第 5 章 符 号 执 行

符号执行技术在 1976 年由 Jame C. King 提出[1]。20 世纪 70 年代，关于软件正确性测试的研究工作都基于一个原则：选择合适的测试用例对程序运行状态进行测试，如果对于提供的输入都能产生正常的结果输出，则认为程序是可靠的。其中的方法可分为两大类。一类是以模糊测试为代表的随机性测试，虽然模糊测试等随机测试方法至今仍活跃在软件安全测试的一线，但其具有的盲目性和随机性使其无法提供完整可靠的测试结果。另一类是以模型检测为代表的形式化证明方法，通过归纳法来证明程序是否具有期望的性质，证明过程的复杂性使其在面对大规模程序的时候几乎不可用。正是在这样的背景下，James C. King 提出了符号执行方法，可以将其看成是上述两类传统方法的折中。King 希望在无法获取程序特性说明等信息的情况下，仍旧能够对其进行快速全面的自动化安全性检测。本章将对符号执行的基本方法进行介绍。

5.1 符号执行基本模型

5.1.1 基本思想

符号执行的基本思想是：使用符号变量代替具体值作为程序或函数的参数，并模拟执行程序中的指令，各指令的操作都基于符号变量进行，其中操作数的值由符号和常量组成的表达式来表示。

对于任意程序，其执行流程是由指令序列的执行语义控制的，执行语义包括：变量定义语句对数据对象的描述，声明语句对程序数据对象的修改，条件语句对程序执行流程的控制。当程序的输入参数确定时，其指令序列被固定下来，因此程序执行语义和控制流也就得到确定。如果不用具体数值，而是用符号值作为程序的输入参数，则指令序列的操作对象就从具体数值变为了符号值，程序的执行语义和控制流程也变成了和符号变量相关的符号表达式。读者可以将符号执行视为程序具体执行的自然扩展，符号变量使得程序执行语义变得不确定，这也使得符号执行技术在理想情况下可以遍历程序执行树的所有路径。也可以将程序的一次具体执行视为符号执行的一个实例，当需要对某条程序路径进行遍历分析时，只需根据符号执行方法对该路径的分析结果，就可以引导控制流遍历该路径的程序输入。

King[1] 在提出符号执行技术的同时，也为其限定了理想的使用场景：

(1) 理想模型中程序只处理有符号整数，在实际测试中这种情况不会出现。

(2) 理想模型中假定程序“执行树”的规模是有限的,在实际测试中,由于程序中存在的循环等原因,很多程序的“符号执行树”可能是无穷大的。

(3) 理想模型中符号执行技术可以处理程序内所有 if 条件语句中的约束表达式,在实际测试中,约束表达式中通常会出现符号执行引擎无法处理的操作和变量类型。

5.1.2 程序语言定义

基于符号执行技术的理想场景对程序语言做如下定义。

(1) 程序变量类型:程序中只包括有符号整数类型。

(2) 程序语句类型:

- 简单的声明语句,例如,a=3。
- if 条件语句(包括 then 和 else),例如 if(a<0),假定程序内所有 if 条件语句中的表达式都可以化简为{arith. expr. }≥0 的形式,例如-a-1≥0。
- 无条件跳转语句,例如 goto 语句。
- 变量操作语句,例如读操作(read)。变量处理操作符中只包含基本的整数运算操作,例如加、减、乘(+、-、*)。

5.1.3 符号执行中的程序语义

虽然程序语义因为符号变量的加入而发生变化,但无论是程序语法还是程序语句的操作流程都不会因为符号变量的存在而发生变化,这就保证了符号执行技术的有效性。下面首先介绍程序执行语义在符号执行模式下产生的变化。

1. 符号数据对象

为了简化描述,假设每次程序需要新的输入时,都从符号列表{a1,a2,a3,…}中选取,程序输入参数中的符号变量通过变量声明、数学运算操作等方式传递至程序中的变量。在 King[1] 设计的理想模型中,程序使用的每个符号变量都应该是一个有符号整数。

