変数に格納されるオブジェクトの型を仮定した仮想メソッド呼び出し解決手法

An approach for virtual method call resolution with an assumption of a variable type

鹿島 悠* 石尾 隆† 井上 克郎‡

あらまし 静的解析において仮想メソッド呼び出しの解決は重要な技術であり，既存研究では呼び出される可能性のあるメソッドすべてを網羅しつつ精度を高める手法が提案されている．しかしプログラム理解では，開発者は変数に格納されるオブジェクトの型を仮定してメソッドの呼び出し関係を調査することがある．本研究では，変数に格納されるオブジェクトが特定の型であると仮定された場合に，その型情報を用いて，他の変数のオブジェクトの型についても一貫した型情報の特定を行い，仮想メソッド呼び出しの解決を行う手法を提案する．実験では，既存手法と提案手法とで作成されるコールグラフの比較を行い，提案手法の有効性を確認した．

Summary. In static analysis, virtual method call resolution is an important technique. Several methods are propsed for detecting all callable methods precisely. In a case of program comprehension, call relations should be taken on assuming a variable type. This paper proposes a new approach for resolving dynamic binding; our approach identifies variables affected by type assumptions and then consistently resolves dynamic binding. In an experiment, we compared call graphs made by existing methods and ones made by our approach, then confirmed the effectiveness of our approach.

1 はじめに

Java に代表されるオブジェクト指向プログラミング言語では，仮想メソッド呼び出しが提供されている．仮想メソッド呼び出しでは，実行時に呼ばれるメソッドは静的には決定されず，実行時のレシーバオブジェクトのクラスにより決定される．

ソースコードに対して静的解析を行う場合は，仮想メソッド呼び出しの解決，すなわち，仮想メソッド呼び出しにより実行時に呼ばれる可能性のあるメソッドの集合を特定する作業を行う必要がある．仮想メソッド呼び出しの解決手法として，クラス階層解析 [1]，Rapid Type Analysis [2]，Variable Type Analysis [3]（VTA）といった方法が提案されている．これらの方法は，主としてコンパイラによるプログラムの最適化を目的としており，仮想メソッド呼び出しで呼び出される可能性があるメソッドをすべて網羅する必要があった．

しかし，プログラム理解やデバッグを目的として静的解析を行う場合，開発者が注目している変数に特定の型のオブジェクトが格納されていると仮定したい場合がある． 例えば，Fluid Source Code Views [4] や Code Bubbles [5] ではメソッド呼び出しをインライン展開した結果を表示するが，利用者は呼び出し文ごとにレシーバオブジェクトの型を選択しなくてならない．また，同一オブジェクトが格納される変数に対しても利用者が個別に同一の型を割り当てていかなければならない．大場ら [6] の提案するパターン抽出手法では，仮想メソッド呼び出しのインライン展開を行なっているが，同一のオブジェクトを使用する複数の呼び出し文で一貫した型の解決がなされておらず，現実には起こり得ない呼び出し系列を抽出する可能性が

*Yu Kashima, 大阪大学 大学院情報科学研究科

†Takashi Ishio, 大阪大学 大学院情報科学研究科

‡Katsuro Inoue, 大阪大学 大学院情報科学研究科

ある．

これらの手法において，開発者が注目している変数 v が特定の型 C のオブジェクトを参照していると仮定したとき，v をレシーバオブジェクトとする仮想メソッド呼び出しの解決は，v の型は常に C であるという仮定の下で行うのが望ましい．また，v の値を代入される変数など，代入されるオブジェクトの型が v に依存する変数がレシーバオブジェクトとして利用される場合についても同様に扱うことが望ましい．さらに，型を仮定することで呼び出し関係がなくなるメソッドについては，引数や戻り値の影響を除去することも重要である．現状，上記のように変数に特定の型のオブジェクトが代入されている仮定の下で，プログラム全体の仮想メソッド呼び出しを一貫して解決する手法は提案されておらず，また手作業で一貫した解決を行うのは，プログラムサイズの肥大化や変数どうしの複雑なエイリアスの問題から困難であると予想される．

そこで本研究では，ある変数に代入されているオブジェクトの型を仮定した場合の，仮想メソッド呼び出しの解決方法を提案する．本手法は VTA でも用いられている Type Propagation Graph（TPG）を利用し，仮定した型に依存する変数を特定し，それらの変数を用いた仮想メソッド呼び出しの解決を行う．さらに，解決の結果呼び出されないことが判明したメソッドがある場合は，TPG を加工し VTA を再度行うことにより，呼ばれなくなったメソッドの影響を除去した仮想メソッド呼び出しの解決結果を得る．

