基于SAT求解的面向对象程序类型分析

曹 璟 徐宝文

(东南大学计算机科学与工程学院 南京 210096) (江苏软件质量研究所 南京 210096)

摘 要 类型分析是面向对象程序分析中的重要环节,精确的类型分析能够提高其它程序分析的精度。由于传统精确分析方法固有的高复杂性,现有的类型分析大都使用粗糙的分析方法。提出了一种基于SAT求解的面向对象程序类型分析方法。该方法用命题逻辑表示类型在变量间的传递关系,将程序抽象成命题公式,并使用高效的SAT求解器求解,从而获得变量运行时的类型集合。该方法是流敏感的,并且具有良好的伸缩性,既可以进行快速但精度低的上下文不敏感分析,也可以进行较慢但精度高的上下文敏感分析。

关键词 命题公式可满足性验证,类型分析,程序分析,面向对象程序

Object-oriented Program Type Analysis Based on SAT Solver

CAO Jing XU Bao-wen

(Department of Computer Science & Engineering, Southeast University, Nanjing 210096, China)

(Jiangsu Institute of Software Quality, Nanjing 210096, China)

Abstract Type analysis plays an important role in object-oriented program analysis. Accurate type analysis will improve the precision of other program analyses. However, due to the inherent high complexity of traditional type analysis, people generally make rapid but coarse analyses. This paper presented a new type analysis method of object-oriented program based on SAT solver, in which we show transfer relationship among variables by the proposition logic method, abstract programs into proposition formulas and adopt highly effective SAT solver so that we can obtain variable runtime types through decoding the results. This method is flow-sensitive with good flexibility, that is to say, it can not only make rapid but rough context-insensitive analysis, but also comparatively slow but highly accurate context-sensitive one.

Keywords Type analysis, SAT, Program analysis, Object-oriented programming

1 引言

面向对象程序的多态性使得运行时变量的类型与其声明类型可能并不一致,前者可以是后者的子类型,并且变量可能具有多种不同的运行时类型。面向对象程序类型分析的目的就是静态分析出运行时变量的所有可能类型。当然,静态分析是对实际情况的一种估计。安全的静态分析,其分析结果应该是实际情况的超集,并且越接近实际情况,分析就越精确。静态类型分析应该在保证安全性的前提下,不断提高精确性。

类型分析的主要作用是用于构造程序的调用图。调用图是进行程序过程间分析时不可或缺的重要信息,而且调用图的精度直接影响了程序分析的精度。因此,类型分析是面向对象程序分析的重要环节。目前典型的类型分析方法主要有CHA[1]分析法、RTA[2]分析法、XTA[3]分析法、VTA[4]分析法、0-CFA[5]分析法和k-CFA($k>0$)[5]分析法。前4种方法是流不敏感、上下文不敏感的,分析速度快但精度较低。而流敏感、上下文不敏感的0-CFA分析法,以及流敏感、上下文敏感的k-CFA($k>0$)分析法,虽然精度较高,但复杂度很高,难以接受,并且存在空间爆炸问题[12]。因此,在大多数程序分析中,人们往往采用前4种快速的分析方法,但同时降低了后继分析的精度,任凭后继分析方法如何改进,也无法弥补在类型分析上损失的精确性。由此,我们认为研究精确的、复杂度可接受的类型分析方法是非常必要的。

SAT(命题公式可满足)求解是一个经典问题,虽然理论上已经证明其具有NP完全的时间复杂度,但是由于人们的不断探索,得益于优异的启发式搜索算法的发明,近年来人们在SAT求解算法的研究上取得了突破,已能满足实用化的需求。例如,目前求解器zChaff的求解规模达到了100万个布尔变量以及1000万个子句[6,7]。因此,SAT求解的应用范围越来越广泛,诸如计算机辅助设计、自动测试模式生成(ATPG)、人工智能等许多领域的目标问题被抽象为SAT问题。

在程序分析和验证领域,SAT的应用也逐渐受到重视。Clarke等人利用SAT求解器进行谓词抽象过程中状态空间的搜索,提高了搜索的速度和精度[8]。Xie等人提出了一种基于SAT的C程序缺陷侦测方法[9],并且据此实现了一个C

到稿日期:2008-02-21 本研究得到国家杰出青年科学基金项目(60425206),国家自然科学基金与微软亚洲研究院联合资助项目(60633010),国家自然科学基金项目(60503033、60403016),江苏省自然科学基金项目(BK2005060),江苏省高技术研究项目(BG2005032)资助。

曹 璟(1980-),男,博士研究生,主要研究领域为程序设计语言、程序分析,E-mail:jing_cao@seu.edu.cn;徐宝文(1961-),男,教授,博士生导师,CCF会员,主要研究领域为程序设计语言、软件工程、并行与网络软件等方向的教学与科研工作。

程序分析系统[10]。Dennis 等人利用 SAT 验证 Java 程序的方法是否满足其规格说明[11]，SAT 的应用使得他们的方法能够适合大规模的程序分析和验证。

本文提出了一种基于 SAT 求解的面向对象程序类型分析方法。该方法通过将程序抽象成命题公式，从而用命题逻辑来表示类型在变量间的传递，求出满足程序公式的所有解即可获得变量的运行时类型集合。该方法是流敏感的，具有良好的伸缩性，既可以进行快速但精度低的上下文不敏感分析，也可以进行较慢但精度高的上下文敏感分析。与传统方法相比，本文所提出的方法除了具有较高的精度外，还解决了传统精确类型分析方法存在的空间爆炸问题[12]，而且由于采用了高效的启发式搜索算法，提高了类型推导的速度。

本文在讨论程序抽象前的预处理过程与类型传递的逻辑抽象的基础上，首先研究了面向基本程序结构的基本分析方法，然后对过程间分析进行了深入探讨。在文章的结尾，我们给出了一个实例研究。

2 程序预处理

预处理是程序抽象前的准备工作，目的是为了便于命题公式的抽取。预处理过程主要对程序进行以下变换：首先是复杂语句的化简，其次是基于属性化(field-based)处理，最后是 SSA(Static Single Assignment)表示形式的转换。

