软件分析技术课程讲义

熊英飞
北京大学

2024 年 12 月 29 日

目录

第一章　总览

1.1　引言

1.1.1　缺陷检测问题

为尽量减少软件中的缺陷，我们希望自动回答如下问题。

定义 1 (缺陷检测问题). *针对给定的缺陷类型，输入程序P，求程序P中是否存在给定类型的缺陷。*

比如，一个内存泄漏缺陷检测问题希望检测出程序中是否存在内存泄漏。这部分我们以该问题为例展现软件分析的困难。

1.1.2　哥德尔不完备定理

1931年提出的哥德尔不完备定理是二十世纪最重要的发现之一，他否定了希尔伯特计划，也间接展示了上述缺陷检测问题不可能自动回答。

哥德尔不完备定理包括两个主要定理。这里我们主要需要用到的是第一不完备定理。是指包含自然数和基本算术运算（如四则运算）的一致系统一定不完备，即包含一个无法证明或证伪的定理。这里一致性是指任意命题和其逆命题不能同时被证明。完备性是指对所有命题，该命题本身或其逆命题一定能被证明。

主流程序设计语言能表示自然数和基本运算，即落在哥德尔不完备定理的范围。注意这里说的主流程序设计语言可以表示自然数并不是指程序设计语言中有Int这样的数据类型，因为Int是有限的，而数学上自然数是通过皮亚诺算数公理定义的。这句话主要是指主流程序设计语言可以通过List等类型定义出一个无限递归的结构，或者熟悉函数式程序设计语言的同学可以很容易用代数数据类型定义出皮亚诺算数公理的自然数类型。

那么根据哥德尔不完备定理，如果我们用的形式系统是一致的，一定存在某定理T不能被证明。那么我们可以构造如下程序。

```
a=malloc();
if (T) free(a);
return;
```

如果T为永真式，则没有内存泄漏，否则就有。但由于T无法被证明，所以我们并不知道T是否为永真式。

哥德尔不完备定理的证明概要可以查阅课程胶片或者网上的一些科普资料[18]。

1.1.3 停机问题

哥德尔不完备定理的完整证明较为复杂，软件分析的难度也可以从更简单的停机问题上来理解。

停机问题是指判断一个程序在给定输入上是否会终止，图灵1936年证明不存在一个算法能回答停机问题的所有实例。

图灵的证明是反证法，通过现代编程语言的语法可以很容易地理解。注意判断一个程序在给定输入上是否会终止等价于判断一个没有输入的程序是否会终止，只要我们将输入定义为程序中常量就行。简单起见，我们只考虑没有输入的程序。假设存在停机问题的判定算法Halt(p)，对于终止的程序p返回true，对于不终止的程序返回false。那么我们可以写出如下邪恶程序。

```
void Evil() {
  if (!Halt(Evil)) return;
  else while(1);
}
```

那么，Halt(Evil)的返回值是什么呢？容易看出，我们没法定义出该返回值。假设返回值为真，也就是判断Evil应该停机，但实际运行Evil会进入while(1)死循环，推出矛盾。假设返回值为假，也就是判断Evil应该不停机，但实际运行Evil会立刻返回，推出矛盾。因此，不存在这样的判定算法Halt。

套用以上证明过程，我们可以很容易证明很多缺陷检测问题也不存在算法能回答。比如我们可以假设存在判定内存泄漏的算法LeakFree(p)，对

没有内存泄漏的程序返回true，对有内存泄漏的程序返回false。那么我们可以写出如下邪恶程序。

```
void Evil() {
    int a = malloc();
    if (LeakFree(Evil)) return;
    else free(a);
}
```

容易看出，无论LeakFree(Evil)返回什么值，都会推出矛盾。

可能有同学会有疑问，图灵的原始停机问题证明是构建在图灵机上的，但上面的证明似乎和图灵机没有关系？实际上，因为现代编程语言都可以用图灵机来模拟，我们可以把上面的证明过程都转到图灵机上。用图灵机模拟函数，就是先在纸带的某个位置写下一函数输入，并将图灵机头对准该位置，切换到某个预定义的函数初态开始执行，然后图灵机运行到某个预定义的终态表示函数执行结束，这时写在在纸带特定位置的内容就是函数输出。我们还可以进一步写一个解释器函数interpret，输入是表示程序的字符串，输出是程序执行结果。换句话说，Halt的作用是，对于任何合法程序的字符串p都能运行结束，并且输出为true表示interpret(p)运行一定能到终态，输出为false表示interpret(p)的运行不能终止。然后我们可以在纸带上写下上面Evil程序的字符串，注意这里把Evil传递给Halt实际是把字符串的起始位置传递给Halt。通过以上论证，可以知道Halt无法返回正确结果。

1.1.4 莱斯定理

套用停机问题的证明，我们可以发现很多软件分析问题都不可判定。

但到底有多少问题是不可判定的呢？1953年的莱斯定理表明，几乎所有有意义的程序分析问题都是不可判定的。

莱斯定理的表述为，给定一个被图灵机M识别的递归可枚举语言[1]，给定一个语言的属性P，如果P是非平凡的，那么该语言是否具有性质P是一个不可判定问题。一个性质P是平凡的，当且仅当要么该性质对所有的语言都为真，要么该性质对所有的语言都为假。

注意原始莱斯定理是定义在形式语言上的，可以看成是输入到布尔的函数。如何将莱斯定理应用到所有可计算的函数上呢？注意一个可计算的

[1]递归可枚举语言即可以被图灵机识别的语言。

函数用图灵机表示的时候，是首先在纸带上写下输入，然后图灵机运行到达终态，这时纸带上的内容就是输出。那么我们可以把输入和输出排列在一起写在纸带上。首先我们运行原始图灵机对输入进行处理，得到输出。然后我们再启动一个图灵机来比较图灵机和期望输出的等价性。把这两个图灵机组合到一起，我们就得到了一个新的图灵机接受该可计算函数的输入输出对，这样我们也就可以把函数的性质转换成递归可枚举语言的性质了。

换句话说，莱斯定理告诉我们，除了不需要检查的平凡属性，所有非平凡属性都是不可判定的。注意这里的属性是定义在可计算函数的属性，这里的函数应该理解为数学上的函数，即输入输出对的集合，而不是程序中一个带语法的函数。举例来说，“永远不会返回0”是一个函数的属性，但“函数的实现代码不超过10行”不是。

注意“程序运行过程中不会发生内存泄漏”这样的运行时性质也可以看成是可计算函数的属性。我们只需要假设程序每一次调用malloc和free的时候都在输出中产生一份日志记录，然后我们可以在这样的日志记录中取定义“内存泄漏”这样的属性。

1.1.5 模型检查

无论是哥德尔不完备定理、停机问题还是莱斯定理，讨论的都是涉及无穷的量的一些性质。比如，哥德尔不完备定理中涉及自然数，自然数的大小是无穷的。停机问题和莱斯定理都涉及图灵机，而图灵机包含一个无穷长的纸袋。但这些理论上的结果在实际中的计算机上是不存在的，因为实际计算机的内存是有限的，如果我们考虑计算机通过网络连接，网络上计算机的总数也是有限的，所以这些理论的结果并不适用实际的系统。

因为内存总大小是有限的，我们可以通过有限状态自动机来表示程序。这里“状态”指一个程序运行过程中某个时刻内存中所有位置的值。给定一个状态，程序的下一个状态是确定的。因此，我们可以得到一个图，图中的节点为程序的状态，而图中的边是状态到状态的转移。可以看出，这样一个图的节点数是有限的，不含环的路径长度和数量也是有限的。对于大量属性，只需要在这样的路径上做遍历就可以完成。比如，对于停机问题，我们检查从起始状态出发的路径是否有环。对于内存泄漏问题，我们检查从起始状态出发的路径在终止或者进入重复状态之前，是否有分配的内存没有释放。因为路径长度有限，这样的算法一定在有限时间内终止。

基于这样思想开发的技术称为模型检查。模型检查被广泛用于检查硬件系统的实现正确性。但是，由于软件的复杂性，虽然内存大小理论上是有限的，但实际状态数仍然是天文数字，采用模型检查的方法基本不可能在有限时间内完成检查。

1.2 软件分析

定义 2 (软件分析技术). *给定软件系统，回答关于系统行为的问题的技术，称为软件分析技术*

在很多情况下，软件分析指回答和系统行为无关的软件性质的问题，比如程序有多少行。但这类问题的回答不在本课程范围内，所以没有包括在软件分析技术的范围内。

在部分文献[5]中也严格软件分析问题和软件验证问题。其中软件分析返回程序符合的属性，比如：程序中有几处内存泄漏（按多少个malloc语句分配的内存有可能泄漏计算）？软件验证判断程序是否符合给定属性，比如：程序中是否只有不超过3处内存泄漏?

在本课程中我们不对这两种问题进行严格区分，统一将其称为软件分析问题。事实上这两类问题通常是可以互相转换的。比如，如果我们有软件分析工具，我们让该工具分析处内存泄漏的数量，那就可以回答对应的验证问题。如果我们有软件验证工具，我们可以通过二分查找判断出程序中内存泄漏的数量。

1.3 近似求解软件分析问题

根据引言部分的分析，软件分析问题无法精确求解。实际中通常采用近似解。对于判定问题，近似的方法就是对于一部分实例回答“不知道”。对于返回值为集合的问题，近似方法就是返回一个和原始集合相近的集合。

由于程序分析常常被用于编译优化中，所以保证优化的正确性是很重要的。因此，通常会对近似的正确性（soundness）提出要求。具体要求取决于应用，常见要求包括：

- **判定问题下近似**：只输出“是”或者“不知道”，即返回“是”的实例集合是真实实例集合的子集。

- **判定问题上近似**：只输出“否”或者“不知道”，即返回“不知道”的实例集合是真实实例集合的超集。
- **集合问题下近似**：返回的集合是实际集合的子集。
- **集合问题上近似**：返回的集合是实际集合的超集。

1.3.1 抽象法

抽象法通常对程序运行状态定义一个抽象域，同时将程序的执行过程重新定义在抽象域上。

下面以符号分析为例介绍抽象法。给定一个整数的四则运算表达式，我们想要避免计算出结果，但尽量分析结果的符号。

我们可以定义如下抽象域{正, 负, 零, 粿}。抽象域中四个值的含义通过下面γ函数定义。

$$
\begin{aligned}
\gamma(\text{正}) &= \{i \mid i > 0\} \\
\gamma(\text{负}) &= \{i \mid i < 0\} \\
\gamma(\text{零}) &= \{i \mid i = 0\} \\
\gamma(\text{粿}) &= \text{所有整数和NaN}
\end{aligned}
$$

为了采用该抽象域进行运算，我们首先定义抽象函数将普通整数映射到抽象域上。

$$
\beta(i) = \begin{cases} \text{正} & i > 0 \\ \text{负} & i < 0 \\ \text{零} & i = 0 \end{cases}
$$

然后针对$+, -, \times, /$等操作定义对应的抽象域上的运算$\oplus, \ominus, \otimes, \oslash$。

$$
a \oplus b = \begin{cases} \text{正} & a = \text{正}, b = \text{正} \\ \text{负} & a = \text{负}, b = \text{负} \\ \text{零} & a = \text{零}, b = \text{零} \\ \text{粿} & \text{其他情况} \end{cases}
\qquad
a \ominus b = \begin{cases} \text{正} & a = \text{正}, b = \text{负} \\ \text{负} & a = \text{负}, b = \text{正} \\ \text{零} & a = \text{零}, b = \text{零} \\ \text{粿} & \text{其他情况} \end{cases}
$$

$$
a \otimes b = \begin{cases} \text{正} & a, b \in \{\text{正}, \text{负}\}, a = b \\ \text{负} & a, b \in \{\text{正}, \text{负}\}, a \neq b \\ \text{零} & a, b \neq \text{粿} \wedge (a = \text{零} \vee b = \text{零}) \\ \text{粿} & \text{其他情况} \end{cases}
\qquad
a \oslash b = \begin{cases} \text{正} & a, b \in \{\text{正}, \text{负}\}, a = b \\ \text{负} & a, b \in \{\text{正}, \text{负}\}, a \neq b \\ \text{零} & a = \text{零} \wedge b \in \{\text{正}, \text{负}\} \\ \text{粿} & \text{其他情况} \end{cases}
$$

然后我们就可以在抽象域上进行运算。比如$5 \times 2 + 6$就可以运算为$\beta(5) \otimes \beta(2) \oplus \beta(6)$，从而在不计算出精确值的情况下计算出符号。注意由于我们做了抽象，所以有时我们会对正常的式子得出粿的结果，即结果不精确。

1.3.2 搜索法

搜索法通过遍历程序的输入空间，查看是否有触发缺陷的输入存在。比如测试就是一种典型的搜索法。搜索法会通过引入各种剪枝和推断算法来减少搜索空间。但由于输入空间可能无限大，同时单个输入可能不停机，所以搜索法只能对程序的部分执行进行分析。

实际分析中常常混合使用抽象法和搜索法。

第二章　数据流分析

在引言部分我们了解了如何在对一个表达式进行抽象分析。这一章我们看看如何在一个命令式程序上进行分析。相比表达式，命令式程序的基本单位是命令，并且任意程序是通过顺序、选择、循环三种方式组成命令构成。分析命令式程序的基本框架为数据流分析框架，主要如何对命令、顺序、选择、循环这些结构进行抽象。

简单起见，本章先不考虑指针、数组、结构体、函数调用、动态内存分配等高级编程语言成分，这些成分将在未来课程中处理。

数据流分析在控制流图上进行。我们这里假设程序已经被转换成了控制流图。转换成控制流图的算法可以参考龙书[1]等编译课本。

2.1　基础

定义 3 (半格)**.** 半格是一个二元组$(S, \sqcup)$，其中S是一个集合，$\sqcup : S \times S \to S$是一个合并运算，并且任意$x, y, z \in S$都满足下列条件：

- 幂等性*:* $x \sqcup x = x$
- 交换性*:* $x \sqcup y = y \sqcup x$
- 结合性：$(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$

引理 1. 有界半格$(S, \sqcup_S, \bot_S)$和$(T, \sqcup_T, \bot_T)$的笛卡尔乘积$(S \times T, \sqcup_{ST}, (\bot_S, \bot_T)$还是有界半格，其中$(s_1, t_1) \sqcup_{ST} (s_2, t_2) = (s_1 \sqcup_S s_2, t_1 \sqcup_T t_2)$

定义 4 (有界半格)**.** 有界半格是一个有最小元$\bot$的半格，满足$x \sqcup \bot = x$。

定义 5 (偏序)**.** 偏序是一个二元组$(S, \sqsubseteq)$，其中S是一个集合，$\sqsubseteq$是一个定义在S上的二元关系，并且满足如下性质：

- 自反性：$\forall a \in S, a \sqsubseteq a$
- 传递性：$\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq y$
- 非对称性：$x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

定理 1. 每个半格都定义了一个偏序关系：$x \sqsubseteq y$ 当且仅当 $x \sqcup y = y$

定义 6 (单调函数). 给定偏序关系$(S, \sqsubseteq_S)$和$(T, \sqsubseteq_T)$，称函数$f : S \to T$为单调函数，当且仅当对任意$a, b \in S$满足

$$a \sqsubseteq_S \Rightarrow f(a) \sqsubseteq_T f(b)$$

如果f接收多个参数，在讨论f的单调性的时候，把f看做是定义在各个输入参数域的笛卡尔乘积上的一个函数。

定义 7 (图). 图是一个二元组(V, E), 其中V是节点的集合，$E \subseteq V \times V$是边的集合。

定义 8 (控制流图). 给定一个程序，控制流图是一个图(V, E)，其中V是程序中基本块和条件表达式的集合，另外包含特殊节点$entry$和$exit$；E表示节点之间控制转移关系，满足：如果存在一条执行序列，执行完v_1之后立即执行v_2，那么$(v_1, v_2) \in E$。对于任意节点v，至少存在一条从$entry$到达该节点的路径，也存在一个从该节点到达$exit$的路径。

我们用$pred(v)$表示图中一个节点的前驱节点，用$succ(v)$表示图中一个节点的后继节点。

定义 9 (不动点). 给定一个函数$f : S \to S$，如果$f(x) = x$，则称x是f的一个不动点。

定理 2 (不动点定理). 给定高度有限的有界半格$(S, \sqcup, \bot)$和一个单调函数f，链$\bot, f(\bot), f^2(\bot), \ldots$必定在有限步之内收敛于$f$的最小不动点，即存在非负整数$n$，使得$f^n(\bot)$是$f$的最小不动点。

证明. 首先证明收敛于f的不动点。因为$\bot$是最小元，所以有$\bot \sqsubseteq f(\bot)$。

两边应用f，因为f是单调函数，得到$f(\bot) \sqsubseteq f^2(\bot)$。

再次应用f，得到$f^2(\bot) \sqsubseteq f^3(\bot)$。

因此，原命题中的链是一个递减链。因为格的高度有限，所以必然存在某个位置前后元素相等，即，到达不动点。

然后证明收敛于最小不动点。假设有另一不动点u，则$\bot \sqsubseteq u$。两边反复应用n次f，可得$f^n(\bot) \sqsubseteq f^n(u) = u$。因此，$f^n(\bot)$是最小不动点。 □

