Open な Java Program に対する Object Flow 解析

千代 英一郎[†1]

本論文では，Java program における object の流れ（object の生成点から各使用点に至る過程）の表現方法およびその解析方法を提案する．提案する表現方法は，program の動作の理解に役立つほか，program の脆弱性や信頼性の検証において，検出した問題点の発生過程を人間に示す手段として有用である．我々の方法の特徴は，object を指示する存在の変化に着目し，object の流れを label 付き状態遷移系上の遷移系列として定式化していることで，これにより object が流れる様々な要因（変数間の代入，他の object の field への書き込みと読み出し，method 呼び出しや復帰等）を統一的に表現することが可能となる．また program とその外部環境の間の object の流れも自然な形で表すことができる．提案する解析は，open program の置かれる任意の文脈の作用について穏健な近似を行う機能を持ち，着目する object の生成点が含まれている場合，健全な解析結果を得るために stub を必要としない．本論文では，open program に対する解析を closed program に対する解析の拡張として定義し，その相対的な健全性を示している．

Object Flow Analysis for Open Java Programs

Eiichiro Chishiro[†1]

In this paper, we propose a new representation of object flows in Java, which are propagation sequences from a statement creating an object to statements using it. This representation can help us understand program's dynamic behaviors, and serve as a convenient means for representing error traces in program verification. The idea is that we formalize object flows as transition sequences on a labeled transition system by focusing on the change of entities which point to objects. This approach enables us to represent uniformly all causes of object flow including variable assignments, field load/store, and method invocation/return. We can also represent object flow between a program and its external environment in the same way. Our analysis can safely approximate effects of all possible contexts in which an open program being analyzed will be put, and needs no stub to get a sound solution if the object focused is created in the open program. We formalize our analysis as an extension of that for closed programs and show the soundness under the assumption that the latter is sound.

1. はじめに

本論文では，Java program における object の流れ（object の生成点から各使用点に至る過程）の表現方法およびその解析方法を提案する．object は Java のような object 指向言語における data の基本単位であり，その流れに関する情報は program の諸性質を調べるうえで重要である．

1.1 object の流れとは

「object の流れ」という概念は一般的なものではないが，Java program に対する様々な dataflow 解析は，縮退した形での object の流れの解析と見なすことができる．

たとえば，代表的な dataflow 解析の一つに pointer 解析[29] がある．pointer 解析は program 中の pointer の指示先（Java の場合，object の集合）を求めるもので，その結果は compiler が行う最適化等に利用される．ここで，「pointer の指示先を求める」という pointer 主体の定義を，指示される object の側を主体としてとらえなおしてみると，pointer 解析は program 内で生成される object の指示**元**（pointer の集合）を求めるものと考えることができる．すなわち，pointer 解析は，program 内で生成される各 object（の参照値）が，その生成点からその（参照値の）使用点である pointer に至る object の流れの解析として定義できる．

また，escape 解析[6] は，method 内で生成され，method 外部で参照される object の集合を求める解析であり，同期の除去や garbage collection の回数削減

†1 日立製作所システム開発研究所
Systems Development Laboratory, Hitachi, Ltd.

等に利用される．ここで，method 内で生成した object が method 外部で参照される条件を，その object が method 内における何らかの流出点（たとえば method 呼び出しの実引数）で使用されていることであると考えれば，escape 解析も object の流れの解析として定義することができる．

そのほか，pointer 解析に基づく class 解析[5]，Java card applet 間の information flow 解析[11]，web application の脆弱性を調べる taint 解析[20] 等も同様に object の流れの解析と見なすことができる．

これらの解析はいずれも，なんらかの条件を満たす object の流れが存在するかどうかのみに関心があり，その過程（経路）自体は問題としない．これに対し，本論文では，object がどういう過程を経て生成点から各使用点に至るかを，人間が理解できる形で得ることを目的とする．得られた過程情報は，program の動作の理解に役立つほか，program の脆弱性や信頼性の検証において，検出した問題点の発生過程を人間に示す手段として有用である．

たとえば**図 1** の program では，「文 6 で生成された `String` object が，文 8 で配列 `xs` に格納され，文 9 で method `put` の呼び出しにより `b` の指示先 object の field `str` に代入され，文 14 で method `get` の呼び出しにより取り出され，文 15 で field `f` に代入される」という object の流れが存在する．これに対し，この program に対する pointer 解析では「文 6 で生成された `String` object が，文 15 の field `f` の参照点に到達する」ことは分かるが，その過程に関する情報は得られない．

詳しくは 6 章で述べるが，program の脆弱性を解析する際に，脆弱性の存在問題を object の流れの存在問題に帰着できることがある．このとき，object の流れが存在することしか分からないと，それが真の脆弱性を表している（解析精度の低さからくる false positive ではない）かどうかを調べたり，その脆弱性を修正したりするのが難しい．このような場合，object の流れに関する詳細な過程情報は事実上不可欠である．

以降では，このような過程情報を含んだ object の流れを object flow と呼び，その解析を object flow 解析と呼ぶ．

1.2 問　　題

object flow を求める際には，大きく 2 つの問題が存在する．1 つ目は，object flow の表現方法である．難しいのは，object の流れる要因が変数間の代入，他の object の field への書き込みと読み出し，method 呼び出しや復帰等多岐にわたることである．3 章では，

```
 1: public class A
 2: {
 3:   static String f;
 4:   public static void main(String[] a) {
 5:     B b = new B();
 6:     String x = new String();
 7:     String[] xs = new String[10];
 8:     xs[1] = x;
 9:     b.put(xs[1]);
10:     m2(b);
11:   }
12:
13:   static void m2(B b) {
14:     String y = b.get();
15:     f = y;
16:   }
17: }
18:
19: class B
20: {
21:   String str;
22:   void put(String s) { str = s; }
23:   String get() { return str; }
24: }
```

図 1 Program 例
Fig. 1 Example program.

従来手法について述べた後，これらを label 付き状態遷移系上の遷移系列として統一的に定式化する方法を示す．

2 つ目は，open program の扱いである．open program とは，program の実行に関わる class（および method）定義の一部が存在しない program のことである．これに対し，参照されている class や method の定義がすべて含まれている program を closed program と呼ぶ．

たとえば，外部の class library を利用する program は，単体では open program である．また，HTTP servlet のように，外部の framework 上で動作する program も open program である．現実に解析対象として与えられる program は，open program であることが多く，これは object flow を単純な program の記号的な実行によって得ることを困難にしている．

open program を解析するための一般的は方法は次の 2 つである．

1 つ目は，open program に欠けている class 定義等を補うための stub を用意し，open program を閉じる（closed program にする）方法である．これに

より，解析は対象が closed program であることを前提とすることができる．

対象を closed program に限定した解析は，より単純で効率的な実現が可能であり，解析精度も高くなるという利点を持つ．一方で，人手で stub を用意するのは手間がかかるうえ，誤りの入りやすい作業である．また，標準 library 等既存の class 定義を stub として利用する場合，解析対象となる program の規模が大きくなり解析効率が低下するという問題が生じる．

open program を解析する 2 つ目の方法は，欠けている部分の作用を近似することである．多くの場合，近似は解析結果が健全になるような方向に行う．ここで健全な解析結果とは，実際に起こりうる object flow をすべて含むような解析結果のことである．object flow の結果を program の信頼性検証等に利用する場合，健全でない解析結果は，重要な不具合を看過してしまう可能性を持つ．また compiler が最適化のために method 単位で行う dataflow 解析では，健全でない結果は不正な最適化の適用につながりうる．

近似による解析は，stub を用いる場合に比べて複雑になり，効率も低下する傾向があるが，前述した stub の持つ欠点は存在しない．一方，解析精度をなるべく低下させないようにしながら，つねに健全な結果を導くような解析の設計は容易ではない．設計時には，open program の置かれるすべての文脈を想定する必要があるが，ad hoc な方法では corner case を見過ごす可能性が高い．

本論文では近似による open program の解析方法を示す．4 章で示す近似を用いることで，着目する object が program 内部で生成されている場合には，stub を用いなくても健全な解析結果を得ることができる．program 外部で生成される object に関する object flow を求める場合には，その生成点を含む method 定義を stub として与えることで，同様に健全な結果が得られる．4.3 節では，近似の正しさを，3 章で与える closed program に対する解析が健全である，という前提のもとで証明する．

近似による解析では，実際には起こりえない object flow が多数検出されうる．5 章では，型情報を用いることで，解析精度を改善する方法を示す．型情報を用いた精度改善は，closed program に対する解析では一般的だが，open program に対する解析の場合，不完全な型情報をどのように補完するかが問題となる．ここでは，つねに健全な補完方法，および特定の前提のもとで健全な補完方法を示す．

1.3 論文の構成

本論文の構成は以下のとおりである．2 章では本論文で用いる program model および deductive system の定義を与える．3 章では提案する object flow の表現方法について述べた後，closed program に対する object flow 解析を示す．4 章では 3 章の解析に program 外部を近似する機能を加えることで，open program に対する object flow 解析を構成する．5 章では型情報に基づき解析精度を向上させる方法を示す．6 章では object flow 解析の応用について述べる．7 章では object flow 解析を実装し，初期評価した結果を示す．8 章では関連研究について述べる．9 章は結論である．

2. 準　　備

本章では，本論文で示すすべての解析が対象とする program model の定義について述べる．本論文で用いる program model は，解析対象 program を predicate の集合として model 化したもので，形式的な扱いが容易である．また，このような program model に対する解析の定式化方法について述べる．本論文で示すすべての解析は deductive system により定式化される．

2.1 Program Model

本論文では，解析対象となる Java program を変数を含まない atomic な predicate（relation）の集合として model 化する．Java program に対する program model の定義を**図 2** および**図 3** に示す．以降ではこれを単に program model と呼ぶ．

program model は Java program を直接 model 化したものではなく，それを compile することで生成される classfile を model 化したものであり，Java の classfile 仕様[19] と密接に関連している．

program model $mprog$ は class model $mclass$ の集合であり，class model $mclass$ は $pred$ の集合である．$pred$ は class の持つ情報を表す predicate である．

各 class は ClassName cn によって識別される．$class(cn, a)$ は class cn の access flags が a であることを表す predicate である．access flags は，Java の classfile 仕様中で定義されている同名の属性に対応するもので，class の access 権限等の情報を表す．$subtyped(cn_1, cn_2)$ は，class cn_1 が cn_2 を直接継承（もしくは実装）していることを表す．

$method(m, cn, mn, dn, a)$ は，class cn で method m が定義され，その名前が mn，型が dn，access flags が a であることを表す．m は，各 method 定義を一意に識別するために導入される識別子であり，関係

$$
\begin{array}{lcl}
mprog & ::= & mclass^* \\
mclass & ::= & pred^* \\
pred & ::= & class(cn, a) \\
& | & subtyped(cn_1, cn_2) \\
& | & method(m, cn, mn, dn, a) \\
& | & codeindex(i, m, p) \\
& | & assign(i, v_1, v_2) \\
& | & new(i, v, h) \\
& | & getfield(i, v_1, fn, v_2) \\
& | & putfield(i, v_1, fn, v_2) \\
& | & getstatic(i, cn, fn, v) \\
& | & putstatic(i, cn, fn, v) \\
& | & callstatic(i, cn, mn, dn) \\
& | & calldynamic(i, cn, mn, dn) \\
& | & formal(m, k, v) \\
& | & actual(i, k, v) \\
& | & incoming(i, v) \\
& | & outgoing(m, p, v) \\
& | & tyv(v, cn) \\
& | & tyh(h, cn) \\
& | & refclass(cn)
\end{array}
$$