结构复杂的语句会给程序的抽象带来不必要的麻烦，为此我们利用中间变量将复杂语句化简。由于类型分析只需考虑类型为引用类型的表达式及相关语句，因此化简的对象是包含复杂访问路径[1]的语句，我们将其转换成只包含两位访问路径的语句。

Java 程序中，引用类型的变量可能指向同一个对象，从而对象属性的取值会因为别名而相互影响。为此我们对程序做基于属性化处理，即将声明类型相同的对象的同名属性当作同一变量。例如，对于形如“$o.f$”的表达式，若对象 o 的声明类型为 C，则用表达式“C_f”替换“$o.f$”。对于形如“$C.f$”的类属性，我们直接把它替换为“C_f”。这样，我们就把对象属性和类属性也当作普通变量(下文称它们为全局变量)处理。

程序中，不同位置的同一变量的取值可能不同，从而不能直接把这些相同变量抽象成一个布尔变量(这样抽象将导致命题公式永假)。例如对于图 1(1)中的两条语句，公式$(a=A)\wedge(a=B)$永不满足。有两种建模方法能够解决该问题：一种是将程序抽象成状态转移系统。如图 1(2)所示，语句“$b=c$;”被抽象成“$S_{i+1}(b)=S_i(c)$”，其中 S_i 表示程序第 i 个状态(程序状态是由所有变量组成的向量)，“$S_i(c)$”表示 S_i 状态时变量 c 的值。另一种是将程序用 SSA 形式来表示，然后将每个变量抽象成单独的布尔变量。SSA 形式的程序具有两个重要的性质：首先，程序中的每个变量只有一个定义点，这样相应的布尔变量就不会导致矛盾；其次，Ø 函数能够区分出从不同控制流上流入的值，使得对循环结构的分析不需要迭代(在本文第 4.1 节论述)。通过对这两种建模方法的比较(详细的比较在最后给出)，我们选择后者。

我们采用 R. Cytron 提出的方法[13]进行程序 SSA 形式的变换。在变换过程中，我们针对程序抽象的需求还要做以下额外的处理：

- 我们给每个变量加上形如“类名_方法名_”的前缀，使得变量名在程序的全局空间中是唯一的。
- 方法体中形式参数和全局变量的下标从 2 开始计数。
- 为每个返回类型不是 void 的方法增加形如“类名_方法名_ret1”变量，并在方法的最后增加形如“类名_方法名_ret1=Ø(返回值 1,返回值 2,…,返回值 n);”的 Ø 函数。

图 1(3)给出了 SSA 形式变换的示例。

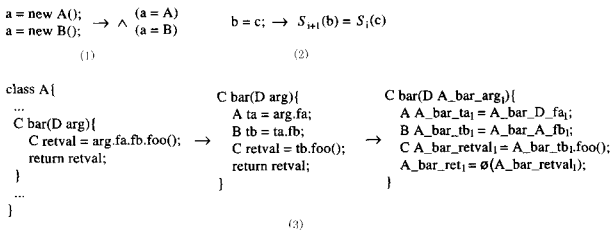


图 1 程序预处理示例

3 类型传递的逻辑抽象

能够引起变量运行时类型发生改变的语句有对象创建语句、赋值语句、强制类型转换语句、方法调用语句，以及 SSA 形式转换增加的 Ø 函数语句(这些语句如图 2 所示)，类型分析只需考虑这些语句。由于方法调用语句的抽象规则和具体的过程间分析方法有关，因此该规则在第 5 节给出。本节介绍其它语句的抽象规则。

$$v = new\ C; \quad (1)$$

$$v_1 = v_2; \quad (2)$$

$$v_1 = (C) < expression >; \quad (3)$$

$$v = o.f(a_1, \ldots, a_n); \quad (4)$$

$$v_n = \phi(v_i, v_j \ldots, v_k); \quad (5)$$

图 2 五类引起变量类型改变的语句

我们对程序中的类采用二进制方式统一编码，假设 CN 是程序中类的个数，则每个类的编码是位数为 $\log_2 CN$ 的二进制数。我们将经过预处理的程序中的每个变量抽象成一个长度为 $\log_2 CN$ 的布尔向量。在下文中，我们用 $\vec{b}_v$ 表示变量 v 抽象成的布尔向量。假设 $\vec{b}_1$ 和 $\vec{b}_2$ 是长度为 n 的布尔向量，b_{1i} 和 b_{2i} 分别表示向量 $\vec{b}_1$ 和 $\vec{b}_2$ 的第 i 个分量，我们记

$$(\vec{b}_1 \rightarrow \vec{b}_2) = (b_{1n} \rightarrow b_{2n}) \wedge \cdots \wedge (b_{12} \rightarrow b_{22}) \wedge (b_{11} \rightarrow b_{21})$$

下面的规则(1)—(5)分别给出了对象创建语句、强制类型转换语句、变量赋值语句以及 Ø 函数语句的抽象规则。其中，对象创建语句的抽象规则中的 $[\![C]\!]_i$ 表示类型 C 的二进制编码的第 i 位。这类语句的命题公式是由变量 v 抽象出的布尔向量 $\vec{b}_v$ 的所有位组成的合取式，并且每一位需根据类型编码相应位的值决定取正还是取反。强制类型转换语句的抽象规则和对象创建语句的抽象规则相同。变量赋值语句的抽象规则中的“↔”是命题逻辑中的相等联接符。这类语句的命题公式的含义是两边的布尔向量的取值相等。需要注意的是，经过预处理以后，程序中的对象属性变成了普通变量，因此赋值语句全都形如“$a=b$”。Ø 函数语句的命题公式可看作是由多个赋值语句公式组成的析取式，表示值可以从控制流的任意一个分支流入。

$$(1)F(\text{“}v=new\ C\text{”}) = \wedge_{i=n}^{1} f_i, \quad f_i = \begin{cases} b_{vi} & \text{if } [\![C]\!]_i = 1 \\ \neg b_{vi} & \text{if } [\![C]\!]_i = 0 \end{cases}$$