2. 程序语句

1) 变量操作语句

(1) 数学运算符。

程序具体执行时,数学运算操作可以描述成操作符、圆括号和整数变量构成的多项表达式,例如,a=1+(2*4)。符号执行模式下同样可进行类似的描述,只需要将表达式中的整数替换成符号值集合即可,例如,a=a1+(a2*a3)。可以看到,符号值的引入并没有修改数学运算符的操作语义,只是运算结果由整数多项式变成了符号多项式。

(2) 数据读写操作。

以读操作 read(addr)为例,当 addr 中的值为具体值时,read(addr)返回具体值;如果 addr 中的值为符号值,则返回符号值。所以符号执行同样没有影响读写操作的语义。

2) 无条件跳转语句

goto 语句也称为无条件跳转语句,其一般书写格式为

```
goto 语句标号
```

其中语句标号是按程序标识符规范书写的符号，放在某一语句行的前面，标号后需要添加冒号(:)，语句标号起标识语句的作用，与 goto 语句配合使用。示例如下，其中 loop 就是语句标号：

```
goto loop:
    …
loop: while(x<7);
```

符号执行模式中 goto 语句与具体执行中的语义是完全一致的。

3) 声明语句及条件跳转语句

具体执行时，无论是条件跳转语句还是声明语句，语句内表达式的取值都是具体值。例如 if(a<0)，表达式 a<0 的真值在语句执行完成后就可以计算出来，并根据真值决定条件分支的跳转，因为 a 的取值是已知的。符号执行时，if 语句的语义没有变化，同样会根据 a<0 的计算结果进行跳转，但此时 a 为符号变量，无法计算出 a<0 的真值，该如何决定条件分支的走向呢？这就是符号执行技术对程序执行语义的最大改变，也是符号执行和具体执行的关键区别：符号变量的引入使得程序执行到路径分支时无法确定程序的走向。

3. 程序执行状态

具体执行时，程序状态中通常包括程序变量的具体值、程序指令计数等描述信息，使用这些信息就可以描述程序执行的控制流向。因为符号变量的引入导致分支走向不确定，仅凭原有的信息已经无法完整描述符号执行的状态，King 为程序状态新添加了一个变量：路径约束条件，下面都用 pc(path constraint)来表示。

简单地说，pc 就是符号执行过程中对路径上条件分支走向的选择情况，根据状态中的 pc 变量就可以确定一次符号执行的完整路径。如前文所述，符号执行过程中，在每个 if 条件语句处并没有实际值决定程序执行哪条分支，这就需要符号执行引擎主动选择执行分支并记录整个执行过程，pc 就辅助完成了这项工作。举例来说，假设符号执行过程中经过 3 个与符号变量相关的 if 条件语句 if_1、if_2、if_3，每个条件语句处的表达式如下所示：

$$if_1: a_1 \geqslant 0$$
$$if_2: a_1 + 2 * a_2 \geqslant 0$$
$$if_3: a_3 \geqslant 0$$

假设引擎在 3 个 if 条件分支处分别选择的是 if_1：true，if_2：true，if_3：false，则 pc 表示为

$$pc = (a_1 \geqslant 0 \wedge a_1 + 2 * a_2 \geqslant 0 \wedge \neg(a_3 \geqslant 0))$$

如上面所示，pc 是一个 bool 表达式，表达式由符号执行路径上涉及的 if 条件语句中的表达式及表达式的真值选择拼接而成。假设 if_n 处的表达式为 $R \geqslant 0$，R 是一个与符号变量相关的多项表达式，把 $R \geqslant 0$ 称为 q，则程序执行到 if_n 处时 pc 可能会表现为下面两种形式之一：

(1) pc 包含 q。

(2) pc 包含 ¬q。

如果符号执行引擎选择进入 then 分支，则 R≥0 的真值为 true，pc 表现为(1)的形式；如果选择 else 分支，则 R≥0 的真值为 false，pc 表现为(2)的形式。需要注意的是，pc 的初始值为 true。