図 2 Java program model の predicate
Fig. 2 Predicates of Java program model.

$$
\begin{array}{lcl}
cn & \in & \text{ClassName} \\
fn & \in & \text{FieldName} \\
mn & \in & \text{MethodName} \\
dn & \in & \text{MethodDescriptorName} \\
m & \in & \text{MethodIdentifier} \\
v & \in & \text{VariableIdentifier} \\
h & \in & \text{ObjectIdentifier} \\
i & \in & \text{ProgramPoint} \\
p & \in & \text{ByteOffset} \\
k & \in & \text{Nat} \\
a & \in & \text{AccessFlags}
\end{array}
$$

図 3 Java program model の domain
Fig. 3 Domains of Java program model.

database における primary key の役割を持ち，他の predicate からの参照に利用される．

$codeindex(i, m, p)$ 以降は，各 method 定義の本体（code 属性）を表す predicate である．

$codeindex(i, m, p)$ は，program point i が method 定義 m の byte offset p に対応することを表す．対象とする classfile が行番号 table を含む場合には，byte offset p から，対応する source program の行番号を得ることができる．

$assign(i, v_1, v_2)$ は，i における，変数 v_2 から v_1 への代入 $v_1 = v_2$ を表す．ここで変数 v は，operand stack 上の値もしくは local 変数を表す．また $new(i, v, h)$ は，i において，object h が生成され，v に代入されることを表す．h は生成点の異なる object を一意に識別するために導入される識別子である．

$getfield(i, v_1, fn, v_2)$ は，i における instance field fn の値の使用 $v_2 = v_1.fn$ を表す．$putfield(i, v_1, fn, v_2)$ は，i における instance field fn の値の定義 $v_1.fn = v_2$ を表す．同様に，$getstatic(i, cn, fn, v)$ および $putstatic(i, cn, fn, v)$ は，i における class field fn の値の使用 $v = cn.fn$ および値の定義 $cn.fn = v$ をそれぞれ表す．なお，本研究では配列要素の区別は行わず，すべての配列参照は仮想的な field `[]` に対する参照として取り扱う．

$callstatic(i, cn, mn, dn)$ は i における静的な method 呼び出し，$calldynamic(i, cn, mn, dn)$ は i における動的な method 呼び出しをそれぞれ表す．呼び出される method 名は mn，型（戻り値および引数の型）は dn である．cn は，$callstatic$ の場合，呼び出す method 定義の属する class を表す．$calldynamic$ の場合，method 呼び出しにおける receiver 変数の宣言型を表す．

$formal(m, k, v)$ は，method m の k 番目の仮引数が v であることを表す．$actual(i, k, v)$ は，i における method 呼び出しの k 番目の実引数が v であることを表す．引数の順序はつねに 1 から始まる．instance method の呼び出しの場合，第 1 引数は暗黙の `this` 引数に対応する．

$incoming(i, v)$ は，i における method 呼び出しの戻り値が v であることを表す．$outgoing(m, p, v)$ は method m の byte offset p における戻り値が v であることを表す．$outgoing$ の位置が program point でなく method および byte offset によって表されているのは，解析の記述を若干簡潔にするためである．

$tyv(v, cn)$ は変数 v の型が cn であることを，$tyh(h, cn)$ は object h の型が cn であることをそれぞれ表す．open program では，これらの型関係は 1 対 1 になるとは限らない．なお，program model では，object flow 解析に無関係な primitive 型の値を扱う命令等は捨象されている．そのため，すべての変数の型は cn によって表現することが可能である．

最後に $refclass(cn)$ は型 cn が program 内で参照されていることを表す．これは program model 中の他の predicate から導出可能であるが，後に解析を定

義するうえでの便宜のため，program model に加えておく．

後に利用するため，program model に関する用語をいくつか定義しておく．

定義 2.1（集合） program P に対し，P の program model を $M(P)$ で表す．$M(P)$ が含む変数 v の集合を $Var(P)$，生成 object h の集合を $Obj(P)$，program point i の集合を $PP(P)$，定義される method m の集合を $Method(P)$，参照される class cn の集合を $RefClass(P)$，定義される class cn の集合を $DefClass(P)$ で表す．□

2.2 Program Model の生成

2.1 節で定義した program model は，Java の classfile から生成することを想定して設計されている．本節では，classfile からの program model の生成方法について簡単に述べる．

classfile には，Java program における class が Java VM が処理可能な形式で表現されている．各 classfile には，そこで定義されている class の継承関係や定義されている field や method の名前，仮引数や戻り値の型情報，各種属性値が含まれており，そこから $class(cn, a)$ や $method(m, cn, mn, dn, a)$ 等の predicate を生成するのは容易である．

program model 生成の中心となるのは method 本体の model 化である．classfile では，各 method の本体は Java VM 命令（しばしば bytecode と呼ばれる）の列として表現されている．Java VM は，いわゆる stack machine であり，大半の演算命令は VM が持つ operand stack 上の値に対して適用され，その結果は operand stack 上に積まれる．また，method 呼び出しの際の実引数や method の戻り値の受け渡しも operand stack を介して行われる．各 method は，operand stack とは別に local 変数 table と呼ばれる領域を持ち，local 変数の値を保持することができる．

operand stack の model 化 method 本体を model 化する際には，operand stack を介した data の流れを図 2 の各種 predicate によって表現する必要がある．これに関しては様々な方法が可能であるが，実際に起こりうる data の流れがすべて含まれてさえいれば，どのような方法であっても解析の健全性という点で問題はない．

単純な方法は，operand stack の深さごとにその位置における値を表す新しい変数（たとえば $\mathtt{os}_0$，$\mathtt{os}_1$，...）を導入するというものである．これにより，operand stack を介した data の流れは変数間の代入となり，predicate *assign* を用いて表現できる．この方法は，実現は容易だが，operand stack の各領域は式の評価や関数呼び出しごとに再利用されるため，本来起こりえない data の流れが多数 model に含まれてしまうという問題がある．

より精度の高い方法は，operand stack に積まれる値ごとに fresh な（他で使用されていない）一時変数を割り当てていくものである．この方法では，制御の流れの合流点で operand stack 状態の併合を行う等の処理が必要となり，実現は複雑になるが，起こりえない data の流れが operand stack を介して生じる可能性は少なくなる．実現は，Java VM における bytecode の検証と同様，method の symbolic な実行に基づき行うことができる．我々は，文献 17) で Brisset-Coglio verification algorithm と呼ばれている方法と類似の方法を用いている．

なお，method 呼び出しにおける引数の値や戻り値の受け渡しは operand stack を介して行われるため，上記のように一時変数を導入して model 化を行う場合，各 method 呼び出しにおける実引数や戻り値を表しているのはどの変数なのかを model 上で表現する必要がある．これらは，method 呼び出し命令を処理する際に，適切な predicate *actual* および *incoming* を作成することで行う．また，method 復帰命令に対しても，その戻り値を表す変数を示すために predicate *outgoing* を作成する[*1]．なお，operand stack に積まれた実引数の値は，Java VM により呼び出し先 method の local 変数 table に copy される．各仮引数が local 変数 table のどの slot に対応するかは method の型定義に基づき一意に定まるため，method の仮引数を表す predicate *formal* の生成に method 本体の情報は不要である．

制御の流れ program point i_1 における命令の実行後，制御は program point i_2 に移る，というような method 内の制御の流れに関する情報は，program model に含まれていない．すなわち，本論文で示す解析はすべて flow-insensitive なものである．より高い精度を得るためには，文献 12) にあるように，SSA 形式への変換を行い，異なる値を持つ local 変数を区別するか，制御 flow を表す predicate $cflow(i_1, i_2)$ を導入し，解析を flow-sensitive なものに拡張する必要

[*1]「戻り値を表す変数」という用語が 2 種類の意味で用いられていることに注意されたい．呼び出し先 method においては，呼び出し元に返す値（これは operand stack に積まれる）に対して割り当てられた変数が「戻り値を表す変数」であり，呼び出し元 method においては，呼び出し後，operand stack に積まれている戻り値に対して割り当てられた変数が「戻り値を表す変数」である．

26 : `aload_1`	$assign(i_1, t_1, \texttt{b})$
27 : `aload_3`	$assign(i_2, t_2, \texttt{xs})$
28 : `iconst_1`	
29 : `aaload`	$getfield(i_3, t_2, [], t_3)$
30 : `invokevirtual#6`	$calldynamic(i_4, \texttt{B}, \texttt{put}, (\texttt{Ljava/lang/String;})\texttt{V})$
	$actual(i_4, 1, t_1)$
	$actual(i_4, 2, t_3)$

図 4 Java program model の例
Fig. 4 An example of Java program model.

がある．

以上が，program model 生成の概略である．本論文では具体的な生成方法については省略し，以降では P の program model $M(P)$ が P における data の流れをすべて含む健全な model であることのみを前提とする．

例 2.1 model 化の例として，図 1 の文 9 `b.put(xs[1])` に対応する Java VM 命令列，および各命令に対して生成される predicate を**図 4** に示す．

左段が Java VM 命令列，右段が各命令に対応する predicate である．左端の数字 26–30 は各命令の位置（method 先頭からの byte offset）を表している．なお，変数 `b` および `xs` は local 変数 table の slot 1 および 3 にそれぞれ割り当てられている．

26 の命令 `aload_1` は，method `put` の呼び出しの 1 番目の実引数として `b` の値（暗黙の `this` 引数）を operand stack に積んでいる．この data の流れは，新しく積まれた値を表す一時変数 t_1 を導入することにより，`b` から t_1 への代入 $assign(i_1, t_1, \texttt{b})$ として model 化されている．

27–29 の命令列は，method `put` の呼び出しの 2 番目の実引数として `xs[1]` の値を operand stack に積んでいる．27，28 で配列 `xs` および添字 `1` を operand stack に積んだ後，29 の `aaload` により，それらの値が operand stack から取り除かれ，代わりに `xs[1]` の値が積まれる．この data の流れは，predicate *assign* および *getfield* によって model 化されている．前述したとおり，program model では配列の個々の要素は区別しないため，添字部分は model 化されない（[] はすべての配列要素を代表する field である）．導入された一時変数 t_2 および t_3 は，operand stack 上の `xs` および `xs[1]` の値をそれぞれ表している．

30 の命令 `invokevirtual#6` は method `put` を呼び出している．これは，呼び出しを表す predicate *calldynamic*，および各実引数の値がどの変数によって表されているかを示す predicate *actual* によって model 化されている．ここでは，t_1 が 1 番目の実引数の値，t_3 が 2 番目の実引数の値をそれぞれ表している．なお `(Ljava/lang/String;)V` は method `put` の descriptor（Java VM が定める型の内部表現）である．

完全な program model には上記の predicate に加えて，各 predicate の生成位置を表す predicate *codeindex* や各変数の型を表す predicate *tyv* が含まれる．

なお，model 生成時に導入した一時変数の一部は，compiler における標準的な最適化手法である copy 伝播[1] を用いて除去することが可能である．たとえば，上記の例に対して，不要な一時変数を除去すると以下のようになる．