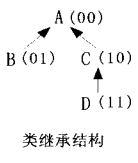
(2) $F(\text{“}v=(C)\langle expression\rangle\text{”})=\wedge_{i=n}^{1} f_i$，$f_i=\begin{cases} b_{vi} & \text{if } [\![C]\!]_i=1 \\ \neg b_{vi} & \text{if } [\![C]\!]_i=0 \end{cases}$

(3) $F(\text{“}v_2=v_1\text{”})=(\overrightarrow{b_{v_2}}\leftrightarrow\overrightarrow{b_{v_1}}),(\overrightarrow{b_{v_2}}\leftrightarrow\overrightarrow{b_{v_1}})=(\overrightarrow{b_{v_2}}\rightarrow\overrightarrow{b_{v_1}})\wedge(\overrightarrow{b_{v_1}}\rightarrow\overrightarrow{b_{v_2}})$

(4) $F(\text{“}v_n=\phi(v_i,v_j,\cdots,v_k)\text{”})=(\overrightarrow{b_{v_n}}\leftrightarrow\overrightarrow{b_{v_i}})\vee(\overrightarrow{b_{v_n}}\leftrightarrow\overrightarrow{b_{v_j}})\vee\cdots\vee(\overrightarrow{b_{v_n}}\leftrightarrow\overrightarrow{b_{v_k}})$

图 3 是 4 种语句抽象的简单示例。左图所示是程序片段的类继承结构，其中每个类节点右边括号内的数字是其两位的二进制编码。右图包含 8 条语句，每个变量都被抽象成一个两位的布尔向量，例如变量 a_1 对应的布尔向量为$\overrightarrow{b_{a_1 2}b_{a_1 1}}$，每条语句的右边是其相应的命题公式。

为了下文论述的方便，我们用公式“(变量名＝类名)”作为对象创建语句以及强制类型转换语句命题公式的简化表示，用公式“(变量名＝变量名)”作为赋值语句命题公式的简化表示。例如，对于图 3 中的语句(1)，我们用“$(a_1=A)$”替代“$(\neg b_{a_1 2}\wedge\neg b_{a_1 1})$”作为其命题公式；对于语句(6)，我们用“$(b_1=c_1)$”代替“$(\overrightarrow{b_{b_1}}\leftrightarrow\overrightarrow{b_{c_1}})$”作为其命题公式。



(1)	a_1 = new A();	$(\neg b_{a_1 2}\wedge\neg b_{a_1 1})$
(2)	a_2 = new B();	$(\neg b_{a_2 2}\wedge b_{a_2 1})$
(3)	c_1 = new C();	$(b_{c_1 2}\wedge\neg b_{c_1 1})$
(4)	d_1 = new D();	$(b_{d_1 2}\wedge b_{d_1 1})$
(5)	a_2 = (C) a_1;	$(b_{a_2 2}\wedge\neg b_{a_2 1})$
(6)	$b_1=c_1$;	$(\overrightarrow{b}_{b_1}\leftrightarrow\overrightarrow{b}_{c_1})$
(7)	$b_2=d_1$;	$(\overrightarrow{b}_{b_2}\leftrightarrow\overrightarrow{b}_{d_1})$
(8)	$b_3=\phi(b_1,b_2)$	$(\overrightarrow{b}_{b_3}\leftrightarrow\overrightarrow{b}_{b_1})\vee(\overrightarrow{b}_{b_3}\leftrightarrow\overrightarrow{b}_{b_2})$

图 3　抽象示例

4　基本分析方法

本节我们首先介绍 3 种基本程序结构(顺序结构、分支结构和循环结构)的分析，然后介绍这些结构的组合分析，并将分析方法扩展到一般的程序结构上。本节假定程序不包含方法调用语句，因此不涉及过程间分析。

4.1　基本结构分析

为了使讨论清晰明了，本小节我们进一步假定基本结构之间不相互嵌套。

顺序结构及分支结构的命题公式的抽取方法相同，都是将结构中相关语句抽象出的公式取交。将结构的命题公式转换成 CNF 形式后，由求解器求出所有解即完成对该结构的分析。

图 4 是分支结构分析的示例。图中给出了分支结构的源程序、SSA 形式(变量名省略了前缀)以及有关语句的公式。图的右上角给出了该结构的公式，求解所得分析结果如图右下角所示。

(1)	a = new A();	a_1 = new A();	$(a_1=A)$
(2)	b = new B();	b_1 = new B();	$(b_1=B)$
(3)	if (P)	if (P)	
(4)	c = a;	$c_1=a_1$;	$(c_1=a_1)$
(5)	else	else	
(6)	c = b;	$c_2=b_1$;	$(c_2=b_1)$
(7)		$c_3=\phi(c_1,c_2)$;	$(c_3=c_1)\vee(c_3=c_2)$

命题公式：
$(a_1=A)\wedge(b_1=B)\wedge(c_1=a_1)\wedge(c_2=b_1)\wedge((c_3=c_1)\vee(c_3=c_2))$

分析结果：
$a_1=\{A\}$　$c_1=\{A\}$
$b_1=\{B\}$　$c_2=\{B\}$
$c_3=\{A,B\}$

图 4　分析结构分析示例

上述示例表明，这种抽取方法能够避免分支结构公式中“∧”联接符和“∨”联接符相互嵌套的情况[9]，大大降低了公式向 CNF 形式转换的复杂度，体现了 SSA 形式给公式抽取带来的又一好处。

循环结构命题公式的分析相对复杂。首先，我们仍然将结构中相关语句抽象出的公式取交作为该结构的原始公式，但该公式不能直接交由求解器求解。在进一步分析其原因之前，我们首先给出若干定义。