提案手法を実装し，実験で次の 3 つの調査を行った．まず，複数の型のオブジェクトを参照しうる変数を調査した．次に，それらの変数に対して，その変数の型に依存して型が決まる変数がレシーバオブジェクトとして用いられるメソッド呼び出し数を調査した．そして，既存手法によるコールグラフと，型の仮定と TPG の加工，そして再度の VTA を行ったあとのコールグラフを比較した．実験の結果，複数の型を取りうる変数の割合は全体に比して少数であったこと，そのうち仮想メソッド呼び出しの解決に影響を与えるような変数はさらに少数であったこと，しかし，適当な変数を選択することでコールグラフのサイズを大きく減少させるような場合があることを確認した．

本論文の主な貢献は以下の通りである．

- 変数が参照するオブジェクトの型を仮定した状況で利用できる，新たな仮想メソッド呼び出しの解決方法を提案した．
- 提案手法を実装し，実際のプログラムに対して効果を確認した．

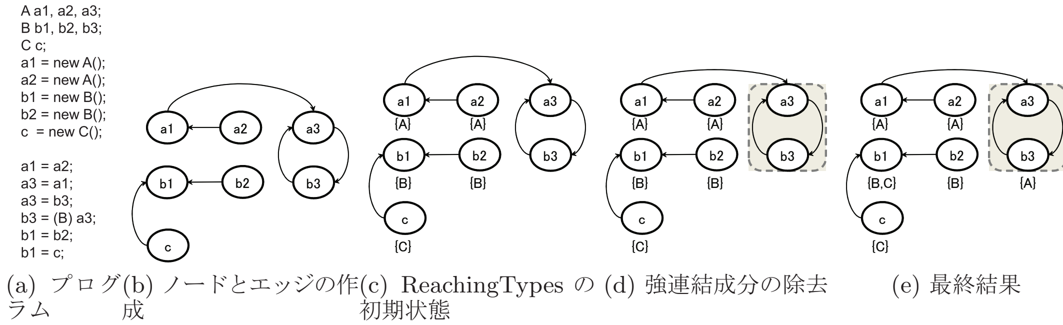
以降，2 節では研究の背景として既存の仮想メソッド呼び出しの解決方法について述べる．3 節では提案手法を，4 節では評価実験を，5 節では関連研究を，6 節ではまとめと今後の課題を述べる．

2　背景

2.1　仮想メソッド呼び出しの解決方法

仮想メソッド呼び出し $v.m(a_1, a_2, ..., a_n)$ があるとき，実行時に呼び出されるメソッドは，変数 v が実行時に参照しているオブジェクト o に実装されている m と同じシグネチャを持つメソッド m_{target} である．m_{target} の検索は次の手順で行う．o の型をクラス C としたとき，C のクラス階層を根に向かって探索する．そして最初に見つかった m と同じシグネチャを持つメソッドが $C.m_{target}$ となる．

静的解析において，レシーバオブジェクト o を一意に特定することはできないため，v が実行時に参照しうるオブジェクトの型の集合 $runtime_types(v)$ を得る必要がある．$runtime_types(v)$ が得られた後は，$runtime_types(v)$ 中のそれぞれの C_i のクラス階層を探索し，呼び出されうるメソッドの集合 $C_i.m_{target}$ が得られる．以上のように，仮想メソッド呼び出しにおいて実行時に呼び出される可能性のあるメソッドの集合を特定する作業を仮想メソッド呼び出しの解決と呼ぶ．そして，仮想



(a) プログラム (b) ノードとエッジの作成 (c) ReachingTypes の初期状態 (d) 強連結成分の除去 (e) 最終結果

図 1: VTA の例

メソッド呼び出しの解決手法とは，$runtime_types(v)$ を求める手法と考えることができる．

2.2 クラス階層解析

クラス階層解析は，レシーバオブジェクトの宣言型と宣言型のクラス階層を分析し，$runtime_types(v)$ を保守的に求める手法である [1]．クラス階層解析では，変数 v の宣言型を d とした時，$runtimes_types(v)$ は，v に代入されるオブジェクトが取りうる型の集合 $hierarchy_types(d)$ となる．

Java における *hierarchy_types(d)* は以下のように得られる．

- d がクラス C の場合，C と C のサブクラスすべての集合．
- d がインターフェース I の場合，次の 2 つの集合の和集合．
  - I または I の派生インターフェイスを実装しているすべてのクラスの集合．以降 *implements(I)* と呼ぶ．
  - *implements(I)* のサブクラスすべて．

クラス階層解析は，宣言型の変数に代入されても型エラーにならないサブクラスすべてを漏れ無く $runtimes_types(v)$ に追加するため，仮想メソッド呼び出しの解決時に呼ばれる可能性のあるメソッドすべてを保守的に取得できるという特徴がある．また，対象のプログラム中の型階層構造と宣言型さえ分かれば良いため非常に高速に動作する．

