北京大学

本科生毕业论文

题目：基于 Datalog 条件摘要的程序分析加速技术

姓　　名：王迪

学　　号：1300012802

院　　系：信息科学技术学院

专　　业：计算机科学与技术

研究方向：程序语言理论

导　　师：熊英飞

二〇一七年五月

北京大学本科毕业论文导师评阅表

学生姓名	王迪	学生学号	1300012802	论文成绩	
学院（系）	信息科学技术学院			学生所在专业	计算机科学与技术
导师姓名	熊英飞	导师单位/所在研究所	软件研究所	导师职称	研究员
论文题目（中、英文）	通过基于 Datalog 的条件摘要加速程序分析 Accelerating Program Analyses by Conditional Summarization with Datalog				

导师评语

王迪的本科毕业论文探索了程序分析中的一个基础问题——库代码摘要问题。现代软件系统常常都带有很大的代码库，导致我们分析很小的一部分程序就不得不分析很大的代码库。对这个问题的一个可行的解决方案是对库进行预先摘要，但目前仍然缺乏通用的库分析技术。

王迪的本科科研期间已经探索了基于上下文无关文法的摘要方法。王迪的毕业设计将该方法进一步扩展到更通用的基于 Datalog 的分析上，大大拓展了方法的适用范围。实验证明方法的效果良好。论文写作规范，逻辑性强，是一篇优秀的本科毕业论文。

导师签名：

年　　月　　日

版权声明

摘要

现实生活中的程序往往会共用很大一部分代码，但是很多程序分析并不能重用对共用代码的分析结果。库摘要通过重用对共用库代码的分析结果，可以有效的加速应用端代码的程序分析。然而，在对库代码进行摘要的时候，关于应用端代码的信息是缺失的，这导致人们不能对库代码做一个完整的摘要。现有技术通过上下文无关语言（Context Free Langauge）可达性分析产生条件摘要，即在不同的对应用端代码的假设条件下的库摘要。不过，上下文无关语言的使用也导致一些重要的分析无法被进行，比如控制流分析。

在这篇论文中，我提出了一个基于 Datalog 的新的条件摘要技术 ConDlog。ConDlog 通过在库代码上进行一个条件化的程序分析计算出一个完成的摘要。然后，在分析应用端代码时，ConDlog 会实例化这个摘要，尽可能地减少分析中的中间计算。我在两个针对 Java 程序的分析上实现了 ConDlog 技术：类层次分析（Class Hierarchy Analysis）和快速类型分析（Rapid Type Analysis）。我在 DaCapo 测试集的 11 个程序上做了实验，并选择 Java 标准库作为库代码。ConDlog 在类层次分析和快速类型分析上分别达到了 3.9 倍和 1.3 倍的平均加速。

关键词：程序分析，条件摘要，声明式分析

Accelerating Program Analyses by Conditional Summarization with Datalog

Di Wang (Computer Science and Technology)
Directed by Prof. Yingfei Xiong

ABSTRACT

Practical programs share large modules of code, but many program analyses are ineffective at reusing analysis results for shared code. Library summarization is an effective way to accelerate the analysis of client code by reusing analysis results for library code. However, information about the client is unknown during library summarization, preventing complete summarization of the library. An existing approach utilizes context-free language (CFL) reachability analysis to provide conditional summaries, enabling the summarization of a library under certain premises. However, the use of CFL rules out important analyses, e.g., Control Flow Analysis.

I present ConDLog, a new conditional summarization technique based on Datalog. ConDLog runs a conditional version of an analysis offline on a library and computes a complete summary. It instantiates the summary online to eliminate as many intermediate computations of the analysis of client code as possible. I have implemented ConDLog to accelerate two analyses for Java programs: Class Hierarchy Analyais (CHA) and Rapid Type Analysis (RTA). I evaluate the analyses on eleven programs from the DaCapo suite that share the Java standard library. ConDLog achieves average speedups of 3.9x for CHA and 1.3x for RTA.

KEYWORDS: Program Analysis, Conditional Summaries, Declaritive Analysis

目录

第一章　绪论

1.1　程序分析

在当今信息时代，每个人的生活都离不开软件。早上起来刷手机上的微信，是软件；制作开会时用的演示文稿，得用软件；宇宙飞船上天，控制也离不开软件。这个世界不仅已经充斥着各种各样的软件，而且新的软件还在日复一日地产生着。随着软件接管人们生活中越来越多的方面，软件的安全性也越来越受到人们的重视。事实上，软件缺陷曾经造成灾难性的事故。2003 年，由于电网管理软件内部实现存在重大缺陷，美国和加拿大发生大面积停电事故，造成至少 11 人丧生；2006 年，由于软件系统没有实现对防碰撞硬件系统故障的检测，巴西两架飞机相撞，造成 154 人丧生；2005 年，由于错误的升级指令，东京证券交易所出现了人类历史上最长停机事故，造成的资金和信誉损失难以估算。墨菲定律告诉我们，对人类来说，会出错的事总会出错，而设计、实现软件程序的也是人，所以软件缺陷几乎无法避免。幸运的是，随着科技的发展，计算机科学家们提出了各种各样的技术检测程序中潜在的缺陷，设计出实用的工具帮助人们避免软件缺陷。

程序分析（Program Analysis）就是软件缺陷检测技术中极为重要的一支。程序分析算法在**不实际运行程序**的前提下，**自动**地对软件的性质进行推断，比如，程序中是否存在内存泄漏，是否存在缓冲区溢出漏洞，等等。根据哥德尔不完备定理（Gödel's Incompleteness Theorem）和莱斯定理（Rice's Theorem），判断一个程序是否满足一个非平凡的性质是图灵不可判定的。所以，程序分析的基本方法是通过对程序执行状态进行抽象，计算出近似解。举例来说，我们想要对程序中变量的符号进行分析，就可以将数字抽象为正、负、零、未知四种抽象状态。接着，我们需要定义抽象状态上的四则运算，使得其相对于实际的四则运算是正确的。以加法为例，我们可以将四种状态间的运算定义如表格表格1.1：

表 1.1　符号分析抽象状态上的加法运算

+	正	负	零	未知
正	正	未知	正	未知
负	未知	负	负	未知
零	正	负	零	未知
未知	未知	未知	未知	未知

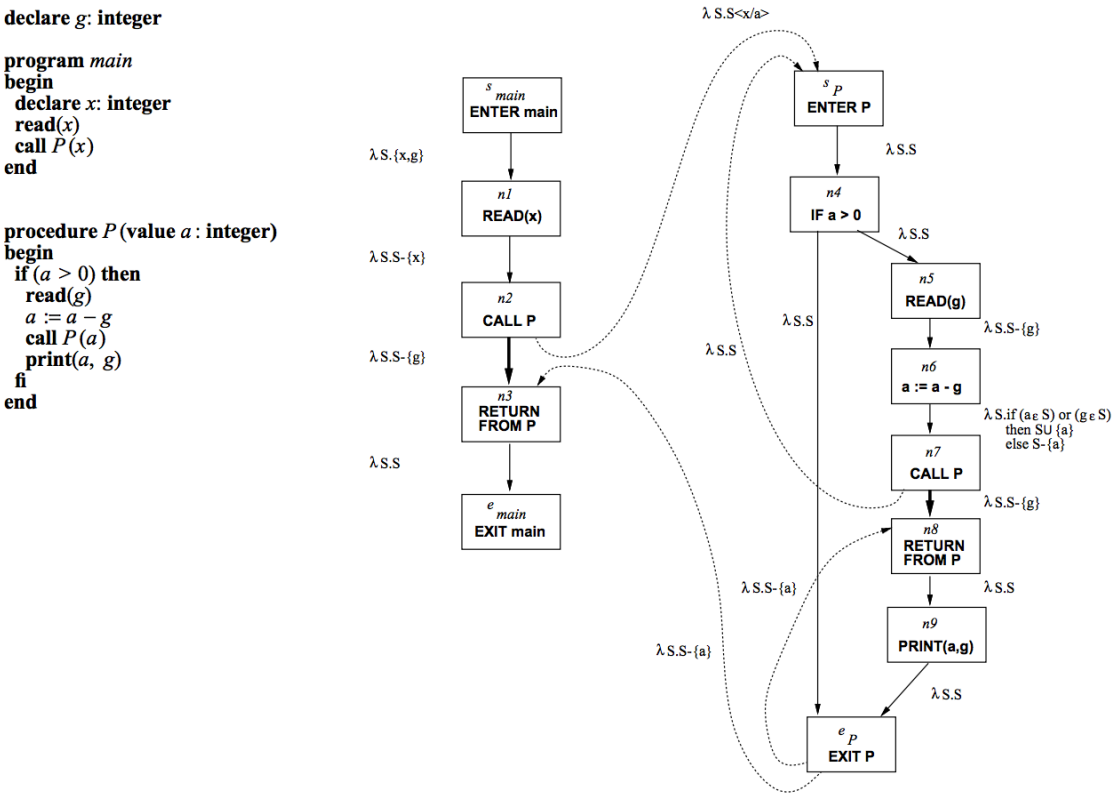
因为程序中的变量个数是有限的，我们设计的每个变量的抽象状态个数也是有限

的，所以我们所需要分析的程序的抽象执行状态空间是有限的，从而我们的分析算法是可行的。以上是一个简单的例子，程序分析经过多年的发展，已经成为了一个百花齐放的领域：数据流分析（Dataflow Analysis），过程间分析（Inter-procedural Analysis），指向分析（Points-to Analysis），控制流分析（Control Flow Analysis），等等。程序分析算法的近似属性自然地给人们带来了一个设计上的问题：如何权衡**精度**和**性能**？早期，因为软件规模不是很大，人们注重提升分析的精确度；但是时至今日，软件的规模已经越来越大，如何将已有的分析算法实现在大规模的程序上，且保证一定的时间效率，成为了新的挑战。

1.2 基于摘要的程序分析

现实生活中的软件往往会共用很大一部分代码，比如 Java 程序大量使用 Java 标准库，安卓软件频繁调用安卓开发框架。基于此观察，人们提出了基于摘要的程序分析的概念，即预先对频繁使用的公共代码进行分析，生成摘要，然后在应用端代码的分析的时候直接使用摘要，避免对共用代码进行重复分析。优秀的摘要技术往往可以在保证精度的情况下大幅提升性能。在程序分析领域，模块化分析（Modular Analysis）和可组合分析（Compositional Analysis）已经得到了充分的理论研究 [10–12, 14, 17, 36]。然而，在实际的程序语言上设计和实现这些分析仍然很困难。动态分配内存和高阶函数等语言特性造成了复杂的程序数据流和控制流，妨碍了人们对摘要进行正确、高效的重用。所以，事实上很多实用的程序分析仍然会从头分析整个应用端程序，在这个过程中生成摘要来加速分析。一些流行的基于摘要的程序分析算法（例如 [13, 23, 27]）已经被包含在了诸如 SLAM [6]，Soot [8]，WALA [15]，Chord [22] 等的程序分析工具里。以 Thomas Reps 等人提出的图可达性方法 [23] 为例简要介绍一下摘要的基本思路。假设我们想要分析程序中可能没有被初始化的变量，考虑示例程序（图1.1a）及其对应的带有数据流转移函数的超级图（Supergraph）（图1.1b）。

每条控制流边上标记了一个形如 $\lambda S.\mathsf{exp}$ 的转移函数，在这个分析中，就是一个 $2^{\mathsf{Var}} \to 2^{\mathsf{Var}}$ 的函数，表示一条语句执行前未初始化变量集合和执行后未初始化变量集合的转移关系，其中 Var 是程序中的变量集合。举例来说，对于 $\mathbf{read}(x)$ 这条语句，因为它给变量 x 读入了一个值，所以执行后必然 x 已经初始化了，从而其转移函数是 $\lambda S.S - \{x\}$。在这个分析中，我们希望建立每个控制流节点与其所属过程的入口之间的数据流关系。因为程序中有多个过程，且允许递归调用，精确分析并不是一件容易的事情。Thomas Reps 等人观察到数据流转移函数可以用图可达性来刻画，即对于一个 $2^D \to 2^D$ 的可分配函数 f（即对任意集合 A, B，满足 $f(A \cup B) = f(A) \cup f(B)$），可以构造一个包含于 $(D \cup \{\mathbf{0}\}) \times (D \cup \{\mathbf{0}\})$ 的二元关系来精确地刻画，从而可以用一个有向图

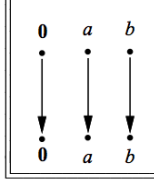
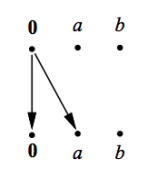
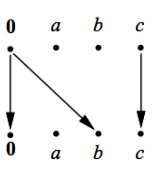


(a) 示例程序　　(b) 其超级图

图 1.1　示例程序及其超级图

来表示。

示例 1.2.1. 下面这个表格描述了三个转移函数以及它们对应的图表示：

$\mathbf{id}: 2^{\{a,b\}} \to 2^{\{a,b\}}$ $\mathbf{id} = \lambda S.S$	$\mathbf{a}: 2^{\{a,b\}} \to 2^{\{a,b\}}$ $\mathbf{a} = \lambda S.\{a\}$	$\mathbf{f}: 2^{\{a,b,c\}} \to 2^{\{a,b,c\}}$ $\mathbf{f} = \lambda S.(S - \{a\}) \cup \{b\}$
		

通过这种方式，我们可以将图1.1b中的带有数据流转移函数的超级图转换成普通的图：把每个转移函数替换成其对应的用二元关系刻画的图即可。我们称转化后所得的图1.2为爆炸超级图（Exploded Supergraph）。此时我们已经将原分析问题转化成一个图可达性问题：如果控制流节点 ℓ_0 处的变量 x 可以到达控制流节点 ℓ_1 处的变量 y，那么这意味着在 ℓ_0 处 x 可能未初始化可以推出在 ℓ_1 处 y 可能未初始化①。此时，摘要技