在程序逻辑中，程序设计人员当然希望一个 if 分支的 then 和 else 分支都能够被执行到，所以当执行到 if 条件语句处时，符号执行需要创建两个“并行”的执行过程：一个进入 if 语句的 then 分支，生成 then 分支对应的 pc；另一个进入 else 分支，同样生成对应的 pc。两个符号执行过程在 if 分支之后相互独立，拥有各自的执行状态，介绍到这里读者可以明白，符号执行过程中产生的分支只和 if 条件语句相关，与其他的程序执行状态无关，如果只是执行普通的程序声明语句或者运算指令引擎，不会产生分支。

当选择 then 分支的时候，假设输入变量是满足 q 的，这个过程可以用表达描述为 pc＝pc∧q，类似的，当选择 else 分支时可以描述为 pc＝pc∧¬q，pc 之所以被称为条件路径就是因为根据其内容就可以确定一条唯一的程序执行路径。每个和符号变量相关的 if 条件语句都会为 pc 贡献一个决定程序执行走向的表达式。pc 的真值恒为 true，当 pc 的表达式为 pc＝pc∧q 时，要确定 pc 对应路径的程序输入参数，只需要使用约束求解器对 pc 进行求解。

5.1.4 符号执行树

执行树是用来描述程序执行路径的树形结构。执行树中的一个节点对应程序中的一条语句，程序语句之间的执行顺序或跳转关系对应执行树中节点间的边。对于每个 if 语句会有两条边与其相连，左子树对应的是 if 语句的 true(then)分支，右子树对应 if 语句的 false(else)分支。执行树中还可以包含指令计数、pc(路径约束条件)、变量值等程序执行状态信息。一个函数与其执行树的对应关系如图 5-1 所示。