定义 1　赋值图是一个有向图 $G=\langle N_{val},N_{var},E\rangle$，其中：

①G 包含两类节点：一类是表示具体值的节点，这类节点的集合表示为 N_{val}；另一类是表示变量的节点，这类节点的集合表示为 N_{var}。

② 边集 $E=N_{val}\cup N_{var}\rightarrow N_{var}$，包含两类边，一类是变量到变量的有向边，另一类是值到变量的有向边。

定义 2　赋值图上 v_1 到 v_2 的路径称为 v_1 到 v_2 的赋值路径。赋值路径表示了值经过中间变量流入目标变量的过程。赋值图上这样的路径可能不只一条，我们用 $Path(v_1,v_2)$表示 v_1 到 v_2 的赋值路径的集合。如果某条赋值路径有可能被执行到，则称该路径是合法路径，否则称该路径是非法路径。

有了上述两个定义，可以得出性质 1 如下。

性质 1　合法赋值路径的头节点一定是表示具体值的节点。

由定义 1 可知，表示值的节点只有出边，没有入边，因此这类节点不可能作为赋值路径的中间节点。那么，如果某条赋值路径以表示变量的节点为头节点，就不会有值流入路径包含的所有变量，即这些变量不会取得具体的值。显然这样的赋值路径是不合法的。

类型的传递可以用赋值图和赋值路径来表示，其中变量的值就是其运行时类型，我们用大写字母表示类型，小写字母表示变量。由程序的命题公式构造赋值图的方法非常简单，首先将公式中的每个变量(或值)表示为单独的节点，其次将形如“(变量名＝变量名或类名)”的子式转换为有向边“变量名或类名→变量名”。赋值路径从赋值图中抽取。例如，图 5(1)是一个公式，图 5(2)是由该公式构造出的赋值图，图 5(3)是从该赋值图上抽取的合法赋值路径。

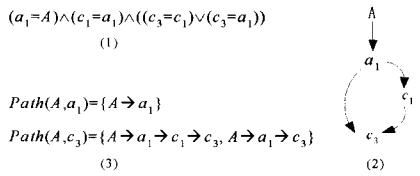


图 5　命题公式及其赋值图

下面的性质给出了循环结构的原始公式不能直接求解的原因。

性质 2　循环结构的原始公式蕴含带环的赋值路径。

该性质不难证明，限于篇幅我们不给出证明过程。

例如在图 6 中，图 6(1)是一循环结构；图 6(2)是其 SSA 形式表示；图 6(3)是结构体中每条语句抽象出的公式，则循环结构的公式是由子公式(1)—(7)组成的合取式；图 6(4)给出了该循环结构公式中含有带环赋值路径的子式，以及相应的带环路径。由于没有值流入环中的变量，因此这些变量就没有受到约束，求解器求解时会用所有可能的类去尝试，从而造成精度的大幅降低，所以不能直接对原始公式求解。

性质 3　循环结构原始公式蕴含的带环赋值路径是不合法的。

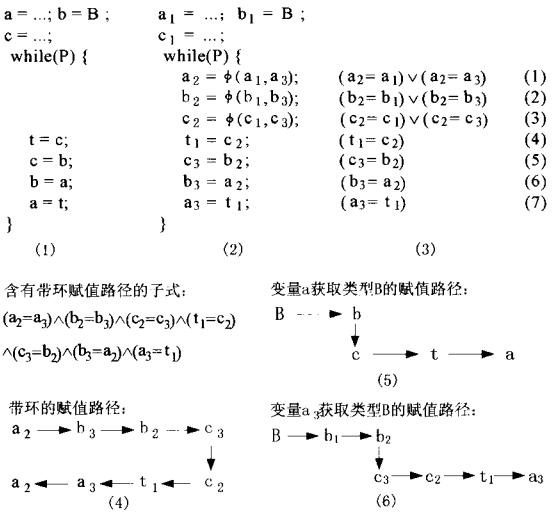


图 6　循环结构分析示例

由 SSA 表示形式的特点可知,结构中的每个变量只被赋值一次,从赋值路径的角度看,即表示变量的节点只有一条入边。如果赋值路径含有环路,那么它就一定不会包含表示值的节点,由性质 1,该赋值路径是不合法的。

因此,将含有带环赋值路径的子式从原始公式中剔除不会影响分析结果的准确性。剔除方法是:首先,将命题公式转换成析取式,然后逐个由合取子式构造赋值图。若赋值图中含有强连通分量,则舍弃该子式。最后,将所有剩下的子式重新组成析取式,作为循环结构最终的命题公式,由求解器求其所有解即可得到分析结果。我们将从原始公式中剔除含有带环赋值路径的子式的过程称为对原始公式的验证。

循环结构分析需要验证的子式的个数是 $\prod_{i=1}^{m} k_i$,其中 m 是公式中 Ø 函数的个数(与结构体中的变量个数有关),k_i 是第 i 个 Ø 函数参数的个数。验证一个子式的复杂度由构造赋值图的复杂度和求强连通分量的复杂度组成,前者的复杂度是 $O(n+e)$,其中 n 是节点个数,e 是边的个数,后者的复杂度也是 $O(n+e)$,因此总的复杂度是 $k^m \cdot O(n+e)$(k 通常等于 2)。

由此可见,循环结构分析的复杂度是指数级的,但是当 m 不大时,这种分析方法是可行的。我们将在 4.2 节介绍减小 m 的方法。

虽然该循环结构分析方法比较复杂,但对比传统分析方法,它具有如下显著优势。

性质 4　循环结构的分析不需要反复迭代至不动点。

限于篇幅,我们仅给出非严格的简要说明。经过预处理,结构中的赋值语句全都如"$v_1 = v_2$;"的形式,其中 v_1 和 v_2 都是简单变量(包括全局变量)。从而对于任一变量,我们只需考查类型第一次流入它的最短赋值路径。这样的路径涉及结构体中的每条相关语句最多一次,但语句的涉及顺序不同于行文顺序,和程序的控制流有关。对于传统分析方法而言,该赋值路径可能需要结构体的多次迭代才会被执行到。而我们的方法,由于对 Ø 函数语句也进行了抽象,结构的公式从而含有控制流信息,以致上述的赋值路径蕴含在其中,因此只需对该公式求解一次即可。