2.3 Variable Type Analysis

Variable Type Analysis（VTA） [3] は，変数間の代入関係や引数渡しの関係を示す Type Propagation Graph（TPG）を作成し，オブジェクトのアロケーションから変数へと型伝播を行い，$runtime_types(v)$ を求める手法である．VTA では TPG 上に変数に対応するノードを作成し，それぞれのノードに $ReachingTypes$ を割り当てる．この $ReachingTypes$ が変数に代入される可能性のあるオブジェクトの型の集合を示しており，$runtime_types(v)$ は，v に対応するノードの $ReachingTypes$ となる．VTA は，変数間の代入関係を用いるためプログラム中で代入される可能性の無いサブクラスの影響を除去できるため，クラス階層解析よりも精度の高い結果を得られる．

2.3.1 Type Propagation Graph の作成方法

TPG は有向グラフであり，ノードはプログラム中の変数に対応して作成され，エッジは変数の代入関係や配列による参照の共有に対応して作成される．

TPG の作成では，まず，クラス階層解析を用いて仮想メソッド呼び出しの解決を行い，コールグラフを作成する．以降このコールグラフを保守的なコールグラフと呼ぶ．次に，TPG を作成する．まず，ノードを以下の手順で作成する．

- プログラム P に含まれるすべてのクラス C に対して，C が持つすべてのオブジェクト型のフィールド f ごとに，ラベル $C.f$ のノードを作成する．
- P の保守的なコールグラフ中に出現するすべてのメソッド $C.m$ に対して以下の処理を行う．
  - $C.m$ のオブジェクト型の仮引数 p_i すべてに対して，ラベル $C.m.p_i$ のノードを作成する．
  - $C.m$ のオブジェクト型のローカル変数 v_i すべてに対して，ラベル $C.m.v_i$ のノードを作成する．
  - $C.m$ が静的メソッドではない場合，擬似変数 this に対応する，ラベル $C.m.this$ のノードを作成する．
  - $C.m$ がオブジェクト型の戻り値を持つ場合，ラベル $C.m.return$ のノードを作成する．

そして，エッジを以下の手順で作成する．

- 代入文 $lhs = rhs$ が存在し，lhs と rhs が共に変数やフィールド参照または配列参照である場合，rhs 中の変数を表すノードから，lhs 中の変数を表すノードへの有向エッジを追加する．
- 代入文 $lhs = o.m(a_1, a_2, ..., a_n)$，または，メソッド呼び出し文 $o.m(a_1, a_2, ..., a_n)$ が存在し，このメソッド呼び出しが $C.m$ と対応するとする．この時，保守的なコールグラフ上で，$C.m$ によって呼び出されうるすべてのメソッド $C'.m'$ に対して，以下の手順でエッジを追加する．
  - o に対応するノードから $C'.m'.this$ に対応するノードへのエッジを追加する．
  - $C'.m'$ の戻り値が void でない場合，$C'.m'.return$ から lhs に対応するノードへのエッジを追加する．
  - オブジェクト型の実引数 a_j に対しては，a_j に対応するノードから $C'.m'$ の a_j と対応する仮引数のノードへとエッジを追加する．

図 1b は，図 1a のソースコードから TPG を生成した例である．各変数に対応するノードが作成され，代入文に併せて有向エッジが作成されている．

2.3.2 型伝播

TPG の作成後，各ノードの *ReachingTypes* を求める．まず，全ノードに空集合の *ReachingTypes* を割り当てる．

そして，$lhs = newA();$ または，$lhs = newA[n];$ というアロケーションを行う代入文に対して，lhs に対応する変数のノードの *ReachingTypes* に A を追加する．次に，解析対象外のメソッドの呼び出し文 $lhs = o.m(a_1, a_2, ..., a_n)$ については，m の宣言時の戻り値の型のサブクラスすべてを lhs の *ReachingTypes* に追加する．同様に，解析対象外のフィールドを用いた代入文 $lhs = o.f$ については，f のサブクラスすべてを lhs の *ReachingTypes* に追加する．図 1c は，図 1b に，*ReachingTypes* を割り当てたもので，図 1a 中の変数へのアロケーションを行っている文に併せて *ReachingTypes* を追加している．

次に，TPG 上の強連結成分を検索し，強連結成分中のノードを 1 つのスーパーノードに変換する．スーパーノードの *ReachingTypes* は強連結成分中のノードの *ReachingTypes* の和集合とし，以後スーパーノード中のノードの *ReachingTypes* は，スーパーノードの *ReachingTypes* とする．なお，強連結成分の折りたたみを行った後の TPG は，無閉路有向グラフ（DAG）となる．図 1c は，図 1b に対して強連結成分の折りたたみを行った例である．図中の b3 と a3 は強連結成分であり，この 2 つのノードをスーパーノードへと変換している．