① 此处的可达性实际上需要考虑过程调用边和返回边的匹配。

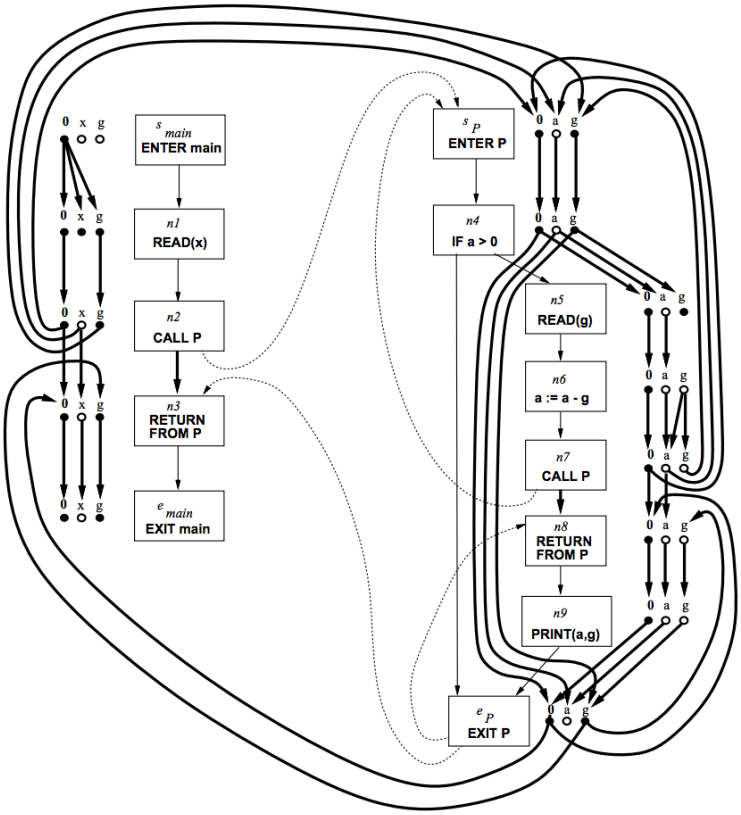


图 1.2　图1.1b对应的爆炸超级图

术自然地产生了：我们可以记录每个过程入口节点和出口节点间的可达性作为这个过程的摘要，当我们分析到一个过程调用语句时，就可以不用重新分析这个过程，而是使用我们计算好的摘要，从而减少了中间计算量。在图1.2中的爆炸超级图上，过程 P 的摘要就是 $\{\mathbf{0} \Rightarrow \mathbf{0}, a \Rightarrow a, g \Rightarrow g\}$。

虽然这样的基于摘要程序分析设计和实现起来更容易，但是不能重用共用代码的分析结果还是拖累了整个工具的性能。举例来说，哪怕是分析一个只向终端输出一行“Hello World”的 Java 程序的函数调用关系，这些工具仍然需要重新分析 Java 标准库里面成千上万个类。想要真正地将摘要做到重用，即重用对不同程序共用代码部分的分析结果，在我看来有两大困难。第一，对于相同的共用代码，如果应用端程序不同，关于共用代码部分的分析结果是不一定相同的。这意味着，将一个程序的分析结果中关于共用代码的部分**不加条件**地应用到另一个程序中，是不可行的。第二，重用摘要的目的在于提高程序分析的性能，为了加速分析，我们必须在分析应用端时尽可能地去除共用代码相关部分的中间计算，但是保留所有其应有的分析结果。中间计算去除得少了，加速效果就不好；中间计算去除得多了，分析结果就可能不正确。为了解决这些困难，研究者们提出了条件摘要（Conditional Summarization）技术，以求正确并

高效地重用程序分析的摘要。

1.3　条件摘要

无条件的摘要在实例化时不会考虑实际应用端代码的信息，在具有复杂数据流和控制流的现代语言上并不适用。与之相对的，条件摘要就是会在摘要中考虑应用端代码中可能具备的条件，不同的条件就会对应地计算出不同的分析结果作为摘要，在分析应用端代码时，再根据应用端的信息实例化这些摘要。Hao Tang 等人提出的条件 Dyck 上下文无关语言可达性分析技术 [31] 和 Sulekha Kulkarni 等人提出的跨程序训练技术 [19] 是目前最先进的两种条件摘要方法。这两个工作都在 Java 上进行，且均选择 Java 标准库作为共用代码，所以我在后文中不加区分地使用“库代码”和“共用代码”。

Hao Tang 等人观察到 Java 程序中灵活的回调（Callback）和虚拟方法调用（Virtual Call）为可复用的摘要造成了阻碍。在不知道应用端代码信息，只分析库代码时，无法分析虚拟方法调用的目标方法。甚至，因为应用端代码可以对库中定义的方法进行重载（Override），所以对于库代码中的回调，它能造成的作用也是无法分析的。条件 Dyck 上下文无关语言可达性分析技术将回调可能的行为和虚拟方法调用可能的目标方法作为条件，从而可以在没有应用端具体信息的时候分析库代码。

```
1 package library;
2 public abstract class
      LibraryClass {
3   public final int pub(int a){
4     int b = m1(a);
5     int c = m2(b);
6     int d = c;
7     int e;
8     if (...) e = m3(d);
9     else e = m4(d);
10    return e; }
11  private final int m1(int x1){
12    return x1; }
13  abstract protected int
        m2(int x2);
14  protected int m3(int x3){
15    return x3; }
16  private final int m4(int x4){
17    int x5 = 100; return x5; }
18 }
19
20 package client;
21 class ClientClass extends
       library.LibraryClass {
22   protected int m2(int x2){
23     return x2; }
24   public static void main(){
25     ClientClass client = new
         ClientClass();
26     int z = 255;
27     int z1 = client.pub(z); }
28 }
```

图 1.3　示例程序

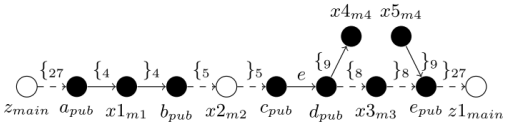


图 1.4　其对应的变量依赖图

以图1.3中的 Java 程序为例，我们进行一个变量依赖关系分析，即一个变量的值是否可能通过数据流影响到另外一个变量。以变量为点，变量间的直接依赖关系（例如赋值语句等）为有向边，我们自然地可以构建出一个变量依赖图（见图1.4）。图中有的边上带有括号：左括号表示方法调用，将方法调用的实际参数传递到方法的形式参数；右括号表示方法返回，将方法的返回值传递到方法调用的接收变量。在只分析库代码时，第 5 行的 m2 调用是一个回调，而第 8 行的 m3 调用是一个虚拟方法调用。条件 Dyck 上下文无关语言可达性分析技术在这个示例中就会将第 5 行调用处变量 b 是否影响到 c 作为一个条件，将第 8 行调用是否会调用 LibraryClass 的 m3 方法作为一个条件。因此，该技术产生的摘要中会包含如下信息：在图1.4上，如果 b_{pub} 可以影响到 c_{pub}，并且第 8 行调用了 LibraryClass 的 m3 方法，那么 a_{pub} 可以影响到 e_{pub}。之后，在分析应用端程序时，就可以根据实际的信息来决定是否实例化该摘要。示例中，从

ClientClass 实现的 m2 方法可以分析出 b_{pub} 影响到了 c_{pub}，从调用关系分析中可知第 8 行调用了 LibraryClass 的 m3 方法，因此该摘要可以被实例化，从而直接推导出 a_{pub} 可以影响到 e_{pub}，利用这个很容易地可以推出 z_{main} 会影响 $z1_{main}$。

条件 Dyck 上下文无关语言可达性分析技术可以**独立**地分析库代码，计算出一个**完整**的摘要，并在分析完成后去除掉库对应的变量依赖图中的大部分点和边，只用保留有可能与应用端代码交互的点和它们之间的条件摘要。美中不足的是，Dyck 上下文无关语言的使用也限制了该技术能适用的程序分析，一些重要的分析（比如控制流分析）无法自然地在这个框架中进行条件摘要。Sulekha Kulkarni 等人则选择不去独立地分析库代码，而是从应用端的全程序分析（Whole-program Analysis）的结果中去学习条件摘要，故称为跨程序训练。该技术使用一种逻辑编程语言 Datalog 来描述程序分析的规约，因而获得了比 Dyck 上下文无关语言更强的表达能力。在程序分析领域中，Datalog 是一种流行的描述规则的方式 [9, 18, 29, 30, 35]。在一种分析上使用跨程序分析技术时，需要指定条件摘要的选取方式和实例化方式。比如在方法调用图分析中，可以将一个库方法是否可能被调用作为前提条件，在这个前提条件下能推出的方法调用关系作为摘要；而实例化则需要在分析应用端代码前，根据应用端的信息对摘要的前提条件是否满足进行判断。仍以图1.3中的代码为例，通过全程序分析我们可以知道第 8 行会调用 LibraryClass 的 m3 方法。跨程序分析技术从这个程序中可以计算出一个条件摘要：如果 LibraryClass 的 pub 方法可被调用，那么第 8 行就会调用 LibraryClass 的 m3 方法。这个摘要显然并不对所有可能的应用端程序成立，因为其他的应用端可能会重载 LibraryClass 的 m3 方法。因此，用户可以规定摘要的前提条件判断方式为 m3 方法有没有在应用端被重载。

跨程序分析技术虽然可以从现实生活中存在的代码中学习出大量的条件摘要，但它并不能保证其条件摘要是完整的。之前我提到，条件 Dyck 上下文无关语言可达性分析技术可以独立的从库代码中分析出完整的条件摘要，但其表达能力不如 Datalog。更加优秀的条件摘要框架应该融合以上两种思路的优点，力求正确、高效、完整地生成库代码的条件摘要，并对多种分析有不平凡的性能加速。

第二章　概观

在这一章，我会通过示例介绍 ConDlog 技术的基本思路，并陈述本工作的主要学术贡献。§2.1通过在一个简单的图可达性分析上实现 ConDlog 技术介绍我的思路和方法。§2.2总结了本工作的创新点和贡献。

2.1 ConDlog 应用示例

ConDlog 技术旨在通过重用共用代码的程序分析结果加速应用端程序的分析。在 ConDlog 中，程序分析规则通过 Datalog 进行表述。ConDlog 可以被设置成库分析模式和端分析模式。在库分析模式下，它独立地分析库代码以生成完整的条件摘要。这个分析同样通过 Datalog 进行规约，该规约是在原有的全程序分析规则上进行一个简单、系统的变换。分析完成后，生成的摘要可以保存在磁盘上。在端分析模式下，它根据应用端代码的信息选取满足前提条件的摘要，将其实例化并用在分析中，减少中间计算量并加速分析。这个分析也通过 Datalog 进行规约。除此之外，ConDlog 需要用户指定一个分析特定的**筛选**函数 *Filter* 用来指导 ConDlog 在端模式下选取满足前提条件的摘要。作为示例，我选取了有向图可达性分析。ConDlog 会先在我们指定的共用图上计算条件摘要，然后将这些摘要应用到两个不同的、包含共用图为子图的应用端图上。

图2.1中包含了共用图 L 和两个应用端图 A 和 B。表格2.1中陈列了图可达性分析的全程序分析规约，以及 ConDlog 需要的库分析、端分析的规约。图中有两类节点：库节点（标号为 L1 到 L11）和端节点（标号为 A0 到 A2 以及 B0 到 B3）。图 A 包含所有库节点和端节点 A0 到 A2。图 B 包含所有库节点和端节点 B0 到 B3。做个类比，这样的每张图其实代表了一个程序：端节点表示应用端特有的信息，库节点表示共用库的信息。A0 和 B0 分别表示这两个程序的入口节点。

问题描述。 图可达性分析的目标是计算出所有的可以被入口节点达到的节点集合。对图 A 来说，就是从 A0 可达的；对图 B 来说，就是从 B0 可达的。我们通过表格2.1中全程序分析部分的 Datalog 程序来解决这个问题。输入关系 edge 包含了图中所有的边。输出关系 reachable 包含了所有可以被入口节点 m_0 达到的节点。规则 (1) 是基础情况，表示 m_0 本身是可达的。规则 (2) 则是一个归纳步骤，表示如果节点 n 可达，且存在一条有向边从 n 到 m，则 m 也可达。为了简便，下文中我将使用 r(n) 表示 reachable(n)，

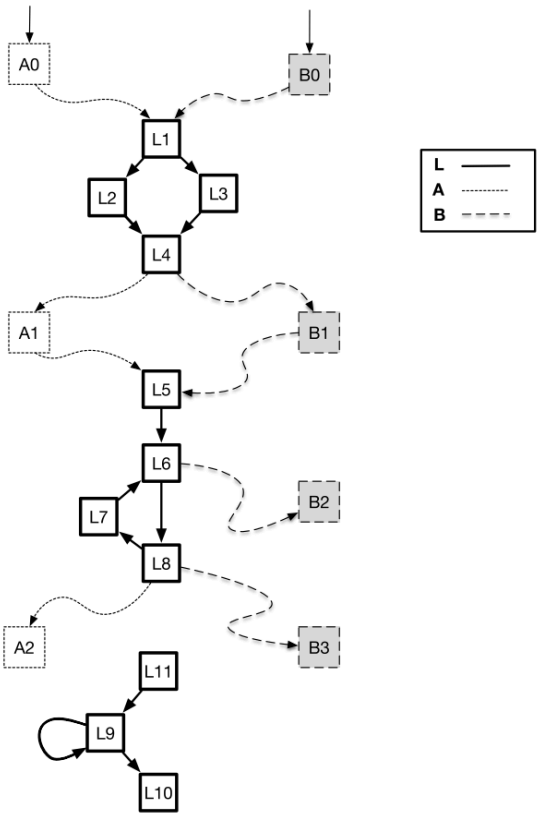


图 2.1　示例图

用 $\mathsf{e}(n, m)$ 表示 $\mathsf{edge}(n, m)$。

在图 A 上执行该 Datalog 程序可得可达性关系 $\{\mathsf{r}(\mathsf{A}i) \mid i \in [0, 2]\} \cup \{\mathsf{r}(\mathsf{L}i) \mid i \in [1, 8]\}$。在图 B 上执行该 Datalog 程序可得可达性关系 $\{\mathsf{r}(\mathsf{B}i) \mid i \in [0, 3]\} \cup \{\mathsf{r}(\mathsf{L}i) \mid i \in [1, 8]\}$。虽然图 A 和 B 都共用了库图 L，但每次执行 Datalog 程序仍然会对库图进行重新分析。我们希望能够通过摘要的方式省掉这部分重复的计算。

库模式分析。　ConDLog 会首先在库图 L 上进行一个库模式分析。库模式分析同样通过 Datalog 进行规约（见表格2.1中库分析部分）。我基于以下思路从全程序分析的规约中变换得到对库分析进行条件摘要的规约。