$getfield(i_3, \texttt{xs}, [], t_3)$
$calldynamic(i_4, \texttt{B}, \texttt{put}, (\texttt{Ljava/lang/String;})\texttt{V})$
$actual(i_4, 1, \texttt{b})$
$actual(i_4, 2, t_3)$

これにより，program 上に存在していた data の流れを保持したままで，より小さな model を得ることができる． □

2.3 Deductive System

本論文では，すべての解析を deductive system として定式化する．deductive system とは，以下で定義される *fact* の導出規則（*rule*）の集合である．

fact および *rule* は以下の形式を持つ論理式である．

$$\begin{array}{lcl} fact & ::= & p(a, \ldots, a) \\ literal & ::= & fact \mid \neg fact \\ rule & ::= & literal \wedge \ldots \wedge literal \Rightarrow fact \end{array}$$

ここで，p は predicate symbol，a は変数もしくは定数である．本論文では *rule* の集合として，以下の条件を満たすもののみを扱う．

- 各 *rule* の前提（⇒ の左側）中に現れる変数は必

ずその *rule* の結論（⇒ の右側）にも現れる．

- 層状（stratified）である[3]．

すなわち，本論文の deductive system が扱う論理式の class は，safe かつ stratified な datalog program[26] の class に一致する．

最初の条件は，*rule* により導出される *fact* が変数を含まないことを保証する．このような *fact* を *ground fact* と呼ぶ．2.1 節で定義した program model は *ground fact* の集合 $\mathcal{M}$ である．

2 つめの条件は，否定形の *fact* の意味の定義や計算を容易にする．*rule* $literal_1 \wedge \ldots \wedge literal_n \Rightarrow fact_0$ において，$fact_0$ 中の predicate symbol p は各 $literal_i$ 中の predicate symbol q_i に依存しているという．q_i が否定形（$\neg q_i$ の形）で用いられている場合を負の依存，そうでない場合を正の依存と呼ぶ．*rule* 集合に含まれるすべての predicate symbol を node とし，predicate symbol 間の依存を edge とする有向 graph を依存 graph と呼ぶ．edge の向きは，p が q に依存しているならば，q から p である．

rule 集合が層状であるとは，その依存 graph が負の依存 edge を含む閉路を持たないことである．これは，ある predicate symbol が負の依存を介して自分自身に依存することがないことを意味している．たとえば，*rule* 集合 $\{q(x) \Rightarrow p(x), \neg p(x) \Rightarrow q(x)\}$ は q が負の依存を介して自分自身に依存しており，層状でない．本論文における解析が層状であることは，その *rule* 集合に関する依存 graph を描くことで容易に確認できる．

依存 graph を強連結成分分解し，成分間に edge がある場合にその始点を含む成分が終点を含む成分より小さくなるように各成分を番号付けることができる．この番号を層（stratum）と呼ぶ．層状な *rule* 集合の依存 graph は負の依存を含む閉路を持たないため，負の依存 edge の始点は必ず終点とは別の成分に属し，かつその層は終点よりも小さい．

rule 集合は，各 *rule* の結論に含まれる predicate symbol の層に基づき，いくつかの group に分けることができる．先に述べた負の依存 edge の性質から，ある group g に属する *rule* の前提に predicate symbol p が否定形で含まれる場合，p を結論に含むすべての *rule* は g より小さな層の group に属することが分かる．

本論文では，上記の条件を満たす解析（*rule* 集合）および解析対象 program model $\mathcal{M}$ に対し，その解を $\mathcal{M}$ および *rule* 集合の標準 model（canonical model）[3] として定義する．標準 model の直感的な意味は，$\mathcal{M}$ および *rule* 集合から導出できるすべての *fact* の集合である．導出は前述した group ごとに，層の小さいものから行う．その際，現在より前の group に属する predicate symbol を含む *fact* のうち，それまでに導出されていないものについては，その否定形が導出されていると見なす．すなわち，この定義は閉世界仮説を自然に取り入れたものとなっている．前述した group の性質から，否定形が導出されていると見なした *fact* の肯定形が以降の group の処理時に導出されることはなく，矛盾は生じない．詳細については文献 3)，34) を参照されたい．

本論文では各 *rule* を以下の形式で表現する．

$$\frac{\mathcal{M} \vdash literal_1 \quad \mathcal{M} \vdash \ldots \quad \mathcal{M} \vdash literal_n}{\mathcal{M} \vdash fact_0}\ (RuleName)$$

RuleName は *rule* の名前である．$\mathcal{M} \vdash literal_i$ は judgement と呼ばれ，解析対象 $\mathcal{M}$ および解析（*rule* 集合）の解（標準 model）に $literal_i$ が含まれることを表す．なお，$\mathcal{M}$ 中の *fact* はつねに標準 model に含まれる．水平線の上部に現れる各 $literal_i$ は *rule* の前提，下部に現れる $fact_0$ は *rule* の結論である．これは「*rule* の前提が標準 model に含まれるならば，結論も標準 model に含まれる」という標準 model の定義に対応している．

3. Closed Program の解析

本章では，提案する object flow の定義について述べた後，closed program に対する object flow 解析を示す．open program に対する解析については，4 章で述べる．

3.1 Object Flow の定義

1 章で述べたとおり，本論文における object flow は，object の生成点から各使用点に至る過程のことである．だが，何を過程（情報）とし，それをどう表現するかについては検討が必要である．

たとえば，過程情報を object の生成点から使用点に至る実行 trace によって定義することが可能である．自動検証技術の 1 つである model 検査[9] では，program が仕様を満たさないことを示す反例は実行 trace の形式で与えられることが多い．実行 trace は object の流れとともに制御の流れを含んでおり，flow-sensitive な情報を得ることができる．

過程情報を実行 trace とする方法の問題は，着目している object に関係のない文や式が trace として含ま

れうることである．そのため，小さな program であっても実行 trace は長くなることが多く，着目している object の流れを理解するのが難しくなる[4]．また，静的に実行 trace を得るためには対象 program を記号的に実行する必要があるが，これは高 cost の処理である．特に open program の場合，object の生成点から使用点に至るすべての経路を記号的に実行するのは容易ではない．

実行 trace の上記の欠点を改善する方法として，過程情報を，object の生成点から各使用点に至る method の呼び出し系列によって定義する方法が考えられる．たとえば，method の呼び出し関係を表す call graph に基づき，着目している object に関わる method 呼び出しおよび復帰の系列を過程情報と見なすことができる．Java では method 呼び出し単位で処理が進行することが一般的なため，このような系列は，着目している object の流れの理解に有用な過程情報を含んでいることが期待できる．また，call graph の計算は，実行 trace に比べて低 cost で実現可能である．

一方，この方法は field 参照を介した object の flow を表現できないという問題がある．たとえば，ある method で object を別の object の field に代入した後，別の method でその object を取り出して使用するような流れは上記の定義では表現できない．これは配列参照に関しても同様である．

また，着目している object が `Map` や `Set` 等の `Collection` class の object（container object と呼ぶ）へ格納される場合，container object の流れと着目している object の流れの区別や関連付けが問題となる．

3.1.1 提　　案

本論文では，method の呼び出し系列に基づく方法を一般化し，過程情報を object の指示関係の変化の系列として定義することでこれらの問題を解決する．method 呼び出しおよび復帰にともなう object の移動は，その object を指示する存在（変数）の変化であり，上記の定義において，指示関係の変化を method 呼び出しおよび復帰の場合に制限したものとみることができる．object の指示関係の変化の系列は，label 付状態遷移系上の遷移系列として定義される．

最初に，object の指示状態 *state* を以下のように定義する．

$$\begin{aligned} state \quad ::= \quad & (h)_I \\ | \quad & (h,v)_V \\ | \quad & (h,h',fn)_H \\ | \quad & (h,cn,fn)_C \end{aligned}$$

h は着目している object である．$(h)_I$ は h がどこからも指示されていない状態，$(h,v)_V$ は h が変数 v から指示されている状態，$(h,h',fn)_H$ は h が別の object h' の instance field fn から指示されている状態，$(h,cn,fn)_C$ は h が class cn の class field fn から指示されている状態をそれぞれ表す．

object の指示状態 *state* を変化させる要因を以下の *event* によって表す．

$$\begin{aligned} event \quad ::= \quad & evNEW(i,h,v) \\ | \quad & evAS(i,h,v_1,v_2) \\ | \quad & evCALL(i,h,v_1,v_2) \\ | \quad & evRET(i,h,v_1,v_2) \\ | \quad & evPF(i,h,v,h',fn) \\ | \quad & evGF(i,h,v,h',fn) \\ | \quad & evPS(i,h,v,cn,fn) \\ | \quad & evGS(i,h,v,cn,fn) \end{aligned}$$

すべての *event* は，program point i における object h の指示状態を変化させる要因を表す．要因は，$evNEW(i,h,v)$ では object h の生成および変数 v への代入，$evAS(i,h,v_1,v_2)$ では変数 v_2 から v_1 への代入，$evCALL(i,h,v_1,v_2)$ では method 呼び出しにおける呼び出し元実引数 v_2 から呼び出し先仮引数 v_1 への代入，$evRET(i,h,v_1,v_2)$ では method 呼び出しからの復帰における呼び出し先で戻り値を保持する変数 v_2 から呼び出し元で戻り値を受ける変数 v_1 への代入，$evPF(i,h,v,h',fn)$ では変数 v から object h' の instance field fn への代入，$evGF(i,h,v,h',fn)$ では object h' の instance field fn から変数 v への代入，$evPS(i,h,v,cn,fn)$ では変数 v から class cn の class field fn への代入，$evGS(i,h,v,cn,fn)$ では class cn の class field fn から変数 v への代入である．

各 *event* は，以下の定義に基づき，label 付状態遷移系を定める．

定義 3.1（Object Flow Graph） program P 内で生成される object h の状態遷移とは，P 内で生じる h に関する各 *event* から，以下のいずれかを満たす形で構成される 3 つ組の集合である．

(1) $((h)_I, evNEW(i,h,v), (h,v)_V)$.
(2) $((h,v_2)_V, evAS(i,h,v_1,v_2), (h,v_1)_V)$.
(3) $((h,v_2)_V, evCALL(i,h,v_1,v_2), (h,v_1)_V)$.
(4) $((h,v_2)_V, evRET(i,h,v_1,v_2), (h,v_1)_V)$.
(5) $((h,v)_V, evPF(i,h,v,h',fn), (h,h',fn)_H)$.
(6) $((h,h',fn)_H, evGF(i,h,v,h',fn), (h,v)_V)$.
(7) $((h,v)_V, evPS(i,h,v,cn,fn), (h,cn,fn)_C)$.
(8) $((h,cn,fn)_C, evGS(i,h,v,cn,fn), (h,v)_V)$.