最後に，DAG に変換した TPG のノード間でトポロジカルソートを実行し，*ReachingTypes* の伝播を行う．具体的には，各エッジ $n_a \rightarrow n_b$ に対して，n_b の *ReachingTypes* を n_a の *ReachingTypes* との和集合とする．図 1e は，図 1d の TPG に対し *ReachingTypes* の伝播を行った例である．この結果，b1 の *ReachingTypes* は，b2 と c の *ReachingTypes* の和集合となり，{B,C} となっている．また，a3 と b3 のスー

パーノードには，a1 から型の伝播が行われ，$ReachingTypes$ は {A} となる．

3 提案手法

提案手法は，与えられた Java プログラムにおいて，変数 v_a に格納されるオブジェクトを特定の型 C_{fixed} と仮定した場合の仮想メソッド呼び出しの解決を行う．提案手法は次の 3 つのステップで構成される．

ステップ 1 プログラム全体に対して VTA を行い，TPG を作成する．

ステップ 2 v_a に依存して型が決まる変数の集合を求める．

ステップ 3 v_a がレシーバオブジェクトとなるメソッド呼び出しを特定し，対応する TPG 上のエッジを除去する．そして，型伝播を行い $ReachingTypes$ を求める．結果求まる v の $ReachingTypes$ が $runtime_types(v)$ である．

以降，ステップ 2 とステップ 3 について解説する．

3.1 ステップ 2: 指定した変数に依存して型が決まる変数の特定

ステップ 2 では，変数 v_a の型に依存して型が決まる変数，すなわち，変数 v_a 以外に型が伝播されることが無い変数の集合を求める。具体的には，v_a に対応するノードを n_a としたとき、次の 2 つの手順で求まるノード集合 N_d に対応する変数の集合を指す．

手順 1 $N_d \leftarrow \{n_a\}$ とする．

手順 2 $N_d \leftarrow N_d \cup \{n | p(n) \subseteq N_d\}$ を N_d が変化しなくなるまで行う．ただし、$p(n)$ はノード n に対する有効辺 $n' \rightarrow n$ を持つノード n' の集合である．

図 2a のプログラムに対して作成した TPG の一部が図 2b である．ここで，クラス M のフィールド f の型を B に固定したとすると，$N_d = \{M.f, M.g.j\}$ であり，$M.g.j$ の型も B に固定される．

3.2 ステップ 3: TPG の加工と型伝播

$v \in N_d$ となる仮想メソッド呼び出し $v.m$ は，v_a の型に依存しているため，$v.m$ で呼ばれるメソッドは，C_{fixed} のクラス階層を探索して得られる，m と同じシグネチャを持つメソッド m_{fixed_target} と特定でき，他のメソッドは呼ばれないことが分かる．以上の結果からコールグラフが変化し，メソッド呼び出しに対応していたエッジを TPG から除去することができる．最後に，型伝播を行い $ReachingTypes$ を求める．

エッジの除去は，以下のように行う．

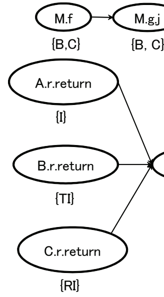
- v ($\in N_d$) がレシーバオブジェクトとなる仮想メソッド呼び出し $v.m(a_1, a_2, ...a_n)$ について，
  - 呼び出されるメソッドの候補であった，$C_{fixed}.m$ 以外の $C.m$ に対して，
    - v に対応するノードから $C.m.this$ に対応するノードへのエッジを削除．
    - 実引数 a_j に対応するノードから，仮引数 $C.m.f_j$ に対応するノードへのエッジを削除
    - $C.m.return$ から，戻り値を受け取る変数に対応するノードへのエッジを削除

そして，エッジ除去後の TPG を用いて型伝播を行い，全変数の $ReachingTypes$ を求める．ただし，N_d に属する変数の $ReachingTypes$ は C_{fixed} のみとする．

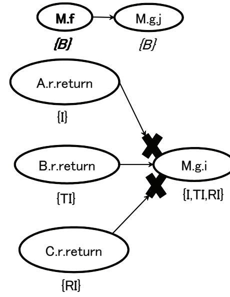
図 2b の例で，型を仮定し N_d を求めたが，この結果 $M.g.j$ がレシーバオブジェクトとなる 18 行目のメソッド呼び出し $i.r()$ で呼び出されるメソッドは，$B.r()$ のみとなる．よって図 2c に示すとおり，$i.r()$ の戻り値によるエッジである $A.r.return \rightarrow M.g.i$ と $C.r.return \rightarrow M.g.i$ を除去する．そして，図 2d が型伝播を行った結果であり，$M.g.$i の $ReachingTypes$ は TI のみとなる．この結果，19 行目の仮想メソッド呼び出し $i.p()$ の解決結果は，$TI.p()$ のみとなる．

```
 1: class A {
 2:         I r() { return new I(); } }
 3: class B extends A {
 4:         I r() { return new TI(); } }
 5: class C extends A {
 6:         I r() { return  new RI(); } }
 7:
 8:  class IA { void p() { ... }; }
 9:  class TI extends I { void p() { ... } }
10: class RI extends I { void p() { ... } }
11:
12: class M {
13:         A f;
14:         void setB() { this.f = new B(); }
15:         void setC() { this.f = new C(); }
16:         void g() {
17:                 A j = f;
18:                 I i = j.r();
19:                 i.p();
20:         }
21: }
```