例如在图 6 中,图 6(5)所示是对于图 6(1)的 $Path(B,a)$ 中的一条最短路径,传统的分析方法需要迭代两次才能走完该路径。而图 6(6)所示是对于图 6(2)的 $Path(B,a_3)$ 中的一条最短路径,该路径蕴含在结构的公式中,对公式求解一次就能走完该路径。

4.2　结构组合分析

本节介绍如何将基本结构组合起来分析,并且将分析方法扩展到一般的程序结构上。

结构间的嵌套可分为其它结构分别嵌套顺序结构、分支结构以及循环结构 3 种情况(分别如图 7(1)、图 7(2)、图 7(3)所示)。假设复合结构中每个部分的公式(即 F_1, F_2, F_3, F_4)已验证过。那么,对于前两种情况,复合结构的命题公式是各部分公式的交,并且无需对复合公式再次验证。对于最后一种情况,复合结构的公式也是各部分公式的交,但是需要对复合公式再次验证。

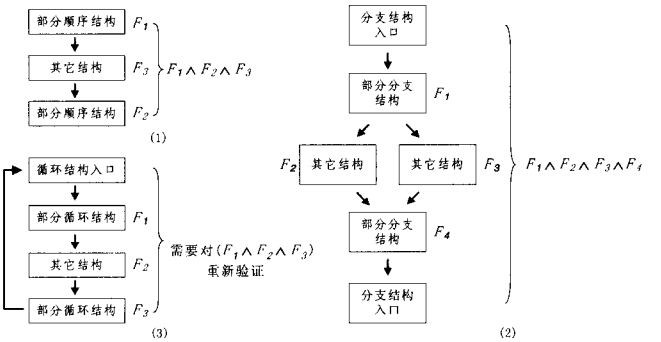


图 7　结构嵌套分析示例

组合分析的优势在于将程序按结构分解成规模较小的多个部分,对每个部分分别单独抽象,然后再将子公式取交,这样就能够降低程序抽象的复杂度。例如,对于一个多层循环结构,或者一个较大规模的分支结构包含了若干个循环结构,此时如果对结构整体抽象并一次验证,则代价势必较高。而通过组合分析,我们可以对多层循环结构分层次验证,或者将分支结构中的若干循环结构分开验证,这样可以较大地降低验证的复杂度。因此,结合组合分析方法,我们对循环结构的分析方法是可行的。

由此我们可以得到抽取程序公式的通用方法如下:

输入:程序

输出:程序抽象出的命题公式

1)构造程序的控制流图,求出图中所有极大强连通分支的集合。

2)若集合为空,表示程序不包含循环结构,则将所有相关语句抽象的公式取交作为程序的公式,并返回。若集合不为空,则进入下一步。

3)针对集合中的每个极大强连通分支,若其规模不大,或者其内部不包含子循环结构,则将其作为一个整体抽取公式并做验证,然后返回。若该强连通分支规模较大,并且其内部包含子循环结构,则进入下一步。

4)将该分支作为子图,取消其最外层的回边,然后求该子图的极大强连通分支的集合,对该集合再次运用步骤 2)至步骤 4)抽象集合中各分支的公式,集合抽象完毕后,将其中所有分支的公式以及该子图包含的其它相关语句抽象出的公式组合起来(取交)并做验证,然后返回。

利用上述方法抽象出程序的命题公式以后,再由求解器求出所有解即完成对程序的分析。

该方法既适用于分析任意结构的 Java 源程序(例如程序

中可以包含异常处理语句，带标号的 continue 或 break 语句），又可以直接分析 Java 字节码程序或者其它形式表示的 Java 程序（例如 Jimple[17]，Shimple[17]等）。

5 过程间分析

过程间分析的基本思想是将若干存在调用关系的方法的公式组合起来，统一求解。显然，组合的方法个数越多，则求解的结果越精确。最精确的过程间分析是根据保守调用图，从 main 方法开始，将整个程序抽象成一个公式求解；而最粗糙的过程间分析则是针对每个方法单独求解。当然，还可以指定调用路径的长度，将长度范围内的方法组合在一起求解，这种分析策略的精度介于前两者之间。实际上，这种过程间分析方法是以调用路径为上下文的上下文敏感分析，而针对方法的独立分析则退化为上下文不敏感分析。

本节将分别介绍上述 3 种分析策略。同时，本节还将给出方法调用语句的抽象规则，以及形式参数、全局变量（对象属性和类属性）类型的抽象方法，它们因采用的分析策略不同而各异。

5.1 独立分析

独立分析方法通过使用 RTA 方法对方法的形式参数类型、全局变量类型以及方法调用返回类型进行全局估计，从而将方法抽象成命题公式，并独立求解。独立分析策略的特点是快速但精度较低。

下面的规则(5)－(8)给出了将方法独立抽象成公式的方法。其中，我们用 $RT[v]$ 表示通过 RTA 方法分析出的变量 v 的所有可能类型的集合。规则(5)表示方法的命题公式主要由 3 部分组成，分别是表示形参类型的公式（规则(6)）、表示全局变量类型的公式（规则(7)）和方法体的公式。规则(6)－(8)表示形参的类型、全局变量的类型、，及方法调用返回的类型是 RTA 分析所得类型集合中的任意一个类型。这 3 处估计是该分析策略精度不高的根源。另外需要注意的是，在规则(6)和规则(7)中，变量的下标是 1，这就是在 SSA 形式变换中，方法体中形式参数和全局变量的下标从 2 开始计数的原因。对于方法体，我们按照 4.2 节介绍的通用方法，利用规则(1)－(4)、(8)，对其进行抽象。

将方法抽象成命题公式之后，由求解器求解即可得到该方法包含的所有变量的类型。

$$(5)F(MethodName(params)Methodbody)=F(params)\wedge(\bigwedge_{\substack{all\ global\\ variable\ v}}F(v))\wedge F(MethodBody)$$

$$(6)F(params)=(\bigwedge_{\forall p\in params}(\bigvee_{\forall C\in RT[p]}(p_1=C)))$$

$$(7)F(v|vis\ global\ variable)=(\bigvee_{\forall C\in RT[v]}(v_1=C))$$

$$(8)F(\text{“}v=o.m(a_1,\cdots,a_k)\text{”})=(\bigvee_{\forall C\in RT[v]}(v=C))$$