2.2 数据流分析框架

定义 10 (数据流分析问题). 给定如下输入：

- 控制流图(V, E)
- 有限高度的有界半格$(S, \sqcup, \bot)$
- 一个起始节点的输入值OUT_{entry}
- 一组单调函数，对任意$v \in V \backslash \{entry\}$存在一个单调函数$f_v$

数据流分析问题的目标是产生下列输出：

- 对任意$v \in V \backslash \{entry\}$，输出分析结果$\mathrm{OUT}_v$，满足$\mathrm{OUT}_v = f_v(\sqcup_{w \in pred(v)} \mathrm{OUT}_w)$

以上定义适用于正向数据流分析。对于反向数据流分析，只需要把数据流图的所有边起点和终点交换，同时交换$entry$和$exit$节点即可。

2.2.1 数据流分析问题的轮询算法

令$V = \{v_1, \ldots, v_n\}$是控制流图的所有节点。首先定义轮询函数F，如下：

$$F(\mathrm{OUT}_{v_1}, \mathrm{OUT}_{v_2}, \ldots, \mathrm{OUT}_{v_n}) = \begin{pmatrix} f_{v_1}(\sqcup_{w \in pred(v_1)} \mathrm{OUT}_w), \\ f_{v_2}(\sqcup_{w \in pred(v_2)} \mathrm{OUT}_w), \\ \ldots \\ f_{v_n}(\sqcup_{w \in pred(v_n)} \mathrm{OUT}_w) \end{pmatrix}$$

数据流分析轮询算法反复将F应用到$(\bot, \ldots, \bot)$上，直到到达不动点，即分析结果

$$(\mathrm{OUT}_{v_1}, \mathrm{OUT}_{v_2}, \ldots, \mathrm{OUT}_{v_n}) = F^k(\bot, \ldots, \bot),$$

其中k满足

$$F^k(\bot, \ldots, \bot) = F^{k+1}(\bot, \ldots, \bot)。$$

定理 3 (正确性). 轮询算法执行结束之后，输出满足$\mathrm{OUT}_v = f_v(\sqcup_{w \in pred(v)} \mathrm{OUT}_w)$。

证明. 根据F的定义和不动点的定义直接可得。 □

定理 4 (终止性). 以上轮询算法必然终止。

证明. 由于f和$\sqcup$都是单调函数，所以F是单调函数，直接应用不动点定理可得。 □

2.2.2 数据流分析问题的工单算法

数据流分析工单算法的伪码如下：

$\forall v \in V \backslash \{entry\} : \mathrm{OUT}_v \leftarrow \bot$;
ToVisit $\leftarrow V \backslash \{entry\}$;
while ToVisit.$size > 0$ **do**
  $v \leftarrow$ ToVisit中任意节点;
  ToVisit $\leftarrow$ ToVisit$\backslash \{v\}$;
  $\mathrm{IN}_v \leftarrow \sqcup_{w \in pred(v)} \mathrm{OUT}_w$;
  if $\mathrm{OUT}_v \neq f_v(\mathrm{IN}_v)$ **then**
    ToVisit $\leftarrow$ ToVisit $\cup\, succ(v)$;
  end
  $\mathrm{OUT}_v \leftarrow f_v(\mathrm{IN}_v)$;
end

定理 5 (终止性). 以上分析算法必然终止。

定理 6. 工单算法终止之后，返回的结果必然和轮询算法相同。

以上两个定理的证明详见胶片。

引理 2 (正确性). 工单算法执行结束之后，输出满足$\mathrm{OUT}_v = f_v(\sqcup_{w \in pred(v)} \mathrm{OUT}_w)$。

证明. 由以上定理和轮询算法的正确性，直接可得。 □

2.3 数据流分析示例

设计一个数据流分析，即设计一套方法，在给定一个控制流图和其他所需条件的时候，给出数据流分析问题的输入。

2.3.1 符号分析

给定程序输入的符号，分析程序输出可能的符号。

分析方向：正向分析

半格元素：从变量到抽象值的函数$X \to \{\text{正}, \text{负}, \text{零}, \text{窏}, \bot\}$，其中$X$是程序中所有变量的集合。

合并运算：

$$(s_1 \sqcup s_2)(x) = s_1(x) \sqcup s_2(x)$$

$$v_1 \sqcup v_2 = \begin{cases} v_2 & \text{if } v_1 = \bot \\ v_1 & \text{elif } v_2 = \bot \\ v_1 & \text{elif } v_1 = v_2 \\ \text{窏} & \text{elif } v_1 \neq v_2 \end{cases}$$

最小元：映射任何变量到$\bot$

输入值：根据输入的符号范围选择合适的抽象值

转换函数：根据相应语句，采用之前定义的抽象域操作进行运算。

2.3.2 可达定值分析

对程序中任意语句，分析运行该语句后每个变量的值可能是由哪些语句赋值的，给出语句标号。要求上近似，即返回值包括所有可能的定值。

分析方向：正向分析

半格元素：从变量到语句标号集合的函数$X \to 2^L$，其中X是程序中所有变量的集合，L是程序中所有语句的标号的集合。

合并运算：对应集合的并 $(s_1 \sqcup s_2)(x) = s_1(x) \cup s_2(x)$

最小元：映射任何变量到空集$\bot$

输入值：最小元

转换函数：$f_v(\text{甲})(x) = (\text{甲}(x) \backslash \text{KILL}_v^x) \cup \text{GEN}_v^x$，其中

- 对于赋值给x的赋值语句，$\text{KILL}_v^x =$ 所有赋值给x的语句编号，$\text{GEN}_v^x =$ {当前语句编号}。
- 对于其他语句，$\text{KILL}_v^x = \text{GEN}_v^x = \emptyset$.

2.3.3 可用表达式分析

给定程序中某个位置p，如果从入口到p的所有路径都对表达式exp求值，并且最后一次求值后该表达式的所有变量都没有被修改，则exp称作p的

一个可用表达式。给出分析寻找可用表达式。要求下近似。
分析方向：正向分析
半格元素：表达式的集合
合并运算：对应集合的交
最小元：全集
输入值：空集
转换函数：$f_v(\text{甲}) = (\text{甲} \backslash \text{KILL}_v) \cup \text{GEN}_v$，其中

- 对于赋值给x的赋值语句，KILL_v = 所有包含x的表达式， GEN_v = 当前语句中求值的不含x的表达式。

- 对于其他语句，$\text{KILL}_v = \emptyset$， GEN_v = 当前语句中求值的表达式.

2.3.4 活跃变量分析

给定程序中的某条语句s和变量v，如果在s执行前保存在v中的值在后续执行中还会被读取就被称作活跃变量。返回所有可能的活跃变量。
分析方向：反向分析
半格元素：变量集合
合并运算：集合的并
最小元：空集
输入值：空集
转换函数：$f_v(\text{甲}) = (\text{甲} \backslash \text{KILL}_v) \cup \text{GEN}_v$，其中

- 对于赋值给x的赋值语句，$\text{KILL}_v = \{x\}$；对于其他语句，$\text{KILL}_v = \emptyset$。

- GEN_v = v中读取的所有变量.

2.3.5 繁忙表达式分析

从执行某个程序节点之前开始，在其中变量被修改之前，在所有终止执行中一定会被读取的表达式。找到每个程序节点的繁忙表达式。要求下近似。

具体定义方式留着练习。

2.4 加宽和变窄

标准数据流分析有可能收敛得很慢，甚至有可能因为半格的高度无限导致不收敛。为了让结果收敛得更快，可以使用加宽的方法。加宽之后结果也会随之变得不精确，这时候可以使用变窄的方法，进一步让结果变精确。

2.4.1 加宽

加宽的基本思路是在每次更新OUT_v的时候，根据更新之前的值和新计算时的值，分析抽象值的变化趋势，然后根据变化趋势推测最终会收敛到的抽象值。

令A为抽象值的空间。加宽算子$\nabla : A \times A \to A$负责完成上述推测操作。给定$o$为更新之前的值，$n$为更新之后的值，则$o \nabla n$表示根据$o$和$n$的差别推测的最终会收敛到的抽象值。

应用加宽之前，数据流分析算法采用如下语句更新OUT_v的值。

$$\mathrm{OUT}_v \leftarrow f_v(\mathrm{IN}_v)$$

应用加宽之后，更新方式变成了：

$$\mathrm{OUT}_v \leftarrow \mathrm{OUT}_v \nabla f_v(\mathrm{IN}_v)$$

以上修改即可以应用于轮询算法也可以用于工单算法。由于工单算法引入了随机性，理论讨论较为复杂，之后的讨论仅限于轮询算法。

定理 7 (加宽安全性). *如果加宽算子满足$y \sqsubseteq x \nabla y$，加宽的轮询算法终止后，满足$\mathrm{OUT}_v \subseteq \mathrm{OUT}_v^w$。其中$\mathrm{OUT}_v$是原始轮询算法返回的结果，$\mathrm{OUT}_v^w$是加宽轮询算法返回的结果，$v$为任意控制流节点。*[1]

定理的证明见胶片。

以上安全性的证明仅限于分析终止的情况，目前没有找到容易的方式来判断分析的终止性。大多数教材、专著和论文中会对加宽算子要求一个比较强的终止性条件：对任意序列都终止。这个条件的形式和分析算法终止性的定义差别不大，要求更强，所以实际对证明帮助有限。

[1] 在很多静态分析的教材[4, 23, 15, 2, 7]中，安全性同时还要求$x \sqsubseteq x \nabla y$，但似乎不需要这个条件也能完成证明。

不过，加宽算子的一些性质通常对实现终止性有帮助，在实际设计时常常遵守。首先，最终达到收敛时，上一轮的值和本轮新计算的值会相等，所以通常加宽算子对两个相等的参数返回相等的参数。其次，由于转换函数和合并函数的单调性，数据流分析的过程中OUT_v值是单调变大的。但∇并不具有单调性，所以有可能这一轮值比上一轮更小，形成振荡。为了避免这种情况，通常要求$x \sqsubseteq x \nabla y$，即加宽分析仍然确保OUT_v值单调变大。注意如果所使用的抽象域仍然形成高度有限的半格的话，$x \sqsubseteq x \nabla y$这个性质就可以保证分析终止了。但由于加宽处理的情况通常是高度无限的半格，所以并不能套用之前的方式来进行证明。

2.4.2 加宽示例：区间分析

假设程序中的变量都是整数，给定输入的上下界，求输出的上下界。要求上近似。

分析方向：正向分析

半格元素：程序中每个变量的区间

合并运算：每个变量的区间对应求并，区间的并定义为

$$[a,b] \sqcup [c,d] = [min(a,c), max(b,d)]$$

最小元：每个变量都映射到$\perp$

输入值：根据具体分析任务的输入上下界确定

转换函数：根据程序语句对区间进行计算。

以上分析是不保证终止的，因为半格的高度是无限的。为了解决这个问题，引入如下加宽算子。

$$\begin{aligned} [a,b] \nabla \perp &= [a,b] \\ \perp \nabla [c,d] &= [c,d] \\ [a,b] \nabla [c,d] &= [m,n] \end{aligned}$$

其中

$$m = \begin{cases} a & c \geq a \\ -\infty & c < a \end{cases}$$

$$n = \begin{cases} b & d \leq b \\ +\infty & d > b \end{cases}$$

易见该加宽算子满足$y \sqsubseteq x \nabla y$，所以保证安全性。同时，该加宽算子也能保证终止。这是因为该算子满足$x \sqsubseteq x \nabla y$，所以在分析过程中OUT_v的值不会减小，同时上下界一旦出现扩大，就会被提升到$\pm\infty$，不会出现无限增大的情况，因此一定保证终止。

2.4.3 变窄

加宽虽然能加快收敛，但也会导致很多分析返回不精确的结果。为了解决这样的问题，变窄通过再次应用原始分析对加宽的结果进行修正。给定加宽分析的OUT_v值作为初值，变窄应用原始的数据流分析对OUT_v值进行多轮迭代更新。可以证明，这样更新之后的结果精度介于加宽分析结果和原始分析结果之间，也就是说保证了安全性。具体证明见课件。

但变窄并不能保证终止。所以实践中通常是限制迭代的轮数，在迭代次数到达上限时终止。

第三章　抽象解释

之前的课程内容中，程序分析的正确性是针对单独应用各自论证的，能否统一论证程序分析的正确性呢？这部分的抽象解释理论就是关于这部分的理论。同时，抽象解释理论也帮助我们理解不同程序分析之间的关系。

3.1　抽象解释理论框架

抽象解释主要关注一个抽象域和一个具体域之间的关系。这里的抽象域和具体域都是带偏序的集合。比如在符号分析中，抽象域的集合是$\{正, 负, 零, 槑, \bot\}$，偏序关系是$\sqsubseteq$；具体域的集合是所有整数集合的幂集，偏序关系是子集关系。本文中采用中文粗体代表抽象域，如$\mathbf{虚}$；采用英文大写字母代表具体域，如$\boldsymbol{D}$.

抽象解释采用抽象化函数和具体化函数来描述抽象域和具体域之间的关系

- 抽象化函数$\alpha : \boldsymbol{D} \to \mathbf{虚}$
- 具体化函数$\gamma : \mathbf{虚} \to \boldsymbol{D}$

我们之前已经见过γ函数，还使用过另一个β函数。β函数是在具体域中元素是集合的时候的特殊的抽象化函数，可以从β导出α，即$\alpha(X) = \sqcup x \in X \beta x$.

抽象解释理论的核心是采用伽罗瓦连接描述了抽象化函数和具体化函数之间的关系。

定义 11 (伽罗瓦连接)**.** 我们称γ和α构成抽象域$\mathbf{虚}$和具体域$\boldsymbol{D}$之间的一个伽罗瓦连接，记为

$$(\boldsymbol{D}, \subseteq) \leftrightharpoons_{\alpha}^{\gamma} (\mathbf{虚}, \sqsubseteq)$$

当且仅当

$$\forall X \in \boldsymbol{D}, 甲 \in \mathbf{虚}, \alpha(X) \sqsubseteq 甲 \iff X \subseteq \gamma(\boldsymbol{D})$$

伽罗瓦连接的定义虽然很简洁，但也意味很多有用的性质。首先，α和γ都是单调的。前面我们已经见过了，单调性在证明分析安全性的过程中起着很重要的作用。其次，$\gamma \circ \alpha$保持或增大输入。这个性质和安全性对应。当我们从具体值得到抽象值的时候，抽象值一定表示了一个包括具体值的范围。最后，$\alpha \circ \gamma$保持或缩小输入。这个性质意味着α函数应该返回尽可能小的抽象值。如果存在一个抽象值能表示当前具体值的时候，那么α函数的返回值不能比这个抽象值更大。关于该定理的定义和证明请参见胶片。

以上抽象域的讨论主要涉及到值，很多时候我们也需要关注函数。

定义 12 (函数抽象)**.** *给定伽罗瓦连接*$(\boldsymbol{D}, \subseteq) \leftrightharpoons_{\alpha}^{\gamma} (\hat{\boldsymbol{D}}, \sqsubseteq)$*，给定*$\boldsymbol{D}$*上的函数*$f$*和*$\hat{\boldsymbol{D}}$*上的函数*$\hat{f}$*，我们说*

- $\hat{f}$*是*f*的安全抽象，当且仅当*

$$\alpha \circ f \circ \gamma(甲) \sqsubseteq \hat{f}(甲)$$

- $\hat{f}$*是*f*的最佳抽象，当且仅当*

$$\alpha \circ f \circ \gamma = \hat{f}$$

- $\hat{f}$*是*f*的精确抽象，当且仅当*

$$f \circ \gamma = \gamma \circ \hat{f}$$

注意精确抽象实际上意味着抽象域的函数不丢失信息，这样的函数抽象不一定存在。最佳抽象意味着返回尽可能精确的值，这样的函数抽象总是存在的，但在实际中不一定容易定义。之后我们会介绍符号抽象技术，用于针对特定具体域函数找到最佳抽象。最后，安全抽象是程序分析的基本要求，之后的讨论中主要使用的是安全抽象的概念。

3.2 抽象解释和分析正确性

一般而言，程序分析是分析程序在给定输入集合下的所有具体执行序列具有的性质。从抽象解释的角度来看，这里的具体域的元素是任意的程序在任意输入集合下的所有具体执行序列集合，抽象域是这组具体执行对应的分析结果。抽象域上的偏序关系就定义了分析的正确性或安全性：如

果分析出来的结果大于等于原结果，那么分析就是正确的。也就是说，我们可以用伽罗瓦连接来定义程序分析结果的正确性。

给定一个具体程序，该程序的具体执行语义定义了如何从一个输入集合产生该程序的所有执行序列，即一个从输入状态集合到具体执行序列集合的函数。那么，如果一个抽象解释对应的程序分析是正确的，该程序分析就应该是上述函数的安全抽象。