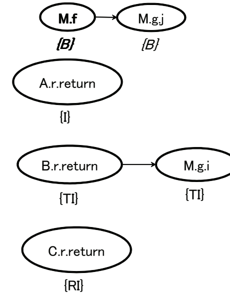
(a) プログラム



(b) 初期の TPG



(c) 変化後の TPG



(d) 再度の型伝播後

図 2: 提案手法の例

表 1: 実験対象のプログラム

プログラム名	クラス数	メソッド数	仮想メソッド呼び出し数
ANTLR	1,201	11,971	14,616
Apache-Ant（ANT）	1,286	11,549	15,972
Apache-Tomcat（Tomcat）	2,681	28,462	36,505
Batik	5,236	41,557	49,825
JEdit	1,132	7,998	17,516
和歌山大学教務システム（KS）	4,446	41,106	46,848
Torque	2,589	24,975	25,267

4 実験

提案手法を実装し，効果を調査するため評価実験を行った．実験にあたって以下のリサーチクエスチョン（RQ）を設定した．

RQ1 格納されるされるオブジェクトの型を仮定することで，仮想メソッド呼び出しの解決に影響を与える可能性のある変数はいくつあるのか

RQ2 変数が仮想メソッド呼び出しの解決に影響を与えている場合，1 つの変数あたりいくつのメソッド呼び出しに影響を与えているのか

RQ3 提案手法を用いた結果，既存手法と比べてコールグラフはどのように変化するのか

実験対象としたのは表 1 に示す 7 個の Java プログラムである．表 1 に，各プログラムのクラス数，メソッド数，仮想メソッド呼び出し数も併せて示している．

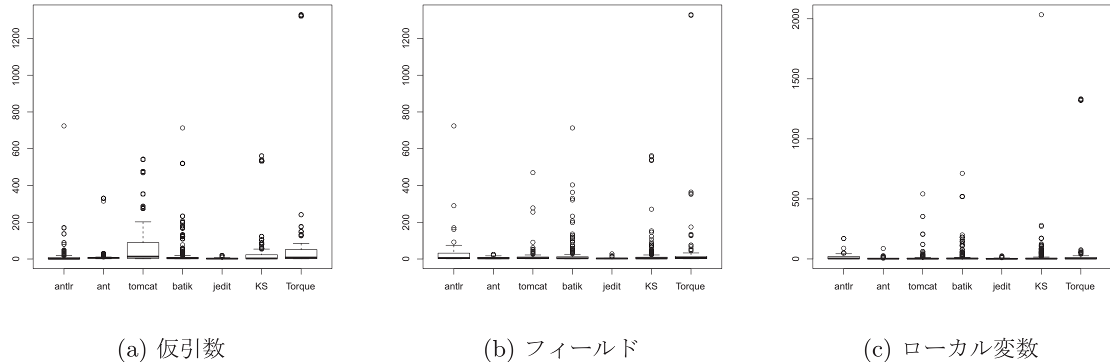
調査対象とした変数は，仮引数（レシーバオブジェクトも含む），フィールド，そしてローカル変数の 3 種類である．それらの変数を対象に以下の調査を行った．

1. $ReachingTypes$ が複数ある変数を数える．(RQ1 に対応)
2. $ReachingTypes$ が型を複数含む変数を対象に，その変数に依存して型が決まる変数の集合 N_d を特定し，N_d に含まれる変数をレシーバオブジェクトとする仮想メソッド呼び出しを特定する．その仮想メソッド呼び出しについて，メソッド呼び出しの候補数を調査し，複数の候補が見つかった仮想メソッド呼び出しを数え上げる．また，そのような仮想メソッド呼び出しが見つかる変数についても数え上げる．(RQ1，RQ2 に対応)
3. メソッド呼び出しの候補数が複数ある仮想メソッド呼び出しが見つかった場合，変数 v のオブジェクトの型を $ReachingTypes$ に含まれる型の 1 つと仮定し，提案手法を用いて仮想メソッド呼び出しの解決を行い，元のコールグラフから除去されたノードとエッジを数え上げる．(RQ3 に対応)