ここで各3つ組 $(state_1, event, state_2)$ は，label $event$ が付加された $state_1$ から $state_2$ への状態遷移を表す．このようにして得られる，h のすべての状態遷移から構成される label 付状態遷移系を h の object flow graph と呼ぶ．□

object の生成点から使用点に至る過程は，object flow graph 上の遷移系列として定義される．

定義 3.2（Object Flow Path） h の object flow graph において，$state_1$ から $state_2$ へ至る遷移系列を（$state_1$ から $state_2$ への）object flow path と呼ぶ．□

object flow graph と一般的な data 依存 graph（たとえば program slicing における program dependence graph（PDG）[13]）の大きな違いは，PDG のような依存 graph では，文や式のような構文要素が node，構文要素間の依存関係が edge として表現されているのに対し，object flow graph では，object の指示状態が node，状態遷移を引き起こす（構文要素に対応した）$event$ が edge として表現されていることである．program slicing との関係については，8 章で詳しく述べる．

なお，program model 中の program point i の集合を表す関数 PP および変数 v の集合を表す関数 Var は，$event$ に対しても自明な形で定義される．これは 4 章で用いる．

3.1.2 Object Flow Graph の意味

本項では，3.1.1 項で定義した object flow graph の意味について述べる．最初に，理解しにくいと思われる「object の状態遷移」および「object flow graph における分岐・合流」に関して直感的な説明を行った後，本論文における object flow 解析が扱う問題を定義し，それに基づき object flow graph の意味を非形式的に与える．なお，object flow graph の意味を厳密に定義するためには，Java program の形式的な意味が必要となる．これに関しては 3.3 節でふれる．

object の状態遷移 object h の object flow graph における，$state_1$ から $state_2$ への $event$ による遷移は，$event$ の生じる program point i において，h の新たな指示状態[*1] $state_2$ が生じうることを意味している．これは，program point i において，$state_1$ が表す指示状態が消滅することを意味しないことに注意されたい．一般に，program の実行過程の各時点（program point）において，object は複数の指示状態を持ちうる．たとえば，代入文 `x = y;` により，それまで `y` に指示されていた object h は `x` に指示されるようになるが，このことは h が `y` に指示されなくなること（遷移前の状態の消滅）を意味しない．直感的には，定義 3.1 の遷移規則は新しい状態の発生順序を定めているといえる．

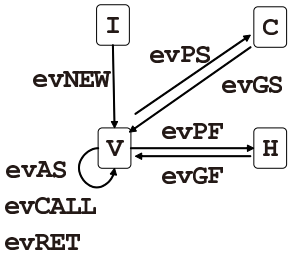

図 5 Object の状態遷移
Fig. 5 State transition of object.

object の 4 種類の状態（I, V, H, C）間で生じうる遷移を**図 5** に示す．object は，生成される（$evNEW$）ことで，どこからも指されていない状態 I から，変数に指されている状態 V へ遷移する（指示状態が増える．以降も同様）．その後，別の object の instance field へ代入される（$evPF$）と，状態 H へと遷移する．状態 H からの遷移は，instance field の読み出し（$evGF$）によってのみ生じ，読み出し先の変数に指されている状態 V へ遷移する．class field についても同様に，class field から指されている状態 C と V との間の遷移（$evPS$ および $evGS$）が存在する．method 呼び出し（$evCALL$）による遷移は，呼び出し元の実引数に指されている状態 V から呼び出し先の仮引数に指されている状態 V への遷移である．同様に，復帰（$evRET$）による遷移は，呼び出し先で返却値を保持している変数に指されている状態 V から，呼び出し元で返却値を受け取る変数に指されている状態 V への遷移である．

object flow graph の分岐・合流 object flow graph における分岐は，ある領域が保持する object の参照値が 2 つ以上の異なる領域に複写される場合に生じる．ここでの領域とは，変数，object の field，および各 object 生成点で生成する object を格納している仮想的な領域（状態 I において object を保持している領域）のことである．

分岐が生じる例として，以下の簡単な program 片（class `A` に含まれているとする）を考える．

```
1: x = new C();
2: y = x;
3: z = x;
```

*1 object flow graph における「状態」とはすべて「指示状態」のことであり，以降ではこの両者を区別せず用いる．

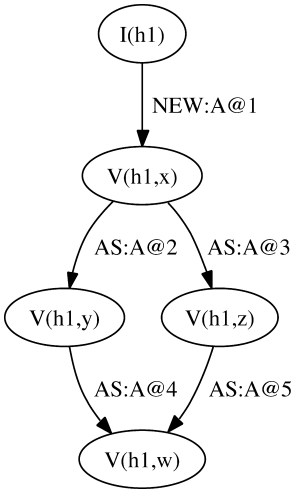

図 6 Object flow graph の例
Fig. 6 Example of object flow graph.

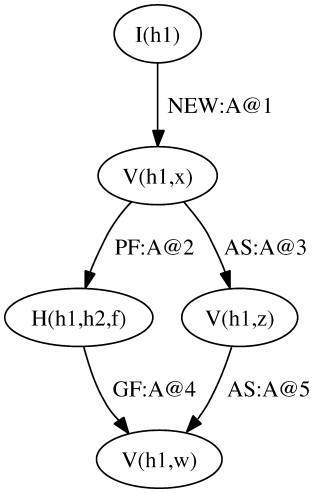

図 7 Object flow graph の例（2）
Fig. 7 Example of object flow graph (2).

```
4: w = y;
5: w = z;
```

これに対する object flow graph は**図 6** のようになる．なお，本論文では object flow graph を以下のように簡略化して表記する．

(1) *event* の名前は短縮形 NEW，ASsign，CALL，RETurn，PutField，GetField，PutStatic，および GetStatic を用いる．
(2) *event* の位置は program point ではなく，その program point に対応する class 名と行番号を用いる．

たとえば NEW:A@1 は class `A` の 1 行目における *event new*，AS:A@2 は同 2 行目における *event assign* を表す．

図 6 の例では，文 2 および 3 により変数 `x` の保持する object $h1$ の参照値が変数 `y` および `z` へと代入された結果，分岐が発生している．

一方，object flow graph における合流は，2 つ以上の異なる領域が保持する同一 object の参照値がある同一領域に複写される場合に生じる．図 6 の例では，文 4 および 5 により変数 `y` および `z` の保持する object $h1$ の参照値が変数 `w` へと代入された結果，合流が発生している．

この例は変数代入のみからなる単純なものであり，変数を node，変数代入を edge とした graph を書くと，program の syntactic な情報のみから同様の object の流れ情報を得ることができる．object flow graph の特徴の 1 つは，生成される object という semantic な要素が取り入れられていることである．たとえば，文 2 が `a.f = y;`，文 4 が `w = b.f;` に置き換えられた例

```
1: x = new C();
2: a.f = x;
3: z = x;
4: w = b.f;
5: w = z;
```

を考える．もし `a` と `b` が同一の object $h2$ を参照しているならば，object $h1$ の流れは object $h2$ の field f が先ほどの例における変数 `y` に置き換わったものになると考えるのが自然である．このような流れは，先ほど述べた syntactic な graph では表現できないが，object flow graph では**図 7** に示すように，図 6 と構造的に同型の graph として表現することができる．

object flow 解析が扱う問題　以上の議論から分かるように，これまで直感的に用いてきた object flow とは，program の実行時における「object の参照値の複写」という事象を抽象化したものである．参照値の複写という概念を用いると，object flow 解析が取り扱う問題は，「解析対象 program P 内で生成される object h に対して，P の実行時に h の参照値が複写される場所，複写元および複写先領域，複写の種類，複写の発生順序を求める」ものであるということができる．解析結果を表す object flow graph では，各 edge が複写場所（program point），複写元および複写先領域，複写の種類（変数代入，method 呼び出し等）を表し，各 object flow path が起こりうる複写の発生順序を表している．本論文で示す解析は近似的であり，解析結果の object flow graph は実際には起こりえない edge や path を含んでいるが，起こりうるものは必ず含まれているという意味で健全である．健全性については，3.3 節で改めて議論する．

$$\frac{\mathcal{M} \vdash method(m, cn, mn, dn, a)}{\mathcal{M} \vdash dispatch(m, cn, mn, dn)}\ (D1) \qquad \frac{\begin{array}{l}\mathcal{M} \vdash \neg\ method(m, cn_1, mn, dn, \_) \\ \mathcal{M} \vdash dispatch(m, cn_2, mn, dn) \\ \mathcal{M} \vdash subtyped(cn_1, cn_2)\end{array}}{\mathcal{M} \vdash dispatch(m, cn_1, mn, dn)}\ (D2)$$

図 8 解析 D の導出規則
Fig. 8 Derivation rule of analysis D.

3.2 Closed Program の解析

本節では，closed program に対する object flow 解析を行う方法について述べる．object flow 解析は，pointer 解析の結果に基づくため，最初にここで用いる pointer 解析を deductive system により定義する．

3.2.1 Pointer 解析

本項で定式化する pointer 解析は，context-insensitive, flow-insensitive, field-sensitive, interprocedural なものである．pointer 解析の目的は，各変数の指示先となる object，各 object の field の指示先となる object，および各 method 呼び出しの呼び出し先となる method 定義を求めることである．動的な method 呼び出しでは，receiver object の型によって呼び出される method が変化するため，呼び出し先の解析には両者の対応関係（dispatch 情報と呼ぶ）も必要となる．

ここでは，これらの情報を以下の predicate によって表現する．

$ptv(v, h)$ 変数 v が object h を指示する．

$pth(h_1, fn, h_2)$ object h_1 の field fn が object h_2 を指示する．

$calling(i, m)$ program point i の method 呼び出しが method m を呼び出す．

$dispatch(m, cn, mn, dn)$ receiver object の型が cn，呼び出し method 名が mn，型が dn である method 呼び出しの呼び出し先は method m である．

図 8，**図 9** に deductive system による pointer 解析を示す．解析 D の各規則は predicate $dispatch$ の導出規則，解析 P の各規則は predicate ptv，pth，$calling$ の導出規則である．解析 P は前提に $dispatch$ を用いる規則を含むため，解析 D に依存している．

規則 $D1$ は，class cn で method 名 mn，型 dn の method m が定義されているなら，cn 型の receiver object に対する method 名 mn，型 dn の method 呼び出しの呼び出し先が m であることを表す．規則 $D2$ は，class cn_1 が親 class cn_2 から method m の定義を継承し，かつそれを override していない場合に，cn_1 は親 class cn_2 の m に関する dispatch 情報を引き継ぐことを表す．

規則 $P1$ は，$new(i, v, h)$ により，変数 v が i で生成される object h を指示することを表す．規則 $P2$ は，$assign(i, v_1, v_2)$ により，右辺 v_2 の指示先 object を左辺 v_1 も指示することを表す．規則 $P3$ は，$getfield(i, v_1, fn, v_2)$ により，v_1 の指示先 object h_1 の field fn の指示先 object h_2 を v_2 が指示することを表す．規則 $P4$ は，$putfield(i, v_1, fn, v_2)$ により，v_2 の指示先 object h_2 を v_1 の指示先 object h_1 の field fn が指示することを表す．規則 $P5$ は，$putstatic(i, cn, fn, v_2)$ により class cn の field fn に代入される v_2 の指示先 object h が，$getstatic(i, cn, fn, v_1)$ により，v_1 の指示先になることを表す．規則 $P6$ は，$callstatic(i, cn, dn, mn)$ の呼び出し先が $dispatch(m, cn, mn, dn)$ により，method m に定まることを表す．規則 $P7$ は，$calldynamic(i, cn_1, dn, mn)$ の呼び出し先が第 1 引数（暗黙の `this` 引数）の指示先 object h の型 cn_2 および $dispatch(m, cn_2, mn, dn)$ により，method m に定まることを表す．規則 $P8$ は，method 呼び出し $calling(i, m)$ により，呼び出し先 method m の返す object h が，呼び出し元で戻り値を受ける変数 v_1 の指示先になることを表す．規則 $P9$ は，method 呼び出し $calling(i, m)$ により，呼び出し元の実引数 v_2 の指示先 object h が，呼び出し先 method m の仮引数 v_1 の指示先になることを表す．