5.2 整体分析

当程序的规模不是很大，或者求解器具有较强的求解能力时，可以把整个程序抽象成一个命题公式来求解，以获得精确的分析结果。抽象整个程序涉及的规则包括规则(1)－(4)和规则(9)－(12)。

如规则(9)所示，抽象过程从 main 方法开始，我们采用 4.2 节介绍的通用方法抽取 main 方法的公式。在抽取过程中，遇到方法调用语句时，通过保守调用图得到所有可能的实际被调方法，在当前上下文环境下抽象出所有这些方法的公式，然后交上表示接受变量值是任意方法返回值的析取式（规则(11)）。对于其它语句，采用规则(1)－(4)抽象。

当前环境下被调方法抽象的公式是 3 部分子式的交（规则(12)）。第一部分子式是对调用环境的抽象，表示实参到形参的类型传递关系。第二部分子式是对方法中全局变量类型的估计（这是一种保守方法，更精确的方法应该按照调用次序，将前后方法中的全局变量链接起来）。第三部分子式是对方法体的抽象，其抽象过程与抽象 main 方法的过程相同，也是采用 4.2 节的通用方法抽取公式。

需要注意的是，对于递归调用语句，应当使用规则(8)，以避免循环抽象，也就是说，我们只进入递归方法抽取公式一次。

将整个程序抽象成命题公式之后，由求解器求解即可得到所有变量的类型。

$$(9)F(program)=F(methodbody\ of\ main)$$

$$(10)F(v|vis\ global\ variable)=(\bigvee_{\forall C\in RT[v]}(v_1=C))$$

$$(11)F(\text{“}v=o.m(a_1,\cdots,a_k)\text{”})=((\bigwedge_{\substack{every\ actual\\ method\ C\_m}}F(C\_m(p_1,\cdots,p_k)\{a_1,\cdots,a_k\}))\wedge(\bigvee_{\substack{every\ actual\\ method\ C\_m}}(v=C\_m\_ret)))$$

$$(12)F(C\_m(p_1,\cdots,p_k)\{a_1,\cdots,a_k\})=\wedge_{i=1}^{k}(p_i=a_i)\wedge(\bigwedge_{\substack{all\ global\ variable\\ v\ in\ method\ C\_m}}F(v)\wedge F(\text{body of }C\_m)C\_m$$

5.3 分部分析

分部分析策略通过指定方法调用路径的长度限定公式包含的方法个数，从而达到分析复杂度和分析精度之间的任意平衡。

假设指定的调用路径长度为 k，程序公式抽象过程如下：从 main 方法开始，依据保守调用图，按照 5.2 节的方法将调用路径长度小于等于 k 以内的方法抽取成一个命题公式，然后利用拓扑排序方法，从保守调用图上剩下的方法中找出起始方法，分别从这些起始方法开始继续按照前面的方法抽取。在抽取过程中，对于不是 main 方法的起始方法，其参数类型和全局变量类型的抽象遵循规则(6)和(7)；对于调用路径长度小于 k 的方法，按照 5.2 节的方法抽取命题公式，对于调用路径长度等于 k 的方法中的方法调用语句按照规则(8)抽象。对于递归调用的抽象，同 5.2 节一样，遵循规则(8)。

将程序抽象成若干命题公式之后，由求解器求解就可以得到所有变量的类型。

6 实例研究

图 9 左边是一个程序实例，我们将分别以第 5 节介绍的 3 种过程间分析策略分析该程序，其中在分部分析策略中，我们规定调用路径的长度为 1。

图 9 右边是程序经过预处理以后的形式，3 种分析策略都在该形式的程序上抽象命题公式，在没有歧义的情况下，我们省去了变量名的前缀。该程序经由 RTA 分析获得的保守调用图如图 8 所示。

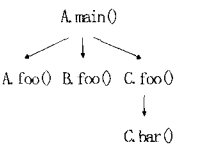


图 8 保守调用图

```
public Class A{
  public A foo(A arg) { return arg; }
  public static void main(String args[]){
    A a = new C();
    A b = a.foo(a); }
}
public Class B extends A{
  public A foo(A arg){
    A retval = new B();
    return retval; }
}
public Class C extends A{
  public A f;
  public A foo(A arg){
    f = arg;
    A retval = bar(f);
    return retval; }
  public A bar(A arg){
    A a, b, c;
    a = new A();
    b = arg; c = new C();
    while(P){
      a = b;
      if (Q) b = c;
      else c = a; }
    return c; }
}
public Class D{
  public D foo(){
    D retval = new D();
    return retval;}
}
```

```
public Class A{
  public A foo(A A_foo_arg1){ A A_foo_ret1 = ø(A_foo_arg1); }
  public static void main(String A_main_args1[]){
    A A_main_a1 = new C();
    A A_main_b1 = A_main_a1.foo(A_main_a1); }
}
public Class B extend A{
  public A foo(A B_foo_arg1){ A B_foo_retval1 = new B();
  A B_foo_ret1 = ø(B_foo_retval1); }
}
public Class C extend A{
  public A f;
  public A foo(A C_foo_arg1){
    C_foo_C_f2 = C_foo_arg1; A C_foo_retval1 = bar(C_foo_C_f2);
    A C_foo_ret1 = ø(C_foo_retval1); }
  public A bar(A C_bar_arg1){
    A C_bar_a1, C_bar_b1, C_bar_c1;
    C_bar_a1 = new A(); C_bar_b1 = C_bar_arg1; C_bar_c1 = new C();
    while(P){
      C_bar_b2 = ø(C_bar_b1, C_bar_b3);
      C_bar_c2 = ø(C_bar_c1, C_bar_c3);
      C_bar_a2 = C_bar_b2;
      if (Q) C_bar_b3 = C_bar_c2;
      else C_bar_c3 = C_bar_a2; }
    C_bar_c4 = ø(C_bar_c1, C_bar_c2, C_bar_c3);
    A C_bar_ret1 = ø(C_bar_c4); }
}
public Class D{
  public D foo(){
    D D_foo_retval1 = new D();
    D D_foo_ret1 = ø(D_foo_retval1); }
}
```

