软件分析技术课程讲义
第一章引言

熊英飞
北京大学

2023 年 9 月 19 日

1 引言

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。

1.2 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[2]。

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。

1.4 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

[1] 递归可枚举语言即可以被图灵机识别的语言。

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。

1.5 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。

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[1]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。

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。

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。

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

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

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

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

然后我们就可以在抽象域上进行运算。比如5×2+6就可以运算为$\alpha(5) \otimes \alpha(2) \oplus \alpha(6)$，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。

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。

参考文献

[1] 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.

[2] 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.