例 3.1 例として，以下の predicate に対して解析 P を適用する．

1 $new(i_1, \mathtt{x}, h_6)$
2 $new(i_2, \mathtt{xs}, h_7)$
3 $putfield(i_3, \mathtt{xs}, [], \mathtt{x})$
4 $getfield(i_4, \mathtt{xs}, [], \mathtt{v})$

predicate 1 および 2 に規則 $P1$ を適用することで predicate 5 $ptv(\mathtt{x}, h_6)$ および 6 $ptv(\mathtt{xs}, h_7)$ が導出できる．predicate 3，5，6 に規則 $P4$ を適用することで predicate 7 $pth(h_7, [], h_6)$ が導出できる．最後に predicate 4，6，7 に規則 $P3$ を適用することで predicate 8 $ptv(\mathtt{v}, h_6)$ が導出できる．これ以上導出できる predicate は存在しないため，以上の 8 つが解析 P の解となる． □

$$\frac{\mathcal{M} \vdash new(i, v, h)}{\mathcal{M} \vdash ptv(v, h)} \ (P1)$$

$$\frac{\mathcal{M} \vdash assign(i, v_1, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash ptv(v_1, h)} \ (P2)$$

$$\frac{\mathcal{M} \vdash getfield(i, v_1, fn, v_2) \quad \mathcal{M} \vdash ptv(v_1, h_1) \quad \mathcal{M} \vdash pth(h_1, fn, h_2)}{\mathcal{M} \vdash ptv(v_2, h_2)} \ (P3)$$

$$\frac{\mathcal{M} \vdash putfield(i, v_1, fn, v_2) \quad \mathcal{M} \vdash ptv(v_1, h_1) \quad \mathcal{M} \vdash ptv(v_2, h_2)}{\mathcal{M} \vdash pth(h_1, fn, h_2)} \ (P4)$$

$$\frac{\mathcal{M} \vdash getstatic(i_1, cn, fn, v_1) \quad \mathcal{M} \vdash putstatic(i_2, cn, fn, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash ptv(v_1, h)} \ (P5)$$

$$\frac{\mathcal{M} \vdash callstatic(i, cn, mn, dn) \quad \mathcal{M} \vdash dispatch(m, cn, mn, dn)}{\mathcal{M} \vdash calling(i, m)} \ (P6)$$

$$\frac{\mathcal{M} \vdash calldynamic(i, cn_1, mn, dn) \quad \mathcal{M} \vdash actual(i, 1, v) \quad \mathcal{M} \vdash ptv(v, h) \quad \mathcal{M} \vdash tyh(h, cn_2) \quad \mathcal{M} \vdash dispatch(m, cn_2, mn, dn)}{\mathcal{M} \vdash calling(i, m)} \ (P7)$$

$$\frac{\mathcal{M} \vdash calling(i, m) \quad \mathcal{M} \vdash incoming(i, v_1) \quad \mathcal{M} \vdash outgoing(m, p, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash ptv(v_1, h)} \ (P8)$$

$$\frac{\mathcal{M} \vdash calling(i, m) \quad \mathcal{M} \vdash formal(m, k, v_1) \quad \mathcal{M} \vdash actual(i, k, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash ptv(v_1, h)} \ (P9)$$

図 9 解析 P の導出規則
Fig. 9 Derivation rule of analysis P.

3.2.2 Object Flow 解析

3.1 節で定義した object flow を求める解析は *event* を導出する deductive system として定式化できる．**図 10** に object flow 導出規則を示す．各規則は，object の指示関係を変化させうる program model の各 predicate と 1 対 1 に対応しており，その意味は明らかである．

object flow 解析は，3.2.1 項で定義した解析 D，P および図 10 の解析 G によって構成される．object h の object flow graph は，導出された *event* および定義 3.1 から求めることができる．

図 1 の program に対する object flow graph の例を**図 11** に示す．解析の入力は，class `A` および `B` の定義に，program が暗黙的に呼び出している `Object` および `String` class の constructor method 定義（空）を加えた closed program（の model）である．ここでは 6 行目で生成している `String` object（h6 と呼ぶ）に着目しており，図 11 は h6 の object flow graph である．図の表記法については，3.1.2 項を参照されたい．

図 11 を見ると，method 呼び出しだけでなく，field 参照（配列参照）や container object への格納等に由来する object flow が object を指示する存在の変化として自然な形で表現できていることが分かる．

class `B` の field `str` は，method `put` の呼び出しによって書き込まれた後，method `get` の呼び出しによって読み出される．`str` の書き込みと読み出しの間には，`put` からの復帰，`m2` の呼び出し，`get` の呼び出しが存在するが，これらは着目する object の指示状態に影響しないため，graph 上では捨象されている．これは実行 trace との大きな相違点である．

$$\frac{\mathcal{M} \vdash new(i, v, h)}{\mathcal{M} \vdash evNEW(i, h, v)} \ (G1)$$

$$\frac{\mathcal{M} \vdash assign(i, v_1, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evAS(i, h, v_1, v_2)} \ (G2)$$

$$\frac{\mathcal{M} \vdash calling(i, m) \quad \mathcal{M} \vdash formal(m, k, v_1) \quad \mathcal{M} \vdash actual(i, k, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evCALL(i, h, v_1, v_2)} \ (G3)$$

$$\frac{\mathcal{M} \vdash calling(i_1, m) \quad \mathcal{M} \vdash incoming(i_1, v_1) \quad \mathcal{M} \vdash outgoing(m, p, v_2) \quad \mathcal{M} \vdash ptv(v_2, h) \quad \mathcal{M} \vdash codeindex(i_2, m, p)}{\mathcal{M} \vdash evRET(i_2, h, v_1, v_2)} \ (G4)$$

$$\frac{\mathcal{M} \vdash putfield(i, v_1, fn, v_2) \quad \mathcal{M} \vdash ptv(v_1, h_1) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evPF(i, h, v_2, h_1, fn)} \ (G5)$$

$$\frac{\mathcal{M} \vdash getfield(i, v_1, fn, v_2) \quad \mathcal{M} \vdash ptv(v_1, h_1) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evGF(i, h, v_2, h_1, fn)} \ (G6)$$

$$\frac{\mathcal{M} \vdash putstatic(i, cn, fn, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evPS(i, h, v_2, cn, fn)} \ (G7)$$

$$\frac{\mathcal{M} \vdash getstatic(i, cn, fn, v_2) \quad \mathcal{M} \vdash ptv(v_2, h)}{\mathcal{M} \vdash evGS(i, h, v_2, cn, fn)} \ (G8)$$

図 10 解析 G の導出規則
Fig. 10 Derivation rule of analysis G.

なお，$(h6, \texttt{main} :: \texttt{x})_V$ から $(h6, < \texttt{init} >:: \texttt{t6})_V$ への edge は，図 1 の program の 6 行目

```
String x = new String();
```

に対して Java の compiler が自動的に生成する constructor method（<init>）の呼び出しに対応している．x はその実引数，t6 は method <init>の仮引数である．

3.3 解析の健全性

一般に，program 解析の健全性を厳密に議論するためには，program の意味を形式的に定義し，解析結果の意味をそれに基づいて与える必要がある．

たとえば，3.2.1 項で示した pointer 解析の健全性を議論するためには，Java program の形式意味論を定義したうえで，解析結果である $ptv(v, h)$ や $pth(h_1, f, h_2)$ 等の predicate の意味を，その意味論と関連付けて定義しなければならない．

Java program の意味の定義は，操作意味論を用いて行う方法[23),25)] が一般的である．操作意味論では，program の意味は，program の実行とともに変化していく stack や heap の状態（変数や object の値）の系列（実行 trace）として定義できる．

このような意味論を前提とすると，たとえば 3.2.1 項で informal に記述した predicate $ptv(v, h)$ の意味（変数 v が object h を指示する）は，「解析対象 program のとりうる実行 trace において，ある時点で変数 v が object h の参照値[⋆1]を持つ可能性がある」と定義できる．このとき，解析の健全性は「解析対象 program P のとりうる実行 trace において，ある時点で変数 v が object h の参照値を持つ場合には必ず，$ptv(v, h)$ が解析結果に含まれる（導出される）」という形で与えることができる．同様に，3.2.2 項で定義した object flow 解析についても，実行 trace における参照値の複写と対応付ける形で健全性の定義が可能である．

これらの証明は，実行 trace の長さに関する帰納法を用いて文献 23) 等と同様の流れで行うことが可能だと思われる．ただし，Java の完全な言語仕様は非常に大きく，意味論の定義およびその証明には多大な作業が必要となる．本論文では証明は行わず，以降では closed program に対する解析の正しさを前提とする．

⋆1 object h は，ある program point で生成されるすべての object を model 化したものなので，より正確には「h に対応する program point で生成される任意の object の参照値」というべきであるが，煩雑であり，ここでの議論の大筋にも影響しないため省略する．

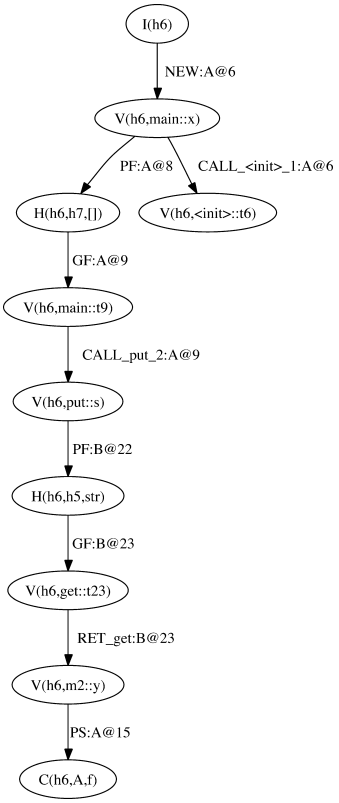

図 11 Closed program に対する object flow graph
Fig. 11 Object flow graph for closed program.