图 9　程序实例

独立分析策略抽象出的各个方法的命题公式如表 1 所示。

表 1　独立分析抽取出的公式

方法	命题公式
A. foo	((arg1=A)∨(arg1=B)∨(arg1=C))∧(ret1=arg1)
A. main	(a1=C)∧((b1=A)∨(b1=B)∨(b1=C))
B. foo	((arg1=A)∨…∨(arg1=C))∧(retval1=B)∧(ret1=retval1)
C. foo	((arg1=A)…)∧(((C_f1=A)∨…∨(C_f1=C))∧(C_f2=arg1)∧((retval1=A)∨…∨(retval1=C))∧(ret1=retval1)
C. bar	((arg1=A)…)∧(a1=A)∧(b1=arg1)∧(c1=C)∧(((b2=b1)∧(c2=c1)∧(a2=b2)∧(b3=c2)∧(c3=a2))∨((b2=b3)∧(c2=c1)∧(a2=b2)∧(b3=c2)∧(c3=a2))∨((b2=b1)∧(c2=c3)∧(a2=b2)∧(b3=c2)∧(c3=a2)))∧((c4=c1)∨(c4=c2)∨(c4=c3))∧(ret1=c4)
D. foobar	(retval1=D)((ret1=retval1)

整体分析策略抽象出的整个程序的命题公式如下所示。其中，第 2 行是 A. foo 抽象出的公式，第 3 行是 B. foo 抽象出的公式，第 5－7 行是 C. bar 方法抽象出的公式，4－8 行是 C. foo 方法抽象出的公式。

1.　(A_main_a1=C)

2.　∧((((A_foo_arg1=A_main_a1)∧(A_foo_ret1=A_foo_arg1))

3.　∧((B_foo_arg1=A_main_a1)∧(B_foo_retval1=B)∧(B_foo_ret1=B_foo_retval1))

4.　∧((C_foo_arg1=A_main_a1)∧((C_foo_C_f1=A)∨(C_foo_C_f1=B)∨(C_foo_C_f1=C))∧(C_foo_C_f2=C_foo_arg1)

5.　∧((C_bar_arg1=C_foo_C_f2)∧(C_bar_a1=A)∧(C_bar_b1=C_bar_arg1)∧(C_bar_c1=C)∧(((b2=b1)∧(c2=c1)∧

6.　(a2=b2)∧(b3=c2)∧(c3=a2))∨((b2=b3)∧(c2=c1)∧(a2=b2)∧(b3=c2)∧(c3=a2))∨((b2=b1)∧(c2=c3)∧(a2=b2)∧(b3

7.　=c2)∧(c3=a2)))∧((C_bar_c4=C_bar_c1)∨(C_bar_c4=c2)∨(C_bar_c4=c3))∧(C_bar_ret1=C_bar_c4))

8.　∧(C_foo_retval1=C_bar_ret1)∧(C_foo_ret1=C_foo_retval1))

9.　∧((A_main_b1=A_foo_ret1)∨(A_main_b1=B_foo_ret1)∨(A_main_b1=C_foo_ret1))

由于在分部分析策略中指定的调用路径长度为 1，因此该策略将整个程序抽象成两个命题公式，一个是由 A. main，A. foo，B. foo，C. foo 组成的公式(如下所示)，另一个是 C. bar 的公式(同表 1 中 C. bar 的公式)。

1.　(A_main_a1=C)

2.　∧(((A_foo_arg1=A_main_a1)∧(A_foo_ret1=A_foo_arg1))

3.　∧((B_foo_arg1=A_main_a1)∧(B_foo_retval1=B)∧(B_foo_ret1=B_foo_retval1))

4.　∧((C_foo_arg1=A_main_a1)∧((C_foo_C_f1=A)∨(C_foo_C_f1=B)∨(C_foo_C_f1=C))∧(C_foo_C_f2=C_foo_arg1)

5.　∧((C_foo_retval1=A)∨…∨(C_foo_retval1=C))∧(C_foo_ret1=C_foo_retval1))

6.　∧((A_main_b1=A_foo_ret1)∨(A_main_b1=B_foo_ret1)∨(A_main_b1=C_foo_ret1))

上述 3 种分析策略的分析结果以及程序运行时的实际情况如表 2 所示。

表 2　分析结果

分析方法	A_foo_arg1	A_foo_ret1	A_main_a1	A_main_b1	B_foo_arg1	B_foo_retval1	B_foo_ret1	C_foo_arg1	C_foo_C_f2
独立分析	{A,B,C}	{A,B,C}	{C}	{A,B,C}	{A,B,C}	{B}	{B}	{A,B,C}	{A,B,C}
整体分析	{C}	{C}	{C}	{B,C}	{C}	{B}	{B}	{C}	{C}
分部分析	{C}	{C}	{C}	{A,B,C}	{C}	{B}	{B}	{C}	{C}
实际情况	Ø	Ø	{C}	{C}	Ø	Ø	Ø	{C}	{C}
分析方法	C_foo_retval1	C_foo_ret1	C_bar_arg1	C_bar_a1	C_bar_b1	C_bar_c1	C_bar_b2	C_bar_c2	C_bar_a2
独立分析	{A,B,C}	{A,B,C}	{A,B,C}	{A}	{A,B,C}	{C}	{A,B,C}	{A,B,C}	{A,B,C}
整体分析	{C}	{C}	{C}	{A}	{C}	{C}	{ C}	{ C}	{ C}
分部分析	{A,B,C}	{A,B,C}	{A,B,C}	{A}	{A,B,C}	{C}	{A,B,C}	{A,B,C}	{A,B,C}
实际情况	{C}	{C}	{C}	{A}	{C}	{C}	{C}	{C}	{C}
分析方法	C_bar_b3	C_bar_c3	C_bar_c4	C_bar_ret1	D_foo_retval1	D_foo_ret1			
独立分析	{A,B,C}	{A,B,C}	{A,B,C}	{A,B,C}	{D}	{D}			
整体分析	{C}	{C}	{C}	{C}	Ø	Ø			
分部分析	{A,B,C}	{A,B,C}	{A,B,C}	{A,B,C}	Ø	Ø			
实际情况	{C}	{C}	{C}	{C}	Ø	Ø			