原则 2.1.1. 选择一个（些）关系作为摘要所使用的条件，然后给所有依赖于这条关系的关系加上额外的参数（将其条件化），最后对规则进行调整：

- 去掉跟应用端相关的基础规则。
- 将剩余规则中的关系替换成条件化的版本。
- 根据需要添加一些与摘要所使用条件相关的规则，比如自反性。

表格2.1中库分析的规约正是如此得到的：我选择 $\mathsf{r}(m)$ 作为条件进行摘要，新定义

出关系 reachableSum(m, k) 表示在 k 可达的条件下可以推出 m 可达；去掉了基础规则 reachable(m_0)，然后将归纳规则中的关系更换为带条件的版本，最后为 reachableSum 添加了自反性规则。在库图 L 上进行此分析，我们可以计算出如下摘要：

reachableSum(L4, L1)
reachableSum(L6, L5)
reachableSum(L8, L5)
reachableSum(L8, L6)
reachableSum(L10, L11)
⋮

端模式分析。 让我们来看 ConDlog 如何将之前在图 L 上计算得出的条件摘要应用在图 A 和 B 的分析上。直觉来说，为了尽可能地去掉中间计算，我们应该忽略所有从库节点出发的边，其相关可达性都用摘要计算。但这样做是不正确的：观察图 B，如果我们忽略了所有库节点出发的边，那么我们会遗漏掉 B2 这个可达节点。从而，ConDlog 首先会对摘要进行筛选，筛选函数 $Filter$ 见表格2.1中的端分析部分。因为我选择的条件是 r(m)，而这个条件成立并且可以忽略掉 m 出发的边当且仅当在应用端图不存在 m 到任意端节点的边。通过 $Filter$ 函数我们可以得到所有这样满足条件的 m，而后将其保存为 validPre 关系。端模式分析也同样使用 Datalog 描述规约，其规约也同样是从全程序分析的规约变换而来。类似地，变换的思路如下：

原则 2.1.2. *根据库模式分析中选择的作为条件的关系，将在右端使用到这种关系的规则拆分成两个规则：一种在摘要不满足条件时使用原来的规则进行推导，一种在摘要满足条件时实例化条件摘要。*

在图 A 中，节点 L1、L5、L6 相关的摘要是满足条件的，于是 ConDlog 可按下列顺序推出 A2 可达：

r(L1) :- ¬validPre(A0), r(A0), e(A0, L1).
r(L4) :- validPre(L1), r(L1), reachableSum(L4, L1).
r(A1) :- ¬validPre(L4), r(L4), e(L4, A1).
r(L5) :- ¬validPre(A1), r(A1), e(A1, L5).
r(L8) :- validPre(L5), r(L5), reachableSum(L8, L5).
r(A2) :- ¬validPre(L8), r(L8), e(L8, A2).

而在图 B 中，节点 L1 和 L5 相关的摘要是满足条件的，于是 ConDlog 可按下列顺

序推出 B2 可达：

$$
\begin{aligned}
&r(L1) \;:\!-\; \neg validPre(B0), r(B0), e(B0, L1).\\
&r(L4) \;:\!-\; validPre(L1), r(L1), reachableSum(L4, L1).\\
&r(B1) \;:\!-\; \neg validPre(L4), r(L4), e(L4, B1).\\
&r(L5) \;:\!-\; \neg validPre(B1), r(B1), e(B1, L5).\\
&r(L6) \;:\!-\; validPre(L5), r(L5), reachableSum(L6, L5).\\
&r(B2) \;:\!-\; \neg validPre(L6), r(L6), e(L6, B2).
\end{aligned}
$$

2.2 主要贡献

我的工作的主要学术贡献如下：

1. 我提出了独立分析库代码生成条件摘要，在应用端对摘要再进行筛选和实例化，以求加速程序分析的思想。这种模式具有相当的灵活性，使得我既可以在信息缺失的情况下进行分析，也能够探索更加复杂的分析。
2. 我根据这个思想，提出了基于 Datalog 的条件摘要技术 ConDLog。我为 ConDLog 的库分析和端分析提出了两个设计原则，揭示了其内在规律。
3. 我在两个分析上实现了 ConDLog 技术，并在实际的 Java 程序上取得了不错的性能提升。

表 2.1　ConDLog 需要的规约

图可达性分析（全程序分析）：

定义域：

$\mathbb{N}$ 表示图节点集合。

输入关系：

edge($m : \mathbb{N}, n : \mathbb{N}$)　　// 从节点 m 到节点 n 的边。

输出关系：

reachable($m : \mathbb{N}$)　　// 节点 m 可被节点 m_0 到达。

规则：

reachable(m_0).　　(1)

reachable(m) :- reachable(n), edge(n, m).　　(2)

图可达性分析（库模式分析）：

定义域：

$\mathbb{N}_{lib} \subset \mathbb{N}$ 表示库图节点集合。

输入关系：

$\text{edge}_{lib}(m : \mathbb{N}_{lib}, n : \mathbb{N}_{lib})$　　// 库图中从节点 m 到节点 n 的边。

输出关系：

reachableSum($m : \mathbb{N}_{lib}, k : \mathbb{N}_{lib}$)　　// 条件摘要：若节点 k 可达，则 m 也可达。

规则：

reachableSum(m, m).　　(1)

reachableSum(m, k) :- reachableSum(n, k), $\text{edge}_{lib}(n, m)$.　　(2)

图可达性分析（端模式分析）：

定义域：

$\mathbb{N}$ 表示图节点集合。

输入关系：

edge($m : \mathbb{N}, n : \mathbb{N}$)　　// 从节点 m 到节点 n 的边。

reachableSum($m : \mathbb{N}_{lib}, k : \mathbb{N}_{lib}$)　　// 库模式分析中生成的关系。

validPre($k : \mathbb{N}_{lib}$)　　// $Filter$ 筛选出的满足条件的节点。

输出关系：

reachable($m : \mathbb{N}$)　　// 节点 m 可被节点 m_0 到达。

规则：

reachable(m_0).　　(1)

reachable(m) :- ¬validPre(n), reachable(n), edge(n, m).　　(2)

reachable(m) :- validPre(n), reachable(n), reachableSum(m, n).　　(3)

筛选：

$Filter = \lambda m.\nexists n.(\text{edge}(m, n) \wedge n \in \mathbb{N}_{clt})$

第三章　背景知识

在这一章，我将简要介绍使用 Datalog 规约的程序分析相关的背景知识。§3.1介绍了 Datalog 的语法（Syntax）。§3.2介绍了 Datalog 的语义（Semantics）。§3.3介绍了 Datalog 的求解算法（Evaluation）。

3.1　Datalog 的语法

$$
\begin{array}{rl}
\text{分析} & C ::= \{c_1, \cdots, c_n\} \\
\text{约束} & c ::= l_0 \texttt{:-} l_1, \cdots, l_n \\
\text{字面量} & l ::= r(\alpha_1, \cdots, \alpha_n) \\
\text{参数} & \alpha ::= v \mid d \\
\text{变量} & v \in \mathbb{V} = \{x, y, \cdots\} \\
\text{常量} & d \in \mathbb{N} = \{0, 1, \cdots\} \\
\text{关系名称} & r \in \mathbb{R} = \{\mathsf{a}, \mathsf{b}, \cdots\} \\
\text{元组} & t \in \mathbb{T} = \mathbb{R} \times \mathbb{N}^* \\
\text{分析输入} & i \in \mathbb{I} \subseteq \mathbb{T} \\
\text{分析输出} & o \in \mathbb{O} \subseteq \mathbb{T}
\end{array}
$$

图 3.1　Datalog 分析的语法

图3.1展示了 Datalog 分析的语法。一个 Datalog 分析 C 由一个约束集合 $\{c_1, \cdots, c_n\}$ 构成。每条**约束**（Constriaint）$c \in C$ 都有一个约束头字面量 l_0，和一个由字面量集合构成的约束体 $\{l_1, \cdots, l_n\}$。一个**字面量**（Literal）由一个关系名称 r 和一列或为**变量**或为**常量**的参数表 $(\alpha_1, \cdots, \alpha_n)$ 构成。我们称一个只包含常量的字面量为**元组**（Tuple）或**地面字面量**（Grounded Literal）。Datalog 分析的输入是一个元组集合 $\mathbb{I}$，我们也称之为**分析输入**。Datalog 分析的输出同样是一个元组集合。为了允许避免特定分析结果的重复计算，我们添加了一个元组集合 $\mathbb{O}$ 用以表示用户感兴趣的输出结果，称之为**分析输出**。

示例 3.1.1. *以表格2.1中全程序分析的规约为例：*

$$
\begin{aligned}
&\mathsf{reachable}(m_0). && (1)\\
&\mathsf{reachable}(m) \texttt{:-} \mathsf{reachable}(n), \mathsf{edge}(n, m). && (2)
\end{aligned}
$$

这个分析包含了两条约束，一条基础规则和一条归纳规则。m_0 是一个常量，表示了整个图的入口节点；而 m, n 则是表示图节点的变量。形如 $\mathsf{reachable}(m)$、$\mathsf{edge}(n, m)$ 的都

是字面量。对于这个分析而言，关系名称为 edge 的元组构成了分析输入，关系名称为 reachable 的元组构成了分析输出。

3.2 Datalog 的语义

$$
\begin{array}{rcl}
[\![C]\!] & \in & 2^{\mathbb{I}} \to 2^{\mathbb{T}} \\
[\![c]\!] & \in & 2^{\mathbb{T}} \to 2^{\mathbb{T}} \\
[\![l]\!] & \in & \Sigma \to \mathbb{T}, \Sigma \stackrel{\text{def}}{=} V \to \mathbb{N} \\
[\![C]\!](I) & = & \textbf{lfp}\ \lambda T.T \cup I \cup \bigcup_{c \in C} [\![c]\!](T) \\
[\![l_0 \texttt{:-} l_1, \cdots, l_n]\!](T) & = & \{[\![l_0]\!](\sigma) \mid \bigwedge_{1 \le k \le n} [\![l_k]\!](\sigma) \in T \wedge \sigma \in \Sigma\} \\
[\![r(\alpha_1, \cdots, \alpha_n)]\!](\sigma) & = & r(sub(\alpha_1), \cdots, sub(\alpha_n)) \\
 & & sub(\alpha) \stackrel{\text{def}}{=} \begin{cases} \sigma(\alpha), & \alpha \in \mathbb{V} \\ \alpha, & \alpha \in \mathbb{N} \end{cases} \\
Gr(C,T) & = & \{[\![l_0]\!](\sigma) \texttt{:-} [\![l_1]\!](\sigma), \cdots, [\![l_n]\!](\sigma) \mid \\
 & & l_0 \texttt{:-} l_1, \cdots, l_n \in C \\
 & & \wedge \bigwedge_{1 \le k \le n} \sigma(k) \in [\![C]\!](T) \wedge \sigma \in \Sigma\}
\end{array}
$$

图 3.2　Datalog 分析的语义

图3.2展示了 Datalog 分析的语义。Datalog 分析通过从输入元组对分析约束计算最小不动点（**lfp**）来推导输出元组。这个推导从输入元组出发，重复地按照如下规则应用每条约束，直到没有新的元组可以再被推出：对于一个变量到常量的赋值 σ，如果约束体字面量在赋值 σ 下对应的元组都已经在当前已有的元组集合中，那么可以推出约束头字面量在赋值 σ 下对应的元组。在这个过程中，我们称只由常量组成的约束为**地面约束**（Grounded Constraint）。我们将分析 C 作用在输入元组集合 T 上产生的所有地面约束记为 $Gr(C,T)$。

示例 3.2.1. *承接示例3.1.1，设 $m_0 = 0$，输入元组集合*

$$\mathbb{I} = \{\mathsf{edge}(0,1), \mathsf{edge}(1,2), \mathsf{edge}(2,3)\}$$

*包含三条边。记该 Datalog 分析对应的转移函数 $f \in 2^{\mathbb{T}} \to 2^{\mathbb{T}}$ 为 $\lambda T.T \cup \mathbb{I} \cup [\![c_1]\!](T) \cup [\![c_2]\!](T)$，其中 c_1 和 c_2 分别表示两条规则，那么分析的结果就可以由 f 的最小不动点来表述，即 **lfp** f。根据克莱尼不动点定理（Kleene Fixed-point Theorem），我们可以反复作用函数 f 直到其收敛（简便起见，我们用 $\mathsf{r}(n)$ 表示 $\mathsf{reachable}(n)$，用 $\mathsf{e}(n,m)$ 表示*

$\mathsf{edge}(n, m))$:

$$
\begin{aligned}
T_0 &= \mathbb{I} \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} \\
T_1 &= f(T_0) = T_0 \cup \mathbb{I} \cup [\![c_1]\!](T_0) \cup [\![c_2]\!](T_0) \\
&= T_0 \cup \mathbb{I} \cup \{\mathsf{r}(0)\} \cup \varnothing \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3), \mathsf{r}(0)\} \\
T_2 &= f(T_1) = T_1 \cup \mathbb{I} \cup [\![c_1]\!](T_1) \cup [\![c_2]\!](T_1) \\
&= T_1 \cup \mathbb{I} \cup \{\mathsf{r}(0)\} \cup \{\mathsf{r}(1)\} \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3), \mathsf{r}(0), \mathsf{r}(1)\} \\
T_3 &= f(T_2) = T_2 \cup \mathbb{I} \cup [\![c_1]\!](T_2) \cup [\![c_2]\!](T_2) \\
&= T_2 \cup \mathbb{I} \cup \{\mathsf{r}(0)\} \cup \{\mathsf{r}(1), \mathsf{r}(2)\} \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3), \mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2)\} \\
T_4 &= f(T_3) = T_3 \cup \mathbb{I} \cup [\![c_1]\!](T_3) \cup [\![c_2]\!](T_3) \\
&= T_3 \cup \mathbb{I} \cup \{\mathsf{r}(0)\} \cup \{\mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\} \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3), \mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\} \\
T_5 &= f(T_4) = T_4 \cup \mathbb{I} \cup [\![c_1]\!](T_4) \cup [\![c_2]\!](T_4) \\
&= T_4 \cup \mathbb{I} \cup \{\mathsf{r}(0)\} \cup \{\mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\} \\
&= \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3), \mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\}
\end{aligned}
$$

因为 $T_4 = T_5$，从而该 *Datalog* 分析在输入 $\mathbb{I}$ 对应的输出元组为 $\{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\}$。