通过这样的方式，我们就把证明一个分析的正确性转为了证明特定伽罗瓦连接上安全函数抽象的问题。下面我们具体看一下如何定义具体语义和证明数据流分析的安全性。

3.2.1 控制流图上具体语义

一个程序的下一步执行由其内部执行状态决定，包括所有变量取值，堆上的值，和PC指针等。我们这里不去区分程序的内存的细节，只是简单假设存在一个内存状态集合M表示程序执行过程中所有可能的内存状态。给定一个控制流图(V, E)，一个用控制流图表示的程序的一个执行状态就是由(v, m)组成的对，称为*执行状态*，其中$v \in V, m \in M$。我们称由执行状态构成的序列为*具体执行序列*，或者称为一个*执行踪迹*（trace）。

我们同时假定每个控制流节点v对应一个*具体转换函数*$trans_v : M \to 2^M$和一个*控制转移函数*$next_v : M \to 2^{succ(v)}$。两个函数返回的都是幂集，是因为程序可能有不确定的情况，一个状态可以有多个下一个状态，比如有随机函数。同时，在程序结束的时候下一个状态是空集。当我们考虑反向分析的时候，我们还需要反向定义程序语义，而反向语义通常是不确定的。

那么我们可以定义出如下的单步执行函数$step : \mathbb{T} \to \mathbb{T}$，其中$\mathbb{T}$是执行踪迹的全集。

$$step(t) = \{t+(v', m') \mid v' \in next_{last(t).node}(last(t).mem), m' \in trans_{v'}(last(t).mem)\},$$

其中$+$表示序列的追加，$last$返回序列的最后一个元素，$s.node$返回状态s对应的控制流节点，$s.mem$返回状态s对应的内存状态。

我们接着将单步执行函数扩展到集合上，即

$$Step(T) = (\bigcup_{t \in T} step(t)) \cup T$$

那么，一个程序从输入状态集合T_I出发所能产生的所有执行踪迹就为$Step^{\infty}(T_I) = \lim_{n\to\infty} Step^n(T_I)$。

这种将程序映射为执行踪迹集合的语义定义方式通常称为迹语义(Trace Semantics)。

3.2.2 数据流分析的安全性

下面我们证明数据流分析是安全的。我们之前定义过轮询函数F，而数据流分析就是反复应用轮询函数直到不动点，即结果为$F^{\infty}(I)$。我们试图论证F是$Step$的安全抽象，然后自然F^{∞}就是$Step^{\infty}$的安全抽象。换个角度，我们可以认为F定义了控制流图上的抽象语义。

因为数据流分析其实是分析出从起始节点到某个节点v的所有执行踪迹所满足的性质，所以我们区分具体分析定义的伽罗瓦连接和数据流分析的伽罗瓦连接。具体分析定义的伽罗瓦连接记为是$(\boldsymbol{D}, \subseteq) \leftrightharpoons_{\alpha}^{\gamma} (\boldsymbol{虚}, \sqsubseteq)$，其中具体域是从起始节点到某个节点的执行踪迹的集合，抽象域是该集合的性质。

因为数据流分析对每个控制流节点返回一个值，所以数据流分析可以看做是两个伽罗瓦连接的复合。第一次将所有执行踪迹的集合抽象为一个映射，映射的输入是节点v，输出是从起始节点到某个节点v的所有执行踪迹。第二次基于$(\boldsymbol{D}, \subseteq) \leftrightharpoons_{\alpha}^{\gamma} (\boldsymbol{虚}, \sqsubseteq)$将每个节点对应的执行踪迹集合抽象为该踪迹集合的性质。第一个伽罗瓦连接用$(\boldsymbol{D}, \subseteq) \leftrightharpoons_{\alpha_w}^{\gamma_w} (\boldsymbol{D}_w, \subseteq_w)$表示，第二个用$(\boldsymbol{D}_w, \subseteq_w) \leftrightharpoons_{\alpha_a}^{\gamma_a} (\boldsymbol{虚}_w, \sqsubseteq_w)$表示。数据流分析产生的完整伽罗瓦连接是两次连接的复合，即$\alpha = \alpha_a \circ \alpha_w$，$\alpha_a(d)(v) = \alpha(d(v))$。$\gamma$类似。

我们要求抽象域上的节点转换函数f_v满足如下条件。

$$\forall t \in \gamma(\text{甲}), (t' \in step(t) \wedge last(t').node = v) \Rightarrow t' \in \gamma(f_v(\text{甲}))$$

即节点转换函数只需要对于会实际执行当前节点的任意踪迹保证安全即可。

基于上面的条件，采用之前具体数据流分析部分讲过的方法，可以很容易证明F是Step的安全抽象，这里不再详细展开。

3.3 流非敏感分析

如果我们不区分不同节点上的OUT值，我们就得到了流非敏感分析，即

$$F_{fi}(\mathrm{OUT}) = \bigsqcup_{v \in V} f_v(\mathrm{OUT})$$

可以证明F_{fi}的分析结果和F的分析结果形成了伽罗瓦连接，即流非敏感分析是对流敏感分析的进一步抽象。该抽象忽略掉了“节点”这个维度。之后会看到不同敏感性的分析，都是在分析中添加/去掉某个维度得到。

第四章　过程间分析

之前的章节中，我们没有考虑过程和过程调用，即分析都限制在一个过程内部，这样的分析被称为过程内分析。这一章我们考虑过程间分析，即考虑过程和过程调用的分析。

之前我们定义的控制流图是针对过程内部的，只有一个$entry$节点和一个$exit$节点。现在我们程序由多个过程组成，则这样的程序就对应多个控制流图，对于过程p，对应控制流图的入口节点为$entry_p$，出口节点为$exit_p$。同时，控制流图上会有两类特殊的节点，过程调用节点负责调用一个其他过程，过程返回节点负责处理调用过程的返回值。过程调用节点没有后继节点，而过程返回节点的前驱节点为过程调用节点的前驱节点。

4.1　上下文不敏感的过程间分析

处理过程调用的方式就是直接的方式就是把不同过程的控制流图连起来，称为超级控制流图。如果有一个过程调用节点$call$调用了过程p，同时对应的过程返回节点为$resume$，那么我们就添加两条边：从$call$到$entry_p$，从$exit_p$到$resume$。

很多分析是针对程序中的每个变量分析出一个值，比如符号分析。由于通常不同过程有不同的本地变量，所以一般对每个过程采用不同的抽象域。这样，$call$和$resume$就负责在不同的抽象域之间转换。$call$根据当前过程的抽象域计算出实参的抽象值，然后转化为被调用过程的实参抽象值。$resume$根据被调用过程的抽象域计算出抽象返回值，然后在被调用进程中替换给合适的变量，类似赋值语句。注意这里$resume$不能直接把前驱节点的值合并起来，而需要区分当前过程的前驱节点和$exit_p$节点。

如果某过程p涉及到读全局变量g，全局变量g要添加到p和所有直接和间接调用p过程的输入中。类似的，如果某过程p涉及到写全局变量g，全局

变量g要添加到p和所有直接和间接调用p过程的输出中。全局变量被大量添加是分析大型软件的一个问题。

4.2 基于克隆的过程间分析

前面这种方式会产生不精确，因为在分析过程中混淆了不同调用上的结果。比如A过程和B过程都调用了C过程，但A过程传入C的值对应的返回值可能会流入B过程。即会考虑实际不可能出现的执行轨迹。

为了避免这种情况，我们需要细化一下具体执行的语义。之前我们认为一个执行状态包括一个控制流图节点和一个内存状态，但引入过程调用之后，具体执行状态需要额外包含一个调用栈，才能进行正确的返回。具体而言，调用栈是一个序列，按顺序包括所有之前执行了但还没有返回的过程调用节点。

为了匹配具体执行中的调用行为，我们需要在抽象域也引入调用栈，这样函数返回的时候就可以根据调用栈进行返回。但是，调用栈是一个无穷的集合，所以我们需要设计一个调用栈的抽象域。通常的做法是取最近k次调用，即调用栈的长度最多为k。这样抽象调用栈就变成了一个有穷集合。简单起见，我们通常认为抽象调用栈的长度固定为k，对于长度不到k的调用栈引入特殊符号补齐。比如$k = 3$,具体调用栈中只包含一次调用v，那么对应的抽象调用栈就是$(-,-,v)$，其中$-$是用来补全的特殊符号。

但是，如果只是简单将抽象调用栈加到抽象域中并不解决问题，因为我们还是无法区分抽象状态中代表的执行踪迹哪些来自于A哪些来自于B，即我们无法写出有实际效果的状态压缩函数。为了区分来自不同调用的执行踪迹，我们进一步细化OUT值。之前对于每个控制流节点有一个OUT值，现在我们针对每一个控制流节点和每一个抽象调用栈有一个OUT值。令v为非过程调用/返回节点的任意控制流节点，c为任意抽象调用栈，$call$为调用p的过程调用节点，$resume$为$call$对应的过程返回节点，$inproc_pred$负责返回当前过程内的前驱节点，$tail$返回序列除了第一个元素之外的子序列，那么我们有如下方程组。

$$
\begin{aligned}
OUT_{v,c} &= f_v\left(\bigsqcup_{w\in pred(v)} OUT_{w,c}\right)\\
OUT_{call,tail(c)+call} &= f_{call}\left(\bigsqcup_{w\in pred(v)} OUT_{w,c}\right)\\
OUT_{resume,c} &= f_{resume}\left(\bigsqcup_{w\in inproc\_pred(v)} OUT_{w,c}, OUT_{exit_p,tail(c)+call}\right)
\end{aligned}
$$

用轮询/工单算法求解这组方程，就得到了过程间分析的解。

这里的抽象调用栈可以看做是当前调用的上下文，因此这种分析被称为*上下文敏感分析*。添加新的OUT值等价于克隆被调用过程的控制流图，因此这种分析被称为*基于克隆的上下文敏感分析*。根据需要，也可以采用不同类型的上下文。比如在面向对象语言中，函数调用是针对某个特定的对象发起，比如x.m()是针对x发起，那么可以用x的具体值作为上下文（如何抽象表达x的值是后续指针分析介绍的内容），这样的分析通常称为*对象敏感分析*。

4.3 基于上下文无关文法可达性的分析

基于克隆的上下文敏感分析只是考虑最近k次调用，能否完整考虑所有调用栈呢？即我们永远不会考虑在函数调用关系上不成立的执行轨迹。这样的分析也被称为精确的上下文敏感分析。

本章介绍一种基于上下文无关文法(CFL)可达性的分析方法。该方法针对满足分配性的数据流分析开展，具有直观的图形表示和较好的理论性质，是目前被广泛使用的一种方法。该方法的主要思路是，精确的上下文敏感分析主要是要正确匹配调用边和返回边。一个执行序列中的调用边和返回边是否匹配和匹配括号一样，是一个上下文无关属性，可以用上下文无关文法来捕获。即如下Dyck上下文无关文法。

$$\begin{array}{rcl} S & \rightarrow & \{_1 S\}_1 \\ & | & \{_2 S\}_2 \\ & | & \dots \\ & | & SS \\ & | & \epsilon \end{array}$$

这里每一个$\{_i$表示一个调用节点，$\}_j$表示一个返回节点，$i = j$表示调用和返回匹配。如果我们把一个执行轨迹上的所有调用节点和返回节点都提取出来组成子序列，如果这个子序列符合上面的文法，就说明执行序列在调用关系上是合法的。

那么怎么做到只分析了这样的执行序列呢？CFL可达性首先利用数据流分析的分配性，把任意转换函数分解为一系列单位元素组成的图的可达性问题。然后在这个图上求解如下的CFL可达性问题：对于图中任意结

点v_1、v_2，确定是否存在从v_1到v_2的路径，使得该路径上的标签组成了给定上下文无关文法中的句子。对于精确的上下文敏感分析，这里的上下文无关文法就是上面的Dyck文法。具体图的转法用latex不太好排版，详见课件。

虽然求解CFL可达性问题存在通用算法，但针对精确的上下文敏感分析，可以发展出更高效的算法，避免计算不可达的路径。基于这样算法形成的分析框架叫做IFDS框架。

4.4 函数摘要分析

换个角度来看，IFDS的求解算法可以认为是对每个过程做了一个函数抽象，在给定输入的抽象值之后，可以利用IFDS算法得到的图计算过程的输出抽象值。这样的函数对于任何上下文都是固定不变的，所以如果A调用了B，在生成A的函数抽象过程中，我们只需要直接使用B的函数抽象就可以了，而不需要对于不同地方对B的调用进行不同的分析。

基于这个思路，我们可以泛化出一套基于函数摘要的精确上下文敏感分析。具体而言，我们首先分析出每个过程对应的抽象域上的函数摘要，然后对于Main函数的摘要传入抽象域的输入，我们就能得到抽象域上的输出。这样，我们可以根据分析的特点来选择函数表示方式，有可能做到比IFDS更高效的分析。

比如我们考虑数据流分析标准型。假设全集集合中有m个元素，那么每个控制流节点就要展开成m个节点。然后假设一个过程中所有的n个节点的Gen和Kill都为空，那么通过选择合适的函数表示，我们可能可以用$O(n)$的时间计算出来这个过程整体的转换函数Gen和Kill都为空，但CFL可达性分析必须在$O(nm)$个节点的图上跑可达性分析。

注意IFDS算法本身是一个不断加边直到到达不动点的过程，所以产生函数摘要的过程也是一个程序分析过程。首先我们需要选择一个合适的抽象域来表示函数。这个抽象域必须能表示函数的复合和函数的合并两种操作，即：

$$\begin{aligned}(f_2 \circ f_1)(x) &= f_2(f_1(x)) \\ (f_1 \sqcup f_2)(x) &= f_1(x) \sqcup f_2(x)\end{aligned}$$

针对数据流标准型，我们可以用Gen和Kill两个集合来表示函数。考虑合并操作为集合的并集，函数的复合和合并分别定义如下。

$$\begin{aligned}(g_2,k_2)\circ(g_1,k_1) &= (g_2\cup(g_1-k_2),k_1\cup k_2)\\ (g_1,k_1)\sqcup(g_2,k_2) &= (g_1\cup g_2,k_1\cap k_2)\end{aligned}$$

可以验证以上计算式符合上面的要求。

然后，我们在程序上做一个数据流分析来得到所有函数的摘要。每个控制流图节点的输出值是从函数入口位置到当前节点的函数摘要。因为该摘要是对原数据流分析的摘要，为了区分，我们用f_v表示原数据流分析的转换函数，用$\hat{f}_v$表示用来产生函数摘要的数据流分析的转换函数，那么对于非过程调用节点，$\hat{f}_v$定义如下：

$$\hat{f}_v((g,k)) = f_v\circ(g,k)$$

对于过程调用节点$call$，假设$call$调用了过程p，那么转换函数定义如下：

$$\hat{f}_{call}((g,k)) = OUT_{exit_p}\circ(g,k)$$

可以归纳证明上面的转换函数都是单调函数。

每个过程Entry节点的初值是一个等价变换函数，即$(\emptyset,\emptyset)$。

通过以上分析，我们可以得到每个过程的函数摘要，再传入初值就得到了分析的结果。

第五章　稀疏分析

大量的程序分析是关于变量中保存了什么值。对于这样的分析，典型的抽象域是一个映射，从变量映射到变量的值。但由于一个程序每个语句一般只读取少部分变量，最多修改一个变量，这样的数据流分析就造成两方面的冗余开销。

- 空间的冗余：每个节点都要对所有变量保存一份值，即使大部分变量的值都是相同的。

- 时间的冗余：如果一个控制流节点不修改变量的x，那么该节点的转换函数只是简单传递x的值。整个分析中大部分都是这样冗余的传递。

稀疏分析采用一个预分析来构建更稀疏的分析方程，避免这样的冗余开销。为了避免空间的冗余，稀疏分析不在每个节点都保存所有变量的抽象值，而是只保存当前修改的变量的抽象值。为了能在控制流节点需要值的时候读取相应的值，稀疏分析不再沿着数据流图传递抽象值，而是在需要某个变量的值的时候直接从该变量最后一次赋值的位置读取相应的值。

为了实现上述分析，我们必须知道每个变量的值是在什么地方最后被赋值的。注意因为有控制流汇合的情况，一个变量最后被赋值的位置可能有多个。为了捕获这样的信息，我们引入“定义-使用”关系的概念。给定变量x，如果节点A可能修改x的值，节点B可能读取由A写入的x的值，我们就说A和B之间存在“定义-使用”关系，记为$def(A) \xrightarrow{x} use(B)$。稀疏分析假设存在一个预分析来获得这个信息。

有了“定义-使用”关系之后，我们就可以利用这个关系构建数据流分析了。给定一个控制流节点v，假设其读取的变量是$\{r_1, \ldots, r_n\}$，写入的变量是$\{w_1, \ldots, w_m\}$, 那么我们得到如下的方程。其中f_v是节点v对应的转换