数え上げを行う際のコールグラフのノードとエッジは次のように定義する．ノードはメソッドに対応するものとし，他のメソッドを呼び出している，または，他のメソッドから呼び出されているものを数え上げる．エッジはメソッド呼び出しに対

表 2: 仮想メソッド呼び出しに影響を与える変数

	仮引数			フィールド			ローカル変数		
	総数	複数の型	影響有り	総数	複数の型	影響有り	総数	複数の型	影響有り
ANTLR	19,487	2,731 (14%)	680 (3.5%)	3,552	121 (3.4%)	42 (1.2%)	14,794	386 (2.6%)	133 (0.9%)
Ant	18,496	2,420 (13%)	873 (4.7%)	4,317	155 (3.6%)	64 (1.5%)	9,030	345 (3.8%)	165 (1.8%)
Tomcat	46,793	7,700 (16%)	2,077 (4.4%)	8,155	348 (4.3%)	136 (1.7%)	24,425	1,054 (4.3%)	272 (1.1%)
Batik	69,093	11,997 (17%)	2,077 (3.0%)	21,284	933 (4.4%)	297 (1.4%)	30,567	1,829 (6.0%)	835 (2.7%)
JEdit	12,907	1,331 (10%)	163 (1.3%)	2,670	137 (5.1%)	45 (1.7%)	6,698	192 (2.9%)	59 (0.9%)
KS	71,253	9,715 (14%)	2,158 (3.0%)	10,481	873 (8.3%)	542 (5.2%)	26,408	1,082 (4.1%)	427 (1.6%)
Torque	40,964	4,841(12%)	756 (1.8%)	5,710	345 (6.0%)	145 (2.5%)	14,731	747 (5.1%)	211 (1.4%)



(a) 仮引数　(b) フィールド　(c) ローカル変数

図 3: 影響を受ける仮想メソッド呼び出し数の分布

応するものとし，呼び出し文ごとに個別にメソッド間にエッジを引くものとする．メソッド m のある呼び出し文で，メソッド g を呼び出している場合，m に対応するノードから g に対応するノードにエッジを引く．複数の呼び出し文が存在する場合は，文の数だけエッジを引く．

4.1 実験結果

4.1.1 RQ1 に対する実験結果と回答

RQ1 に対応した調査を行った結果を表 2 に示す．表 2 には，仮引数，フィールド，ローカル変数のそれぞれについて，オブジェクト型の変数の総数と，そのうち $ReachingTypes$ に複数の型が含まれるものの数（「複数の型」列），それらの変数のうち仮想メソッド呼び出しに使用され，かつ，その仮想メソッド呼び出しに複数のメソッド呼び出し候補が存在するものの数（「影響有り」列）を示している．なお，表中の括弧内の数値は，総数に対する割合を示している．

表 2 より，複数の型が $ReachingTypes$ に含まれる変数は，全体に比して仮引数の場合は 10%から 17%，フィールドやローカル変数の場合は 2.6%から 8.3%であった．この結果から，複数の型を取りうる変数は，割合としては全体に比して少数であることがわかる．

また，複数の型が含まれ，さらにメソッド呼び出しに影響を与えるような変数は全体に比して非常に少数であり，仮引数とフィールドでは 1 つを除きすべて 5%を下回り，ローカル変数の場合はすべて 3%以下という割合になっている．

以上から型の仮定によりコールグラフが変化する可能性のある変数は，全体に比した場合には非常に少数であることが分かる．

4.1.2 RQ2 に対する実験結果と回答

次に RQ2 に対応する調査を行った結果を図 3 と表 3 に示す．図 3 は，変数の型の固定により影響を受ける仮想メソッド呼び出し数の分布を箱ひげ図で示している．表 3 は，図 3 で示した値の中央値と平均値を示している．

1 つの変数の型を固定した場合に，呼び出される可能性のあるメソッドの候補が変

表 3: 影響を受ける仮想メソッド呼び出し数の平均値と中央値

	仮引数		フィールド		ローカル変数	
	中央値	平均値	中央値	平均値	中央値	平均値
ANTLR	1	9.323	5.5	44.93	4	15.83
Ant	6	11.41	3.5	5.906	2	3.564
Tomcat	13	93.8	5	16.47	2	12.86
Batik	4	19.15	4	20.15	3	18.48
JEdit	2	3.92	3	4.089	1	4.051
KS	3	85.41	3	15.32	2	19.39
Torque	8	51	6	75.48	3	171.7