3.3 Datalog 的求解算法

在这一节中，我们只关注自底向上（Bottom-up）的 Datalog 求解算法 [1]。

3.3.1 朴素求解

在 §3.2中我们定义了 Datalog 的不动点语义，实际上克莱尼不动点定理给出了一个简单直接的算法，即示例3.2.1中使用的反复作用 Datalog 分析对应的转移函数直至其收敛。在此基础上稍作变换，就得到了 Datalog 的**朴素求解**（Naive Evaluation）算法。

设关系名称集合为 $r_1, \cdots, r_m$，同时我们将元组集合 $\mathbb{T}$ 按关系名称拆分为 $\mathbb{T}_1, \cdots, \mathbb{T}_m$。我们为每个关系 r_i 定义一个转移函数 $q_i \in 2^{\mathbb{T}_1} \times \cdots \times 2^{\mathbb{T}_m} \to 2^{\mathbb{T}_i}$。设以 r_i 关系名称作为头部字面量的规则集合为 C_i，那么

$$
q_i = \lambda(T_1, \cdots, T_m). \bigcup_{c \in C_i} [\![c]\!]\Big(\bigcup_{1 \le j \le m} T_j\Big)
$$

即计算满足某条 C_i 中约束的元组集合的并。算法1列出了该算法的伪代码。这是一个基于迭代的算法：$R_i^{(k)}$ 表示第 k 轮迭代后关系名称为 r_i 的元组集合。

算法 1 Datalog 的朴素求解算法

```
for all i do
    R_i^(0) ← ∅
end for
k ← 0
repeat
    k ← k + 1
    同时进行 R_i^(k) ← q_i(R_1^(k−1), ··· , R_m^(k−1)) 操作
until ∀i.(R_i^(k) = R_i^(k−1))
输出 R_1^(k), ··· , R_m^(k)
```

示例 3.3.1. 让我们再来看示例*3.2.1*。在这个示例上使用 *Datalog* 的朴素算法求解，我们可得如下的迭代序列（E 表示关系 e，R 表示关系 r）：

$$
\begin{array}{lcllcl}
E^{(0)} & = & \varnothing & R^{(0)} & = & \varnothing \\
E^{(1)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(1)} & = & \{\mathsf{r}(0)\} \\
E^{(2)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(2)} & = & \{\mathsf{r}(0), \mathsf{r}(1)\} \\
E^{(3)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(3)} & = & \{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2)\} \\
E^{(4)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(4)} & = & \{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\} \\
E^{(5)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(5)} & = & \{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\}
\end{array}
$$

3.3.2 半朴素求解

观察示例3.3.1中的计算，我们可以发现有大量的重复计算：r(0) 被重复产生了四次，r(1) 被重复产生了三次，而所有的以 e 为名称的元组在每次迭代都重新产生了一遍。为了减少这种冗余计算，人们提出了 Datalog 的**半朴素求解**（Semi-naive Evaluation）算法。这个算法的核心思路在于尽量只计算出新的元组，减少旧元组被重复推出的次数。仍以示例3.1.1中的 Datalog 分析为例。在朴素求解中，设我们想要进行 $R^{(k+1)} \leftarrow q(E^{(k)}, R^{(k)})$ 这一步计算。令 $T^{(k)} = T^{(k-1)} \cup \Delta T^{(k)}$，其中 T 表示任一关系名称对应的元组集合。那么，在半朴素求解中，在第 k 轮迭代中我们希望求解 $\Delta R^{(k)}$，即新的元组集合。根据转移函

数 q 的定义，易知其对每个参数都是关于集合并可分配的，从而

$$\begin{aligned} R^{(k+1)} &= q(E^{(k)}, R^{(k)}) \\ &= q(E^{(k-1)} \cup \Delta E^{(k)}, R^{(k-1)} \cup \Delta R^{(k)}) \\ &= q(E^{(k-1)}, R^{(k-1)}) \cup q(E^{(k-1)}, \Delta R^{(k)}) \cup q(\Delta E^{(k)}, R^{(k-1)}) \cup q(\Delta E^{(k)}, \Delta R^{(k)}) \\ &= R^{(k)} \cup q(E^{(k-1)}, \Delta R^{(k)}) \cup q(\Delta E^{(k)}, R^{(k-1)}) \cup q(\Delta E^{(k)}, \Delta R^{(k)}) \end{aligned}$$

由此可见，我们并不需要在每一轮迭代时重新计算出所有的元组，而是**增量**地计算出新的元组。我们对每个转移函数 q_i，记 Δq_i 表示其增量计算的版本

$$\Delta q_i = \lambda(T_1, \cdots, T_m, \Delta T_1, \cdots, \Delta T_m). \bigcup_{c \in C_i} \bigcup_{1 \le j \le m} [\![c]\!](\bigcup_{1 \le k < j} T_k \cup \bigcup_{j \le k \le m} \Delta T_k)$$

算法2列出了该算法的伪代码。这同样是一个基于迭代的算法，不过我们将迭代计算换成了增量版本，从而减少了冗余计算量。

算法 2 Datalog 的半朴素求解算法

for all i **do**
    $R_i^{(0)} \leftarrow \varnothing$
end for
$k \leftarrow 1$
同时进行 $\Delta R_i^{(1)} \leftarrow q_i(R_1^{(0)}, \cdots, R_m^{(0)})$ 操作
repeat
    $k \leftarrow k + 1$
    for all i **do**
        $R_i^{(k-1)} \leftarrow R_i^{(k-2)} \cup \Delta R_i^{(k-1)}$
    end for
    同时进行 $\Delta R_i^{(k)} \leftarrow \Delta q_i(R_1^{(k-2)}, \cdots, R_m^{(k-2)}, \Delta R_1^{(k-1)}, \cdots, \Delta R_m^{(k-1)}) - R_i^{(k-1)}$ 操作
until $\forall i.(\Delta R_i^{(k)} = \varnothing)$
输出 $R_1^{(k-1)}, \cdots, R_m^{(k-1)}$

示例 3.3.2. 同样以示例*3.2.1*为例，我展示一下 *Datalog* 的半朴素求解算法的迭代序列：

$$
\begin{array}{lcllcl}
E^{(0)} & = & \varnothing & R^{(0)} & = & \varnothing \\
\Delta E^{(1)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & \Delta R^{(1)} & = & \{\mathsf{r}(0)\} \\
E^{(1)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(1)} & = & \{\mathsf{r}(0)\} \\
\Delta E^{(2)} & = & \varnothing & \Delta R^{(2)} & = & \{\mathsf{r}(1)\} \\
E^{(2)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(2)} & = & \{\mathsf{r}(0), \mathsf{r}(1)\} \\
\Delta E^{(3)} & = & \varnothing & \Delta R^{(3)} & = & \{\mathsf{r}(2)\} \\
E^{(3)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(3)} & = & \{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2)\} \\
\Delta E^{(4)} & = & \varnothing & \Delta R^{(4)} & = & \{\mathsf{r}(3)\} \\
E^{(4)} & = & \{\mathsf{e}(0,1), \mathsf{e}(1,2), \mathsf{e}(2,3)\} & R^{(4)} & = & \{\mathsf{r}(0), \mathsf{r}(1), \mathsf{r}(2), \mathsf{r}(3)\} \\
\Delta E^{(5)} & = & \varnothing & \Delta R^{(5)} & = & \varnothing
\end{array}
$$

虽然仍然进行了五轮迭代，但是实际的运算量都在 ΔE 和 ΔR 上，从而诸如 r(0)、r(1) 等都只被推出过一次。

Datalog 求解上也有不少的优化技巧。其中之一就是规定一个关系的计算优先顺序：我们构造一个关系种类上的有向图，如果关系 r 在某条头部关系为 s 的约束的约束体中出现，那么添加一条从 r 到 s 的有向边。然后我们在这个图上进行拓扑排序，直观来看排序更靠前的关系也更加基础，被更多关系所依赖。从而我们按照拓扑序，依次在每个强连通分支上进行半朴素求解算法，最后计算出答案。

第四章　ConDlog 及其两个实例

在这一章，我将详细介绍我提出的基于 Datalog 的条件摘要技术 ConDlog 及其在 Java 程序上的两个分析实例。§4.1介绍了 ConDlog 技术的计算流程。§4.2介绍了 Java 上的类层次分析，以及我为其实现的 ConDlog 技术。§4.3介绍了 Java 上的快速类型分析，以及我为其实现的 ConDlog 技术。

4.1　ConDlog 技术流程

ConDlog技术包含库分析和端分析两部分。库分析流程如图4.1。在库分析时，ConDlog 的输入为独立的库代码和一个 Datalog 描述的库分析规约，应用 Datalog 的求解算法，计算出条件摘要。这个条件摘要会被存储在磁盘中，由端分析加载使用。

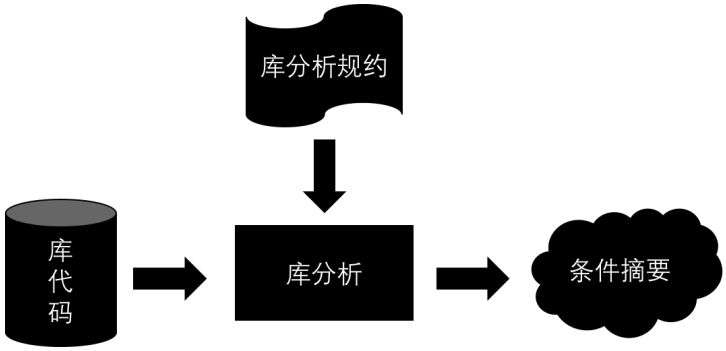


图 4.1　ConDlog 库分析流程

端分析流程如图4.2。在端分析时，ConDlog 的输入为库代码、端代码、预先计算好的条件摘要、摘要合法性筛选函数和一个 Datalog 描述库分析规约。ConDlog 先根据条件摘要和摘要筛选函数选出可以使用的摘要，然后将满足条件的摘要实例化，根据库分析规约，应用 Datalog 的求解算法，计算出最终答案。

在 ConDlog 技术的工作流程中，程序分析用户需要设计的主要是库分析规约、端分析规约和摘要合法性筛选函数。其中，筛选函数是在库分析规约和端分析规约的基础上制定，相对比较简单。库分析规约和端分析规约都是由原程序分析的全程序分析规约变换得到。在第二章中我给出了库分析规约和端分析规约的设计规律和原则，我在这里重复如下：

原则 4.1.1 (库分析规约设计)**.** 选择一个（些）关系作为摘要所使用的条件，然后给所有依赖于这条关系的关系加上额外的参数（将其条件化），最后对规则进行调整：

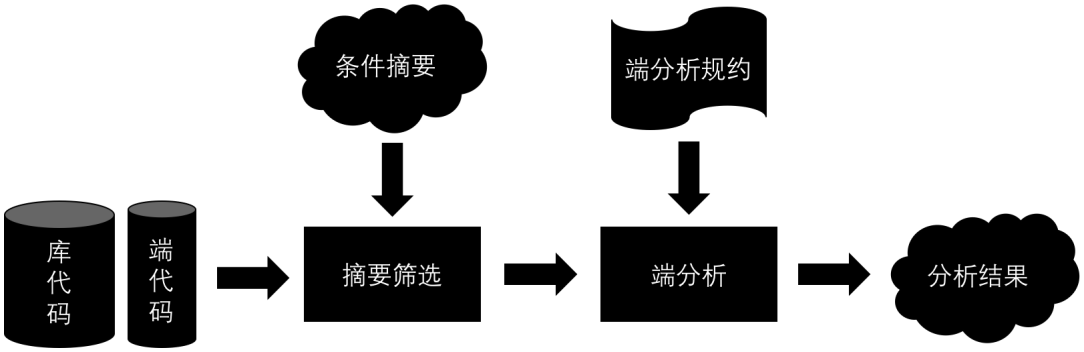


图 4.2 ConDlog 端分析流程

- 去掉跟应用端相关的基础规则。
- 将剩余规则中的关系替换成条件化的版本。
- 根据需要添加一些与摘要所使用条件相关的规则，比如自反性。

原则 4.1.2 (端分析规约设计)**.** 根据库模式分析中选择的作为条件的关系，将在右端使用到这种关系的规则拆分成两个规则：一种在摘要不满足条件时使用原来的规则进行推导，一种在摘要满足条件时实例化条件摘要。

4.2 实例一：类层次分析

表格4.1中呈现了 Java 程序的**类层次分析**（Class Hierarchy Analysis）全程序分析的 Datalog 规约。这个分析基于类型层次计算出程序的方法调用图，即规约中的 callGraph 关系。这个关系中的元组 (i, m) 表示从主方法出发可以到达方法 m，且方法 m 是方法调用指令 i 可能的调用目标。顾名思义，类层次分析是仅通过程序中的类型层次关系算出所有方法调用目标的上近似（或者说超集）。设方法调用 i 的接受者类型为 t，调用的方法签名为 m，要分析其实际可能的目标方法，我们先获取 t 的所有子类型。然后我们观察这些子类型，我们选择那些重载了方法签名 m 的方法作为可能的调用目标（参考表格4.1中的规则 (3)）。

我接下来讨论将 ConDlog 库分析技术应用到类层次分析上。我们需要先选择一个关系作为条件摘要中所使用的条件。观察表格4.1中的规则，很多关系都是从 rMethod 关系推出的，而且在独立分析库代码时，因为缺乏主方法的信息，可达方法这部分信息是无法计算的。因此，我们选择 rMethod 关系作为条件。接下来，我们需要将依赖于 rMethod 的关系都转化成带条件的版本。如表格4.2中所示，我们将关系 rMethod、rInvoke、target、callGraph 变换成了带条件的关系 rMethodSum、rInvokeSum、targetSum、callGraphSum。然后，我们去掉与应用端信息相关的基础规则，再把剩下的规则中的关系替换为带条件的版本。与第二章中使用的例子不同的是，我们由于考虑到计算摘要的性能问题，定

表 4.1　类层次分析全程序分析的 Datalog 规约

类层次分析（全程序分析）：

定义域：

$\mathbb{M}$ 表示程序中方法的集合。
$\mathbb{I}$ 表示程序中方法调用指令的集合。
$\mathbb{T}$ 表示程序中引用类型的集合。

输入关系：

dispatch($m_1 : \mathbb{M}, t : \mathbb{T}, m_2 : \mathbb{M}$)	// 虚拟调用类型 t 中签名为 m_1 的方法 // 会实际调用 m_2。
body($m : \mathbb{M}, i : \mathbb{I}$)	// 方法 m 包含方法调用指令 i。
binding($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 的方法签名为 m。
receiver($i : \mathbb{I}, t : \mathbb{T}$)	// 方法调用指令 i 的接受者类型为 t。
subtype($t_1 : \mathbb{T}, t_2 : \mathbb{T}$)	// 类型 t_1 是类型 t_2 的子类型。