函数，根据读取变量的抽象值返回写入变量的抽象值；$i \in \{1 \dots n\}$。

$$\mathrm{IN}_v^{r_i} = \bigsqcup_{def(v') \xrightarrow{r_i} use(v)} \mathrm{OUT}_{v'}^{r_i}$$

$$(\mathrm{OUT}_v^{w_1}, \dots \mathrm{OUT}_v^{w_m}) = f_v(\mathrm{IN}_v^{r_1}, \dots, \mathrm{IN}_v^{r_n})$$

求解该方程组，就得到了稀疏分析的结果。

5.1 获得“定义-使用”关系

获得“定义-使用”关系通常有两种方法。一种是执行可达定值分析，可达定值分析的结果就对应了程序中所有的“定义-使用”关系。另外一种是把程序转换成静态单赋值的形式，该形式保证所有变量只被赋值一次，顺着变量名就能直接找到所有的“定义-使用”关系。

需要注意的是，因为标准的可达定值分析和静态单赋值形式都只考虑栈上的变量，稀疏分析通常只针对栈上的变量。堆上的值通过指针等结构进行间接访问，虽然也可以通过抽象的方式定义出“定义-使用”关系，但构建高效的预分析较为困难。

第六章　指针分析

指针分析是关于回答指针之间关系的程序分析的统称。在指针分析中，被研究得最多的问题就是指向分析(points-to analysis)，即分析每个指针变量可能指向的地址。其他很多分析，比如别名分析（判断两个指针是否可能指向同一位置），通常也是以指向分析为基础构建。

指向分析有两种基础算法，分别被称为Anderson指向分析算法和Steensgaard指向分析算法。Anderson算法更精确，Steensgaard算法速度更快。

6.1　Anderson指向分析

Anderson指向分析就是按程序分析标准流程对指针操作做抽象之后产生的分析。

首先我们假设程序中没有堆上分配的内存，没有结构体、数组等数据结构，没有*(p+1)等指针运算。这样，程序中可能保存值的地址就是局部和全局变量在栈上的地址。程序分析的结果就是从指针变量到变量地址集合的映射。我们用OUT_v^a来表示a指针变量在v节点执行之后保存的变量地址的值，在不引起混淆的情况下直接用a表示变量a的地址，那么一个流敏感的指针分析就是对每个节点v和每个变量x计算所有的OUT_v^x值，每个值是由变量地址组成的集合。

在忽略上述复杂结构之后，和指针的操作可以看做由如下四种基本语句构成：

- a=&b
- a=b
- a=*b

- *a=b

其他语句可以看做是由这四条复合而成，比如*a=**b,可以写成：

```
c=*b;
d=*c;
*a=d;
```

因此我们只需要对这四种基本语句定义转换函数即可。我们这里采用方程的表示形式。下表给出了我们针对每个语句产生的约束。我们假设对于每个节点v和每个变量x，都有$\mathrm{IN}_v^x = \bigcup_{v' \in pred(v)} \mathrm{OUT}_v^x$。同时，对于表格中没有出现在等号左边的变量，默认方程为$\mathrm{OUT}_v^x = \mathrm{IN}_v^x$。

赋值语句	约束
a=&b	$\mathrm{OUT}_v^a = \{b\}$
a=b	$\mathrm{OUT}_v^a = \mathrm{IN}_v^b$
a=*b	$\mathrm{OUT}_v^a = \bigcup_{x \in \mathrm{IN}_v^b} \mathrm{IN}_v^x$
*a=b	$\forall x \in \mathrm{IN}_v^a, \mathrm{OUT}_v^x = \mathrm{IN}_v^b \quad \lvert\mathrm{IN}_v^a\rvert = 1$ $\forall x \in \mathrm{IN}_v^a, \mathrm{OUT}_v^x = \mathrm{IN}_v^x \cup \mathrm{IN}_v^b \quad \lvert\mathrm{IN}_v^a\rvert > 1$

对于最后一列的两种情况，一般把第一种情况称为强更新（strong update），第二种情况称为弱更新（weak update）。

对于流非敏感分析，将同一变量在不同控制流节点的值对应合并，同时转换函数也对应合并即可。注意这样合并之后强更新实际就不存在，因为a在其他位置的值一定是直接传递的。

接下来我们考虑堆上分配的内存。这里的主要问题是每次执行malloc或者new语句就会产生一个新的地址，这样整个程序中的地址是无穷的。为了解决这个问题，一般是对地址坐抽象，常用方法是针对每个malloc语句创建一个抽象地址，代表由这个语句分配的所有对象的具体地址。这样，对于一个分配语句，我们产生如下约束。

赋值语句	约束
a=malloc()//id:1	$\mathrm{OUT}_v^a = \{1\}$

接下来我们考虑结构体。假设我们有如下结构体。

```
struct Node {
    int value;
    Node* next;
    Node* prev;
};
```

该结构体内部又包含两个指针，next和prev，因此，我们需要额外对这两个指针添加指针变量。由于在所有出现该结构体的地方都包含这两个指针，所以我们需要对所有Node类型的地址x都添加两个指针变量：$x.next$和$x.prev$。注意这里x即可以是栈上的变量也可以是malloc分配在堆上的地址。

类似地，因为结构体中的每个字段都可以被取地址，所以对每个Node类型的地址x我们都需要增加三个新的地址：$x.value$、$x.next$和$x.prev$，分别表示三个域的地址。

添加这些变量和地址之后，产生约束的时候对应考虑这些约束和地址就可以。同之前的情况类似，访问字段的情况可以用如下三种情况概括，其对应约束和之前类似，主要需要考虑指向结构体的指针的所有可能性。

赋值语句	约束
a=&b.f	$\mathrm{OUT}_v^a = \{x.f \mid x \in \mathrm{IN}_v^b\}$
a=b->f	$\mathrm{OUT}_v^a = \bigcup_{x \in \mathrm{IN}_v^b} \mathrm{IN}_v^{x.f}$
a->f=b	$\forall x \in \mathrm{IN}_v^a, \mathrm{OUT}_v^{x.f} = \mathrm{IN}_v^b \quad \lvert\mathrm{IN}_v^a\rvert = 1$ $\forall x \in \mathrm{IN}_v^a, \mathrm{OUT}_v^{x.f} = \mathrm{IN}_v^x \cup \mathrm{IN}_v^b \quad \lvert\mathrm{IN}_v^a\rvert > 1$

6.1.1 Steensgaard指向分析

Anderson算法的开销主要来自于顺着在集合之间传递地址，如果能取消这个传递就有望显著提升算法分析效率。Steensgaard指向分析算法的基本思路是通过牺牲精度来避免这个传递。具体而言，如果两个指针变量的指向集合有可能需要传递地址，那么Steensgaard算法就会认为这两个指针变量所指向的集合完全相同。在实际实现中，可以直接将这两个指针变量合并成一个。对比Anderson算法的三次方复杂度，Steensgaard算法的复杂度为$O(n\alpha(n))$，接近线性时间。

具体而言，针对一个流非敏感的指向分析，Steensgaard指向分析产生如下约束。

赋值语句	约束
a=&b	$b \in \mathrm{OUT}^a$
a=b	$\mathrm{OUT}^a = \mathrm{OUT}^b$
a=*b	$\mathrm{OUT}^a = \mathrm{OUT}^{*b}$
*a=b	$\mathrm{OUT}^{*a} = \mathrm{OUT}^b$

同时，Steensgaard算法添加一条额外的等价关系$\forall y, \forall x \in \mathrm{OUT}^y, \mathrm{OUT}^x = \mathrm{OUT}^{*y}$。即对于任意一个二级指针，其指向的指针变量对应的集合都相等。

这是因为这个二级指针的值一旦进行了读取或者写入，所有被指向的变量的值就会和同一个值产生关联，按照Steensgaard算法的标准就应该相等。

不同于Anderson算法，Steensgaard算法为*a等间接访问的指针变量也创建集合，因为别名指针对应的集合会被快速合并，所以并不会产生额外的维护开销。

Steensgaard算法的具体计算过程见课件。

6.1.2 基于CFL可达性的指向分析

指向分析也可以转成CFL可达性的问题求解。具体而言，我们首先创建一个指向关系的图，其中图上的有两类节点。第一类是所有的抽象地址。第二类是指针变量，然后针对赋值语句创建不同类型的边。

赋值语句	边
a=&b	$b \xrightarrow{new} \text{OUT}^a$
a=malloc()//id:1	$1 \xrightarrow{new} \text{OUT}^a$
a=b	$\text{OUT}^b \xrightarrow{assign} \text{OUT}^a$
a=*b	$\text{OUT}^b \xrightarrow{get[*]} \text{OUT}^a$
a=b->f	$\text{OUT}^b \xrightarrow{get[f]} \text{OUT}^a$
*a=b	$\text{OUT}^b \xrightarrow{put[*]} \text{OUT}^a$
a->f=b	$\text{OUT}^b \xrightarrow{put[f]} \text{OUT}^a$

对于图上的每条边$x \xrightarrow{l} y$，同时添加反向边$y \xrightarrow{\bar{l}} x$。然后针对如下上下文无关文法进行CFL可达性分析。

$$
\begin{aligned}
FlowTo &= new\ (assign \mid put[f]\ Alias\ get[f])^* \\
PointsTo &= (\overline{assign} \mid \overline{get[f]}\ Alias\ \overline{put[f]})^*\ \overline{new} \\
Alias &= PointsTo\ FlowTo
\end{aligned}
$$

可以证明基于CFL可达性的指向分析和Anderson指向分析算法等价

6.2 控制流分析

之前我们一直假设对于某个函数调用，我们能静态知道该调用是调用的哪个函数。但实际由于函数指针的存在，这个假设是不成立的。因此，产生完整控制流图的过程必须和指针分析同时进行，该分析通常被称为控制流分析。

具体来说，在存在函数调用的时候，我们需要根据对应程序设计语言的语义执行添加“对于函数指针指向的任意函数，添加调用边和回边”这样的约束。因为这个约束涉及到动态添加边，所以无法预先转成图，所以在进行上下文敏感分析的时候通常采用展开多层的方式进行。

第七章　关系型抽象域

之前学习过有大量分析是关于变量中可以保存什么值的，比如指向分析、区间分析、符号分析等。我们目前学习过的分析都是对每个变量单独进行抽象，不考虑变量之间的关系。这类不考虑变量之间关系的抽象称为非关系抽象。

非关系型抽象由于忽略了变量之间的关系，在进行分析的时候常常无法得到精确的结果。比如，我们可以考虑下面的程序。

a=x ;
b=x ;
c=a−b ;

如果起始状态中x的区间为$[0,1]$，那么区间分析的结果是$[-1,1]$。但其实在这个例子中，a和b之间始终都有等价关系，所以精确结果应该是$[0,0]$。

本章我们介绍关系抽象，其基本特点是考虑了变量之间的关系。

7.1　简单仿射关系抽象

简单仿射关系抽象[27]是区间抽象的改进版本，其基本思路是对区间抽象域进行扩展，将变量的值记录为由一系列抽象符号组成的线性表达式，而针对这些抽象符号记录区间。这里介绍流敏感的简单仿射关系抽象。

为了限制抽象符号空间的大小，简单仿射关系抽象采用为每个变量在每个控制流节点记录一个抽象符号，即$s_{v,x}$，其中v为控制流节点，x为变量名。

抽象域的每个值为两个函数$(申,酉)$。其中申记录了每个变量对应的抽象符号线性表达式，即申是一个从变量到抽象符号线性表达式的函数，给定任意变量x，其对应的申的函数值要么是$\perp$，要么是$\sigma w_{v,x} s_{v,x}$，其中$s_{v,x}$为

抽象符号，$w_{v,x}$为对应的线性系数。 酉记录了每个抽象符号对应的区间。

对于每个节点v，我们首先要合并其前驱节点的值。这个合并我们针对申和酉分别进行。对于申来说，如果任意一个前驱的表达式是$\perp$，则忽略这个前驱；如果某个变量x在所有前驱节点对应的表达式都相同，则保留这个表达式，否则则令表达式直接为$s_{v,x}$，即采用一个新的抽象符号来表示该节点的抽象值。这样，申所对应的格的高度实际只有2，确保分析收敛。对于酉来说，因为酉是从抽象符号到区间的映射，那么和区间抽象类似，直接合并对应符号的区间即可。由于区间的范围是无限的，所以在分析的时候需要再加上之前加宽的方法来确保收敛。

对于节点v的转换函数主要看该转换函数执行的是线性运算还是不是线性运算。如果赋值语句执行的是线性运算，比如x=a+b，那么就在申中记录对应的线性运算，即：

$$\begin{aligned} &OUT^{\text{申}}(x) = IN^{\text{申}}(a) + IN^{\text{申}}(b) \\ &OUT^{\text{申}}(y) = IN^{\text{申}}(y) \qquad\qquad \forall y \neq x \\ &OUT^{\text{酉}} = IN^{\text{酉}} \end{aligned}$$

如果赋值语句执行的不是线性运算，如x=a*b，那么就采用一个新的抽象符号来表示该节点的抽象值，即$OUT^{\text{申}}(x) = s_{v,x}$，并且$s_{v,x}$在酉中记录的区间根据区间分析的计算得出。

初试把所有节点所有变量的申值都设置为$\perp$，所有抽象符号的酉值都设置为空集合，然后用数据流分析的方法分析即可。

在下面这个例子中：

1: a=x;
2: b=x;
3: c=a−b;

我们首先会为x创建抽象符号$s_{1,x}$，并且记录该抽象符号的区间为$[0,1]$，然后记录a和b对应的表达式为$s_{1,x}$，最后c对应的表达式就为$s_{1,x} - s_{1,x} = 0$，从而得到c的区间为$[0,0]$。

简单仿射关系抽象这个名字来源于仿射关系抽象[12]。简单仿射关系抽象不能直接得到线性关系的时候（如两个表达式合并、非线性计算）直接创建新的符号值，但完整的仿射关系抽象在这些情况仍然会试图找到尽可能精确的线性关系来刻画程序执行轨迹集合。

7.2 八边形抽象

简单仿射关系抽象的关系主要依靠赋值语句推出，如果两个变量之间没有赋值关系，简单仿射关系抽象就无法推出两个变量之间的关系了。我们考虑这个如下程序：

```
x=0;
y=0;
while (x<10){
  x++;
  y--;
}
```

在这个程序中，x和y之间一直有互为相反数的关系，但这个关系无法由简单仿射关系推出，因为二者之间没有互相赋值。八边形抽象对于任意两个变量之间都用一组区间来刻画两个变量满足的关系，因此可以克服该问题。

具体而言，八边形抽象对于任意两个变量x和y记录如下四个区间：

- x+y的区间
- x-y的区间
- x的区间
- y的区间

在一个由x和y组成的二维坐标系统中，x和y的区间可以看做是水平和竖直的四条直线，而x+y和x-y可以看做是斜45度的四条直线，这些直线合在一起组成了一个八边形，八边形内部就是x和y对应的可能取值，所以叫做八边形抽象。

八边形抽象域上的合并操作就是对应区间的并。

八边形抽象的转换函数设计的基本思路和之前类似：给定抽象域限定的所有取值，考虑语句执行之后的所有取值，然后重新计算八边形的范围。因为八边形抽象的整体计算比较复杂，详细的计算规则可以参考原始论文[14]。

如果采用八边形抽象域，可以确保在上面的程序中推导出x和y的相反数关系，即x+y的区间始终是[0, 0]。详细分析过程见课程胶片。

第八章　符号执行

到目前为止的内容主要关注基于抽象法进行程序分析。本小节介绍如何用搜索法进行程序分析。搜索法的核心是约束求解工具。

8.1　约束求解工具

给定一个包含自由变量的逻辑公式，约束求解工具判断是否存在一组自由变量赋值，使得该逻辑公式可以满足。换句话说，约束求解工具是判断$\exists x_1, \ldots, x_n, P(x_1, \ldots, x_n)$的逻辑公式是否成立，其中$P$是一个不含自由变量的逻辑公式。比如，一个约束求解工具可以回答针对$x+y=10 \wedge x-y=5 \vee x+z=s.length()$这样的式子，是否存在$x, y, z, s$的值，让式子满足，或者是回答$\exists x, y, z, s, (x+y=10 \wedge x-y=5 \vee x+z=s.length())$是否成立。

一大类约束求解工具被称为“可满足性模理论（SMT）”求解工具。标准的逻辑系统，比如一阶逻辑系统，是只为与、或、非等逻辑运算符赋予了相应的语义，而逻辑运算符之外的运算符，比如加号、减号等，都统一视为某种函数名（在逻辑上称为函词），没有特定的含义。为了在公式中使用数学上这些常用的符号，SMT求解工具假设存在若干理论，这些理论给逻辑公式中的部分谓词和函词给与了相应的语义解释；或者从语法角度来看，针对一些特定的谓词和函词提供了公理。这样，我们就可以基于数论上加减法的含义判断$x+y=10 \wedge x-y=5$是否可满足等。