4. Open Program の解析

本章では 3 章で定義した closed program に対する解析を open program に対する解析に拡張する．

4.1 近似方法

open program の解析では，program 内部の object の流れに加え，program の外部との間の object の流れを考慮する必要がある．

ある method 内部で生成した object が method の外部へ到達する（escape する）かどうかを調べる解析は escape 解析と呼ばれている．escape しないことが静的に分かる object は，heap でなく stack 上に生成する等の最適化が可能となる．

本論文では escape 解析の概念に基づき，program の外部との間の object の流れを近似する．通常の escape 解析と異なる点は，object が escape するかどうかの境界が個々の method でなく program 全体であること，および program の外部へ escape した object が再び program 内部へ戻ってくる流れの解析が必要になることである．

以降では，object が program の外部へ到達することを object の流出と呼び，program の外部へ到達した object が再び program 内部へ戻ってくることを object の流入と呼ぶ．また，program 外部で定義される class および method をそれぞれ外部 class，外部 method と呼ぶ．

object の流出および流入が起こる箇所は，program point および変数によって識別できる．以下にこれらの定義を示す．

定義 4.1（Outflowing Point） P を open program とする．P の program model $M(P)$ 内の program point i における変数 v の出現 (i, v) が以下のいずれかを満たすとき，(i, v) を P の outflowing point と呼ぶ．

(1) i における，外部 method の呼び出しの実引数が v である．
(2) i における，外部から呼ばれる method からの復帰にともなう返却値保持変数が v である．
(3) i における，外部から参照可能な object の instance field への代入の代入元変数が v である．
(4) i における，class field への代入の代入元変数が v である．

□

定義 4.2（Inflowing Point） P を open program とする．P の program model $M(P)$ 内の program point i における変数 v の出現 (i, v) が以下のいずれかを満たすとき，(i, v) を P の inflowing point と呼ぶ．

(1) i における，外部 method の呼び出しの返却値の代入先変数が v である．
(2) i が外部から呼ばれる method m の先頭 program point で，v が m の仮変数である．
(3) i における，外部から参照可能な object の instance field からの代入の代入先変数が v である．
(4) i における，class field からの代入の代入先変数が v である．

□

図 12 に，これらの定義に基づき，outflowing point および inflowing point を導出する規則を示す．各規則が用いている predicate の意味は以下のとおりである．

$outflow(i, v)$ (i, v) は outflowing point である．
$inflow(i, v)$ (i, v) は inflowing point である．
$callaux(i)$ program point i の method 呼び出しは外部 method を呼び出す．
$escape(h)$ h は外部から参照可能な object である．

規則 $A1$ から $A4$ は定義 4.1 の各項目，規則 $A5$ か

$$\frac{\mathcal{M} \vdash callaux(i) \quad \mathcal{M} \vdash actual(i,k,v)}{\mathcal{M} \vdash outflow(i,v)}\ (A1) \qquad \frac{\mathcal{M} \vdash outgoing(m,p,v) \quad \mathcal{M} \vdash codeindex(i,m,p)}{\mathcal{M} \vdash outflow(i,v)}\ (A2)$$

$$\frac{\mathcal{M} \vdash putfield(i,v_1,fn,v_2) \quad \mathcal{M} \vdash ptv(v_1,h) \quad \mathcal{M} \vdash escape(h)}{\mathcal{M} \vdash outflow(i,v_2)}\ (A3) \qquad \frac{\mathcal{M} \vdash putstatic(i,cn,fn,v)}{\mathcal{M} \vdash outflow(i,v)}\ (A4)$$

$$\frac{\mathcal{M} \vdash callaux(i) \quad \mathcal{M} \vdash incoming(i,v)}{\mathcal{M} \vdash inflow(i,v)}\ (A5) \qquad \frac{\mathcal{M} \vdash formal(m,k,v) \quad \mathcal{M} \vdash codeindex(i,m,0)}{\mathcal{M} \vdash inflow(i,v)}\ (A6)$$

$$\frac{\mathcal{M} \vdash getfield(i,v_1,fn,v_2) \quad \mathcal{M} \vdash ptv(v_1,h) \quad \mathcal{M} \vdash escape(h)}{\mathcal{M} \vdash inflow(i,v_2)}\ (A7) \qquad \frac{\mathcal{M} \vdash getstatic(i,cn,fn,v)}{\mathcal{M} \vdash inflow(i,v)}\ (A8)$$

図 12 解析 A の導出規則
Fig. 12 Derivation rule of analysis A.

ら $A8$ は定義 4.2 の各項目にそれぞれ対応している．各規則の意味は定義から明らかである．ここで定義した解析 A は，次節で open program に対する object flow 解析を定義するのに利用する．

本節の以降では，*callaux* および *escape* に関する解析 E について述べる．3.2.1 項で定義した closed program に対する pointer 解析（D，P）に E を加えた解析は，open program に対する健全な pointer 解析となる．健全性の証明は 4.3 節で行う．

図 13 に解析 E の導出規則を示す．大半の規則は，定義 4.1 および 4.2 に基づく object の流出および流入を表している．これらの規則を解析 A が導出する predicate *outflow* および *inflow* を用いて定義することも可能であるが，ここでは pointer 解析と object flow 解析の境界を明確にするため，解析 A に依存しない定義を行っている．

open program に対する解析では，これまで述べた program 内部で生成される object の流出および流入に加え，program 外部で生成される object を考慮する必要がある．ここでは，program 内部で生成されるすべての object と異なる新しい object h_0 を導入し，h_0 を用いてすべての外部生成 object を表す．以降では h_0 を universal object と呼ぶ．h_0 の型は任意の class の subclass とする．

規則 $E1$ は h_0 が外部から参照可能であることを表す．規則 $E2$ から $E5$ は定義 4.1 の各項目，規則 $E6$ から $E8$ は定義 4.2 の（4）以外の各項目を表す．項目（4）に対応する規則は存在しないが，その作用は主に規則 $E9$ によって包摂されている．規則 $E10$ および $E11$ は *callaux* の導出規則である．$E10$ は静的な method 呼び出し，$E11$ は動的な method 呼び出しにそれぞれ対応している．解析 D では，外部 method に対する dispatch 情報 *dispatch* を生成しないため，method 呼び出しが外部 method を呼び出しうるかは *dispatch* の有無によって判定することができる．なお，この判定は $\neg calling(i,m)$ という前提によっても行えそうにみえるが，この predicate は i に対して 1 つでも method 呼び出し先を導出できていれば成立しなくなる点で，規則 $E10$ および $E11$ の前提とは異なる．

4.2 Object Flow 解析の拡張

open program に対する object flow 解析を定式化するためには，3.2 節で定義した object flow を，program の外部を考慮した形に拡張する必要がある．最初に object の指示状態 *state* を以下のように拡張する．

$$\begin{aligned} state \quad &::= \quad \ldots \\ &\mid \quad (h)_A \end{aligned}$$

状態 $(h)_A$ は，object h が program の外部に流出している状態を表す．

$$\frac{}{\mathcal{M} \vdash escape(\mathrm{h}_0)}\ (E1)$$

$$\frac{\mathcal{M} \vdash escape(h_1) \quad \mathcal{M} \vdash pth(h_1, fn, h_2)}{\mathcal{M} \vdash escape(h_2)}\ (E2)$$

$$\frac{\mathcal{M} \vdash outgoing(m, p, v) \quad \mathcal{M} \vdash ptv(v, h)}{\mathcal{M} \vdash escape(h)}\ (E3)$$

$$\frac{\mathcal{M} \vdash actual(i, k, v) \quad \mathcal{M} \vdash ptv(v, h) \quad \mathcal{M} \vdash callaux(i)}{\mathcal{M} \vdash escape(h)}\ (E4)$$

$$\frac{\mathcal{M} \vdash putstatic(i, cn, fn, v) \quad \mathcal{M} \vdash ptv(v, h)}{\mathcal{M} \vdash escape(h)}\ (E5)$$

$$\frac{\mathcal{M} \vdash escape(h) \quad \mathcal{M} \vdash formal(m, k, v)}{\mathcal{M} \vdash ptv(v, h)}\ (E6)$$

$$\frac{\mathcal{M} \vdash escape(h) \quad \mathcal{M} \vdash incoming(i, v) \quad \mathcal{M} \vdash callaux(i)}{\mathcal{M} \vdash ptv(v, h)}\ (E7)$$

$$\frac{\mathcal{M} \vdash escape(h) \quad \mathcal{M} \vdash getstatic(i, cn, fn, v)}{\mathcal{M} \vdash ptv(v, h)}\ (E8)$$

$$\frac{\mathcal{M} \vdash escape(h_1) \quad \mathcal{M} \vdash escape(h_2)}{\mathcal{M} \vdash pth(h_1, fn, h_2)}\ (E9)$$

$$\frac{\mathcal{M} \vdash callstatic(i, cn, mn, dn) \quad \mathcal{M} \vdash \neg dispatch(m, cn, mn, dn)}{\mathcal{M} \vdash callaux(i)}\ (E10)$$

$$\frac{\mathcal{M} \vdash calldynamic(i, cn_1, mn, dn) \quad \mathcal{M} \vdash actual(i, 1, v) \quad \mathcal{M} \vdash ptv(v, h) \quad \mathcal{M} \vdash tyh(h, cn_2) \quad \mathcal{M} \vdash \neg dispatch(m, cn_2, mn, dn)}{\mathcal{M} \vdash callaux(i)}\ (E11)$$

図 13 解析 E の導出規則
Fig. 13 Derivation rule of analysis E.

次に，$(h)_A$ とそれ以外の状態間の遷移を表す *event* を追加する．

$$\begin{array}{rcl} event & ::= & \ldots \\ & | & evOF(i, h, v) \\ & | & evIF(i, h, v) \end{array}$$

$evOF(i, h, v)$ は，object h の outflowing point (i, v) からの流出，$evIF(i, h, v)$ は，object h の inflowing point (i, v) からの流入を表す *event* である．

新しい状態遷移の定義は，定義 3.1 に，以下の 3 つ組を追加したものになる．

(9) $((h, v)_V, evOF(i, h, v), (h)_A)$.
(10) $((h)_A, evIF(i, h, v), (h, v)_V)$.

追加した *event* *evOF* および *evIF* の導出規則を**図 14** に示す．各規則の意味は，これまでの議論から明らかである．

open program に対する object flow 解析は，closed program に対する解析 D，P，G に解析 E，A を加えたものとして構成される．いずれの解析における judgement も同じ導出記号 $\vdash$ を用いて表記するが，その区別は文脈から明らかである．

図 1 の program から class B の定義を取り除いた open program に対する object flow graph を**図 15** に示す．図 11 と同様，図には着目している object h6 の object flow graph である．

図 11 と比較すると，欠けている class B の method

$$\frac{\mathcal{M} \vdash outflow(i, v) \quad \mathcal{M} \vdash ptv(v, h)}{\mathcal{M} \vdash evOF(i, h, v)} \ (G9) \qquad \frac{\mathcal{M} \vdash inflow(i, v) \quad \mathcal{M} \vdash escape(h)}{\mathcal{M} \vdash evIF(i, h, v)} \ (G10)$$

図 14 解析 G の導出規則（追加）
Fig. 14 Additional derivation rule of analysis G.

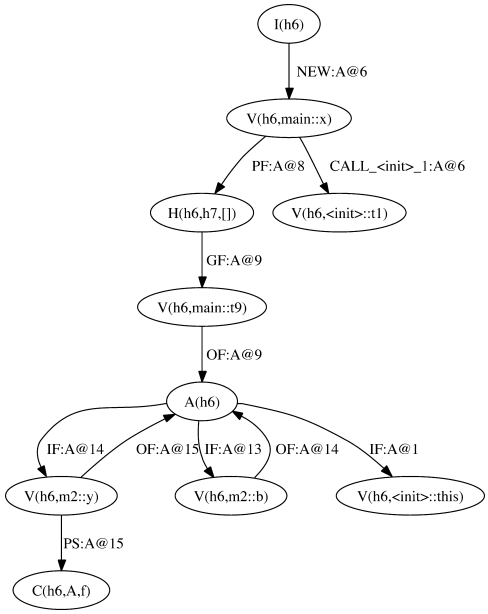

図 15 Open program に対する Object flow graph
Fig. 15 Object flow graph for open program.

