静态程序分析技术

MF1933108-许端琛

# 研究背景

程序编程语言方面的研究已有60年的研究历史，同时也是非常活跃的一个方向。程序编程语言的研究大致可以分为3个部分，包括理论部分，环境部分以及应用部分。其中理论部分包含语言的设计，类型系统，形式语义等。环境部分包含编译器，运行时系统等。而应用部分包含程序分析，程序验证，程序合成。其中在PL的框架下，程序分析多指静态程序分析，也即该课程报告所要讨论的主要内容。

在过去十多年，静态代码分析取得了长足的进步，并且出现了许多新颖的应用：最初主要用于运行前程序的优化，现在它也已成为理解程序，程序设计，查找软件质量缺陷，特别是安全漏洞的通用工具。 这项成功归功于这数十年来的静态分析研究，这些研究催生了新颖的算法，数据结构和设计原理，使静态分析比以往任何时候都更加精确。

而这段时间里，编程语言的核心部分改变的非常少，然而由编程语言构建的程序变的越来越大，而且越来越复杂。因此，尽管从绝对意义上说，静态分析研究的进展是显着的，但人们必须意识到，随着软件应用程序的规模和复杂性不断增加，静态分析技术的研究应该依然得到持续的突破和进步。这也是静态程序分析技术仍有很大研究价值的原因之一。

# 问题定义

静态程序分析推理计算机程序的行为而不实际运行它们。这不仅对优化编译器以产生高效代码很有用，而且对于自动错误检测和其他可以帮助程序员的工具也很有用。静态程序分析器是引起其他程序行为的程序。

众所周知，测试，即具体地运行程序并检查输出，这种方法可能会发现错误，但通常不能表明没有错误。相反，静态程序分析可以（使用正确的近似值）检查程序的所有可能执行情况，并提供有关程序属性的保证。

开发此类分析的主要挑战之一是如何确保高精度和高效率在实际中有用。例如，如果它报告许多误报，或者如果它太慢而无法适应现实世界的软件开发过程，则没有人会使用这样设计的分析。

目前程序静态分析可用于例如一下方面：

1. 程序优化分析。优化的编译器（包括解释器中的即时编译器）需要了解正在编译的程序的许多不同属性，以便生成有效的代码。
2. 程序正确性和安全性分析。设计用于检测错误（或验证是否没有错误）的最成功的分析工具都针对以特定编程语言编写的程序的通用正确性属性。在不安全的语言（如C）中，此类错误有时会导致严重的安全漏洞。在更安全的语言（如Java）中，此类错误通常不会那么严重，但它们仍可能导致程序崩溃。
3. 程序开发分析。现代IDE执行各种程序分析以支持调试，重构和程序理解。

# 方法和技术

## 中间表示

中间表示是编译器或虚拟机内部用来表示源代码的数据结构或代码。 IR的设计有利于进一步处理，例如优化和转换。一个好的中间表示必须是准确的，能够代表源代码而不丢失信息，并且必须独立于任何特定的源语言或目标语言。

通常有多种中间表示，例如抽象语法树，三地址代码等。由于三地址码更接近于机器代码，通常与具体分析语言无关，更加简洁且形式统一（一条指令的右侧最多有一个运算符，没有组合的算数表达式），且可以表示控制流信息。因此在静态程序分析中，多采用三地址码的中间表示。

## 数据流分析

数据流分析是一种用于收集在计算机程序中各个点计算的可能值集的信息的技术。在优化程序时，编译器通常会使用收集到的信息。

执行程序的数据流分析的一种简单方法是为控制流图的每个节点设置控制流方程，通过在每个节点上反复计算输入和输出，直到整个系统稳定，到达一个固定点为止。

数据流分析涉及很多术语，例如基本块，控制流图，控制流路径等。有四个实例包括到达定值分析，活跃变量分析，可用表达式分析，忙碌表达式分析很好的解释了数据流分析的几种情况。

## 过程内分析和过程间分析

过程内分析是仅使用可用于该功能和编译单元的信息来对编译单元内的每个功能执行优化的机制。过程内分析保守地假设被调用的过程有可能改变过程可见的所有变量的状态，并且它们还可能产生某种副作用，比如改变此过程可见的任何变量的值，或产生导致调用栈释放的异常。过程内分析虽然不精确，但是却相对简单。适合于不需要过程间分析的优化。但有些优化不借助过程间几乎不会产生有用的信息。

