程序可对循环中变量进行分层（至少可以分为两层），6个程序不能分层，能够分层的程序比例为84.2%。从这一结果来看，在大部分程序中，循环中只有部分变量（即0层变量）控制循环的迭代次数，而程序实现的数值功能主要是在0层变量的基础上通过其他高层变量来实现的。这一结论对于本文所采用的基于分层提高循环分析精度的思路提供很好的实践支撑，同时对于循环分析领域的其他问题，如分析循环最大迭代次数、算法复杂度、WCET以及算法终止性分析等（这些分析都只依赖0层变量）都能够提供一个剔除无关变量，降低分析开销的通用方法。

关于循环分析精度方面，从结果可以看出，对于这些包含非线性不变式的程序，经典的抽象解释分析受限于Polyhedron抽象域的线性表达能力以及析取、加宽等操作带来的精度损失，很容易导致变量的值范围被近似到无穷边界，这也是溢出等运行时错误检查结果中高误报率的主要来源。本文的方法通过对语义方程的Relaxing，使得原本线性抽象域Polyedron和Interval无法表示的非线性不变式，能够被一个足够大的多面体限定住，从而得到变量的有穷边界，能够消除由于无穷边界而导致误报。通过对比本文方法的#Bounded列和程序中变量总数#Var列，38个程序中，基于Relaxing的方法相比于经典的方法能够增加27个程序的bounded variables的数量，约占总数的71%，其余11个程序bounded variables个数相同。对于这11个程序，本文通过变量区间的大小进一步分析了循环不变式的精度，如P列所示。其中4个程序提高了变量精度，5个精度一样（其中4个程序的参数实例化以后实质上是线性程序，即#Degree 为1p），2个降低了精度（这是因为Polyhedron刚好能够得到精确的循环不变式，分层并Relaxing后反而使得精确的不变式在新的语义方程中不成立）。变量分层为Relaxing提供了清晰的脉络，但在变量不能分层时，本文在实现时也会考虑基于同层变量已经获得的部分变量的不变式来Relaxing其余变量的赋值语句中的表达式，同样也能提高不变式精度，如上述6个不能分层的程序，本文方法能够增加4个程序的Bounded变量的个数。38个程序中变量的总数有146个，经典AI能够获得27个变量的有穷边界，约占总数的18.5%，而本文方法能够获得105个变量的有穷边界，约占总数的71.9%，本文方法分析得到的bounded variables的个数是经典方法的3.89X，即分析精度提升明显。

在时间开销方面，由于本文方法进行了多遍构建语义方程以及不动点迭代，故相对于经典方法的一遍分析时间开销要大。本文方法分析38个程序总共耗时4082ms，平均耗时107.4ms，而经典AI总共耗时1484ms，平均耗时39.1ms，本文时间开销是经典AI的2.75X，可见本文方法的分析效率相比于标准AI并没有降低太多。

1. **相关工作**

Antoine Miné在文[14]中提出了两种符号化的方法提高抽象解释得精度，即“linearization”和“symbolic constant propagation”，其主要思想是on-the-fly的利用上一步迭代的结果来简化当前语句迁移函数的表达式。本文Relaxing的思想和其符号化的思想有很多相似的地方，本文也在多处借鉴了其思想。但其方法主要作用在于通过符号化利用局部的信息提高单条语句分析的精度，对于减少循环迭代过程中的加宽导致的精度损失帮助不大，也无法得到非线性循环不变式的有穷边界。本文方法得到的部分不变式是具体语义下循环迭代过程中始终会满足的约束，利用这一全局信息，可以对循环中的所有语句进行更加有针对性的可靠Relaxing，从而可以使得Relaxed迁移函数能够较好用线性抽象域表示，避免加宽带来的精度损失。