结束语　面向对象程序类型分析的经典方法主要有以下几种：

Bacon 提出的 RTA 方法通过查找程序中所有对象创建语句，构建可能创建实际对象的类集合，变量的类型则估计为

该集合中所有是其声明类型的子类型(包括自身)的元素。RTA 方法粒度较粗,没有深入考虑类型在变量间的传递。

Frank 提出的 XTA 方法在 RTA 的基础上考虑了方法调用以及属性访问引起的类型传递,但没有深入考虑方法内涉及到局部变量的类型传递。

Sundaresan 提出的 VTA 方法则通过类型传播图考虑了由变量赋值语句、对象创建语句、方法调用语句等引起的类型传递,但该方法是流不敏感、上下文不敏感的,因此精度不高。

Shivers 提出的 0-CFA 方法基本思想和 VTA 方法类似,但它是流敏感的。改进的 k-CFA($k>0$)方法是以调用路径为上下文的上下文敏感分析方法。以 k-CFA($k\geqslant 0$)为代表的传统流敏感分析方法,需要在控制流图上迭代分析至不动点,因此代价较大。另外,在做上下文敏感分析时,存在空间爆炸问题。

我们的分析方法是一种流敏感、上下文敏感的方法,精度较高,由于不需要在控制流图上迭代,因此做上下文敏感分析时不存在空间爆炸问题。同时,由于求解过程中使用了高效的启发式搜索算法,因此类型推导也更加迅速。

在研究过程中,除了基于 SSA 形式的抽象以外,我们还考虑了另一种建模方式,即将程序抽象成状态转移系统。我们比较了两种建模方式的优缺点。抽象成状态转移系统的缺点是:1)系统状态包含程序中所有的变量,使得公式规模急剧膨胀;2)需按照语法提取程序公式,且存在"∨"、"∧"相互嵌套,增加了转换成 CNF 形式的复杂度;3)循环结构需要迭代分析。其优点是不需要对循环结构的公式做验证。

基于 SSA 形式抽象的优点是:1)公式规模小,布尔变量的个数较前者少很多;2)简化了程序命题公式的生成,无需按照语法提取程序公式,并且不存在"∨"、"∧"相互嵌套情况;3)循环结构无需迭代分析。其缺点是需要对循环结构的公式做验证。鉴于此,我们采用了后一种建模方式。

在今后的工作中,我们将进一步研究提高公式抽取速度以及验证效率的方法。同时,我们打算将该类型分析方法应用到 AspectJ 程序编译优化[18]等其它的程序分析和优化工作中。

参 考 文 献

[1] Dean J, Grove D, Chambers C. Optimization of object-oriented programs using static class hierarchy analysis // Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP95). Aarhus, 1995: 77-101

[2] Bacon D F, Wegman M, Zadeck K. Rapid type analysis for C++. Technical Report: RC number pending. IBM Thomas J. Watson Research Center, 1996

[3] Tip F, Palsberg J. Scalable Propagation-based Call Graph Construction Algorithms // Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, languages, and Applications (OOPSLA'00). Minneapolis, 2000: 281-293

[4] Sundaresan V, Hendren L, Razafimahefa C, et al. Practical virtual method call resolution for Java // Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00). Minneapolis, 2000: 264-280

[5] Shivers O. Control-flow Analysis of Higher-order Languages. Technical Report: CMU-CS-91-145. Carnegie Mellon University, 1991

[6] Fu Zhaohui, Mahajan Y, Malik S. New features of the SAT'04 versions of zChaff, 2004. http://www.princeton.edu/~chaff/zchaff/sat04.pdf

[7] Moskewicz M, Madigan C, Zhao Y, et al. Chaff: Engineering an efficient sat solver // Proceedings of the 38th Conference on Design Automation Conference (DAC'01). Las Vegas, 2001: 530-535

[8] Clarke E, Kroening D, Sharygina N, et al. SAT-based Predicate Abstraction of Programs. Technical Report: CMU/SEI-2005-TR-006. Carnegie Mellon University, 2005

[9] Xie Yichen, Aiken A. Scalable Error Detection Using Boolean Satisfiability // Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05). Long Beach, 2005: 351-363

[10] Xie Yichen, Aiken A. Saturn: A scalable framework for error detection using Boolean satisfiability. ACM Transactions on Programming Languages and Systems (TOPLAS), 2007, 29(3): 1-43

[11] Dennis G, Chang F Sheng-Ho, Jackson D. Modular verification of code with SAT // Proceedings of International Symposium on Software Testing and Analysis (ISSTA'06). Portland, 2006: 109-119

[12] Grove D, Chambers C. A Framework for Call Graph Construction Algorithms. ACM Transactions on Programming Languages and Systems (TOPLAS), 2001, 23(6): 685-746

[13] Cytron R, Ferrante J, Rosen B K, et al. An Efficient Method of Computing Static Single Assignment Form // Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89). Austin, 1989: 25-35

[14] Allen F E. Control flow analysis. SIGPLAN. Notices, 1970, 5(7): 1-19

[15] Davis M, Putnam H. A computing procedure for quantification theory. Journal of the ACM, 1960, 7(3): 201-215

[16] Chauhan P, Clarke E M, Kroening D. Using SAT based image computation for reachability analysis. Technical Report: CMU-CS-03-151. Carnegie Mellon University, 2003

[17] Vallée-Rai R, Hendren L, Sundaresan V, et al. Soot-a Java Optimization Framework // Proceedings of IBM Center for Advanced Studies Conference (CASCON'99). Toronto, 1999: 125-135

[18] 曹璟,徐宝文,周晓宇,等. 基于面向方法调用图的 AspectJ 动态通知编译优化. 软件学报(已录用)