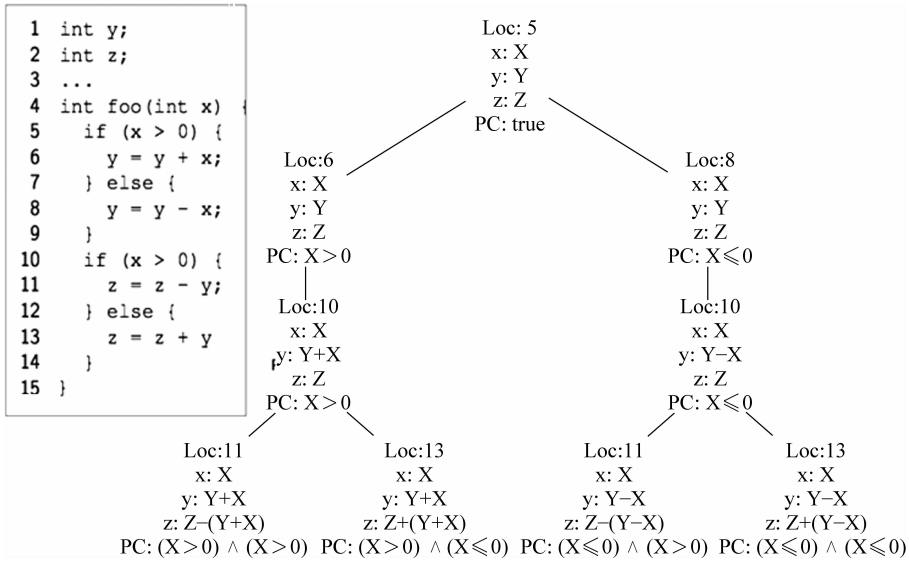

图 5-1 符号执行树实例

执行树描述了执行路径在各程序指令处的状态，且具有如下特性：

- 对于执行树中的每个叶节点，都对应一组具体输入值能够让程序执行到当前状态，即当被测试程序在设计和编码都没有出现错误的情况下，每个叶节点上对应的 pc 表达式都应是恒真的，pc 中的所有符号变量一定可以求得一组解使 pc 为真，这组解就是指导程序执行到该叶节点对应语句处的实际输入值。如果在测试中出现某个叶节点上 pc 表达式无解的情况，说明该路径是存在逻辑问题的，该叶节点对应的路径不可达。
- 执行树中任何两个叶节点上的执行状态都是有区别的，因为任意两个叶节点对应的执行路径都是从 root 节点起始的，并在执行树的某个节点处分支成为两个不同的路径，一条路径选择了该节点的 true(then)分支，另一个进入了 false(else)分支，所以两个路径的最终状态必然不相同。在图 5-2 的示例代码中，虽然程序中有循环，但当初始值不同时，程序的执行路径完全不相同，不会因为循环而产生重合的情况。执行路径的唯一性使得测试过程中不会产生冗余的用例。假设图 5-2 中的输入用例为 a_1，圆圈中数字代表对应的代码行数。

```
1    TWOLOOPS:PROCEDURE(N);
2              DO J=1 TO N;
3                  (body of statements)END;
4              DO K=1 TO N;
5                  (body of statements)END;
6              END;
```

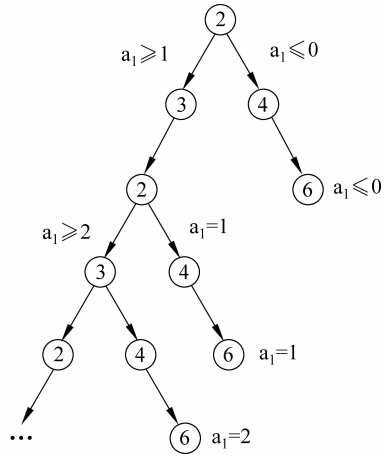

图 5-2　循环程序执行树实例

5.1.5　约束求解

通过 5.1.4 节的介绍可以知道，每个叶节点对应的执行路径可以由一组具体输入指导程序运行得到，而这组具体值正是借助约束求解器对 pc 求解得到的。符号执行过程中，执行树中每个叶节点对应的 pc 会被输入约束求解器进行求解，如果 pc 表达式有解，则求解器会输出满足 pc 的一组符号变量的具体值，如果无解则说明该叶节点对应的执行路径是不可达的。

1. 约束求解问题

关于约束满足问题求解(CSP)的研究最早是由 Montanari 在 1974 年发起的[15]。最初它被用于描述图像处理中的一些问题，随后，便作为一种通用模型被广泛用于各类理论和实际问题的研究中[14]。

约束求解问题可以形式化表示为一个三元组$<V,D,C>$，其中的3个要素分别为：变量V、变量的论域D和约束条件C。变量V是变量的有限集合，表示为$V=\{v_1,v_2,\cdots,v_n\}$；变量的论域D是变量可能取值的有限集合，变量v_i只能在它的值域即论域D_i中取值；约束C是一个有限约束集合，某个约束关系C_i包含V中一个或多个变量，若C_i包含k个变量，则称C_i为在这k个变量集合上的k元约束[14]。

约束求解就是找到约束问题的一个解，该解对变量集合中所有变量都赋一个取自其论域的值，并且这些变量的取值满足该问题所有的约束条件。对于约束问题$P=(V,D,C)$，若P至少存在一个解，则称P是可满足的，否则，称P为不可满足的。在符号执行技术中，约束求解器被用来求解pc的可满足问题[14]。现在主流的约束求解器主要基于两种理论模型：SAT和SMT。

1) SAT问题[16]

SAT问题(The Satisfiability problem，可满足性问题)，是指求解由布尔变量集合所构成的布尔函数，是否存在变量的一种分布使得该函数的取值为1。举例来说，假设布尔函数为$\Phi=(\alpha\vee\neg\beta)\wedge(\beta\vee\gamma)\wedge(\gamma\vee\neg\alpha)$，其中$\alpha$、$\beta$、$\gamma$是布尔变量，求使得$\Phi$值为1时的$\alpha$、$\beta$、$\gamma$的取值分布。在此例中，当$(\alpha,\beta,\gamma)$的取值分布为(1,0,1)时可以满足因此$\Phi=1$，因此，该问题是布尔可满足的(satisfiable)。当不存在任何一种分布满足$\Phi=1$时，称该问题是布尔不可满足的。SAT问题是计算机科学领域中非常重要的一项研究，在人工智能(Artificial Intelligence，AI)、软件设计、形式化验证以及硬件设计方面，如集成电路验证、组合电路等价性验证等，都有着重要的应用。