过程间分析是一种跨功能单元边界执行优化的机制。一个过程间分析处理的是整个程序，他将信息从调用者传送到被调用者，或者反向传递。过程间分析通常涉及调用图的概念，调用图表示程序中过程之间的调用关系。 每个节点代表一个过程，每个边如（f，g）表示过程f调用过程g。过程间分析很有挑战性，通常各个过程的行为和他调用时所在的上下文相关，还需要考虑上下文问题等。

# 现有工具

Soot最初作为Java优化框架。到目前为止，世界各地的研究人员和从业人员都使用Soot来分析，检测，优化和可视化Java和Android应用程序，在静态程序分析中，Soot是一个字节码操作和优化框架，由Java的中间语言组成。

Soot通过其API提供了四种中间表示形式，供其他分析程序访问和使用：

1. Baf：近字节码表示。

2. Jimple：适合优化的 3地址中间表示形式。

3. Shimple：Jimple的SSA变体（类似于GIMPLE）。

4. Grimp：Jimple的聚合版本，适用于反编译和代码检查。

Soot可用于调用图构造，指针分析，定义/使用链构造，过程内数据流分析以及过程间数据流分析。Soot的编程主要涉及一下类型结构：

1. SootClass。 表示加载到Soot或使用Soot创建的单个类。

2. SootMethod。 表示类的单个方法。

3. SootField。 表示类的成员字段。

4. Body。 表示方法主体，并具有不同的风格，分别对应于不同的IR（例如JimpleBody）

Soot为用户提供丰富的API，可以使用户轻松的实现各类自定义Java程序静态分析。

# 研究趋势

该部分主要介绍一篇论文的研究成果，自适应静态分析。改论文发表在2018年的ICSE，论文提出了一种全新的工程方法来设计和实施分析，用以解决静态分析自动优化和自适应时遇到的问题，同时解决当前由于软件系统的规模和复杂性大大增加，使得静态分析系统面临可扩展性不足的问题。

为了将可扩展性提高到一个新的水平，文章将静态分析与即时优化技术融合在一起，首次引入可管理的，本质上自适应的静态分析。 这些分析会自动进行调整，以产生性能/精度的折衷，这对于被分析的软件系统和分析本身而言是最佳的。

系统主要分为5个部分：

1. 第一部分是静态分析的声明式定义语言。系统不能以Java或C / C ++之类的通用编程语言或者Datalog之类的通用逻辑编程语言来表示程序分析本身，而是使用最适合领域特定优化的领域特定语言，也就是仅在考虑到有关静态程序分析领域的特定知识时才是正确的语言。

它必须具有精确的表达水平：具有足够的表达能力，来涵盖静态分析可能出现的情况，但同时必须限制其表达能力，使得不会阻碍自动优化的实现。而且，该语言必须对于静态分析用户是足够容易使用和理解的，同时也必须是机器可读的。

2. 第二部分是High-level的中间表示。要实现对静态分析的自动优化，就需要设计静态分析的专用的特定中间表示（IR），这种中间表示需要非常适合特定的优化。文章提出的High-level中间表示的重点在于在程序执行之前对静态分析进行优化。

3. 第三部分是Low-level的中间表示。第二部分的提前优化是为了在不考虑被分析程序的情况下提高分析性能。但是，所分析的程序可能会对静态分析的最佳配置产生很大影响。 因此，需要有一种机制，允许考虑程序特征，允许在进行分析过程中将分析配置调整为最佳。

与High-level的中间表示相反，为了允许及时的适应，对于Low-level的中间表示，必须允许其更改目前已经确定的分析表示形式，即时的重新生成分析代码。

4. 第四部分是静态分析profiler。一般一个自我优化的静态分析器，需要一名静态分析专家，他需要深入了解分析本身的执行情况。自适应性分析实际上比正常情况下更难，因为实际执行的分析代码不是分析设计者手写的代码，而是由分析的中间表示生成的。因此，这种自适应设计可能会失去分析定义和执行之间的联系。所以需要用于自适应静态分析的专用配置文件的工具。它能够在自动识别导致分析性能降低的执行热点。profiler还应该将这些性能热点链接回静态分析的相关片段，也就是找到该执行模块对应的IR或者实现语言的部分。

5. 最后一步是开发一个优化引擎，该引擎使用关于其执行的静态分析的领域知识，再结合来自profiler分析运行的信息，来确定最佳的分析执行策略，并自动触发这些策略进行自我适应。该引擎应该有效地实现一个控制循环，在该循环中连续处理性能分析信息来确定当前策略的最优性，并找出在剩余时间段内可能更有效的其他策略。

可以说，文章最终就是为了同时利用静态分析和程序属性进行实时优化，提出了一个新的的体系结构。