软件分析技术课程讲义

熊英飞
北京大学

2023 年 9 月 29 日

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[7]。

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[3]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, 槑}。抽象域中四个值的含义通过下面γ函数定义。

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

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$\oplus, \ominus, \otimes, \oslash$。

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

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

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槑的结果，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。实际中的搜索法会通过引入各种剪枝和推断算法来减少搜索空间。

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：$\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x = y$
- 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$X \to \{\text{正}, \text{负}, \text{零}, \text{跟}, \bot\}$，其中$X$是程序中所有变量的集合。

合并运算：

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

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

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[2, 8, 6, 5, 4]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

挖坑待填

3.1　抽象解释理论框架

3.2　抽象解释和分析正确性

3.3　转换函数设计方法

3.3.1　语句对应的转换函数设计方法

3.3.2　条件对应的转换函数设计方法

3.4　流非敏感分析

第四章　基于Datalog的分析

挖坑待填

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

[3] 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.

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

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

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

[7] 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.

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