中间关系：

rMethod($m : \mathbb{M}$)	// 方法 m 可达。
rInvoke($i : \mathbb{I}$)	// 方法调用指令 i 可达。
target($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 可能调用方法 m。

输出关系：

callGraph($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用图，与 target 一样。

规则：

rMethod(m_{main}). (1)

rInvoke(i) :- rMethod(m), body(m, i). (2)

target(i, m_2) :- rInvoke(i), receiver(i, t_1), binding(i, m_1),
subtype(t_2, t_1), dispatch(m_1, t_2, m_2). (3)

rMethod(m) :- target(_, m). (4)

callGraph(i, m) :- target(i, m). (5)

义的 rMethodSum 并不是传递闭包，而只是单步的。根据这个设计，我们希望在端分析时，如果一个库方法可达，那么我们可以通过 callGraphSum 知道这个方法里的方法调用指令可能的目标函数，而不用再考虑其中的中间计算。

然后我们讨论将 ConDlog 端分析技术应用到类层次分析上。为了保证计算出的输出关系与直接计算全程序分析得出的输出关系相同，我们需要对摘要进行筛选。上文中提到，如果一个库方法 k 可达，那么我们希望可以使用摘要 callGraphSum(_, _, k) 直接计算出其中的方法调用关系，而不用去考察其方法体。要保证这样做的正确性，我们需要确认对方法 k 中的任意方法调用 i，如果 i 的接受者类型为 t，方法签名为 m，在应用端代码中没有任何类型重载了 t 的 m 方法。因此，我们可以将筛选函数 $Filter$ 设计为：如果应用端代码中没有定义任何新的这些 t 的子类，那么方法 k 是满足条件的。端分析的 Datalog 规约和 $Filter$ 的定义见表格4.3。因为在库模式分析中我们没有为 rMethodSum 计算传递闭包，所以我们将 rMethodSum 和 callGraphSum 都作为端模式分析的输入。对于 rInvoke 关系来说，因为这不是分析所需的输出关系，所以我们只用

表 4.2　类层次分析库模式分析的 Datalog 规约

类层次分析（库模式分析）：

定义域：

$\mathbb{M}_{lib} \subset \mathbb{M}$ 表示库程序中方法的集合。
$\mathbb{I}_{lib} \subset \mathbb{I}$ 表示库程序中方法调用指令的集合。
$\mathbb{T}_{lib} \subset \mathbb{T}$ 表示库程序中引用类型的集合。

输入关系：

$\text{dispatch}_{lib}(m_1 : \mathbb{M}_{lib}, t : \mathbb{T}_{lib}, m_2 : \mathbb{M}_{lib})$	// 虚拟调用类型 t 中签名为 m_1 的方法 // 会实际调用 m_2。
$\text{body}_{lib}(m : \mathbb{M}_{lib}, i : \mathbb{I}_{lib})$	// 方法 m 包含方法调用指令 i。
$\text{binding}_{lib}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib})$	// 方法调用指令 i 的方法签名为 m。
$\text{receiver}_{lib}(i : \mathbb{I}_{lib}, t : \mathbb{T}_{lib})$	// 方法调用指令 i 的接受者类型为 t。
$\text{subtype}_{lib}(t_1 : \mathbb{T}_{lib}, t_2 : \mathbb{T}_{lib})$	// 类型 t_1 是类型 t_2 的子类型。

输出关系：

$\text{rMethodSum}(m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib})$	// 条件摘要：若方法 k 可达， // 则方法 m 可达。
$\text{rInvokeSum}(i : \mathbb{I}_{lib}, k : \mathbb{M}_{lib})$	// 条件摘要：若方法 k 可达， // 则方法调用指令 i 可达。
$\text{targetSum}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib})$	// 条件摘要：若方法 k 可达， // 则方法调用指令 i 可能调用方法 m。
$\text{callGraphSum}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib})$	// 带条件方法调用图，与 targetSum 一样。

规则：

$$\text{rInvokeSum}(i, k) \text{ :- } \text{body}_{lib}(k, i). \quad (1)$$

$$\text{targetSum}(i, m_2, k) \text{ :- } \text{rInvokeSum}(i, k), \text{receiver}_{lib}(i, t_1), \text{binding}_{lib}(i, m_1), \text{subtype}_{lib}(t_2, t_1), \text{dispatch}_{lib}(m_1, t_2, m_2). \quad (2)$$

$$\text{rMethodSum}(m, k) \text{ :- } \text{targetSum}(\_, m, k). \quad (3)$$

$$\text{callGraphSum}(i, m, k) \text{ :- } \text{targetSum}(i, m, k). \quad (4)$$

按需计算，即满足条件的方法内部的方法调用不用考虑，从而表格4.3中的规则 (2) 只考虑了不满足筛选条件的可达方法。最后，我添加了规则 (5) 和 (7) 用以实例化满足条件的摘要，分别计算出 rMethod 和 callGraph。

4.3　实例二：快速类型分析

表格4.4中呈现了 Java 程序的**快速类型分析**（Rapid Type Analysis）全程序分析的 Datalog 规约。快速类型分析之所以被称为“快速”，就是相较于类层次分析，它有更好的精度和速度。我注意到，在类层次分析分析中，一个方法调用的目标有可能是方法调用接受者类型的任意一个子类型的重载方法，这是相当不精确的。实际上，我们只用考虑程序中可达的新建过实例的类型就可以了（参考表格4.4中的规则 (4)）。要计算程序中可达的新建过实例的类型 t，需要考察所有从主方法可达的方法 m，和方法 m 中所有新建实例的指令 h，这些 h 的类型都是可达的（参考表格4.4中的规则 (3)）。

我接下来讨论将 ConDlog 库分析技术应用到快速类型分析上。与上文中的类层次分析不同，我们不太容易选择一个关系作为条件摘要中所使用的条件。观察表格4.4的规则 (4)，约束右端同时具有 rMethod 和 rType，所以我们可以将这两个关系同时作为条件。接下来，我们需要将依赖这些条件的关系都转化成带条件的版本。具体的变化已经陈列在表格4.5中。我们将 rMethod(m) 替换成了带有方法 k 可达和类型 t 可被新建的条件的 rMethodSum(m,k,t)，将 rInvoke(i) 替换成了带有方法 k 可达的条件的 rInvokeSum(i,k)，将 rType(t) 替换成了带有方法 k 可达的条件的 rTypeSum(t,k)，将 target(i,m) 和 callGraph(i,m) 替换成了带有方法 k 可达和类型 t 可被新建的条件的 targetSum(i,m,k,t) 和 callGraphSum(i,m,k,t)。然后，同样的，我们去掉与应用端信息相关的基础规则，再把剩下的规则中的关系替换成带条件的版本。我们希望这样计算出来的摘要，能够使得在端分析时，知道一个库方法 k 可达和一个库类型可被新建的时候，可以直接使用 callGraphSum 关系推出调用图关系，而不用再考虑其中的中间计算。

然后我讨论将 ConDlog 端分析技术应用到快速类型分析上。同样的，在进行关系计算前，也要对摘要进行筛选。上文中提到，如果一个库方法 k 可达和一个库类型 t 可被新建实例，那么我们希望可以使用摘要 callGraphSum(_, _, k,t) 直接计算出其中的方法调用关系，而不用去考察其方法体。要保证这样做的正确性，我们需要确认对方法 k 中的任意方法调用 i，如果 i 的接受者类型为 t，方法签名为 m，在应用端代码中没有新的 t 的子类 s，或者没有新建子类 s 的实例，或者 s 中没有重载方法 m。因此，我们可以将筛选函数 $Filter$ 设计为：如果应用端中没有新的 t 的子类 s 或者没有新建 s 的实例，那么方法 k 是满足条件的。端分析的 Datalog 规约和 $Filter$ 的定义见表格4.6。表格4.6中的规则 (6) 和规则 (9) 实例化满足条件的摘要，分别计算出 rMethod 和 callGraph。与类层次分析的端分析不同，因为我对快速类型分析的摘要同时考虑了可达方法和可被新建类型，所以在规则中除了判断库方法满足条件 validPre，还需要判断库类型可被新建 rType。

表 4.3　类层次分析端模式分析的 Datalog 规约

类层次分析（端模式分析）：

定义域：

$\mathbb{M}$ 表示程序中方法的集合。
$\mathbb{I}$ 表示程序中方法调用指令的集合。
$\mathbb{T}$ 表示程序中引用类型的集合。

输入关系：

dispatch($m_1 : \mathbb{M}, t : \mathbb{T}, m_2 : \mathbb{M}$)	// 虚拟调用类型 t 中签名为 m_1 的方法 // 会实际调用 m_2。
body($m : \mathbb{M}, i : \mathbb{I}$)	// 方法 m 包含方法调用指令 i。
binding($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 的方法签名为 m。
receiver($i : \mathbb{I}, t : \mathbb{T}$)	// 方法调用指令 i 的接受者类型为 t。
subtype($t_1 : \mathbb{T}, t_2 : \mathbb{T}$)	// 类型 t_1 是类型 t_2 的子类型。
rMethodSum($m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}$)	// 库模式分析中生成的关系。
callGraphSum($i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}$)	// 库模式分析中生成的关系。
validPre($k : \mathbb{M}_{lib}$)	// $Filter$ 筛选出的满足条件的库方法。

中间关系：

rMethod($m : \mathbb{M}$)	// 方法 m 可达。
rInvoke($i : \mathbb{I}$)	// 方法调用指令 i 可达。
target($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 可能调用方法 m。

输出关系：

callGraph($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用图，与 target 一样。

规则：

rMethod(m_{main}). (1)

rInvoke(i) :- ¬validPre(m), rMethod(m), body(m, i). (2)

target(i, m_2) :- rInvoke(i), receiver(i, t_1), binding(i, m_1),
subtype(t_2, t_1), dispatch(m_1, t_2, m_2). (3)

rMethod(m) :- target(_, m). (4)

rMethod(m) :- validPre(k), rMethod(k), rMethodSum(m, k). (5)

callGraph(i, m) :- target(i, m). (6)

callGraph(i, m) :- validPre(k), rMethod(k), callGraphSum(i, m, k). (7)

筛选：

$$Filter = \lambda m.\forall i, t.((\text{body}(m, i) \wedge \text{receiver}(i, t)) \implies \nexists t'.(t' \in \mathbb{T}_{clt} \wedge \text{subtype}(t', t)))$$

表 4.4　快速类型分析全程序分析的 Datalog 规约

快速类型分析（全程序分析）：

定义域：

$\mathbb{M}$ 表示程序中方法的集合。
$\mathbb{I}$ 表示程序中方法调用指令的集合。
$\mathbb{T}$ 表示程序中引用类型的集合。
$\mathbb{H}$ 表示程序中新建实例指令的集合。

输入关系：

dispatch($m_1 : \mathbb{M}, t : \mathbb{T}, m_2 : \mathbb{M}$)　// 虚拟调用类型 t 中签名为 m_1 的方法会实际调用 m_2。
bodyI($m : \mathbb{M}, i : \mathbb{I}$)　// 方法 m 包含方法调用指令 i。
bodyH($m : \mathbb{M}, h : \mathbb{H}$)　// 方法 m 包含新建实例指令 h。
hType($h : \mathbb{H}, t : \mathbb{T}$)　// 新建实例指令 h 创建的实例类型为 t。
binding($i : \mathbb{I}, m : \mathbb{M}$)　// 方法调用指令 i 的方法签名为 m。
receiver($i : \mathbb{I}, t : \mathbb{T}$)　// 方法调用指令 i 的接受者类型为 t。
subtype($t_1 : \mathbb{T}, t_2 : \mathbb{T}$)　// 类型 t_1 是类型 t_2 的子类型。

中间关系：

rMethod($m : \mathbb{M}$)　// 方法 m 可达。
rType($t : \mathbb{T}$)　// 类型 t 的实例可能被创建。
rInvoke($i : \mathbb{I}$)　// 方法调用指令 i 可达。
target($i : \mathbb{I}, m : \mathbb{M}$)　// 方法调用指令 i 可能调用方法 m。

输出关系：

callGraph($i : \mathbb{I}, m : \mathbb{M}$)　// 方法调用图，与 target 一样。

规则：

rMethod(m_{main}). (1)
rInvoke(i) :- rMethod(m), bodyI(m, i). (2)
rType(t) :- rMethod(m), bodyH(m, h), hType(h, t). (3)
target(i, m_2) :- rInvoke(i), receiver(i, t_1), binding(i, m_1), subtype(t_2, t_1), rType(t_2), dispatch(m_1, t_2, m_2). (4)
rMethod(m) :- target(_, m). (5)
callGraph(i, m) :- target(i, m). (6)

表 4.5　快速类型分析库模式分析的 Datalog 规约

快速类型分析（库模式分析）：

定义域：

$\mathbb{M}_{lib} \subset \mathbb{M}$ 表示库程序中方法的集合。
$\mathbb{I}_{lib} \subset \mathbb{I}$ 表示库程序中方法调用指令的集合。
$\mathbb{T}_{lib} \subset \mathbb{T}$ 表示库程序中引用类型的集合。
$\mathbb{H}_{lib} \subset \mathbb{H}$ 表示库程序中新建实例指令的集合。

输入关系：

$\text{dispatch}_{lib}(m_1 : \mathbb{M}_{lib}, t : \mathbb{T}_{lib}, m_2 : \mathbb{M}_{lib})$	// 虚拟调用类型 t 中签名为 m_1 的方法 // 会实际调用 m_2。
$\text{bodyI}_{lib}(m : \mathbb{M}_{lib}, i : \mathbb{I}_{lib})$	// 方法 m 包含方法调用指令 i。
$\text{bodyH}_{lib}(m : \mathbb{M}_{lib}, h : \mathbb{H}_{lib})$	// 方法 m 包含新建实例指令 h。
$\text{hType}_{lib}(h : \mathbb{H}_{lib}, t : \mathbb{T}_{lib})$	// 新建实例指令 h 创建的实例类型为 t。
$\text{binding}_{lib}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib})$	// 方法调用指令 i 的方法签名为 m。
$\text{receiver}_{lib}(i : \mathbb{I}_{lib}, t : \mathbb{T}_{lib})$	// 方法调用指令 i 的接受者类型为 t。
$\text{subtype}_{lib}(t_1 : \mathbb{T}_{lib}, t_2 : \mathbb{T}_{lib})$	// 类型 t_1 是类型 t_2 的子类型。