但是SAT求解只能解决命题逻辑公式问题，使得许多实际应用问题无法直接转换为SAT问题来求解。并且在SAT问题中必须使用布尔变量来表示，要把实际应用中对应的逻辑关系转换为布尔函数，转换开销很大，转换后的布尔函数结构也非常复杂，导致最后的求解过程可能无法完成。SAT求解的种种不足限制了其应用范围，因此后续的研究提出了SMT理论。

2) SMT问题[16]

SMT(Satisfiability Modulo Theories，可满足性模理论)，是在可满足性问题(SAT)的基础上扩展而来的，它将SAT求解从只能解决命题逻辑公式扩展为可以解决一阶逻辑所表达的公式。SMT包含有多种理论，如定长位向量理论(fixed-size bit-vector)、数组(array)、未定义函数(uninterpreted function)等，通过组合使用这些基本理论，SMT在硬件验证、定理证明以及本书提到的约束求解和自动化测试用例生成等领域都得到了广泛的应用。

近年来，对SMT的研究和应用得到了很大的发展，许多高校和科技企业开发出越来越高效的SMT求解器，如麻省理工学院的STP求解器、林茨大学的Boolector求解器以及微软研究院的Z3求解器。表5-1给出了当前主要的SMT求解器及其支持的SMT求解理论。

表 5-1 SMT 求解器[16]

SMT 求解器	支持的操作系统	支持的求解理论
ABsolver	Linux	线性计算、非线性计算
Beaver	Linux/Windows	位向量
Boolector	Linux	位向量、数组
CVC4	Linux/Mac OS	线性计算、数组、位向量、有理数与整数、元组、数组
MathSAT	Linux	空理论、线性计算、位向量、数组
MiniSmt	Linux	非线性计算
OpenSMT	Linux/Mac OS/Windows	空理论、线性计算、位向量
SMT-RAT	Linux/Mac OS	线性计算、非线性计算
STP	Linux/OpenBSD/Windows/Mac OS	位向量、数组
UCLID	Linux	空理论、线性计算、位向量
Yices	Linux/Windows/Mac OS	
Z3	Linux/Windows/Mac OS/FreeBSD	空理论、线性计算、非线性计算、位向量、数组、量化

2. Z3 求解器

在大量 SMT 求解器当中，最出众的莫过于由微软研究院 Leonardo de Moura 主持设计的 Z3 求解器，其被设计作为其他应用程序的底层工具，在大量和定理证明、程序测试的项目中都得到应用，包括 Spec♯、Boogie、Pex、Yogi、Vigilante、SLAM、SAGE、VS3 等。Z3 致力于解决软件验证和软件分析中的问题，它为大量的理论提供了支持，使用全新的算法进行量词实例化和理论合并，在 2007—2011 年的各项大赛中取得了优异成绩。相比于 Yices、STP 等求解器，Z3 不仅性能卓越，其提供的 API 也更加简洁，所以成为大多数符号执行工具的首选。

Z3 是用 C++ 实现的，其可以使用多种编程语言来描述所要求解的问题，如 C 语言格式、Python 语言格式、SMT-LIB 格式、. NET 语言格式、Simplify 格式等。Z3 的结构如图 5-3 所示。

- 化简模块(Simplifier)。Z3 求解器首先会对表达式进行化简处理，这一步骤不要求完善，但要求高效。化简模块使用的是标准代数化简原则，如线性变换、变量数值化等，如 $p \wedge \text{true} \mapsto p$。同时也会对条件表达式做一些有限度的文本简化，如将表达式中的符号用数值来代替：$x=4 \wedge q(x) \mapsto x=4 \wedge q(4)$。
- 编译模块(Compiler)。将经过初步简化处理的表达式转换为特定的语法树和数据结构。
- 核心模块(Congruence Closure Core)。调用理论求解器(Theory Solver)和 SAT 求解器处理经过编译的表达式，并实现了两个求解模块的数据共享。
- 理论求解模块(Theory Solver)。其中包括了 SMT 求解器常用的基本理论，线性