表 4: 既存手法によるコールグラフ

	クラス階層解析		VTA	
	ノード数	エッジ数	ノード数	エッジ数
ANTLR	10,101	62,463	8,057	38,029
Ant	9,246	32,263	7,751	19,109
Tomcat	22,701	131,544	18,572	60,005
Batik	31,809	207,412	25,928	78,550
JEdit	7,151	23,816	6,463	18,195
KS	31,959	123,136	25,973	74,381
Torque	18,850	77,045	14,249	32,926

表 5: ノード減少量とエッジ減少量の平均値と中央値

	仮引数				フィールド				ローカル変数			
	除去ノード数		除去エッジ数		除去ノード数		除去エッジ数		除去ノード数		除去エッジ数	
	中央値	平均値	中央値	平均値	中央値	平均値	中央値	平均値	中央値	平均値	中央値	平均値
ANTLR	1	43.91	53	526.9	13	133	840	1666	13	140.8	592	1500.9
Ant	3	146.3	69	1044	2	38.29	13	268	1	25.58	11	181.2
Tomcat	31	385.1	548	1954	3	531.3	50.5	2386.8	1	309.5	27	1429
Batik	7	464.8	105	2647	5	290.5	44	1587	1	493.2	21	2622
JEdit	0	41.84	8	225.1	6	14.77	42	51.69	1	16.33	6	77.22
KS	2	326.63	48	1880	2	355.9	30	2067.8	11	529	75	3073
Torque	14	509.4	168	1830	3	245.1	42	1007	2	472.7	18	1575

化するような，仮想メソッド呼び出しの数は，図 2 に示されるの分布の形状と，表 3 に示している中央値の値から，多くの場合数個であることが分かる．

ただし，この値は非常に分散が大きく，例えば Torque の仮引数では，最大値として 1200 個以上の仮想メソッド呼び出しに影響を与える変数が存在することが図 3a から分かる．そのため，適切な変数を選択することで，多くの仮想メソッド呼び出しに対して影響を与えることができると考えられる．

4.1.3 RQ3 に対する実験結果と回答

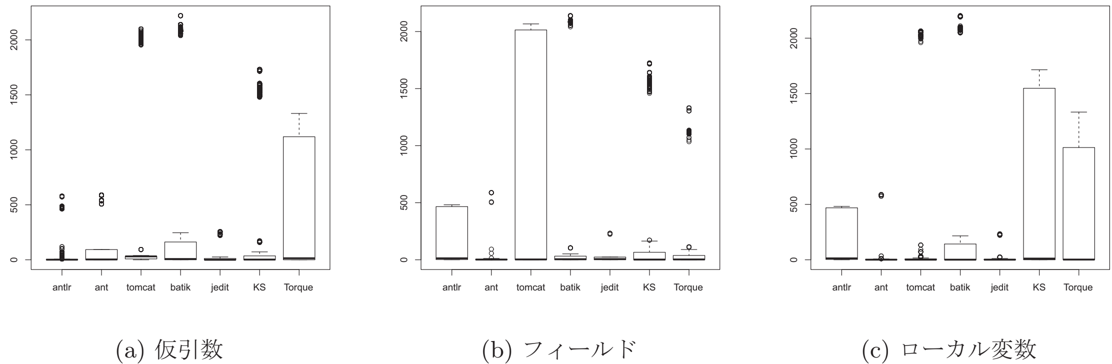
最後に，RQ3 に対応する調査を行った結果を示す．まず，表 4 に，クラス階層解析を用いて作成したコールグラフと，VTA を用いて作成したコールグラフについて，ノード数とエッジ数を示す．次に，図 4 と図 5 に，メソッド呼び出しに影響を与えるような変数を対象に 1 つの型に固定した場合の，VTA のコールグラフと比較した場合のノードとエッジの減少量の分布についてそれぞれ示す．また，表 5 に，図 4 と図 5 に示した値の中央値と平均値を示す．

ある 1 つの変数の型をある 1 つの型に固定し，かつ，メソッド呼び出し候補を減少させることができた場合，コールグラフのサイズを減少させることができた．特にエッジ数については，JEdit の場合以外で，多くの場合数十本以上のエッジを除去できたことが表 5 の除去エッジ数の中央値から分かる．また，図 4 と図 5 からコールグラフのサイズが大きく減少させるような場合があることが分かり，そのような変数に対して型の指定を行った場合，提案手法は有効に働くと考えられる．