输出关系：

$\text{rMethodSum}(m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}, t : \mathbb{T}_{lib})$	// 条件摘要：若方法 k 可达和 // 类型 t 可被新建，则方法 m 可达。
$\text{rInvokeSum}(i : \mathbb{I}_{lib}, k : \mathbb{M}_{lib})$	// 条件摘要：若方法 k 可达， // 则方法调用指令 i 可达。
$\text{rTypeSum}(t : \mathbb{T}_{lib}, k : \mathbb{M}_{lib})$	// 条件摘要：若方法 k 可达， // 则类型 t 可被新建。

$\text{targetSum}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}, t : \mathbb{T}_{lib})$	// 条件摘要：若方法 k 可达和
	// 类型 t 可被新建，则方法调用
	// 指令 i 可能调用方法 m。

$\text{callGraphSum}(i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}, t : \mathbb{T}_{lib})$	// 带条件方法调用图，与 // targetSum 一样。

规则：

$$\text{rInvokeSum}(i, k) \text{ :- } \text{bodyI}_{lib}(k, i). \tag{1}$$

$$\text{rTypeSum}(t, k) \text{ :- } \text{bodyH}_{lib}(k, h), \text{hType}_{lib}(h, t). \tag{2}$$

$$\text{targetSum}(i, m_2, k, t_2) \text{ :- } \text{rInvokeSum}(i, k), \text{receiver}_{lib}(i, t_1), \text{binding}_{lib}(i, m_1), \text{subtype}_{lib}(t_2, t_1), \text{dispatch}_{lib}(m_1, t_2, m_2). \tag{3}$$

$$\text{rMethodSum}(m, k, t) \text{ :- } \text{targetSum}(\_, m, k, t). \tag{4}$$

$$\text{callGraphSum}(i, m, k, t) \text{ :- } \text{targetSum}(i, m, k, t). \tag{5}$$

表 4.6　快速类型分析端模式分析的 Datalog 规约

快速类型分析（端模式分析）：

定义域：

$\mathbb{M}$ 表示程序中方法的集合。
$\mathbb{I}$ 表示程序中方法调用指令的集合。
$\mathbb{T}$ 表示程序中引用类型的集合。
$\mathbb{H}$ 表示程序中新建实例指令的集合。

输入关系：

dispatch($m_1 : \mathbb{M}, t : \mathbb{T}, m_2 : \mathbb{M}$)	// 虚拟调用类型 t 中签名为 m_1 的方法会实际调用 m_2。
bodyI($m : \mathbb{M}, i : \mathbb{I}$)	// 方法 m 包含方法调用指令 i。
bodyH($m : \mathbb{M}, h : \mathbb{H}$)	// 方法 m 包含新建实例指令 h。
hType($h : \mathbb{H}, t : \mathbb{T}$)	// 新建实例指令 h 创建的实例类型为 t。
binding($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 的方法签名为 m。
receiver($i : \mathbb{I}, t : \mathbb{T}$)	// 方法调用指令 i 的接受者类型为 t。
subtype($t_1 : \mathbb{T}, t_2 : \mathbb{T}$)	// 类型 t_1 是类型 t_2 的子类型。
rMethodSum($m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}, t : \mathbb{T}_{lib}$)	// 库模式分析中生成的关系。
rTypeSum($t : \mathbb{T}_{lib}, k : \mathbb{M}_{lib}$)	// 库模式分析中生成的关系。
callGraphSum($i : \mathbb{I}_{lib}, m : \mathbb{M}_{lib}, k : \mathbb{M}_{lib}, t : \mathbb{T}_{lib}$)	// 库模式分析中生成的关系。
validPre($k : \mathbb{M}_{lib}$)	// $Filter$ 筛选出的满足条件的库方法。

中间关系：

rMethod($m : \mathbb{M}$)	// 方法 m 可达。
rInvoke($i : \mathbb{I}$)	// 方法调用指令 i 可达。
rType($t : \mathbb{T}$)	// 类型 t 的实例可能被创建。
target($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用指令 i 可能调用方法 m。

输出关系：

callGraph($i : \mathbb{I}, m : \mathbb{M}$)	// 方法调用图，与 target 一样。

规则：

rMethod(m_{main}). (1)

rInvoke(i) :- ¬validPre(m), rMethod(m), bodyI(m, i). (2)

rType(t) :- ¬validPre(m), rMethod(m), bodyH(m, h), hType(h, t). (3)

target(i, m_2) :- rInvoke(i), receiver(i, t_1), binding(i, m_1),
subtype(t_2, t_1), rType(t_2), dispatch(m_1, t_2, m_2). (4)

rMethod(m) :- target(_, m). (5)

rMethod(m) :- validPre(k), rType(t), rMethod(k), rMethodSum(m, k, t). (6)

rType(t) :- validPre(k), rMethod(k), rTypeSum(t, k). (7)

callGraph(i, m) :- target(i, m). (8)

callGraph(i, m) :- validPre(k), rType(t), rMethod(k), callGraphSum(i, m, k, t). (9)

筛选：

$$Filter = \lambda m.\forall i, t.((\text{body}(m, i) \wedge \text{receiver}(i, t)) \implies \nexists t'.(t' \in \mathbb{T}_{clt} \wedge \text{subtype}(t', t) \wedge \exists h.(\text{hType}(h, t'))))$$

第五章　实验研究

在这一章，我将介绍 ConDlog 技术的具体实现，在 Java 上两个程序分析的实验设计，以及实际的实验效果。§5.1介绍了 ConDlog 工具的设计与架构。§5.2介绍了我为两个分析选取的 Java 程序测试集，和其他实验细节。§5.3介绍了 ConDlog 工具在两个分析上取得的实际效果，并进行了分析。

5.1　工具实现

我以 Java 代码的程序分析为目的设计并实现了 ConDlog 工具。ConDlog 使用 WALA [15] 作为 Java 字节码分析前端，用于解析库代码和端代码中的类型、方法、控制流图等。ConDlog 使用 bddbddb [34] 作为 Datalog 求解的后端。

WALA 是 IBM 公司管理的一个开源项目，为 Java 字节码上的程序分析提供了各种基本功能。我用到的 WALA 的主要特性如下:

- 提供 Java 上的类型系统和类型层次构建，可以从指定的类路径（Classpath）中加载编译好的 Java 字节码，解析其中的类型和接口的定义，构建数据结构表示类型与类型、接口与接口之间的继承关系，以及类型与接口之间的实现关系。
- 采用基于静态单赋值（Static Single Assignment）、寄存器传递（Register Transfer）的中间表达形式，可以用来构建 Java 方法的控制流图（Control Flow Graph）。
- 提供通用的基于迭代的数据流分析求解框架,可以用来实现各种过程内分析(Intra-procedural Analysis)，比如分析过程内每个寄存器的声明类型。
- 提供各种表达 Java 程序信息的数据结构，例如类型、类、方法、方法的中间表达形式、基于静态单赋值的指令等。

我还发现 WALA 是开源框架里面最适合实现 Java 上基于库摘要的程序分析的。在工具实现的过程中，我还尝试了 SOOT [8] 和 Chord [22]。WALA 相比这两个工具有如下优点:

- WALA 生成的中间表达形式和控制流图是稳定的。SOOT 采用 Jimple [33] 作为其中间表达形式，对同一 Java 程序生成的 Jimple 代码不一定相同。因为我希望做基于库摘要的分析，所以需要独立分析库代码，再在端分析时分析库代码和端代码，这要求库分析和端分析中解析 Java 字节码时，生成的关于库的中间表达形式是相同的。
- WALA 的功能解耦设计良好。Chord 的主要设计目的是做全程序分析，无法方便

地进行独立的库代码分析。

bddbddb 是一个基于二元决策图（Binary Decision Diagram）的、自底向上的 Datalog 求解工具。bddbddb 的输入分为**定义域**（Domain）和**多元关系**（Relation）两种。bddbddb 中的每个定义域都用自然数从零开始连续地进行编号，在此基础上可以用自然数元组的集合来表示多元关系。在 ConDlog 中，在对定义域中元素定顺序时，我约定库代码中信息从零开始进行连续编号，所有的应用端代码中信息编号在库信息编号之后。因为 ConDlog 的库分析和端分析时两个阶段，所以为了保证库分析中库代码信息的编号和端分析中库代码信息的编号一致，我需要在库分析时保存定义域中的映射关系到磁盘上。之后，在端分析时，ConDlog 会从磁盘读入每个定义域的映射关系，重建每个定义域。在完成定义域的元素填充和映射关系后，bddbddb 使用二元决策图来表示、存储多元关系。bddbddb 接受 Datalog 程序作为分析的规约，ConDlog 将分析所需的定义域和多元关系保存到磁盘后，调用 bddbddb 的 Datalog 求解器进行求解，输出关系的元组也会通用采用二元决策图格式存储在磁盘上。二元决策图的具体数据结构和算法实现不在本文的讨论范围之中。

我使用 Java 语言编写 ConDlog 工具。在 ConDlog 的实现中，我对 WALA 和 bddbddb 提供的诸多功能进行了封装。我提供了一个 `Program` 类，封装了 WALA 的 Java 字节码解析、中间表达形式构建的功能，其提供的公有方法包括但不限于：

表 5.1 `Program` 类的部分方法

`static Program g()`	获取全局共享的被分析 Java 程序的信息。
`Set<IClass> getClasses()`	获取程序中所有声明的类。
`Set<IMethod> getMethods()`	获取程序中所有声明的方法。
`Set<TypeReference> getTypes()`	获取程序中所有的类型。
`IR getIR(m)`	获取方法 `m` 程序体的中间表达形式。
`boolean isSubtypeOf(s, t)`	判断类型 `s` 是否是 `t` 的子类型。
`Set<IClass> getImplementorsOf(i)`	获取实现了接口 `i` 的所有类。

对于 bddbddb，我将定义域和多元关系分别封装成了 `Dom` 类和 `Rel` 类。对于 `Dom` 类，我提供的公有方法包括但不限于：

表 5.2 `Dom` 类的部分方法

`Dom(name)`	构建一个名字为 `name` 的定义域。
`saveToBDD(dirName)`	将定义域的信息和映射存储在磁盘的 `dirName` 目录中。
`loadLibFromBDD(dirName)`	端模式使用，将定义域中库相关的信息从磁盘解析读入。

对于 `Rel` 类，我提供的公有方法包括但不限于：

表 5.3　Rel 类的部分方法

Rel(name, sign)	构建一个名字为 name、签名为 sign 的多元关系。
saveToBDD(dirName)	将关系按二元决策图格式存储在磁盘的 dirName 目录中。
add(val0)	添加一元元组。
add(val0, val1)	添加二元元组。
add(val0, val1, val2)	添加三元元组。

为了使得 ConDlog 的用户能方便地进行自定义的分析，我提供了 Project 类对程序分析流程进行控制，并提供了 ITask 接口作为可执行任务的规约。ITask 接口中包含方法 String getName() 和 void run()，且我约定一个任务执行结束后会将其结果保存在自己体内。Project 全局只有一个实例，用户可通过该实例提供的 void runTask(task) 方法执行一个分析任务，任务结束后该实例会在内部将 task 的结果和其名称映射起来，从而在之后的分析中，用户可以使用 ITask getTask(name) 来获取已经完成的分析的结果。

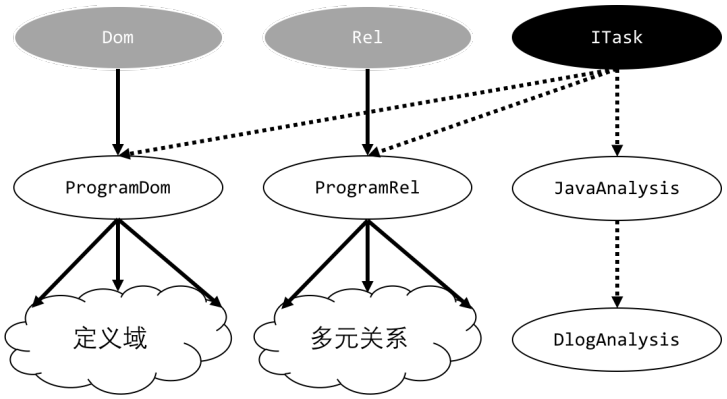


图 5.1　ConDlog 中分析任务实现的关系图

在目前的ConDlog工具中,我实现的任务种类主要是四种:定义域分析(ProgramDom),多元关系分析（ProgramRel），用 Java 实现的程序分析（JavaAnalysis），用 Datalog 实现的程序分析（DlogAnalysis）。他们的关系如图5.1所示。图中的实线箭头表示继承关系，虚线箭头表示实现关系。ProgramDom 类表示程序信息相关的定义域。目前该类型包括但不限于如下子类:

- $\mathbb{T}$: 实现为 DomT 类，表示程序中所有的类型，元素实际的类型是 WALA 提供的 TypeReference，其中编号 0 的类型是 TypeReference.Null。
- $\mathbb{M}$: 实现为 DomM 类，表示程序中所有的方法，元素实际的类型是 WALA 提供的 IMethod，除了具体实现的方法，也包括本地方法（Native Method）和抽象方法（Abstract Method）。
- $\mathbb{I}$: 实现为 DomI 类，表示程序中所有的方法调用指令，元素实际的类型是二元组

`Pair<IMethod,SSAInstruction>`，其中 `SSAInstruction` 类由 WALA 提供。

- $\mathbb{V}$：实现为 `DomV` 类，表示程序中所有的局部变量（寄存器），元素实际的类型是二元组 `Pair<IMethod,Integer>`，其中 WALA 是使用整型对方法内部的局部变量进行编号的。
- $\mathbb{H}$：实现为 `DomH` 类，表示程序中所有的新建实例指令，元素实际的类型是二元组 `Pair<IMethod,SSAInstruction>`，其中编号 0 表示 `null` 对象。
- $\mathbb{F}$：实现为 `DomF` 类，表示程序中所有的域，元素实际的类型是 WALA 提供的 `IField`，包括静态域（Static Field）和实例域（Instance Field），其中编号 0 是一个特殊的域，在实现中用于在不关心数组下标的分析中将数组的读取和写入看做特殊的域读取和写入。