约束求解工具通常采用一种搜索算法去寻找让逻辑公式满足的值，或者搜索出不满足的证明。因此基于约束求解工具的程序分析方法称为搜索法。关于约束求解工具的求解算法，可以参考相关教材[13]。

除了判断逻辑公式是否可以满足，通常约束求解工具对于可满足的公式可以给出一组赋值，使得公式可满足。对于一个不可满足的合取公式集合（假设公式之间用$\wedge$连接），约束求解工具一般还能返回一个尽可能小的

不可满足子集，称为“最小矛盾子集”。

8.2 符号执行

8.2.1 基础符号执行

由于约束求解工具的强大能力，一个基本的思路是利用约束求解工具来完成程序分析。符号执行就是这样一种将程序转换为逻辑公式并且由约束求解工具判断其可满足性的技术。

具体而言，符号执行把输入中的变量值替换成符号，然后沿着某条路径去执行该程序，最终得到包含符号值的程序结束状态。然后我们可以用约束求解工具判断这样的结束状态是否一定满足我们想要的条件。

比如，对于下面的程序，如果我们知道输入中x>0，y为任意值，然后我们想要知道程序执行结束之后x是否一定是大于0的。

```
y *= y;
x += y;
```

那么我们可以引入符号值a和b表示输入时x和y的值。即输入状态为$\{x \mapsto a, y \mapsto b\}$。第一条语句执行结束之后的状态为$\{x \mapsto a, y \mapsto b * b\}$。第二条语句执行结束之后的状态为$\{x \mapsto a + b * b, y \mapsto b * b\}$。然后我们可以得到一个逻辑式子$a > 0 \to a + b * b > 0$。我们想要判断这个式子是不是恒成立，可以把这个式子取反之后判断可满足性。如果取反之后的式子是可满足的，就说明式子不是恒成立，即程序不满足我们提出的规约。我们还可以进一步要求约束求解工具提供反例。

8.2.2 分支和循环

以上程序只包含顺序执行。对于带有条件分支语句的程序，符号执行每次只分析一条路径。比如对于下面程序：

```
if (x > 0) y++;
else y--;
```

假设初试状态x和y的值仍然是a和b，然后对于if语句选择了为真的分支，那么符号执行会记录下一个分支条件$a > 0$和直接结束后y的符号值$b + 1$，其中$a > 0$称为路径条件。假设我们要判断结束后y的值是否大于0，我们需要

检查如下逻辑公式是否恒成立$a > 0 \rightarrow b + 1 > 0$。注意路径条件作为前提的一部分加到了公式中。

上面只是检查了程序的一条路径。对于这个程序，我们需要对程序的两条路径都做这样的判断才能确定程序中没有缺陷。对于一般带循环的程序，我们无法遍历所有的路径，所以符号执行所进行的分析只是针对部分路径而言，所以是所有执行轨迹的一个下近似。

8.2.3 指针

以上方法只是为基础数据类型创建了符号值，但很多时候我们的输入是一个指针，应该如何处理指针呢？我们还是可以为每个指针创建一个符号值。一旦我们需要对指针解引用的时候，我们就从当前的状态分裂出更多的执行路径，每一条路径代表该指针的所指向的一个可能性。比如考虑一个Java程序，输入中包含两个class A类型对应的指针x和y。假设我们首先对x解引用的时候，我们会在堆上创建一个新的对象，让x指向该对象，同时新入新的符号表示该对象的域。然后对y解引用的时候，我们会遍历所有可能性：(1) y和x指向的是同一个对象 (2) y指向的是一个新的对象。每个可能性对应一条新的执行路径。注意我们遍历所有可能性的时候只考虑输入符号解引用时创建的对象，而不考虑程序中间的创建的对象，因为输入的指针不可能指向这些对象。

8.2.4 特殊数据结构

部分数据结构在SMT求解器中提供了直接支持，比如数组、字符串等，对于这些数据结构我们也可以编码到求解器的对应理论直接求解。相应数据结构用SMT求解器提供的类型编码的条件是该数据结构内部元素是无法被指针指向的。比如Java的String类就符合这个条件，但C的字符串数组就不行。

Java的String类还具备创建之后就无法修改的特性，使得我们可以直接把指向String对象的指针建模成String值，很多时候可以显著简化约束和减少探索的状态。

8.3 符号执行的优化

上面介绍的是基础的符号执行算法。学术界也提出了很多符号执行的优化技术。这里介绍三种优化技术。

提前求解

上面介绍的符号执行过程是每次遍历一条路径，然后交给约束求解工具判断该路径对应的路径约束是否可以满足，以及满足的情况下对应的约束是否会被违反。但实际上，程序中很多路径的条件是互斥的，因此我们可以在每一次条件分支的时候都调用约束求解器。如果当前路径条件已经不可达，那么这条路径也就不用继续探索了。加入这条路径往后还有n个条件语句，那么就可以节约2^n次路径探索，形成可观的加速。这样的求解方式叫做提前求解(eager evaluation)。作为对应，之前在路径末尾求解的方式叫做推迟求解(lazy evaluation)。

但是，提前求解相比推迟求解，调用求解器的次数有了显著增加。因此，不一定能起到加速效果，所以提前求解和推迟求解是目前存在的两种求解策略，并没有哪种一定比另外一种更优。针对提前求解，一个可能的优化是采用SMT求解器的增量求解功能，每次针对新增的条件增量计算，提高求解速度。针对推迟求解，一种可能的方式是从冲突学习，将在下一小节介绍。

从冲突学习

如果路径上部分条件组合已经产生了冲突，那么其他包含这部分条件组合的路径也会产生冲突。为了避免对这些不可行的路径反复求解，我们可以在每次出现冲突的时候要求约束求解工具返回一个尽可能小的矛盾集，表示出现冲突的条件。之后某个路径条件也包含这几个出现冲突的条件时，我们就不用额外调用约束求解器，而是直接可以判断这个路径冲突。

注意采用这个方式并不能一定保障推迟求解优于提前求解，因为约束求解工具返回的矛盾集不一定是最小的。

动态符号执行

虽然现在约束求解工具的能力已经比较强，但是分析实际中的程序常常也会遇到很多约束求解工具不能求解的情况。常见的情况包括程序中使

用了约束求解不支持的运算符，比如求余数运算符；形成了约束求解工具无法求解的约束，比如高次方程；或者调用了没有源码或者依赖外部环境的函数，比如操作系统文件调用。

这些约束主要是不被约束求解工具支持，但大多数时候这些约束本身并不难解。比如代码中常有这样的约束：$x\%5 = y$。因为不支持取余数操作符，约束求解工具会直接无法求解，但实际上我们随机生成一个x的值，比如令x=1，就能把约束化简为$1 = y$，然后直接调用约束求解工具就能求解。

但是，一个路径约束通常并不包括这一单一约束，在之前可能还有比较复杂的其他约束。比如，一个完整的约束可能是$x < 0 \wedge x\%5 = y$。直接随机x的值会导致无法满足前面的约束。

因此，动态符号执行将一个具体执行过程和符号执行过程结合起来，当出现不可求解的约束的时候，就代入具体执行的值，就解决了这个问题。首先，动态符号执行对于输入随机生成一些具体值，然后顺着这次具体执行的路径执行符号执行。假如这次随机生成了$x = -1, y = 5$，满足了$x < 0$但没有满足$x\%5 = y$，那么符号执行也会收集到这样的路径约束：$x < 0 \wedge \neg(x\%5 = y)$。为了探索新的路径，动态符号执行会把最后一个没有被取反过的条件取反，这样就得到了$x < 0 \wedge x\%5 = y$。因为该约束无法求解，动态符号执行就对于%涉及的变量x引入具体值-1，得到$-1 < 0 \wedge -1\%5 = y$，化简得到$4 = y$，可以很容易被约束求解工具求解。然后再采用新得到的$x = -1, y = 4$开启新一轮动态符号执行，就可以再新的路径上执行符号执行。重复这个过程，可以不断遍历各种不同的路径，同时对于很多约束求解工具无法求解的约束也可以代入具体指求解。

第九章　程序合成

程序合成是程序分析的典型应用。一方面，程序合成可以直接转成程序分析问题求解。另一方面，实用的程序合成方法也大量使用程序分析的技术。同时，程序合成技术也在很多领域存在大量应用。因此，在本课程中，我们也介绍程序合成。

9.1　语法制导的程序合成

程序合成问题有多种不同的定义。经典的定义是语法制导的程序合成。本章中的大部分求解算法都是求解语法制导的程序合成问题。

给定一个程序空间$Prog$（通常用文法表示），一个逻辑规约$spec$，语法制导的程序合成寻找一个程序$prog$，满足$prog \in Prog \wedge prog \vdash spec$，即$p$符合程序文法且满足逻辑规约。

9.2　归纳程序合成的基本框架

程序合成可以看做演绎合成和归纳合成两大类。演绎合成采用一系列推导规则，从逻辑规约出发推导出一个满足规约的程序。归纳合成主要针对一系列样例，合成一个满足样例的程序。程序合成历史上是从演绎合成开始发展，但演绎合成的主要问题是很难写出一份全面的推导规则，满足各种情况的要求。进入新世纪以来，程序合成的发展逐步转向归纳程序合成。

归纳程序合成的基础框架可以看做是一个搜索框架。首先产生一个程序，然后查看程序是否满足规约的要求。如果满足，则输出该程序，如果不满足，就搜索下一个程序。这样，归纳程序合成就形成了两个关键问题：

1. 如何验证一个程序是否满足规约

2. 如何产生下一个被验证的程序

下面的章节分别回答这两个问题。

9.3 验证程序正确性

因为程序合成以逻辑规约作为输入，所以对于任意合成的程序，可以直接通过求解器判断规约程序是否满足规约。具体而言，给定任意关于输入输出的规约$spec$，程序$prog$满足该规约可以通过如下逻辑公式来表示：

$$out = prog(in) \rightarrow spec(in, out)$$

上述公式取反后，通过约束求解工具判断可满足性，如果不可满足说明公式成立。

但是，上述判断过程每次都要调用约束求解工具，开销较大。实际上，程序空间中大部分程序只需要用一些简单的测试就能排除掉。为了加速判断过程，反例制导的归纳合成*(CEGIS)*[20]采用测试集来加速这个过程。对于每个程序，首先在一组能较快执行完的测试集上做验证，如果通过了测试集再调用约束求解工具。因为测试执行较快，CEGIS能显著加速正确性验证的过程。

但这个方案带来一个问题，就是测试集从何而来？如果只是简单随机生成测试的话，一方面很难知道这些测试输入对应的输出是什么，另一方面测试集也不一定高效：新增加的测试并不一定能让测试集的检测错误程序的能力变得更强。CEGIS利用约束求解工具返回反例的能力来产生这个测试集。每次约束求解工具判断规约不满足的时候，可以同时返回一组不被满足的输入输出值，这一组值就可以作为测试样例加入到测试集中。这样加入的测试样例确保可以让测试集的能力变得更强，因为这个测试集至少能排除一个之前的测试集没有排除的程序。

采用了CEGIS框架之后，虽然程序合成整体仍然是针对一个逻辑规约在合成，但每一轮只需要满足一组测试样例，即程序合成的基本方法是归纳合成而不是演绎合成。

但是，虽然CEGIS保证新的样例可以增强测试集，但并不保证所增加的能力是最强的。最新工作也尝试从求解器获得多个样例，然后再通过一些启发式方法判断哪个样例能从程序空间中排除更多的错误程序，然后只把效果最好的样例加入测试集[10]。

9.4 枚举合成

现在我们来到第二个问题：如何产生下一个被验证的程序？最简单的方法是枚举法，即不断遍历程序空间中的程序直到找到正确的程序。

枚举法通常有自顶向下和自底向上两种形式。自顶向下枚举从非终结符开始顺着语法展开程序，展的过程中会遍历所有文法规则，一旦展到一个完整程序就验证。由于文法空间通常是无限的，所以自顶向下枚举通常设置一个遍历程序大小的上限。

自底向上枚举通常针对表达式程序，从最小表达式开始逐步组合成更大的表达式。一开始程序空间中只有x, y, 0, 1等原子表达式，每一轮自底向上枚举从程序空间中挑选一条语法规则和一些表达式，将这些表达式组合成更大的表达式。相比自顶向下，自底向上的好处是每次合成的都是完整的可执行的表达式，同时可以很容易控制大小从小到大合成。不过第一点好处与程序设计语言有关，如果程序空间不仅仅是表达式，就不一定成立了。

枚举法本身是比较慢的，但在枚举执行过程中可以通过各种方法减少枚举的数量。减少枚举的数量通常有两种方式，一种是等价性削减：当程序和之前枚举过的程序等价时，我们就可以削减程序。另一种是剪枝：给定一个不完整的程序，如果我们知道从该程序出发肯定不能到达一个正确程序的时候，我们就可以抛弃该程序。

等价性削减需要判断两个程序的等价性，一般有如下几种方式。

- 约束求解：调用约束求解工具来判断程序等价性。这样做代价比较大，通常较少采用。

- 预定义规则：通过预定义一些规则（如加法交换律）对程序进行变换，如果变换之后的程序文本相同，即说明两个程序等价。

- 可观察等价性（Observational Equivalence）：如果两个程序在当前测试样例上的返回值都相同，就认为两个程序等价。注意这个方法是不保证正确性的，但在CEGIS框架下并没有关系。因为我们的目标是满足当前测试，去掉测试上等价的程序并不会影响我们完成目标。

上述三种方式中，约束求解和预定义规则可以同时应用到自顶向下和自底向上，但可观察等价性需要完整的可执行程序，只能用于自底向上。

剪枝需要判断从当前程序出发不能到达最终程序。通常有如下几种方式：

- 约束求解：调用约束求解工具来判断从当前程序出发能否到达最终程序。因为调用约束求解工具的开销比较大，也可以用冲突制导的思想学习一些容易判断的约束，当新来的程序违反这样的约束的时候，就不调用约束求解器直接剪枝。[6]
- 抽象解释：在程序文法上做一个正向/反向抽象解释预分析，分析出针对当前测试样例输入，程序的非终结符的可能取值范围（自顶向下），或者针对当前测试输出，程序的非终结符的必须提供的输入值（自底向上）。然后当枚举到一个程序的时候，根据预分析的结果判断。[24]

以上两种方式都可以同时用于自顶向下和自底向上，但通常自顶向下的效果更好，分析也更容易设计。

9.5 归一程序合成

虽然程序合成理论上可以针对任意的编程语言进行合成，但实际上大部分程序合成方法都针对没有循环的语言，因为带循环的程序通常SMT求解器无法直接验证。如果一个语言中没有循环，其实就只剩下顺序和分支两种控制结构。归一程序合成就是针对这样的程序设计的一个专用合成算法。

归一程序合成的主要思想是，如果一个程序中只有顺序和分支两种结构，其实可以把程序的合成分成两部分，一部分采用条件语句来对输入进行分类，另一部分不含条件语句，用来将具体类别的输入转换成输出。这两部分可以分别枚举，减少枚举的空间。具体而言，归一程序合成方法合成符合如下语法的程序：

```
Expr  →  if BoolExpr then Expr else Expr
      |  AtomicExpr
```

其中BoolExpr是条件表达式，AtomicExpr是不含If的原子表达式。

其他程序可以换成这种形式。比如`1 + (if b then x else y)`可以换成`if b then (1+x) else (1+y)`。

归一化程序合成遵循CEGIS框架，对一组对于条件表达式和原子表达式分别枚举。首先，归一化程序合成枚举原子表达式。对于每个枚举到的

原子表达式，合成系统检查该原子表达式可以通过多少输入输出样例。然后，合成系统选择一组能覆盖所有输入输出样例的原子表达式。

接下来合成系统要找到一组条件，这组条件可以区分不同的样例，将样例和原子表达式对应起来。这是一个典型的决策树构建问题[3]。归一化程序合成系统通常枚举一系列的文字（不带与、或、非等逻辑表达式的条件），然后尝试用这组条件组成决策树，决策树最后分类的目标就是采用什么原子表达式将输入转换为输出。

由于采用了CEGIS框架，归一化程序合成的一个关键是在每轮合成尽量小的程序，这样有望提升程序的泛化能力，使得采用较少的CEGIS论数就可以达到收敛。为了保证这个问题，归一化程序合成在寻找能覆盖输入输出样例的所有表达式的时候，需要同时限制表达式的大小和所使用的表达式的个数，然后用迭代加深的方法依次查找。在构造决策树的时候，也同时限制文字的个数和单个文字的大小，然后用迭代加深的方法搜索。[11]

9.6 基于约束求解的程序合成

由于约束求解工具在各类问题上已经展现出良好性能，所以研究人员试图在更大的范围发挥约束求解的能力。约束求解程序合成的基本思路是，将程序合成问题整体转换为一个约束求解问题，然后调用SMT求解器求解。