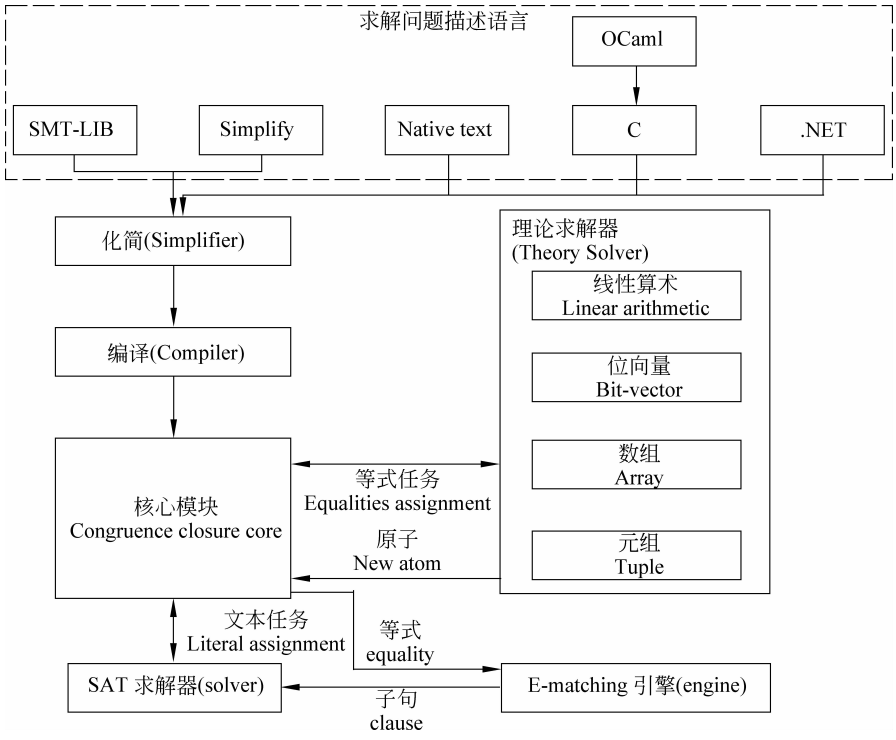

图 5-3 Z3 求解器的结构

算术模块是基于 Yices 中使用的算法实现的，其他的如数组模块、位向量模块等也都是基于经典算法实现的。

- SAT 求解模块：该模块使用的是经典 SAT 求解技术。

这里以 Z3 提供的 Python 接口为例说明其使用方法。

```
>>>a=Int('a')
>>>solve(a>0, a<2)
[a=1]
```

上面代码片段中的 Int('a')函数创建了一个整数变量，并将该变量命名为 a。solve 函数对括号中的约束条件集合进行求解。Z3 提供的 Python 接口允许用户使用操作符＜、＞、＝＝来描述表达式间的关系。默认情况下，solve 函数中的约束条件是逻辑与的关系。Z3 根据变量类型和约束条件集合对变量进行求解，上例中求解得到 a＝1。

5.1.6 符号执行实例

前面已经介绍了符号执行技术的基本原理和方法，下面用实例说明其实际执行过程，首先通过一个简单的实例说明符号执行与具体执行的区别。

```
1  SUM: PROCEDURE(A, B, C);
2       X←A+B;
3       Y←B+C;
```

```
4       Z←X+Y-B;
5       RETUREN(Z);
6   END;
```

上面是用类 PL/1 语言的语法编写的一个计算三数之和的代码，代码中对每条指令进行了编号，表 5-2 中的编号都是与指令编号相对应的。如果函数的初始输入为 1，3，5，则程序执行过程中各变量的变化如表 5-2 所示，程序输出为 9。

表 5-2　SUM 函数在程序执行过程中各变量的变化

指令行数	X	Y	Z	A	B	C
1	?	?	?	1	3	5
2	4	?	?	1	3	5
3	4	8	?	1	3	5
4	4	8	9	1	3	5
5	（返回 9）					