`ProgramRel` 类表示程序信息相关的多元关系。目前该类型包括但不限于如下子类：

- `RelCHA` 类：三元关系 (m_1, t, m_2) 表示虚拟调用类型 t 中签名为 m_1 的方法会实际调用 m_2。
- `RelSub` 类：二元关系 (t_1, t_2) 表示类型 t_1 是类型 t_2 的子类型。
- `RelClinitTM` 类：二元关系 (t, m) 表示类 t 的类初始化函数是 m。
- `RelStaticTM` 类：二元关系 (t, m) 表示类 t 中有静态方法 m。
- `RelVirtIM` 类：二元关系 (i, m) 表示虚拟调用指令 i 调用的方法签名为 m。
- `RelSpecIM` 类：二元关系 (i, m) 表示特殊调用指令（例如私有调用）i 调用的方法为 m。
- `RelVT` 类：二元关系 (v, t) 表示局部变量 v 具有类型 t。在静态单赋值下，这个关系是一个函数。
- `RelHT` 类：二元关系 (h, t) 表示新建实例指令 h 创建的实例类型为 t。
- `RelStaticTF` 类：二元关系 (t, f) 表示类 t 中有静态域 f。
- `RelMobjValAsgnInst` 类：三元关系 (m, v, h) 表示方法 m 中有一条新建实例指令 h，这条指令将新创建的实例赋给了变量 v。

ConDlog 工具项目目前托管在基于 Git 的版本控制网站 Bitbucket 上，仍在进行积极的开发和拓展。当前该项目总代码行数为 4588 行。

5.2 实验设计

实验测试集 我选择了 DaCapo 测试集 [7] 中的 11 个 Java 程序作为我的实验对象。这 11 个 Java 程序功能各异，被广泛地使用在了现实的代码中，而且它们共用的代码主要是 Java 标准库。我选择了 Java 标准库中的主要部分来作为实验中用来计算条件摘要的库代码；具体而言，是 `rt.jar` 和 `jce.jar` 中的字节码。这两个 JAR 包涵盖了绝大部

表 5.4　DaCapo 测试集简介其统计信息

	简介	端类数	端方法数
antlr	生成词法分析器和语法分析器	153	1,656
bloat	Java 字节码分析优化工具	318	3,700
chart	绘制图像并渲染成 PDF	543	7,333
eclipse	Eclipse 集成开发环境	379	3,468
fop	解析 XSL-FO 文件并格式化为 PDF 文件	1,633	11,741
hsqldb	关系型数据库引擎	23	256
jython	Python 语言的 Java 实现	584	6,049
luindex	文件索引工具	181	1,468
lusearch	文本搜索工具	217	1,751
pmd	Java 源代码分析器	1,034	9,600
xalan	XML 至 HTML 转换工具	24	259

分常用的 Java 包，例如 `java.util`，`java.io`，`java.lang` 等。选择 Java 标准库中的主要部分建立摘要是程序分析研究领域中的常见做法，这方面的工作 [5, 20, 25, 31] 通过这种方式来减小计算摘要的时间。DaCapo 测试集的 11 个 Java 程序的简介和统计信息如表格5.4所示。Java 标准库的统计信息如表格5.5所示。

表 5.5　Java 标准库的统计信息

库类数	16,517
库方法数	140,758

实验细节　我的实验分为两部分，分别研究 ConDlog 技术在类层次分析和快速类型分析上通过条件摘要达到的加速效果。在每部分实验中，除了测量 ConDlog 技术的运行时间外，我还需要测量全程序分析的时间作为基线来计算加速比。我在实验中用到的 Datalog 规约与第四章中的有所出入，不同之处在于：

- Java 中每个类可以有一个类初始化函数，在类层次分析和快速类型分析中，除了主方法，这些方法也需要作为整个程序的入口方法。
- Java 中除了虚拟函数调用，还有特殊调用（Special Call）和静态调用（Static Call），在类层次分析和快速类型分析中，计算调用图关系时，还需要把这些调用指令考虑进去。
- 由于 Java 中的特殊调用和静态调用的目标方法是可以唯一确定的，所以在快速类型分析的库分析模式中，如果 i 是特殊调用或静态调用，$\mathsf{targetSum}(i, m, k, t)$ 中的类型 t 可被新建这个条件应该被修正成真。在我的实现中，我将这时的 t 赋为 `null` 的类型，并约定在任意程序中认为这个类型可以被新建。

实验平台 我的实验在一台戴尔 Precision T3610 工作站上进行。该工作站配置了 8 核因特尔 Xeon E5-1620 v2 的 CPU（每个核的频率都是 3.70GHz）。该工作站拥有 128GB 的内存。操作系统为 Ubuntu 14.04.5，Java 版本为 1.8.0_101。

5.3 结果分析

我分别对类层次分析和快速类型分析的实验结果进行分析。

5.3.1 类层次分析结果分析

ConDlog 技术在类层次分析库模式分析上的实验数据如表格5.6所示。表格5.6的第三行和第四行分别列出了库分析产生的条件摘要关系 rMethodSum 和 callGraphSum 中的元组数目。可以看到，虽然 ConDlog 计算的是完整的条件摘要，但是时空开销均在可以接受的范围之内。

表 5.6 ConDlog 在类层次分析库模式分析的实验数据

库分析时间（秒）	1,256
库分析内存（GB）	10.98
rMethodSum 关系大小	2,440,343
callGraphSum 关系大小	3,277,495
摘要读取时间（秒）	14

ConDlog 技术在 DaCapo 的 11 个 Java 程序进行类层次分析端模式分析的实验数据如表格5.7所示。表格5.7的第二列列出了类层次分析计算得出的 callGraph 关系中的元组数目。第三列列出了类层次分析的全程序分析计算所需的时间，同时这也是我计算加速比的基线。第四列列出了使用了条件摘要的类层次分析的端模式分析的时间，这其中包括了摘要读取和筛选的时间（括号中的是筛选时间，摘要读取时间在表格5.6的第五行）。第五列列出了使用 ConDlog 端分析相较于全程序分析所取得的加速比。第五列的最后一行还列出了 ConDlog 技术在 DaCapo 的 11 个 Java 程序上的平均加速比。在这些测试程序上，ConDlog 技术获得了 2.58 倍至 6.11 倍的加速比，平均值为 3.91 倍。可以看到，ConDlog 技术在类层次分析分析上的加速效果是显著的。

为了进一步分析 ConDlog 技术在不同 Java 程序上加速效果的差异，我统计了摘要筛选阶段选出的满足条件的库方法数目，如表格5.8所示。我对加速比和满足条件的库方法比例的关系进行了线性拟合，结果如图5.2所示。可以看到，加速比与满足条件的库方法比例正相关，这也再次说明了我的基于摘要的 ConDlog 技术是有效的。

表 5.7　ConDlog 在类层次分析上的实验数据（ConDlog 时间中括号内为摘要筛选时间）

测试程序	调用图大小	全程序分析时间（秒）	ConDlog 时间（秒）	加速比
antlr	1,821,590	924	172(0.8)	5.37X
bloat	2,079,888	989	323(0.8)	3.06X
chart	1,966,943	1,016	270(0.8)	3.76X
eclipse	1,902,567	963	330(0.8)	2.91X
fop	1,975,336	1,089	423(0.8)	2.58X
hsqldb	1,811,973	904	151(0.8)	5.99X
jython	1,902,328	979	291(1.2)	3.36X
luindex	1,820,424	915	262(0.8)	3.48X
lusearch	1,828,215	977	272(0.8)	3.59X
pmd	1,995,652	1,067	379(0.8)	2.81X
xalan	1,809,099	916	150(0.8)	6.11X
				3.91X

表 5.8　库方法筛选相关实验数据

测试程序	满足条件的库方法数	所占比例
antlr	136,262	96.81%
bloat	130,674	92.84%
chart	133,504	94.85%
eclipse	128,584	91.35%
fop	128,791	91.50%
hsqldb	136,589	97.04%
jython	132,939	94.45%
luindex	132,276	93.97%
lusearch	131,908	93.71%
pmd	131,194	93.21%
xalan	136,513	96.98%

5.3.2　快速类型分析结果分析

ConDlog 技术在快速类型分析库模式分析上的实验数据如表格5.9所示。表格5.9的第三行、第四行和第五行分别列出了库分析产生的条件摘要关系 rMethodSum、rTypeSum 和 callGraphSum 中的元组数目。同类层次分析的库模式分析一样，虽然 ConDlog 计算的是完整的条件摘要，但是时空开销均在可以接受的范围之内。

ConDlog 技术在 DaCapo 的 11 个 Java 程序进行快速类型分析端模式分析的实验数据如表格5.10所示。表格5.10的格式与表格5.7相同。在这些测试程序上，ConDlog 技术获得了 0.92 倍至 2.13 倍的加速比，平均值为 1.27 倍。可以看到，ConDlog 技术在快速类型分析分析上的加速效果并不如在类层次分析分析上的显著。我认为有两个原因导致了这一现象：

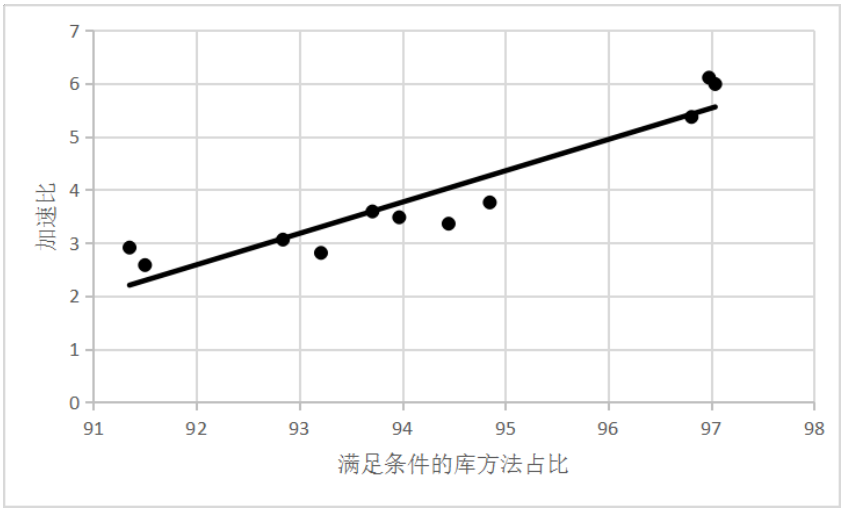


图 5.2　加速比与库方法所占比例关系拟合图

表 5.9　ConDLog 在快速类型分析库模式分析的实验数据

库分析时间（秒）	1,314
库分析内存（GB）	12.93
rMethodSum 关系大小	27,755,414
rTypeSum 关系大小	63,095
callGraphSum 关系大小	22,058,878
摘要读取时间（秒）	36

- 快速类型分析分析比类层次分析分析复杂，其摘要 rMethodSum 和 callGraphSum 所选取的条件也比类层次分析复杂，在摘要实例化时带来更大的时间开销。
- 快速类型分析分析的库摘要较大，从磁盘读取的时间较长（类层次分析读取时间为 14 秒，而快速类型分析读取时间为 36 秒）。

类似的，我也分析了 ConDLog 技术在不同 Java 程序上的效果的差异。摘要筛选阶段选出的满足条件的库方法数目如表格5.11所示。我对加速比和满足条件的库方法比例的关系进行了线性拟合，结果如图5.3所示。同样的，加速比与满足条件的库方法比例呈正相关关系。

表 5.10　ConDlog 在快速类型分析上的实验数据（ConDlog 时间中括号内为摘要筛选时间）

测试程序	调用图大小	全程序分析时间（秒）	ConDlog 时间（秒）	加速比
antlr	228,717	412	298(0.8)	1.38X
bloat	332,773	451	387(1.1)	1.17X
chart	272,405	483	392(0.8)	1.23X
eclipse	239,579	441	438(1.1)	1.01X
fop	285,826	584	622(1.2)	0.94X
hsqldb	221,288	389	221(0.9)	1.76X
jython	247,799	461	388(0.9)	1.19X
luindex	224,870	405	306(0.8)	1.32X
lusearch	230,886	421	428(0.8)	0.98X
pmd	259,465	491	534(0.8)	0.92X
xalan	220,769	510	240(0.8)	2.13X
				1.27X

表 5.11　库方法筛选相关实验数据

测试程序	满足条件的库方法数	所占比例
antlr	136,045	96.65%
bloat	130,465	92.69%
chart	133,288	94.69%
eclipse	128,374	91.20%
fop	128,578	91.35%
hsqldb	136,300	96.83%
jython	132,726	94.29%
luindex	132,079	93.83%
lusearch	131,700	93.56%
pmd	130,987	93.06%
xalan	136,288	96.82%

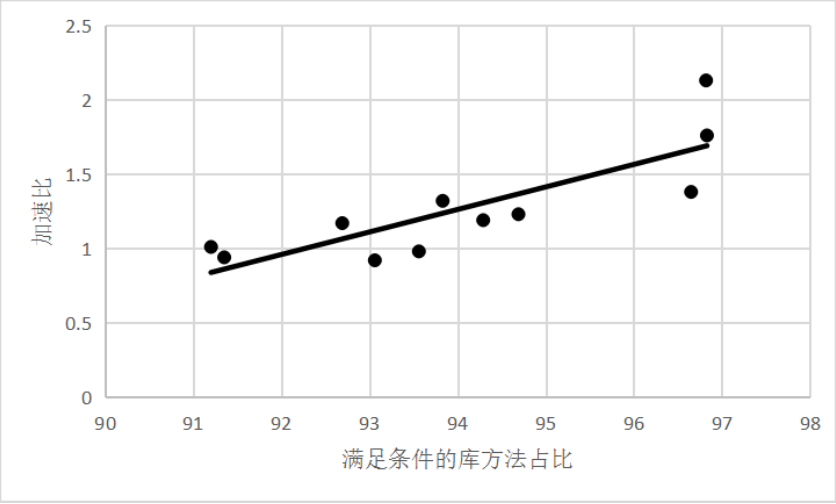


图 5.3　加速比与库方法所占比例关系拟合图

第六章　相关工作

在这一章，我将简要介绍程序分析领域中基于摘要的技术的一些相关工作。我按照基于完整摘要、基于部分摘要、基于预处理库、基于非条件库摘要、基于条件库摘要的顺序进行介绍。