4.2 妥当性への脅威

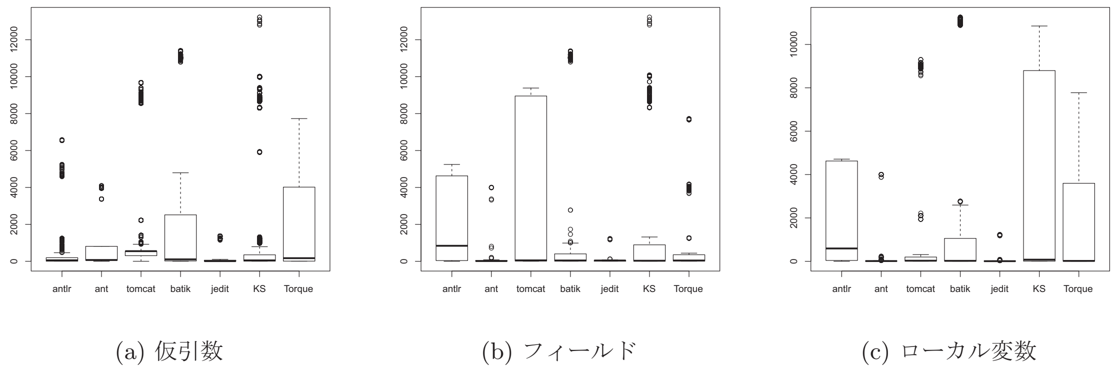
今回実験対象にしたプログラムは実用的なサイズのプログラムであり，実験対象の規模については妥当であると考えられる．ただし，実験対象としたプログラムの個数は，高々7 個であり，対象のプログラムを増やすことで結論が変わる可能性がある．

また，今回対象にしたプログラムは完全なプログラムではなく，JDK などを一部解析対象外として VTA を実行している．そのため，完全なプログラムを実験対象にした場合には，今回の実験結果が変わる可能性がある．



(a) 仮引数　(b) フィールド　(c) ローカル変数

図 4: ノード減少量の分布



(a) 仮引数　(b) フィールド　(c) ローカル変数

図 5: エッジ減少量の分布

仮想メソッド呼び出しにおけるメソッド呼び出し候補を特定する手法として，ポインタ解析 [7] [8] [9] を用いる方法もある．Fluid Source Code Views や Code Bubbles では，インライン展開を行う際にメソッドを呼び出すメソッドが自明に求まることを利用し，ポインタ解析のうち文脈を考慮した解析を用いることで，メソッド呼び出し候補をより正確に求めることができる可能性がある．しかし，Laurie ら [10] の報告によれば，文脈を考慮したポインタ解析は，文脈非依存なポインタ解析と比べ，メソッド呼び出しの候補があまり減少しないことが報告されている．VTA はフロー非依存かつ文脈非依存なポインタ解析と原理的によく似た手法であり，文脈を考慮した手法を用いても，VTA に比べて精度が大きく向上することはあまりないと考えられる．そのため，本実験ではポインタ解析との比較を行わなかった．

5　関連研究

ある変数の型を指定した際に，その変数に依存して型が決まる変数を求める別の方法として，エイリアス解析 [8] [11] を用いる手法が考えられる．エイリアス解析では，ある変数と同じオブジェクトを参照する変数を求めることができる．そのため TPG を使わなくとも，エイリアス解析により指定した変数と同一のオブジェクトを指す変数の集合を得ることができると考えられる．

しかし，エイリアス解析は一般に解析コストが高く，大規模なプログラムに適用するのは難しい．そのため，提案手法では TPG を用いることにした．

6 まとめと今後の課題

本研究では，注目する変数が参照するオブジェクトの型を仮定した状態での仮想メソッド呼び出しの解決手法を提案した．実験では，複数の型のオブジェクトを参照する変数，型を仮定したときに影響を受ける可能性のある仮想メソッド呼び出し，そして，実際に型を仮定したときのコールグラフの変化について調査を行った．実験の結果，複数の型を取りうるような変数自体は全体に比して少数であるが，型を 1 つに固定した場合にコールグラフを大きく減少させる場合があることを確認した．

今後の課題としては，実験対象の規模の拡大や，型を固定する変数を 1 つではなく複数にした場合のコールグラフの変化の調査が挙げられる．

謝辞 本研究は，文部科学省科学研究費補助金若手研究 (A)(課題番号:23680001) の助成を得た．

参考文献

[1] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pp. 77–101, 1995.

[2] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pp. 324–341, 1996.

[3] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 264–280, 2000.

[4] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on*, pp. 260 –263, 2006.

[5] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pp. 455–464, 2010.

[6] 大場光明, 渥美紀寿, 小林隆志, 阿草清滋. メソッド境界を越えた呼び出しパターン抽出のためのコールグラフ探索戦略. Technical Report 18, 名古屋大学大学院情報科学研究科, 2012.

[7] Ana Milanova. Light context-sensitive points-to analysis for java. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 25–30, 2007.

[8] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pp. 131–144, 2004.

[9] Jianwen Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pp. 150–157, 2002.

[10] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: Is it worth it? In *Compiler Construction*, Vol. 3923 of *Lecture Notes in Computer Science*, pp. 47–64. 2006.

[11] Dacong Yan, Guoqing Xu, and Atanas Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 155–165, 2011.