`get` および `put` の呼び出しに由来する flow が，program 外部への流出および流入を表す flow によって近似されていることが分かる．一方で，存在する class `A` に含まれている flow はそのまま保持されており，pointer 解析の結果のように，過程情報が完全に失われてしまうことはない．

4.3 解析の健全性

本節では，open program に対する解析の健全性を，3 章で定義した closed program に対する解析が健全である，という前提のもとで証明する．証明の中心となるのは，pointer 解析の健全性である．object flow 解析の健全性は pointer 解析の健全性から導かれる．

最初に，open program が置かれる文脈に関する条件を定義する．

定義 4.3 （well-formed な文脈） open program P に対し，以下の条件を満たす program E を P の well-formed な文脈と呼ぶ．

(1) E，P は closed program を構成する．
(2) $Var(E) \cap Var(P) = \phi$.
(3) $Obj(E) \cap Obj(P) = \phi$.
(4) $PP(E) \cap PP(P) = \phi$.
(5) $Method(E) \cap Method(P) = \phi$.

E，P が構成する closed program を $E[P]$ で表す．$M(E[P]) = M(E) \cup M(P)$ である． □

次に，外部で生成される object および外部でのみ参照される class と，解析におけるそれらの近似表現を関連付ける抽象化関数を定義する．

定義 4.4 （抽象化関数） open program P に対する抽象化関数 β_h および β_c を以下のように定義する．

$$\begin{aligned} \beta_h(P)(h) &= h \quad \text{if } h \in Obj(P) \\ \beta_h(P)(h) &= \mathrm{h}_0 \quad \text{otherwise} \\ \beta_c(P)(cn) &= cn \quad \text{if } cn \in RefClass(P) \\ \beta_c(P)(cn) &= \mathrm{cn}_0 \quad \text{otherwise} \end{aligned}$$

h_0 は open program の外部で生成されるすべての object，cn_0 は open program の外部でのみ参照されるすべての class を表す識別子であり，$\mathrm{h}_0 \notin Obj(P)$, $\mathrm{cn}_0 \notin RefClass(P)$ を満たす． □

証明には，以下の諸性質を用いる．

補題 4.1 以下が成り立つ．

(1) $M(P) \vdash escape(h)$ ならば
$M(P) \vdash escape(\beta_h(P)(h))$.
(2) $M(E[P]) \vdash tyh(h, cn)$ ならば
$M(P) \vdash tyh(\beta_h(P)(h), \beta_c(P)(cn))$.
(3) $M(P) \vdash dispatch(m, cn, mn, dn)$ かつ
$m \in Method(P)$ ならば $cn \in RefClass(P)$.
(4) $M(E[P]) \vdash dispatch(m, cn, mn, dn)$ かつ
$m \in Method(P)$ ならば
$M(P) \vdash dispatch(m, cn, mn, dn)$.
(5) $M(E[P]) \vdash dispatch(m, cn, mn, dn)$ かつ
$m \notin Method(P)$ ならば，
$\forall m'.M(P) \vdash \neg dispatch(m', cn, mn, dn)$.
(6) $\forall m, mn, dn, a.$
$M(P) \vdash \neg dispatch(m, \mathrm{cn}_0, mn, dn)$.

□

以上の定義および補題から，次の主補題が証明できる．

補題 4.2 （主補題） open program P および P の well-formed な文脈 E に対して以下が成り立つ．

(a) $M(E[P]) \vdash ptv(v,h)$ かつ $v \in Var(P)$ ならば $M(P) \vdash ptv(v, \beta_h(P)(h))$.
(b) $M(E[P]) \vdash pth(h_1, fn, h_2)$ ならば $M(P) \vdash pth(\beta_h(P)(h_1), fn, \beta_h(P)(h_2))$.
(c) $M(E[P]) \vdash calling(i,m)$ かつ $i \in PP(P)$ かつ $m \in Method(P)$ ならば $M(P) \vdash calling(i,m)$.
(d) $M(E[P]) \vdash ptv(v,h)$ かつ $v \notin Var(P)$ ならば $M(P) \vdash escape(\beta_h(P)(h))$.
(e) $M(E[P]) \vdash calling(i,m)$ かつ $i \in PP(P)$ かつ $m \notin Method(P)$ ならば $M(P) \vdash callaux(i)$.

□

証明 付録 A.1 参照. □

open program に対する pointer 解析の健全性は補題 4.2 から導かれる.

補題 4.3 （open program に対する pointer 解析の健全性） open program P に対する pointer 解析は，P が置かれる任意の well-formed な文脈 E のもとで健全である．すなわち，以下が成り立つ.

(1) 任意の $v \in Var(P)$ および $h \in Obj(P)$ に対し，$M(E[P]) \vdash ptv(v,h)$ ならば $M(P) \vdash ptv(v,h)$.
(2) 任意の $h_1 \in Obj(P)$, $h_2 \in Obj(P)$ および fn に対し，$M(E[P]) \vdash pth(h_1, fn, h_2)$ ならば $M(P) \vdash pth(h_1, fn, h_2)$.
(3) 任意の $i \in PP(P)$ および $m \in Method(P)$ に対し，$M(E[P]) \vdash calling(i,m)$ ならば $M(P) \vdash calling(i,m)$.

□

証明 補題 4.2 および定義 4.4 による. □

補題 4.4 open program P および P が置かれる任意の well-formed な文脈 E に対し，$M(E[P]) \vdash event$, $PP(event) \in PP(M(P))$, $Var(event) \subseteq Var(M(P))$ ならば $M(P) \vdash event$ が成り立つ. □

証明 定義 4.3，補題 4.3 および object flow 解析の定義による．なお $PP(event) \in PP(M(P))$ であって，$Var(event) \subseteq Var(M(P))$ を満たさない $event$ は，外部 method の呼び出し，および外部 method への復帰のみである. □

定理 4.1 （open program に対する object flow 解析の健全性） open program P，P が置かれる任意の well-formed な文脈 E，object $h \in Obj(P)$ および変数 $v \in Var(P)$ に対し，$E[P]$ に closed program に対する解析を適用することで $(h)_I$ から $(h,v)_V$ へ至る object flow path が得られるならば，P に open program に対する解析を適用することで $(h)_I$ から $(h,v)_V$ へ至る（同一とは限らない）object flow path を得ることができる. □

証明 付録 A.2 参照. □

5. 解析精度の改善

本章では，型情報を用いて object flow 解析の精度を向上させる方法について述べる．4 章で示した open program に対する解析では，outflowing point から program の外部に流出する object はすべて inflowing point へ流入すると見なしていた．これに対し，Java の言語仕様が定める型に関する制約を利用することで，実際には起こりえない object の流れを除去することが可能である.

言語仕様が定める型に関する制約は，class 間の継承関係に基づくもので，直接的もしくは間接的に継承関係にない class 間の代入を禁止する．この制約を用いることで，たとえば method 呼び出しの戻り値を class A 型の変数に代入している場合，そこに到達する object は A 型もしくは A の subclass 型の object に限定することができる．ここでは，このような限定を type filtering と呼ぶ．また継承関係から定まる代入可能な型の間の関係を subtype 関係と呼ぶ.

type filtering を open program に対する解析に導入する際の問題は，open program には一部の class 定義（正確には class 定義に含まれる class 継承関係の定義）が存在しないため，完全な subtype 関係が分からないことである．不完全な subtype 関係をもとに type filtering を行うと，実際に起こりうる object の流れを除去してしまう可能性がある.

以降では，最初に完全な型情報に基づく type filtering について述べた後，open program から得られる不完全な型情報のもとで，近似的な type filtering を行う方法を示す.

5.1 完全な型情報に基づく Type Filtering

program P 内の class 集合上の subtype 関係は，P 内で定義されている直接的な class 継承関係の反射的推移的閉包である．なお，直接的な class 継承関係は明示的な `extends`，`implements` による継承，および暗黙的な `java.lang.Object` の継承も含め，すべて predicate *subtyped* によって表されているものとする.

図 16 に subtype 関係を表す predicate *subtype* を

$$\frac{\mathcal{M} \vdash subtyped(cn_1, cn_2)}{\mathcal{M} \vdash subtype(cn_1, cn_2)}\ (T1) \qquad \frac{\mathcal{M} \vdash refclass(cn)}{\mathcal{M} \vdash subtype(cn, cn)}\ (T2)$$

$$\frac{\mathcal{M} \vdash subtype(cn_1, cn_2) \quad \mathcal{M} \vdash subtype(cn_2, cn_3)}{\mathcal{M} \vdash subtype(cn_1, cn_3)}\ (T3)$$

図 16 Subtype 解析規則
Fig. 16 Subtype analysis rule.

$$\frac{\mathcal{M} \vdash \neg class(cn, a)}{\mathcal{M} \vdash auxclass(cn)}\ (S1) \qquad \frac{\mathcal{M} \vdash auxclass(cn_1)}{\mathcal{M} \vdash subtype(cn_1, cn_2)}\ (S2a)$$

$$\frac{\mathcal{M} \vdash auxclass(cn_1) \quad \mathcal{M} \vdash auxclass(cn_2)}{\mathcal{M} \vdash subtype(cn_1, cn_2)}\ (S2b)$$

図 17 Subtype completion 規則
Fig. 17 Subtype completion rule.

求める規則を示す．規則 $T1$ は *subtype* が *subtyped* を含むこと，規則 $T2$ および $T3$ は *subtype* がそれぞれ反射的，推移的であることを表す．

type filtering は，3 章で定義した pointer 解析規則のうち，ptv の導出規則の前提に型に関する制約を付加することで行える．

たとえば規則 $P3$ に type filtering を適用した規則は以下のようになる．

$$\frac{\begin{array}{l}\mathcal{M} \vdash getfield(i, v_1, fn, v_2)\\ \mathcal{M} \vdash ptv(v_1, h_1)\\ \mathcal{M} \vdash pth(h_1, fn, h_2)\\ \mathcal{M} \vdash tyh(h_2, cn_1)\\ \mathcal{M} \vdash tyv(v_2, cn_2)\\ \mathcal{M} \vdash subtype(cn_1, cn_2)\end{array}}{\mathcal{M} \vdash ptv(v_2, h_2)}\ (P3T)$$

前提 $subtype(cn_1, cn_2)$ により，object h_2 が変数 v_2 の指示先となるのは，h_2 の型 cn_1 が v_2 の型 cn_2 の subtype である場合に限定されている．他の規則についても同様に type filtering を適用することができる．

5.2 不完全な型情報に基づく Type Filtering

open program に対する解析では program から完全な型情報が得られないため，type filtering を正しく行うためには実際に起こりうる subtype 関係を包含するような subtype 関係を近似的に求める必要がある．以降では subtype 関係を近似的に求める操作を subtype completion と呼ぶ．

自明な subtype completion は，すべての class 間に subtype 関係があると見なすことである．これは type filtering を行わないことと等価である．ここでは自明でない 2 種類の subtype completion を示し，その正しさを証明する．1 つ目の subtype completion は open program が置かれる任意の文脈に対して健全な近似を行う．2 つ目の subtype completion は open program が置かれる文脈にある仮定を設けることで近似精度を向上させる．