基于完整摘要　经典的基于摘要的过程间分析 [23, 27, 36] 会为每个过程计算完整的摘要，并在自顶向下分析中，处理过程调用时使用这些摘要。对于很多分析来说，这样的摘要难以高效计算或紧凑表示 [35, 36]。Greta Yorsh 等人 [39] 通过符号化的方法表示摘要，从而在不损失精度的同时提升了摘要表示的紧凑性。Thomas Ball 等人 [6] 则是使用了二元决策图来表示摘要。还有一些工作通过对摘要中关于过程被调用上下文的无关信息进行剪枝，来优化指向分析（Points-to Analysis）[11] 和形状分析（Shape Analysis）[24]。这些方法产生的摘要在重用时有非常高的实例化成本。与此相对，ConDlog 提倡使用简单的条件进行摘要，并把实例化的过程也实现为 Datalog 规约，这样使得重用摘要时实例化带来的开销不大。

基于部分摘要　Patrice Godefroid 等人 [16] 提出针对特定的程序执行路径构建部分摘要，这个特定的程序执行路径被选择为会潜在地破坏一个分析相关的性质的路径。Xin Zhang 等人 [40] 提出了一个结合自顶向下和自底向上分析的过程间分析框架，在自顶向下的分析中可以利用自底向上阶段计算出的部分摘要。部分转移方程 [21, 37] 只对部分过程的输入-输出关系进行摘要，其主要动机在于弥补一些基于完整摘要方法的缺点，例如无法自然处理程序中的高阶函数。这些方法都没有生成可以跨程序重用的摘要，而 ConDlog 生成的摘要可以在不同的程序上使用。

基于预处理库　Yannis Smaragdakis 等人 [28] 提出了一种优化技术，可以将源代码变换成更适合用来进行指向分析的形式，且不改变分析的结果。这种源代码层面的转换时通过一个简单的预处理分析得到，从而也可以应用到共用库代码的场景中：先对共用库代码进行一次预处理，那么这之后所有的应用端程序分析都可以使用已经处理好的库程序。NicholasăAllen 等人 [4] 提出的需求导向的指向分析也涉及到了源代码变换。他们使用了静态程序切片（Static Program Slicing）技术来减小被分析程序的大小，然后再在减小后的程序上进行全程序分析。

基于库非条件摘要 Atanas Rountev 等人 [26, 38] 提出了一种为库代码独立生成摘要的方法。他们解决了一类过程间可分配环境（Inter-procedural Distributive Environment）数据流分析问题。这个工作中生成的摘要是不带条件的，从而他们的摘要只能处理那种调用目标唯一确定的调用指令（例如 Java 中的静态调用指令）。Steven Arzt 等人 [5] 将这一方法扩展应用到了安卓程序的污点分析（Taint Analysis）中。Karim Ali 等人 [2, 3] 提出了一种对库代码做上近似的方法。该方法为应用端代码生成了方法调用图，而整个库则被抽象为一个单独的库方法 $library$，所用应用端中调用库方法的指令都被指向 $library$。这些方法因为生成的摘要都是非条件的，所以摘要的粒度和精度往往都比较粗糙。ConDlog 使用的条件摘要技术则没有这样的问题。

基于库条件摘要 Hao Tang 等人 [32] 提出了树邻近语言（Tree Adjoining Language）可达性问题，并在此基础上发展了一个库摘要技术，使得在回调存在下的库独立分析成为了可能，但是树邻近语言的使用也限制了摘要中所使用条件的形式。作为后续研究，Hao Tang 等人 [31] 提出了带条件 Dyck 上下文无关语言可达性分析（Conditional Dyck-CFL Reachability Analysis），放宽了摘要中可以使用的条件的形式，且允许多种条件进行组合。这两个工作都对库代码和端代码之间的交互进行了限制，难以处理 Java 的一些复杂特征，例如动态内存分配。Sulekha Kulkarni 等人 [19] 提出了跨程序训练方法，针对任意通过 Datalog 规约的分析，可以从一个程序训练集中学习出关于这些程序共用代码部分的条件摘要。该方法的通用性较强，但是必须从已有代码的分析结果中去学习条件摘要，因而不能保证条件摘要是完整的。ConDlog 则结合了这些工作的思想，在可以生成完整的条件摘要的同时，可以有较大的灵活性和通用性。

总结

在这个软件越来越多、越来越大的时代，将程序分析应用到巨大的程序上毋庸置疑是一个极大的挑战。在这篇论文中，我提出了 ConDlog 技术，一种基于 Datalog 的新的条件摘要方法，在程序共用很大一部分代码的情景下来处理这一挑战。ConDlog 的基本思路是为共用库代码创建条件摘要，并将其重用在不同的应用端程序的分析上。ConDlog 可被设置为两个模式: 库分析模式，在指定的库代码上进行独立的分析，并根据 Datalog 规约创建条件摘要后保存到磁盘；和端分析模式，加载库分析模式产生的条件摘要，并根据应用端代码的信息筛选出满足条件的摘要进行实例化，再按照 Datalog 规约计算分析结果。我在 Java 程序上的两个分析上实现了 ConDlog 技术，并在 DaCapo 测试集的 11 个程序上进行了实验。ConDlog 技术在类层次分析上取得了 3.9 倍的平均加速，在快速类型分析上取得了 1.3 倍的平均加速。未来，我期望将 ConDlog 技术应用到更多的程序分析问题上，并为其建立一般化的抽象框架。

参考文献

[1] Serge Abiteboul, Richard Hull and Victor Vianu. *Foundations of Databases*. Addison-Wesley Longman Publishing Co., **1995**.

[2] Karim Ali and Ondej Lhoták. "*Application-Only Call Graph Construction*". In: *European Conf. on Obj.-Oriented Prog.* **2012**: 688–712.

[3] Karim Ali and Ondej Lhoták. "*Averroes: Whole-Program Analysis without the Whole Program*". In: *European Conf. on Obj.-Oriented Prog.* **2013**: 378–400.

[4] Nicholas Allen, Bernhard Scholz and Padmanabhan Krishnan. "*Staged Points-to Analysis for Large Code Bases*". In: *Comp. Construct.* **2015**: 131–150.

[5] Steven Arzt and Eric Bodden. "*StubDroid: Automatic Inference of Precise Data-flow Summaries for the Android Framework*". In: *Int. Conf. on Softw. Eng.* **2016**: 725–735.

[6] Thomas Ball and Sriram K. Rajamani. "*Bebop: A Path-sensitive Interprocedural Dataflow Engine*". In: *Prog. Analysis for Softw. Tools and Eng.* **2001**: 97–103.

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann *et al.* "*The DaCapo Benchmarks: Java Benchmarking Development and Analysis*". In: *Object-oriented Prog. Syst. Lang. and Appl.* **2006**: 169–190.

[8] Eric Bodden. "*Inter-procedural Data-flow Analysis with IFDS/IDE and Soot*". In: *Int. Workshop on State of the Art Java Prog. Analysis*, **2012**: 3–8.

[9] Martin Bravenboer and Yannis Smaragdakis. "*Strictly Declarative Specification of Sophisticated Points-to Analyses*". In: *Object-oriented Prog. Syst. Lang. and Appl.* **2009**: 243–262.

[10] Cristiano Calcagno, Dino Distefano, Peter O'Hearn *et al.* "*Compositional Shape Analysis by means of Bi-Abduction*". In: *Princ. of Prog. Lang.* **2009**: 289–300.

[11] Ramkrishna Chatterjee, Barbara G. Ryder and William A. Landi. "*Relevant Context Inference*". In: *Princ. of Prog. Lang.* **1999**: 133–146.

[12] Patrick Cousot and Radhia Cousot. "*Modular Static Program Analysis*". In: *Comp. Construct.* **2002**: 159–178.

[13] Patrick Cousot and Radhia Cousot. "*Static Determination of Dynamic Properties of Recursive Procedures*". In: *IFIP Conf. on Formal Desc. of Prog. Concepts*, **1977**.

[14] Isil Dillig, Thomas Dillig and Alex Aiken. "*Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs*". In: *Prog. Lang. Design and Impl.* **2011**: 567–577.

[15] Julian Dolby, Stephen Fink and Manu Sridharan. *T. J. Watson Libraries for Analysis*, **2006**.

[16] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani *et al.* "*Compositional May-Must Program Analysis: Unleashing the Power of Alternation*". In: *Princ. of Prog. Lang.* **2010**: 43–56.

[17] Bhargav S. Gulavani, Supratik Chakraborty, G. Ramalingam *et al.* "*Bottom-up Shape Analysis using LISF*". In: *ACM Trans. on Prog. Lang. and Syst.* **2011**.

[18] George Kastrinis and Yannis Smaragdakis. "*Hybrid Context-Sensitivity for Points-To Analysis*". In: *Prog. Lang. Design and Impl.* **2013**: 423–434.

[19] Sulekha Kulkarni, Ravi Mangal, Xin Zhang *et al.* "*Accelerating Program Analyses by Cross-Program Training*". In: *Object-oriented Prog. Syst. Lang. and Appl.* **2016**: 359–377.

[20] Ravichandhran Madhavan, G. Ramalingam and Kapil Vaswani. "*Modular Heap Analysis For Higher Order Programs*". In: *Static Analysis Symp.* **2012**: 370–387.

[21] Brian R. Murphy and Monica S. Lam. "*Program Analysis with Partial Transfer Functions*". In: *Part. Eval. and Semantics-Based Prog. Manip.* **2000**: 94–103.

[22] Mayur Naik. *Chord: A Versatile Program Analysis Platform for Java Bytecode*, **2006**.

[23] Thomas Reps, Susan Horwitz and Mooly Sagiv. "*Precise Interprocedural Dataflow Analysis via Graph Reachability*". In: *Princ. of Prog. Lang.* **1995**: 49–61.

[24] Noam Rinetzky, Jörg Bauer, Thomas Reps *et al.* "*A Semantics for Procedure Local Heaps and its Abstractions*". In: *Princ. of Prog. Lang.* **2005**: 296–309.

[25] Atanas Rountev and Barbara G. Ryder. "*Points-to and Side-effect Analyses for Programs Built with Precompiled Libraries*". In: *Comp. Construct.* **2001**: 20–36.

[26] Atanas Rountev, Mariana Sharp and Guoqing Xu. "*IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries*". In: *Comp. Construct.* **2008**: 53–68.

[27] Micha Sharir and Amir Pnueli. "*Two Approaches to Interprocedural Data Flow Aalysis*". In: *Program Flow Analysis: Theory and Applications*. Prentice-Hall, **1981**.

[28] Yannis Smaragdakis, George Balatsouras and George Kastrinis. "*Set-Based Pre-Processing for Points-To Analysis*". In: *Object-oriented Prog. Syst. Lang. and Appl.* **2013**: 253–270.

[29] Yannis Smaragdakis and Martin Bravenboer. "*Using Datalog for Fast and Easy Program Analysis*". In: *Datalog 2.0 Workshop*, **2010**.

[30] Yannis Smaragdakis, Martin Bravenboer and Ondrej Lhoták. "*Pick Your Contexts Well: Understanding Object-Sensitivity*". In: *Princ. of Prog. Lang.* **2011**: 17–30.

[31] Hao Tang, Di Wang, Yingfei Xiong *et al.* "*Conditional Dyck-CFL Reachability Analysis for Complete and Efficient Library Summarization*". In: *European Symp. on Programming*, **2017**.

[32] Hao Tang, Xiaoyin Wang, Lingming Zhang *et al.* "*Summary-Based Context-Sensitive Data-Dependence Analysis in Presence of Callbacks*". In: *Princ. of Prog. Lang.* **2015**: 83–95.

[33] Raja Vallee-Rai and Laurie J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*, **1998**.

[34] John Whaley, Dzintars Avots, Michael Carbin *et al.* "*Using Datalog with Binary Decision Diagrams for Program Analysis*". In: *Asian Symp. on Prog. Lang. and Systems*, **2005**: 97–118.

[35] John Whaley and Monica S. Lam. "*Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams*". In: *Prog. Lang. Design and Impl.* **2004**: 131–144.

[36] John Whaley and Martin Rinard. "*Compositional Pointer and Escape Analysis for Java Programs*". In: *Object-oriented Prog. Syst. Lang. and Appl.* **1999**: 187–206.

[37] Robert P. Wilson and Monica S. Lam. "*Efficient Context-Sensitive Pointer Analysis For C Programs*". In: *Prog. Lang. Design and Impl.* **1995**: 1–12.

[38] Dacong Yan, Guoqing Xu and Atanas Rountev. "*Rethinking Soot for Summary-Based Whole-Program Analysis*". In: *Int. Workshop on State of the Art Java Prog. Analysis*, **2012**: 9–14.

[39] Greta Yorsh, Eran Yahav and Satish Chandra. "*Generating Precise and Concise Procedure Summaries*". In: *Princ. of Prog. Lang.* **2008**: 221–234.

[40] Xin Zhang, Ravi Mangal, Mayur Naik *et al.* "*Hybrid Top-down and Bottom-up Interprocedural Analysis*". In: *Prog. Lang. Design and Impl.* **2014**: 249–258.

致谢

感谢我的父母，在各方面一直支持着我。感谢他们任何时候都尊重我的决定，无论是高中时选择信息学竞赛，保送时清华与北大的抉择，还是大三时准备出国留学。大学四年里，当我感觉坚持不下去，自己一无是处的时候，感谢他们一直相信我，以及一直以来对我的鼓励。

感谢北京大学这四年对我的培养。从一名懵懂的刚刚从象牙塔中走出的大一新生，到一个依然懵懂但是已经有坚定理想的大四学生，北京大学提供的良好的学习氛围和优秀的教师团队，都让我获益匪浅。

感谢张路老师和熊英飞老师在科研方向上的指导，让我这样一个本科生能够参与到科研的核心工作当中，切实体会科研的理论和方法。在研究的过程中，为我这样一个科研新手把握研究的方向，减少研究的误区，并未研究提出了许多宝贵的意见。这一篇论文的完成离不开两位老师的谆谆教诲。

感谢 13 级博士生唐浩师兄带领我进行了对基于摘要的程序分析的研究。在研究的过程中，唐浩师兄提出了许多新颖的看法，在理论分析和工程实践方面都给予了我很多帮助。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在 □ 一年 / □ 两年 / □ 三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日