将程序合成转换为约束求解问题的一个基本方法是，将推导规则看做数量有限的构件，程序合成问题就是找到合适的方法将这些构件组合起来，使得满足规约。由于构件的数量是有限的，所以组合方式的空间也是有限的，因此可以转成SMT问题。基于该方法转换成约束求解问题的程序合成方法称为基于构件的程序合成[9]。

具体而言，该方法假设合成的都是没有副作用也没有高阶函数的表达式，每条产生式都是产生表达式中的一个运算操作，并且都有一个返回值。这样，表达式的执行过程中，每个非终结符的位置都一定会产生一个值。我们可以把产生式右边的值看做输入，把产生式左边的值看做输出。比如，

$$Expr_{o1} \rightarrow Expr_{i11} + Expr_{i22}$$

可以看做对应一个叫做“加法”的构件，该构件有两个输入$Expr_{i11}$和$Expr_{i22}$，有一个输出$Expr_{o1}$。其中，$o1$、$i11$、$i12$是输入输出位置的编号，$o1$表示1号产生式的输出，而l_{i12}表示1号产生式的第2个输入。如果有一个表达式$x +$

x，可以看做是由以下三条产生式对应的构件组合而成，而组合的方式是把$o2$连接到$i11$，把$o3$连接到$i22$。

$$Expr_{o1} \to Expr_{i11} + Expr_{i22}$$

$$Expr_{o2} \to x$$

$$Expr_{o3} \to x$$

为了表示构件之间输入输出之间的连结，我们对每个构件的输入输出都赋予一个单独的标签变量。一个构件的输出标签变量和另一个构件的输入标签变量的值相同，表示它们之间有连接。比如，对于产生式：

$$Expr_{o1} \to Expr_{i11} + Expr_{i22}$$

我们可以添加输入标签变量l_{i11}和l_{i12}，以及输出标签变量l_{o1}。除了构件上的标签变量，我们还需要添加一个标签变量l_o表示整个表达式的输出。这样，程序合成的过程就变成了一个为这些变量赋值的过程。

为了保证合成的程序是满足规约的，我们还需要添加一些变量。这里我们还是假设CEGIS框架，每轮我们要满足的是一组测试。针对测试t，我们对于每个产生式左边和右边的非终结符都添加一个变量，表示这些非终结符在这组测试执行过程中的取值。在上面这个例子中，我们需要添加的变量是v^t_{i11}、v^t_{i12}和v^t_{o1}。对于程序中的输入变量，比如x、y，我们也添加相应的变量v^t_x和v^t_x，表示对应输入的值。

最后，我们还针对每个产生式左边和右边的非终结符都添加一个常量，表示这个终结符本身。这个常量主要是为了保证合成的程序符合语法。在上面的例子中，我们可以添加N_{i11}、N_{i12}和N_{o1}，表示$Expr_1$、$Expr_2$和$Expr_0$的非终结符，并且$N_{i11} = N_{i12} = N_{o1} = Expr$。

有了基本的变量和常量之后，接下来我们添加一些约束，保证生成的程序是正确的。首先，我们要要求生成的程序符合语法，所以我们要求连结上的输入输出应该有相同的非终结符。对于任意输入输出编号o和i，我们有

$$l_o = l_i \to N_o = N_i$$

然后，我们要求生成的程序要满足测试。第一，我们需要对测试输入输出产生约束，比如：

$$v^t_x = 1 \wedge v^t_y = 2 \to v^t_o = 2$$

第二，我们需要根据产生式的语义产生约束。比如，对于1号加法产生式和任意测试t，我们产生

$$v_{o1}^t = v_{i11}^t + v_{i12}^t$$

第三，我们需要保证连结上的构件输入输出的值是一样的。对于任意输入输出编号o和i和任意测试t，我们有

$$l_o = l_i \rightarrow v_o^t = v_i^t$$

最后，我们需要保证所有构件连结成一个合法的表达式。第一我们需要保证所有输入标签变量和输出标签变量的值都落在同一个范围内。假设总共能使用n个构件，那么对于任意的输出位置编号o，我们有如下约束：

$$l_o \geq 1 \wedge l_o \leq n$$

对于任意的输入位置编号i，我们有如下约束：

$$l_i \geq 1$$

对于输入位置编号我们不需要限制其上限是因为稍后我们会用别的约束来限制上限。

第二我们要保证所有输出标签的编号都是不同的，这样一个输入永远不会接受两个输出。对于任意两个的输出编号oi和oj，我们有如下约束：

$$l_{oi} \neq l_{oj}$$

第三，我们还需要保证最后形成的是一个树形结构的表达式，即连结必须是无环的。对于任意表达式编号x和该表达式上的输出位置ox和输入位置ixy，我们用如下约束保证无环的性质：

$$l_{ixy} < l_{ox}$$

9.7 基于空间表示的程序合成

基于可观察等价性剪枝的程序合成方法是一种有效的策略，因为大量程序在样例上都返回相同的结果。比如，当某个非终结符对应类型是布尔类型的时候，从该终结符展开的不同程序最多只能有两个返回值。但该方法有两个问题：1. 当样例数量增多的时候，返回相同结果的程序数量就

显著减少了。比如当前有n个样例，但一个布尔表达式可能的结果就变成了2^n中。2. 在CEGIS框架中，每轮都会新增一个样例。但标准的可观察等价性剪枝无法重用之前的计算，每次都会对之前的样例重新计算。

基于空间表示的合成[8, 22]可以有效解决上述两个问题。首先，该方法基于某种数据结构来表示程序的集合，然后对于每个样例都产生一个满足该样例的集合表示，最后对所有集合求交得到满足所有样例的集合，再从集合中随机采样一个程序就是最终答案。该方法可以有效避免可观察等价性剪枝的问题。首先，程序集合针对单个样例产生，可以有效利用可观察等价性剪枝的优势，同时在对多个程序集合求交的时候，因为我们考虑的只有至少通过一个测试的程序，程序空间大大小于所有程序的空间。其次，在CEGIS框架中，每轮新增样例的时候，只需要对新增的样例产生一个集合表示，然后和之前的集合求并即可。无需重新计算之前的样例。

这里的核心问题是，用什么样的数据结构来表示程序空间，既可以做到对单个样例高效计算出整个程序空间，又可以对多个样例求交。为了回答这个问题，我们可以回顾一下可观察等价性的定义：如果对样例输入返回相同的输出，同时又是从同一个非终结符展开的两个程序是等价的。换句话说，对于这样的一组程序，我们在后续合成中只需要关注输出和非终结符这两个信息，也就是说我们可以用加上返回值的非终结符来表示该程序集合，如：

$$[2]Expr$$

该表示也可以整体看做一个新的非终结符，称为带约束非终结符。

为了记录带约束非终结符对应的程序集合，我们可以把原上下文无关文法产生式扩展为带约束的上下文无关文法产生式，比如下面的文法产生式表示返回2的两个表达式可以从两个返回1的表达式相加得到：

$$[2]Expr \rightarrow [1]Expr + [1]Expr$$

之前的文法产生式$Expr \rightarrow Expr + Expr$就叫做该产生式对应的原始文法产生式。

这样，我们可以用一组带约束的上下文无关文法产生式来记录符合某个样例的程序空间，然后对这些上下文无关文法产生式求交，就可以得到最终的程序空间。那么这里有两个问题需要解决：1. 如何产生单个样例对应的文法产生式集合？2. 如何对多个文法产生式集合求交？

和枚举的情况类似，产生符合单个样例的文法也可以采用自底向上和自顶向下两种方式进行。自底向上的方式和普通的自底向上枚举相似。该

算法维护一个非终结符集合和产生式集合。初始非终结符集合只包括输入变量，比如

$$[2]x, [1]y$$

然后系统尝试反复原始应用推导产生式从非终结符集合得到新的非终结符和产生式。比如，应用$Expr- > x$产生式到$[2]x$得到新的产生式

$$[2]Expr \rightarrow [2]x$$

和新的非终结符

$$[2]Expr$$

如果再应用$Expr- > Expr + Expr$到$[2]Expr$可以得到产生式

$$[4]Expr \rightarrow [2]Expr + [2]Expr$$

和非终结符

$$[4]Expr$$

不断重复这个过程，直到非终结符不再增加，或者达到搜索上限。注意起始符号和期望输出产生后，就不再对其应用别的产生式。产生出起始非终结符和期望输出后，再删除从起始符号和期望输出不可到达的产生式，就得到了对单个样例的产生式集合。

自顶向下的方式假设存在一个witness函数，该函数在给定非终结符和期望输出的时候，给出该终结符的可能一步展开和展开的非终结符上的对应输出。比如，给定

$$[2]Expr$$

witness函数可能给出下面的输出

$$[2]x, [2]y,$$

$$[0]Expr + [2]Expr, [1]Expr + [1]Expr, [2]Expr + [0]Expr,$$

$$ite([true]BoolExpr, [2]Expr, [*]Expr), ite([false]BoolExpr, [*]Expr, [2]Expr)$$

其中[*]表示任意的返回值。

自顶向下方式就反复调用witness函数来产生新的产生式和非终结符直到收敛，或者达到搜索上限。注意这个过程中可能产生无法构造的非终结符和期望输出，即witness函数返回为空。对于这类非终结符和使用他们的产生式，最后要再加上一轮进行删除。

对所有单个样例都产生出文法产生式集合后，剩下问题是如何对产生式求交。注意一般的上下文无关文法对求交运算是不封闭的，即求交之后不一定还是上下文无关语言。但我们这里求交想要保留的其实是原始文法对应的AST树。如果我们认为带约束的文法表达的是原始文法的AST树的集合，然后我们试图对这样的集合求交，这样的求交操作仍然是封闭的，可以进行。

具体而言，给定原始产生式

$$N_0 \to P(N_1, N_2, \ldots)$$

如果两个文法中分别存在两个带约束的产生式

$$[c_0]N_0 \to P([c_1]N_1, [c_2]N_2, \ldots)$$

$$[c_0']N_0 \to P([c_1']N_1, [c_2']N_2, \ldots)$$

那么求交得到

$$[c_0, c_0']N_0 \to P([c_1, c_1']N_1, [c_2, c_2']N_2, \ldots)$$

对两个文法中所有这样的产生式求交，然后删掉无法从起始符号到达的产生式即可。由于这样会导致较多无效运算，从效率考虑，可先从起始符号出发，只对从起始符号可达的产生式求交。

9.8 基于概率的程序合成

基于概率的程序合成将经典的程序合成问题扩展了一个概率模型，称为*程序估计问题*。具体而言，给定一个程序空间$Prog$，一条规约$Spec$，一个概率模型Pæ$Prog \to [0,1]$，程序估计问题是求一个程序$prog$，满足

$$prog = \text{argmax}_{prog \in Prog \wedge prog \vdash Spec} P(prog)$$

这里给定不同的P，可以实现不同的任务。比如，如果P估计程序满足给定自然语言需求的概率，那么可以完成从自然语言的代码生成。这样的概率模型可以从一个包含自然语言作为输入的元概率模型中实例化得到。如果P估计程序满足输入规约的概率，那么可以用来加速传统程序合成。

这一小节介绍两种求解程序估计问题的方式，第一种基于枚举，另一种基于空间表示。

9.8.1 基于枚举的方法

在原始程序估计问题中，概率模型被定义为从程序到$[0,1]$之间实数的函数。但是，如果我们不对概率模型的结构做任何假设，我们可能很难给出任何高效的算法。

一种对概率模型进行假设的方法是把程序看做一组文法规则的序列。任何程序都是从起始符号开始，不断应用文法规则展开非终结符得到的。如果我们固定某种非终结符的选择顺序，比如每次都选择当前最左边的非终结符展开，我们就可以把程序表示为一个文法规则的序列。换句话说，一个程序的概率就等于该文法规则序列出现的概率。

$$P(prog) = \prod_i P(rule_i \mid rule_1, \ldots, rule_{i-1})$$

我们也可以推导出$rule_1, \ldots, rule_{i-1}$对应的部分语法树$prog_i$，将该式子转换成：

$$P(prog) = \prod_i P(rule_i \mid prog_i$$

这样，我们就只需要给出一个函数返回$P(rule_i \mid rule_1, \ldots, rule_{i-1})$或$P(prog) = \prod_i P(rule_i \mid prog_i$，就得到了整体的概率模型。该函数可以采用任意统计模型或机器学习模型实现。比如，我们可以Seq2Seq的深度学习模型来实现。我们也可以从$prog_i$上提取特征，采用传统机器学习模型来实现。

有了这样的概率模型之后，我们就可以把程序合成问题建模成一个图上的带权路径查找问题。图上的节点为部分或完整程序，边为文法规则。如果一个部分程序p_1采用文法规则r展开按某种固定策略选定的节点之后就成为了p_2，那么p_1和p_2之间就存在边r。边r的权为$P(r \mid p_1)$。路径的权为路径上边的权的乘积。目标节点为满足规约的完整程序。该路径查找问题就是要找到一条权最大的能达到目标节点的路径。

该问题可以用任意路径查找算法求解，比如迪杰斯特拉算法、定向搜索、A*算法等。A*算法的应用需要一个额外的启发式函数，对于简单的概率模型，该函数可以通过在概率模型和程序文法上做静态分析得到。但对于较复杂的模型应用A*算法则比较困难。

9.8.2 基于空间表示的方法

和前一小节类似，这里我们考虑自顶向下的文法构造方法。自定向下构造文法和之前的概率计算的矛盾是，如果一个非终结符展开了两个非终

结符，比如：

$$[acc]S \to [a]S + [cc]S$$

这里的$[a]S$和$[cc]S$将分别调用witness函数展开，形成两个独立的子问题。但如果我们考虑类似之前的概率模型，那么在计算展开$[cc]S$的产生式的概率的时候，我们需要知道$[a]S$是由哪个产生式展开的，导致概率无法独立计算。

为了解决这个问题，我们首先简化概率模型，使得产生式的概率只依赖部分AST中的祖先节点，不依赖兄弟节点是如何展开的。比如，$P(x + y) = P(E \to E + E \mid \bot)P(E \to x \mid E)P(E \to y \mid E)$。这样，在记录祖先节点的情况下，$[a]S$和$[cc]S$的展开概率就可以独立计算了。我们称这样的概率模型为自顶向下的概率模型。为了让这个概率模型容易统计，我们通常只保留有限个祖先节点，比如最近k个。

有了自顶向下的概率模型，我们可以对原来的非终结符进行改造，除了包含返回值，我们还加上之前的祖先节点，比如$[ac|S, X]S$表示返回ac，由S展开，祖先节点从高到底分别是$[S, X]$的所有表达式。这样，每个终结符所对应的概率最大的表达式就变成了一个独立子问题，可以单独求解。这样，相对基于枚举求解程序估计问题的方法，我们就实现了加速。

但目前产生的子问题比针对传统程序合成问题的空间表示法要多，无法对传统空间表示法进行加速。为了进一步加速，我们进一步引入一个迭代加深的过程。即每次我们设置一个最优程序的概率下界，然后逐步放宽。比如，一开始是0.1，之后每次除以10。这样，我们每轮只考虑出现概率大于等于这个概率下界的程序，就可以大大减少搜索空间。

具体而言，如果进一步把非终结进行改造，加上概率下界，比如$[acc||0.3]S$表示概率下界为0.3，返回acc，由S展开，没有祖先节点的表达式。这样，我们在展开这个非终结符的时候就可以剪枝了。比如，如果$P(S \to x \mid \bot)$的概率小于0.3，那么就不用尝试这个产生式来展开了。

更进一步，我们可以为每个终结符和每个祖先序列分析一个概率上界，即在给定祖先序列下，从这个终结符能展开的程序的最大概率。继续上面的例子，如果$P(S \to S + S \mid \bot)$的概率等于0.9，同时在祖先S继续展开S的概率上界为0.5，那么0.9*0.3*0.3=0.225已经小于0.3了，就可以放弃这条展开式。

这个方法虽然能实现根据概率的有效剪枝，但概率下界很少相同，不利于复用子问题。因此我们需要能复用概率下界不一样的子问题。让我们

考虑两个除了概率下界不同以外，其他都一样的子问题$(P, 0.2)$和$(P, 0.1)$。这里有两种情况。

- 第一种情况，如果$(P, 0.2)$先于$(P, 0.1)$求解。那么前者如果有解，则也是后者的解。如果前者无解，那么可以更新P的估价函数。
- 第二种情况，如果$(P, 0.1)$先于$(P, 0.2)$求解。那么前者如果有解，则用前者解的概率和后者的概率下界对比，如果高于后者的概率下界，则前者的解也是后者的解，否则后者无解。如果前者无解，则后者无解。

基于以上策略，我们仍然能在很大程度上复用子问题，实现对传统给予表示的程序合成的加速。

第十章　错误定位

在软件开发过程中，错误定位是一个至关重要的环节。当软件系统中出现缺陷时，开发人员需要快速、准确地定位到导致问题的代码片段，并进行修复。错误定位技术旨在帮助开发人员在复杂的代码库中快速找到问题的根源，从而提高调试效率和软件质量。