现在用 3 个符号来表示 A、B、C 3 个整数输入，符号执行过程中的变量变化如下（表 5-3）：

表 5-3　SUM 函数在符号执行过程中各变量的变化

指令行数	X	Y	Z	A	B	C	pc
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1+\alpha_2$	--	--	--	--	--	--
3	--	$\alpha_2+\alpha_3$	--	--	--	--	--
4	--	--	$\alpha_1+\alpha_2+\alpha_3$	--	--	--	--
5	（返回 $\alpha_1+\alpha_2+\alpha_3$）						

（1）第一条语句是函数的入口，符号执行引擎将 3 个输入参数符号化为 α_1、α_2、α_3，同时将 pc 初始化为 true。

（2）第二条语句为 X＝A＋B，使用符号进行数学运算，并将符号表达式赋予变量 X，X＝$\alpha_1+\alpha_2$。

（3）第三条语句为 Y＝B＋C，使用符号进行数学运算，并将符号表达式赋予变量 Y，Y＝$\alpha_1+\alpha_3$。

（4）第四条语句为 Z＝X＋Y－B，将各变量的符号值带入运算得 Z＝$\alpha_1+\alpha_2+\alpha_3$。

（5）函数将符号表达式 $\alpha_1+\alpha_2+\alpha_3$ 作为返回值。

上面的简单例子已经说明了符号执行与实际执行中程序变量的区别。下面再用一个实例说明符号执行的完整流程及可能遇到的问题。

```
1   POWER: PROCEDURE(X, Y);
2         Z ← 1;
3         J ← 1;
```

```
4  LAB:   IF Y≥J THEN
5           DO; Z ← Z * X;
6              J ← J+1;
7              GO TO LAB; END;
8         RETURN(Z);
9  END;
```

对上面示例中 POWER 函数的符号执行过程说明如下。在第 4 行代码中遇到 IF 条件语句，约束条件 Y≥J 转换成符号表达式就是 $\alpha_2 \geqslant 1$，符号执行引擎会分别探索分支的两条路径，选择 true 分支的会在路径条件 pc 中添加 $\alpha_2 \geqslant 1$，相反，选择 false 分支的会在 pc 中添加 $\neg(\alpha_2 \geqslant 1)$。

选择 false 分支的探索过程会在第 8 行代码处结束，如果需要构造执行 false 分支的 case，只需要对 pc 进行求解即可，例如一组解为 X=0，Y=0。选择 true 分支的探索会进入循环结构，在执行完第 5～7 行代码后，程序控制流又回到第 4 行的分支处，和上面的操作相同，符号执行引擎再次添加两条探索路径。

上面的例子在符号执行过程中各变量的变化如表 5-4 所示。

程序中有循环的情况很普遍，但对于符号执行来说，循环语句就不太友善了。对于本例，因为条件语句中的符号变量 α_2 并不受其他约束条件的控制，所以 IF 条件语句的 true 分支可以无限探索下去，即符号执行引擎是无法正常终止的。循环问题是符号执行技术中的重要问题，本书会在后面的章节中进行详细介绍。

表 5-4 POWER 函数符号执行过程中各变量的变化

指令行数	J	X	Y	Z	pc
1	?	α_1	α_2	?	true
2	--	--	--	1	--
3	1	--	--	--	--
4	执行过程： ① 处理判断语句 Y≥J 得到约束条件 $\alpha_2 \geqslant 1$。 ② 生成两个分支的路径约束条件： • true⊃$\alpha_2 \geqslant 1$ • true⊃$\neg(\alpha_2 \geqslant 1)$ ③ 两个路径约束都可满足，分别对两个路径进行探索。				
分支 $\neg(\alpha_2 \geqslant 1)$：					
4	1	α_1	α_2	1	$\neg(\alpha_2 \geqslant 1)$
8	探索完成(returns 1 when $\alpha_2 < 1$)				
分支 $\alpha_2 \geqslant 1$：					
4	1	α_1	α_2	1	$\alpha_2 \geqslant 1$
5	--	--	--	α_1	--