Rahul Sharma 等人在[24]通过假定循环不变式关于一组参数满足特定的形式，再通过多遍测试分析获得参数满足的约束，进而Guess得到candidate invariant，再对该invariant进行check是否满足不变式的条件，不满足则继续通过Guess获得新的invariant，直到获得一个有效的invariant。但该方法通常只用来生成Polynomial invariant（具体而言更擅长等式关系的不变式），并需要提前假定不变式的Degree。此外还有很多工作通过Gröbner bases[23]、Abstrct Interpretation[30]、interpolating theorem[31]、Counterexample-guided approach[29]等方法获得非线性的不变式，但本文并不是直接获得非线性不变式，而是通过基于Polyhedron通过线性不变式来近似的可靠表达非线性循环的具体行为。

Bertrand Jeannet等人[32]采用Abstract acceleration来替换基于Kleene和加宽的迭代策略，将循环体中的赋值语句用一个矩阵来刻画，并假设循环迭代次数为n，将循环迭代n次的不变式转换成求矩阵的n次幂，再抽象成多面体抽象域能够表达的约束。该方法也能够用多面体抽象域刻画非线性循环不变式的部分边界，但存在很多限制，一是循环中语句必须是LRA的；二是对于循环中的分支，需要构建多个矩阵来分别计算，存在组合爆炸的问题，故还是不可避免的需要通过加宽提高效率。Colas Le Guernic[33]在它的基础上通过改进部分算法提高了其精度，但并不能避免上述问题。本文工作通过对语义方程进行抽象，能处理一部分非线性赋值语句，也能处理循环中的多个分支，能够在精度和效率间取得较好的平衡。

Gulavani[34] present a framework for extending an abstract domain with的两个提升操作，即表达式抽象和max 操作，利用non-linear axioms (可以是 log, multiplication, square root, and exponentiation)来提升Polyhedron抽象域获得的循环不变式，从而能够分别处理复杂度的界分析中的非线性不变式和带析取的界。Zachary Kincaid 等人[6]提出了一个wedge abstract domain来获取程序中的非线性不变式，wedge treat non-polynomial terms as independent dimensions， 它也是通过polyhedron抽象域的线性约束来刻画non-polynomial terms的线性组合。上述两种方法都是在现有的polyhedron的基础上加入非线性元素来表达非线性不变式，故其产生的非线性不变式的精度很大程度上依赖于Polyhedron抽象域推导出的线性不变式的的精度，而本文的方法就致力于通过Relaxing得到一个polyhedron表示的更精确的线性不不变式，对于提升Gulavani和Zachary的方法精度也有帮助。

为了获得循环的指数型不变式，Isabella等[7]提出了分析代数幂的抽象域,主要用于随机化程序的静态分析和因式分解.它能够处理本文提出的含有单变量区间系数的赋值语句的循环,但是它的计算效率比本文差许多。

1. **总结与展望**

为了提高抽象解释对于包含非线性不变式的循环分析精度，本文提出基于程序部分不变式的语义方程Relaxing思想，而部分不变式获取则是来自于对循环序列中的只包含低层变量的循环进行分析的结果。为此，本文首先对循环中的变量进行依赖分析，构建了变量依赖的Hasse图并对变量分层，进一步基于变量的层次关系对循环进行切片获得一个循环序列； 然后介绍Relaxing的定义和可靠性，并提出了一些Relaxing策略；最后为了避免加宽带来精度损失，本文提出了基于通项公式的不动点求解方法。初步实验结果表明，很多程序的循环体中的变量都和循环迭代次数无关，变量与变量之间存在单向依赖关系。基于部分不变式的Relaxing策略，可以将循环迭代过程中的非稳定变量通过替换，使其变成一个稳定的区间变量，从而提高了加宽的精度，能够获得更加精确的循环不变式。

本文所枚举Relaxing策略只是众多策略中的几种，还有更多的可能更有效的Relaxing策略等待发掘。基于部分不变式的Relaxing是一个非常General的对语义方程进行转化思想，其应用场景不只局限于本文提到的获得变量的有穷边界，也不只局限于抽象解释框架，应该可以视为一个程序静态分析过程中的一个优化操作，应用于多种程序分析和验证技术。这也是下一步有待研究的方向。