10.1　基于测试的错误定位：问题定义

错误定位最常见的类型是基于测试的错误定位。给定如下的输入：

- 软件系统的源代码
- 一组测试用例，其中至少有一个测试用例没有通过

基于测试的错误定位输出一个可能有错误的程序元素列表，这些元素根据出错概率进行排序。程序元素根据错误定位方法的不同可以定义在不同的级别上，如表达式、语句、方法、类、文件等。

10.2　基于测试的错误定位：程序切片

程序切片是一种常用的错误定位技术，它通过分析程序的控制流和数据流，找到与特定语句或变量相关的所有语句。程序切片可以帮助开发人员缩小错误范围，从而更快地定位问题。

切片准则是程序中的一条语句或某个语句中的变量，通常是在错误发生的位置。例如，一个失败的断言语句或抛出的未捕获异常。给定切片准则，程序切片问题就是要找到所有可能影响S或者S可能影响的语句。

根据切片的方向，程序切片可以分为：

- **后向切片**：找到所有影响切片准则的语句。
- **前向切片**：找到所有切片准则可能影响的语句。

根据切片的类型，程序切片可以分为：

- **静态切片**：找到在任何输入下可能被影响的语句。
- **动态切片**：给定特定输入，只考虑该输入下可能被影响的语句。

10.2.1 程序切片示例

考虑以下代码片段：

```
1  a = 0;
2  b = 3;
3  if (x > 0) {
4      a = 1;
5      b++;
6  } else {
7      a = 2;
8      b--;
9  }
10 z = a;
11 b = b + z;
12 b = b - a;
13 if (b > 0) {
14     // ...
15 }
```

假设切片准则是第11行`b = b + z;`，则静态后向切片包括所有可能影响 `b` 和 `z` 的语句，即第3, 4, 5, 7, 8, 10行。注意第10行使用了a的值，所以对a的赋值要被包括进来。第三行决定后面哪些赋值语句被执行，所以也被包括进来。第1行和第2行不被包括因为不管他们赋了什么值，之后都会被覆盖。

假设输入是x=10，则动态后向切片包括第3, 4, 5, 10行。与静态切片不同，在实际运行中没有被执行的部分被排除。

10.2.2 依赖关系和切片计算方法

程序切片依赖于以下三种依赖关系：

- **数据依赖**：语句 a 读取了由语句 b 写入的变量。
- **控制依赖**：语句 a 的执行由语句 b 的执行结果决定。
- **同步依赖**：在多线程程序中引入的依赖。

以上情况仅考虑了语句间的依赖关系。如果切片准则是变量的时候，还要考虑语句和变量之间的依赖关系。

在计算切片的时候，一般把程序上的依赖关系表示为一个图，称为程序依赖图。程序依赖图是一个有向图，节点为语句，边为依赖关系。切片从切片准则开始求图的可达性，然后保留所有可达的语句，就实现了切片。

10.2.3 构造过程内静态依赖关系

构造过程内静态依赖关系包括数据依赖和控制依赖的识别。

对于数据依赖，其本质上是Def-Use关系，可以通过可达定值分析来进行。之前讲的可达定值分析主要用于栈上的变量，这里还要考虑通过指针间接访问的值。一般来说，我们假设已经完成了指针分析，得到了指针变量所指向的内存位置。之后我们就把内存位置当作变量，然后执行可达定值分析。根据需要，可达定值分析可以是流敏感也可以是流非敏感的。

识别控制依赖对于结构化程序相对简单，可以直接从程序的语法结构中提取。If,switch每个分支中的语句依赖if条件。While, for循环体中的语句依赖循环条件。

对于非结构化程序，控制依赖的识别则更为复杂。非结构化程序中的控制流可能包含跳转语句，如 goto。

考虑以下非结构化程序：

```
A: if (x > 0) {
       goto C;
   }
B: a = 1;
C: b = 2;
```

由于B是否执行由A的执行结果确定，所以B控制依赖A。

在控制依赖的计算中，反向支配是一个关键概念。反向支配是指在控制流图中，从某个节点 A 到程序出口的所有路径都必须经过节点 B。如果 B 反向支配 A，那么 B 的执行结果会影响 A 的执行。如果 B 反向支配 A，并且 $B \neq A$，则称 B 严格反向支配 A。

在反向支配的基础上，可以进一步定义反向支配边界。节点 B 的反向支配边界包括所有节点 A，使得 B 反向支配 A 的某个后继节点，但 B 不严格反向支配 A。反向支配边界存在高效算法，具体可以查看编译相关教材[1]。

基于上述概念，控制依赖可以定义为：语句 B 控制依赖于语句 A，当且仅当 A 位于 B 的反向支配边界中。

在上面这个例子中，节点B反向支配自己，但不严格支配A，所以A在B的反向支配边界中，因此B控制依赖A。

10.2.4 过程间依赖关系

过程间的数据依赖包括函数调用和返回值带来的数据依赖，以及通过读写全局变量和堆上数据的数据依赖。我们还是首先假设有一个控制流分析帮我们分析除了调用点和被调用函数之间的关系。对于前者，被调函数入口语句依赖调用语句，调用语句依赖被调函数返回语句。对于后者，我们需要分析出被调函数中对全局变量和堆上内存位置的读写，进行过程间的可达定值分析。

过程间的控制依赖主要是函数中所有顶层语句依赖函数入口，函数入口依赖调用该函数的语句。这样，如果调用语句的执行再依赖于别的语句，可以形成可传递的控制依赖关系。

10.2.5 动态依赖关系

动态依赖关系通常比静态依赖关系容易，因为我们可以拿到程序执行过程中的精确信息。首先我们需要在程序运行时记录每条语句的执行情况，包括读写的内存地址和执行顺序。这通常通过代码中插装追踪代码来实现。在数据依赖上，语句执行a依赖于语句执行b等价于a读了一个最近由b写的内存位置。注意这里的内存位置是指执行过程中的具体的地址，而不是静态分析出来的抽象内存位置。而控制依赖则是静态控制依赖中实际被执行的部分。

10.2.6 程序切片在错误定位中的应用

由于出错的语句只可能出现在动态反向切片中，所以动态反向切片可以有效缩小错误语句的范围，实现错误定位。

在错误定位中，切片准则通常选择为发现错误的位置，如失败的断言语句或抛出的未捕获异常。这些位置通常是运行追踪信息中的最后一条语句。从错误位置出发，构造动态反向切片，就实现了错误定位。

10.3 基于测试的错误定位：基于频谱的错误定位

基于频谱的错误定位是一种基于测试覆盖信息的错误定位方法。该方法通过计算每条语句和测试的关联，计算程序元素的“怀疑度”。怀疑度越高的程序越有可能有错误。

程序频谱是指程序执行过程中的统计量，最早由威斯康星大学的Tom Reps在1997年处理千年虫问题时提出。佐治亚理工大学的James Jones和Mary Jean Harrold等人于2002年将Tom Reps的方法通用化，提出了基于频谱的错误定位技术。

基于频谱的错误定位的基本思想是：被失败的测试用例执行的程序元素更有可能有错误，而被成功的测试用例执行的程序元素更有可能没有错误。基于该思想，基于频谱的错误定位首先对于每个程序元素统计下面四个数据：

- a_{ef}：执行程序元素 a 的失败测试的数量
- a_{nf}：未执行程序元素 a 的失败测试的数量
- a_{ep}：执行程序元素 a 的通过测试的数量
- a_{np}：未执行程序元素 a 的通过测试的数量

然后，基于频谱的错误定位基于公式计算出每个程序元素的怀疑度。一个最常用的怀疑度计算公式为Ochiai：

Ochiai

$$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$$

该公式可以看做是两个分数乘起来。第一个是所有失败测试用例中执行到元素a的比例，另一个是所有执行元素a的测试用例中失败的比例。因此，执行a的失败测试用例越多，占比越高，a元素的怀疑度越高。

学术界还提出了很多其他公式，一些常见的公式如下：

Tarantula

$$\frac{a_{ef}}{a_{ef}+a_{nf}}\left(\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}\right)$$

Jaccard

$$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$$

D

$$\frac{a_{ef}^{*}}{a_{nf}+a_{ep}},\quad *\text{通常设置为2或者3}$$

Naish1

$$\begin{cases}-1 & a_{nf}>0\\ a_{np} & a_{nf}=0\end{cases}$$

10.4 基于测试的错误定位：基于状态覆盖的错误定位

在基于频谱的错误定位中，错误定位是依靠通过测试和失败测试在测试覆盖上的差异来寻找出错程序元素的。但是，如果通过的测试用例和失败的测试用例在测试覆盖上没有差异，基于频谱的错误定位方法就无法发挥作用了。然而，虽然在语句覆盖上没有差异，如果我们详细分析通过测试用例和失败测试用例覆盖的程序状态，我们会发现它们覆盖的程序状态是不同的。基于状态覆盖的错误定位方法就是基于这个思想。

例子 考虑以下代码片段：

```
a = abs(a);
...
if (...) {
    b = sqrt(a);
}
```

在这个例子中，首先 a 的值经过 abs 函数被转成了正数，然后再对 a 进行开方。但是，abs 函数并不能把所有输入都转换为正数。因为在补码的二进制表示中，负数比正数多一个，所以 abs(-2^{31}) 是无法转换为正数的。在大多数数学库的实现中，abs 会原样返回该负数，导致之后调用 sqrt 出错。假设所有通过的测试样例和失败的测试样例都进入了该if分支，那我们无法从测试覆盖上区别出不同程序元素之间的差异。

但是，如果我们考虑程序状态，我们会发现通过的测试样例和失败的测试样例覆盖的程序状态是不同的。通过的测试样例中，a 的值是非负数，而失败的测试样例中，a 的值是负数。如果我们能对程序的执行状态进行详细的划分，再在程序状态上计算覆盖并采用类似于基于频谱的公式来计算怀疑度，我们就能找到导致错误的程序状态。

具体而言，基于状态覆盖的错误定位会设置一系列预定义谓词，然后在程序的若干关键位置插入这些谓词。常见的谓词包括：

- 对整形变量a: $a > 0, a < 0, a == 0$
- 对布尔变量b: $b == true, b == false$
- 对对象o: $o == null, o! = null$

比如，对于上面这个例子，我们会在 abs 函数调用行插入如下谓词：

- $a > 0$
- $a == 0$
- $a < 0$

每个谓词就代表了一个抽象状态。对于每个抽象状态，我们同样测量之前的四个值，表示测试对该抽象状态的覆盖情况。

- a_{ef}：执行到谓词 a 的位置并且满足谓词的失败测试的数量

- a_{nf}：未执行到谓词 a 的位置或者执行到但不满足谓词的失败测试的数量
- a_{ep}：执行到谓词 a 的位置并且满足谓词的通过测试的数量
- a_{np}：未执行到谓词 a 的位置或者执行到但不满足谓词的通过测试的数量

然后，我们再用于基于频谱的公式来计算每个抽象状态的怀疑度。由于$a < 0$这个状态只被失败的测试覆盖，所以它的怀疑度会很高，从而我们可以判断出$a < 0$是缺陷状态，引入该状态的语句为缺陷语句。

10.5 基于测试的错误定位：基于变异的错误定位

在基于频谱的错误定位中，不同测试只要覆盖了语句，对结果的效果就是相同的。但不同测试受同一个语句的影响是不同的，不同测试触发错误的概率、传播错误的概率、捕获错误的概率都不同，基于频谱的错误定位无法利用这些精细信息。基于变异的错误定位依靠变异分析捕获了测试和语句之间的关系，从而更好地实现错误定位。在实际测试中，基于变异的错误定位可以在语句级别实现显著更好的错误定位，不过时间开销也显著更高。

变异分析是一种广泛应用于测试领域的技术，用于衡量测试集的好坏。变异是指对程序的任意随机修改，变异分析通过收集变异后程序上原测试用例的执行结果来评估测试集的质量。如果一个测试集中的任意测试在变异后的程序上执行失败，称为该变异体被这个测试杀死。能杀死越多变异体的测试集越好。

基于变异的错误定位把变异分析应用到了错误定位上。如果我们对一个语句进行大量变异，同时观察特定测试的运行结果变化情况，我们可以得到该语句和该测试之间的关联，并利用该信息完成错误定位。基于变异的错误定位有两个代表性方法，Metallaxis[17]和MUSE[16]。我们这里以MUSE为例介绍基于变异的错误定位。

MUSE方法基于如下两个假设：

- **假设1**: 变异错误语句时导致失败测试变成通过的概率 > 变异正确语句时。

- **假设2**: 变异正确语句时导致通过测试变成失败的概率 > 变异错误语句时。

MUSE方法的基本思路是，首先对程序进行变异，然后对每个变异体执行测试集，记录每个测试用例在变异前后测试结果的变化。给定变异体m，该方法计算两个数据：m_{f2p}，表示m上从失败变成通过的测试数，m_{p2f}，表示m上从通过变成失败的测试数。

然后MUSE方法采用如下公式计算每个变异体的怀疑度：

$$m_{f2p} - m_{p2f}\frac{\sum_m m_{f2p}}{\sum_m m_{p2f}}$$

根据假设1，如果一个变异体导致了更多的失败测试变成通过，那么这个变异体就更有可能是错误的，所以m_{f2p}越大，越有可能有错。根据假设2，如果一个变异体导致了更多的通过测试变成失败，那么这个变异体就更有可能是正确的。所以m_{p2f}越大，该变异体越有可能是正确的。但是由于这两个值通常不在同一个量级，直接相减的话会导致结果被其中较大的量主导，所以上述公式首先把两个值正规化到同一个量级，然后再相减。

最后，每个程序元素的怀疑度就是该元素上变异体的怀疑度的平均。

10.6 基于测试的错误定位：错误概率建模

如前所述，基于变异的错误定位方法通过变异来获取每个语句和测试之间的关联性。但是，由于变异分析的高计算开销，导致基于变异的错误定位方法执行效率很低。实际上，程序语句和测试执行之间的关联可以通过程序语义进行建模。所以一个思路是基于程序语义，建立概率模型捕获测试与语句出错的关系，同时基于测试结果计算语句出错的后验概率，从而实现错误定位。该方向的代表性方法是SmartFL[26]。利用概率建模，SmartFL同时在效率上和效果上都超过了基于频谱的错误定位方法和基于变异的错误定位方法。

让我们首先考虑下面的程序作为例子：

```
1    public class CondTest {
2        public static int foo(int a) {
3            if (a <= 2) { // buggy, should be a < 2
4                a = a + 1;
```

```
5               }
6               return a;
7           }
8
9           @Test
10          void pass() {
11              assertEquals(2, foo(1));
12          }
13
14          @Test
15          void fail() {
16              assertEquals(2, foo(2));
17          }
18      }
```

在该程序中，`foo` 方法中的条件判断语句 `if (a <= 2)` 是错误的，应该是 `if (a < 2)`。在测试 `fail` 中，`foo(2)` 的返回值是3，与期望值2不符，导致测试失败。我们的目标是找到导致测试失败的语句，即条件判断语句 `if (a <= 2)`。

SmartFL方法首先添加一系列随机变量。对于第i行的语句和每个测试t，添加随机变量V_i^t，表示测试t中第i行的表达式计算结果是否正确；对于第i行的语句添加随机变量S_i，表示第i行的语句是否正确。

一开始，我们对哪些语句可能有错并没有先验知识，所以我们假设每个语句都有相同的出错概率，即：

$$P(S_3) = 0.5, \quad P(S_4) = 0.5$$

首先我们考虑通过的测试，用p表示。我们总是假设测试的输入是正确的，所以我们有如下概率：

$$P(V_2^p) = 1$$

接着我们考虑测试在代码第三行的执行。如果第三行本身的代码是正确的，同时第三行中读入的变量值也都是正确的，那么第三行表达式的计算结果也是正确的。所以我们有：

$$P(V_3^p \mid S_3 \wedge V_2^p) = 1$$

如果第三行本身的代码或者读入的值有错，那么第三行表达式的计算结果也有可能是错的。由于错误的分布不容易建模，我们可以简单假设这一行发生的错误会导致返回一个均匀分布上的随机值。由于表达式的返回类型是布尔型，那么这一行的表达式计算结果是正确的概率是0.5。所以我们有：

$$P(V_3^p \mid \neg S_3 \wedge \neg V_2^p) = 0.5$$

接下来我们考虑测试在代码第四行的执行。第四行和第三行情况不同的是，第四行位于条件判断语句的分支中。因此，除了第四行读入的变量，我们还要考虑第三行条件表达式的计算结果是否正确。因为如果不正确的话，第四行根本就不会执行。另一方面，由于第四行表达式的返回类型是int，所以如果产生一个随机值的时候，这个随机值恰好正确的概率很低。这里我们用0.01表示。所以我们有：

$$P(V_4^p \mid S_4 \wedge V_2^p \wedge V_3^p) = 1$$

$$P(V_4^p \mid \neg S_4 \wedge \neg V_2^p \wedge \neg V_3^p) = 0.01$$

以此类推我们可以完成对于两个测试在所有语句上的执行的建模。最后，根据测试结果，我们可以通过贝叶斯定理计算出每个语句出错的后验概率。

$$P(S_4 \mid V_4^p, \neg V_4^f) = ?$$

$$P(S_3 \mid V_4^p, \neg V_4^f) = ?$$

这里的计算结果会告诉我们S_4正确的概率更高，S_3出错的概率更高，从而我们可以确定第三行最有可能是错误的源头。

熟悉概率图模型的读者可以发现，上述条件概率可以用一个贝叶斯网来建模，然后采用多种现有的概率推断算法来计算后验概率。由于这些都有比较成熟的概率编程语言和概率推断工具来支持，我们就不在这里展开了。

10.7 算法式调试

和之前讲过的方法不同，算法式调试（Algorithmic Debugging）[19]是一种交互式调试技术，该技术通过询问程序员“是”或“否”的问题来找到出错的函数。由以色列魏茨曼科学研究所的Ehud Shapiro教授在1982年

提出，并获得ACM杰出博士论文奖。该技术主要针对函数语言设计，并在Haskell等函数语言上得到了广泛实现。

下面用一个C程序的例子来说明算法式调试的过程。

Listing 10.1: 算法式调试示例

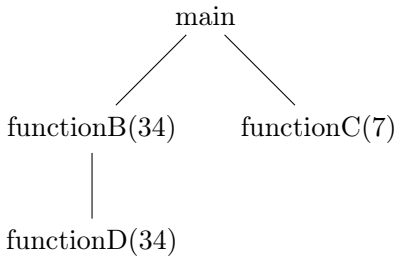
```
int main() {
    printf(functionB(34) + functionC(7));
}

int functionB(int a) {
    return functionD(a);
}

int functionC(int b) {
    return b % 7;
}

int functionD(int a) {
    // incorrect code, should be a+1
    return a++;
}
```

这个程序中在functionD中包含一个Bug，导致程序执行的输出不对。我们现在用算法式调试的过程定位到出错的位置。

- 系统提问：“调用functionB(34)返回34，正确吗？”
- 程序员回答：“不正确。”
- 系统提问：“调用functionD(34)返回34，正确吗？”
- 程序员回答：“不正确。”
- 系统提示：缺陷在functionD(34)中。

为了实现上述过程，算法式调试需要构建一个调用树，表示测试执行过程中的函数调用过程，如下图所示。这样，每次询问就是从调用树上寻找一个节点，询问改节点的函数调用的输入输出是否正确。当用户对某个

节点回答“是”时，表示该节点为根的子树中包含错误，因此可以排除该子树以外的所有节点。相反，当用户对某个节点回答“否”时，表示该节点为根的子树中不包含错误，因此可以排除该子树本身。通过不断重复这个过程，算法能够逐步缩小错误可能存在的范围，直到找到具体的错误位置。



注意上面例子是一个简化的场景。在实际应用中，由于命令式语言具有副作用，算法式调试的每次询问需要找到合适的方式来表示函数调用的输入输出。因此，算法式调试在函数式语言上的应用更为广泛。

10.8 差异调试

差异调试（Delta Debugging）是一种针对版本间差异的错误定位技术，由德国Saarland大学的Andreas Zeller教授在1999年提出[25]。差异调试的核心思想是通过比较两个不同版本（或状态）的程序，找出导致行为差异的最小修改集合。这些修改可以是代码变更、输入数据的变化，或是系统状态的不同。差异调试的基本步骤包括：

1. **选择比较对象**：确定需要比较的两个版本或状态，例如，一个是通过测试的版本，另一个是不通过测试的版本。
2. **确定修改集合**：列出从第一个版本到第二个版本的所有修改。
3. **搜索最小修改**：在修改集合上应用某种搜索算法，查找一个修改的子集，使得该子集尽可能地小，同时仍然不通过测试。

为了更直观地理解差异调试，以下是两个具体的场景示例：

场景1：代码变更导致测试失败 假设昨晚代码还通过所有测试，但今天加班改了1000行代码后，测试不通过了。问题是，哪些修改是导致测试失败的罪魁祸首？

在这个场景中，我们可以将昨晚的代码视为第一个版本，今天的代码视为第二个版本。所有1000行代码的变更构成了修改集合。通过差异调试，我们可以逐步缩小这个集合，最终找到导致测试失败的最小修改子集。

场景2：编译器崩溃 用户编译了一个1000万行代码的项目，编译器崩溃了。问题是，哪些输入代码导致编译器崩溃？

在这个场景中，我们可以将导致编译器崩溃的输入代码视为第二个版本，而一个不会导致崩溃的输入（例如空输入）视为第一个版本。所有输入代码的差异构成了修改集合。通过差异调试，我们可以找到导致编译器崩溃的最小输入子集。

10.8.1 差异调试问题定义

- 输入：
  - 完整修改的集合C
  - 测试谓词$\texttt{test} \subseteq 2^C$, 满足$\neg\texttt{test}(\emptyset)$
- 输出：集合$C' \subseteq C$, 满足
  - $\texttt{test}(C')$为真

给定一个完整的修改集合C，我们的目标是找到C的一个子集，满足测试要求。

同时我们还希望返回的集合尽可能的小。由于不同算法在保证尽可能小这一点上能力不同，所以对于这一点有多个不同级别的属性定义。不同的差异调试算法可能满足其中一个属性，或者一个都不满足。

- **最小性**：不存在$C'' \subset C$，使得$|C''| < |C'| \wedge \texttt{test}(C'')$为真
- **极小性**：不存在$C'' \subset C'$，使得$\texttt{test}(C'')$为真
- **1-极小性**：$\forall c \in C', \texttt{test}(C' - \{c\})$为假

10.8.2 ddmin算法

在本章中，我们介绍两个典型的差异调试算法。第一个是ddmin算法。该算法采用一个固定的顺序尝试从集合中删除元素。为了更直观地理解ddmin 算法的工作原理，以下是一个具体的运行示例。

假设我们有以下修改集合：

$$\Omega = \{a, b, c, d, e, f, g, h, i, j, k, l\}$$

ddmin首先尝试从集合中去掉前一半元素，即尝试：

$$\Omega = \{g, h, i, j, k, l\}$$

测试没有通过。说明前一半元素包含通过测试所需要的关键元素，不能删除。ddmin算法接着尝试去掉后一半元素，即尝试：

$$\Omega = \{a, b, c, d, e, f\}$$

测试还没有通过。说明后一半元素也包含通过测试所需要的关键元素，前后元素的共同作用导致了测试无法通过。这样，我们必须尝试删除更小的集合。ddmin算法接着尝试去掉前四分之一的元素，即尝试：

$$\Omega = \{d, e, f, g, h, i, j, k, l\}$$

测试没有通过，然后尝试删除第二个四分之一的元素：

$$\Omega = \{a, b, c, g, h, i, j, k, l\}$$

这次测试通过了。说明删除的这两个元素和测试是无关的，可以放心删除。算法继续，先后尝试删除第三个和第四个四分之一，都测试失败。接下来，算法尝试删除1/8的元素。由于无法整除，算法会把剩下元素分成6份（因为有2份已经删掉了），然后依次尝试删除。比如一种划分方式是如下这样：

$$\Omega = \{a \mid b, c \mid g \mid h, i \mid j \mid k, l\}$$

依次尝试删除这个划分中的元素，我们得到如下的集合。

$$\Omega = \{a, g, k, l\}$$

最后，算法进一步减少删除的粒度，依次尝试删除单个元素。得到结果：

$$\Omega = \{a, g, l\}$$

为了确保1-极小性，算法会再次尝试删除单个元素，确认所有单一元素无法被进一步删除。得到最终的结果：

$$\Omega = \{a, g, l\}$$

ddmin算法的优点是简单易懂，且能够找到满足测试要求的1-极小性修改集合。然而，ddmin算法的缺点是采用固定的二分尝试顺序，导致该算法无法根据不同情况灵活选择更优的尝试策略。

10.8.3 ProbDD算法

为了克服ddmin算法的缺点，北京大学熊英飞团队于2021年提出了ProbDD（Probabilistic Delta Debugging）算法[21]。ProbDD算法建立概率模型，根据概率模型决定下一次尝试策略，并根据测试结果动态调整概率模型，实验表明在时间和结果上均优于ddmin算法。

ProbDD算法采用概率模型来描述每个修改是否必要。具体来说，每个修改都有一个概率值，表示其是否是导致测试通过的必要修改。ProbDD假设不同修改之间的必要性是彼此独立的。当且仅当所有必要修改都存在时，测试才能通过。

ProbDD算法包括以下步骤来实现：

1. **初始化概率**：为每个修改分配一个初始概率值（例如，均匀分布或基于先验知识的分布）。
2. **选择删除方案**：根据当前概率值选择一个删除方案，即决定删除哪些修改以进行下一次测试。方案应该保证在删除元素个数的期望值越高越好。
3. **运行测试**：应用选定的删除方案后运行测试，观察测试结果。
4. **更新概率模型**：根据测试结果计算后验概率，更新每个修改的概率值。如果测试通过，则降低被删除修改的必要性概率；如果测试不通过，则提高剩余修改的必要性概率。
5. **重复步骤2-4**：直到找到满足条件的最小修改集合或达到预定的迭代次数。

下面结合刚才的例子分别介绍每一步。

初始化概率

在初始化概率时，ProbDD算法为每个修改分配一个初始概率值。这些概率值一般基于先验知识选择。比如，如果我们预期调试结束之后可能剩下1/4的元素，同时不同元素之间不存在差异，可以把所有元素的概率都设置为0.25。

$$p_a = p_b = p_c = p_d = p_e = p_f = p_g = p_h = p_i = p_j = p_k = p_l = 0.25$$

我们用θ_x表示元素x留在最终结果中的随机事件，即

$$P(\theta_x) = p_x$$

选择删除方案

在选择删除方案时，ProbDD算法尝试最大化删除元素个数的期望值，即最大化$|X|\prod_{x\in X}(1-p_x)$。这里X为待删除集合。

接下来我们考虑删除哪些元素可以最大化这个期望值。首先，我们先固定X集合的大小，考虑选择什么元素放入该集合。容易看出，这里应该优先选择概率最小的元素。因为概率越小，$(1-P(x))$ 越大，从而期望值越大。

接着我们考虑X集合的大小。可以证明，集合大小在从小到大变化的过程中存在一个极值点。期望值总是先上升，然后再下降。因此，我们可以通过往X集合中添加概率最小的元素，直到期望值开始下降为止。

在上述例子中，由于所有元素的概率都相同，我们随机选择元素添加到集合中，添加到第四个的时候期望最大。不妨假设我们选择了$\{a,b,c,h\}$，测试不通过。

更新模型概率

接着ProbDD根据测试结果更新概率，更新的方式是把概率设置为根据当前测试结果计算出来的后验概率。如果一个元素没有被删除，那么该元素和这轮测试的结果无关，无需被更新。接下来我们考虑元素x被删除的情况。首先考虑测试通过的情况。根据贝叶斯定理，我们有：

$$P(\theta_x \mid test(X)) = \frac{P(test(X) \mid \theta_x)\cdot P(\theta_x)}{P(test(X))}$$

因为元素x被删除了，即$x \notin X$，那么当该元素是必要的时候，测试就不可能通过，因此$P(\neg test(X) \mid \theta_x)$为0。因此我们把$p_x$设置为0。

然后我们考虑测试未通过的情况。根据贝叶斯定理，我们有：

$$P(\theta_x \mid \neg test(X)) = \frac{P(\neg test(X) \mid \theta_x)\cdot P(\theta_x)}{P(\neg test(X))}$$

因为元素x被删除了，那么当该元素是必要的时候，测试必然不可能通过，所以$P(\neg test(X) \mid \theta_x)$为1。因此

$$P(\theta_x \mid \neg test(X)) = \frac{P(\theta_x)}{P(\neg test(X))} = \frac{p_x}{1-\prod_{x\in X}(1-p_x)}$$

即我们把p_x更新为$\frac{p_x}{1-\prod_{x\in X}(1-p_x)}$。

在上面的例子中，由于测试失败，所以我们根据上面的公式更新被删除元素的概率。即

$$p_a = p_b = p_c = p_h = 0.3657$$

接下来我们进一步选择删除方案。现在上一轮没有删除的八个元素的概率最小，同时和前一次相同，删除四个时期望最大。假设我们选择了删除d, e, f, i。这次测试通过。我们根据上述公式更新概率，得到

$$p_d = p_e = p_f = p_i = 0$$

接下来我们继续选择删除方案。我们还剩下四个元素没有被删除过，同时概率也和一开始保持一样，所以我们会选择这四个元素。这一次测试也失败，所以他们的概率也被升高了。

$$p_g = p_j = p_k = p_l = 0.3657$$

由于剩下元素的概率都升到了0.3657，删除2个就能达到最大期望。我们就这样反复重复这个过程，直到所有元素的概率都升到0或者1。

可以证明，ProbDD的时间复杂度为$O(n)$，其中n是修改集合的大小。同时，在原问题满足一定性质的时候，ProbDD还可以保证结果是极小和最小的。

可以看出，通过概率建模和基于概率模型进行最优化采样，ProbDD算法能避免ddmin算法中的很多低效的尝试。比如ddmin算法一开始会尝试删除一半元素，但两次尝试都是失败的。ProbDD根据先验概率的知识，只会尝试删除4个元素，成功的概率就高了很多。同时，ProbDD算法的概率变化和当前进行的测试有关，比如测试是3个元素还是2个元素，引起的概率变化是不同的，而ddmin算法无论怎么测试，下一轮都将分组的粒度提升一倍。

参考文献

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] J. Aldrich, C. L. Goues, and R. Padhye. Program analysis, 2022. Carnegie Mellon University, `https://cmu-program-analysis.github.io/`.

[3] K. P. Bennett and J. A. Blue. Optimal decision trees. *Rensselaer Polytechnic Institute Math Report*, 214(24):128, 1996.

[4] P. Cousot. *Principles of Abstract Interpretation*. The MIT Press, 2021.

[5] P. Cousot, R. Giacobazzi, and F. Ranzato. Program analysis is harder than verification: A computability perspective. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, volume 10982 of *Lecture Notes in Computer Science*, pages 75–95. Springer, 2018.

[6] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In J. S. Foster and D. Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 420–435. ACM, 2018.

[7] C. H. Flemming Nielson, Hanne Riis Nielson. *Principles of Program Analysis*. Springer, 1999.

[8] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330. ACM, 2011.

[9] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In J. Kramer, J. Bishop, P. T. Devanbu, and S. Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224. ACM, 2010.

[10] R. Ji, C. Kong, Y. Xiong, and Z. Hu. Improving oracle-guided inductive synthesis by efficient question selection. *Proc. ACM Program. Lang.*, 7(OOPSLA1):819–847, 2023.

[11] R. Ji, J. Xia, Y. Xiong, and Z. Hu. Generalizable synthesis through unification. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–28, 2021.

[12] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[13] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.

[14] A. Miné. The octagon abstract domain. *High. Order Symb. Comput.*, 19(1):31–100, 2006.

[15] A. Møller and M. I. Schwartzbach. Static program analysis, 2023. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

[16] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST*

2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA, pages 153–162. IEEE Computer Society, 2014.

[17] M. Papadakis and Y. L. Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test. Verification Reliab.*, 25(5-7):605–628, 2015.

[18] P. Raatikainen. Gödel' s Incompleteness Theorems. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Spring 2022 edition, 2022.

[19] E. Y. Shapiro. *Algorithmic program debugging*. Yale University, 1982.

[20] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[21] G. Wang, R. Shen, J. Chen, Y. Xiong, and L. Zhang. Probabilistic delta debugging. In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 881–892. ACM, 2021.

[22] X. Wang, I. Dillig, and R. Singh. Synthesis of data completion scripts using finite tree automata. *Proc. ACM Program. Lang.*, 1(OOPSLA):62:1–62:26, 2017.

[23] K. Y. Xavier Rival. *Introduction to Static Analysis: An Abstract Interpretation Perspective*. The MIT Press, 2020.

[24] Y. Xiong and B. Wang. L2S: A framework for synthesizing the most probable program under a specification. *ACM Trans. Softw. Eng. Methodol.*, 31(3):34:1–34:45, 2022.

[25] A. Zeller. Yesterday, my program worked. today, it does not. why? *ACM SIGSOFT Software engineering notes*, 24(6):253–267, 1999.

[26] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang. Fault localization via efficient probabilistic modeling of program semantics. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 958–969. ACM, 2022.

[27] Y. Zhang, L. Ren, L. Chen, Y. Xiong, S. Cheung, and T. Xie. Detecting numerical bugs in neural network architectures. In P. Devanbu, M. B. Cohen, and T. Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 826–837. ACM, 2020.