最初に，任意の文脈において健全な subtype completion を示す．

定義 5.1（Full Subtype Completion） open program P に対し，**図 17** の規則 $S1$，$S2a$ によって構成される解析を P に対する full subtype completion と呼ぶ． □

規則 $S1$ は program 外部で定義される class を表す predicate *auxclass* の自明な定義である．full subtype completion の本質は規則 $S2a$ で，外部で定義される class はすべての class の subtype となることを表している．

以下の補題は full subtype completion が健全な近似であることを示す．

補題 5.1 （Full Subtype Completion の健全性） open program P に対する full subtype completion は，P が置かれる任意の well-formed な文

脈 E のもとで健全である．すなわち，任意の cn_1，cn_2 に対し，$M(E[P]) \vdash subtype(cn_1, cn_2)$ ならば $M(P) \vdash subtype(cn_1, cn_2)$． □

証明 付録 A.3 参照． □

full subtype completion の結果は，任意の文脈で健全である．一方，open program とそれが置かれる文脈の間の依存関係に方向性がある場合，その近似精度を向上させることが可能である．

前提 5.1（Downward Closed Assumption） 文脈に対する以下の前提を downward closed assumption と呼ぶ．

$$\forall cn_1, cn_2, a.$$
$$M(P) \vdash auxclass(cn_1) \Rightarrow$$
$$M(P) \vdash class(cn_2, a) \Rightarrow$$
$$M(E[P]) \vdash \neg subtype(cn_1, cn_2).$$

□

前提 5.1 は，解析対象 program 内の class を継承する外部 class はない，ということを意味している．たとえば，既存の framework から呼び出される plug-in のような program が解析対象の場合，framework を構成する各 class が plug-in program 中の class を継承する可能性はきわめて低く，downward closed assumption が成り立つ可能性は高い．一方で，解析対象が framework や library の場合，この前提が破られる可能性は十分にある．

前提 5.1 を仮定できる場合，以下の subtype completion が可能である．

定義 5.2 （DCA Subtype Completion） open program P に対し，図 17 の規則 $S1$，$S2b$ によって構成される解析を P に対する DCA subtype completion と呼ぶ． □

以下の補題は DCA subtype completion が downward closed assumption のもとで健全な近似であることを示す．

補題 5.2 （DCA Subtype Completion の健全性） open program P に対する DCA subtype completion は，P が置かれる任意の well-formed かつ downward closed assumption を満たす文脈 E のもとで健全である．すなわち，任意の cn_1，cn_2 に対し，$M(E[P]) \vdash subtype(cn_1, cn_2)$ ならば $M(P) \vdash subtype(cn_1, cn_2)$． □

証明 付録 A.4 参照． □

open program に対する解析を，full subtype completion に基づく type filtering を行うように修正し，

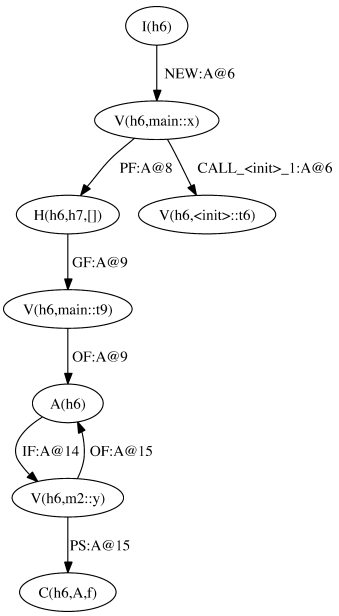

図 18 Open program に対する Object flow graph（with type filtering）
Fig. 18 Object flow graph for open program (with type filtering).

図 1 の program に適用した結果を**図 18** に示す．type filtering を行わない場合（図 15）と比較すると，実現不可能な *event* がすべて除去されていることが分かる．

6. 応　　用

本章では，これまで述べた object flow 解析の応用例として，Java program に対する taint 解析への適用方法を示す．

taint 解析とは，信頼できない外部からの入力（tainted data）が，外部に対して有害な作用をもたらしうる処理で使用される可能性を調べる program 解析である．tainted data の生成点は source（point），使用点は sink（point）と呼ばれる．

たとえば，よく知られている SQL injection 脆弱性は，taint 解析によって検出できる問題の 1 つである．HTTP servlet のような Java program の場合，SQL injection 脆弱性の source は

```
javax.servlet.http.HttpServletRequest:
    String getParameter(String);
    String[] getParameterValues(String);
```

等の外部入力値取得 method の戻り値，sink は

```
java.sql.Statement:
    boolean execute(String);
    ResultSet executeQuery(String);
```

等の SQL 発行 method の実引数となる．tainted data は `String` object によって表される．

Java program に対する object 単位の taint 解析は，tainted data を表す object に着目し，それが source から sink へ至る流れを調べる object flow 解析と見なすことができる．解析の結果得られる source から sink に至る object flow path は，過程情報を *event* 系列の形で含んでおり，利用者が脆弱性の原因を把握し修正するうえで有用である．

なお，object 単位の taint 解析では，`String` object を文字（`char` 型 data）単位に分解した後，再び `String` object を構成するような，object 以外の data を介した流れは追跡できない．これは，object を tainted data の基本単位とする taint 解析に共通する弱点である．

実際に object 単位の taint 解析を行う際には，上記に加えて tainted data を表す object の変化を考慮する必要がある．特に，tainted data が文字列の場合，それを表す object は文字列の連結等の操作により変化していく．たとえば，以下のような code 片において，

```
1:  s1 = getParameter("id");
2:  s2 = "select pw from db where id=" + s1;
```

tainted data を表す object は，文 1 では `getParameter` の戻り値だが，文 2 では，それを文字列 literal と連結した結果の文字列を表す別の `String` object である．

本論文で示した object flow 解析では，tainted data の伝播を表す predicate および *event* を追加することでこのような tainted data を表す object の変化を表現可能である．すなわち，伝播を表す predicate $stem(i, v_1, v_2)$ を program model に加え，解析 G に以下のような *event* 導出規則を追加すればよい．

$$\frac{\begin{array}{l}\mathcal{M} \vdash stem(i, v_1, v_2)\\ \mathcal{M} \vdash ptv(v_1, h_1)\\ \mathcal{M} \vdash ptv(v_2, h_2)\end{array}}{\mathcal{M} \vdash evPR(i, h_1, v_2, h_2)}\ (G11)$$

$stem(i, v_1, v_2)$ は，v_2 の指示先 object から v_1 の指示先 object への tainted data の伝播を表す predicate，$evPR(i, h_1, v_2, h_2)$ は，v_2 の指示先 object h_2 から object h_1 への tainted data の伝播を表す *event* である．解析時には，伝播の起こりうる箇所に対して，predicate $stem(i, v_1, v_2)$ を記述し program model に加えればよい．なお，専用の Java の具体構文を用意することで記述の手間を減らすことも可能だが，詳細は本論文の範囲外である．

7. 評　　価

本章では，これまでに述べた object flow 解析を実装し，簡単な servlet program に適用した結果を示す．評価に用いる servlet program を**図 19** に示す．

servlet は，Apache Tomcat [2] 等の web container 上で動作する Java program である．servlet は `main` method を持たず，web container から必要に応じて生成され，event-driven に各種 method が呼び出される点で，典型的な open program の 1 つである．

また，servlet は web container を介して Internet 上の不特定多数に service を提供することが多いため，悪意のある利用者からの攻撃にさらされやすくなっている．SQL injection をはじめとする数多くの脆弱性問題が報告されており [33]，6 章で述べた object flow 解析に基づく検証の対象としても重要である．

図 19 は，`doPost` event を受け取り，信頼できない外部からの data に基づき危険な処理を行う servlet の例である．この program は，信頼できない外部からの data を method `getParameter` の呼び出しにより受け取り，`ArrayList xs` に 1 度格納した後，method `process` でそれを取り出し method `unsafe` に渡している．`unsafe` は，database 参照や file 入出力等の危険な処理を行う method であると想定している．

ここでは，図 19 の program に対し，これまでに述べた解析を様々な条件で適用した結果を比較する．

すべての解析に共通の入力は，図 19 の program および web container の必要最低限の動作を模倣する stub である．stub は，`doPost` method の引数となる `HttpServletRequest` および `HttpServletResponse` interface を実装した class と，`Servlet` class を生成し `doPost` method を起動する class からなる 116 行の Java program である．以降では図 19 の program および stub からなる program を servlet program と呼ぶ．

期待する出力は，`getParameter` 内で生成される object が，`unsafe` method の呼び出しの引数に至る危険な object flow path を含む object flow graph である．

解の計算は，解析の各導出規則および，解析対象 program の model を datalog program として記述し，その閉包を求めることで行う．閉包の計算には Stanford 大学で開発された `bddbddb` library [24] を利用している．評価環境の CPU は Opteron 165 1.8 GHz，memory は 2 GB，OS は Linux（CentOS 4.3）である．servlet program は JDK1.5 付属の `javac` を用いて class file を生成した後，2.1 節で述べた program model に変

```
 1: import javax.servlet.*;
 2: import javax.servlet.http.*;
 3: import java.io.*;
 4: import java.util.*;
 5:
 6: public class Servlet extends HttpServlet
 7: {
 8:   public void doPost(
 9:     HttpServletRequest req,
10:     HttpServletResponse res)
11:       throws ServletException {
12:     String p = req.getParameter("param");
13:     Wrapper w = new Wrapper();
14:     w.addw(p);
15:     process(w);
16:   }
17:
18:   static void process(Wrapper w) {
19:     String s = w.getw(0);
20:     unsafe(s);
21:   }
22:
23:   static void unsafe(String u) {
24:     // do unsafe operation on u
25:   }
26: }
27:
28: class Wrapper
29: {
30:   ArrayList xs;
31:   Wrapper() { xs = new ArrayList(); }
32:   void addw(String s) { xs.add(s); }
33:   String getw(int i) {
34:     return (String)xs.get(i);
35:   }
36: }
```

図 19 Servlet program
Fig. 19 Servlet program.

換し，解析の入力としている．なお，変換の際には，copy 伝播による program の簡単化を行い，変数間の単純な代入文は除去している．

表 1 に評価結果を示す．表中の analysis は解析の種類，#class は解析対象 class の数，#pred は解析対象 class の model の predicate の数，time（s）は解析に要した総時間（秒）を表す．また NEW から IF までの各欄は解析が検出した，着目している object（`getParameter` の戻り値）に関する *event* の数を表す．path は前述した危険な object flow path を検出できたかどうかを表す．

表の 1 行目（closed）は，3 章で述べた closed program に対する解析を，servlet program に対して適用した結果である．servlet program では，Java の標準 library class である `ArrayList` class の object `xs` に着目する object を格納し（32 行目），取り出し（34 行目）ている．だが，servlet program は `ArrayList` class の定義を含んでいないため，closed program に対する解析では `xs` を介した object flow path を検出できない．

表の 2 行目（closed+lib）は，servlet program が参照しているすべての class 定義を加えた program に対して，closed program に対する解析を適用した結果である．新しく加えた class 定義が参照している class についても，その class 定義を再帰的に加えている．再帰的な class 参照の解決に利用したのは，JDK1.5 の標準 library（`rt.jar` および `jce.jar`）および Apache Tomcat 5.5.20 の servlet API library（`servlet-api.jar`）である．

これにより，`ArrayList` class の object `xs` を介した object flow が検出可能となり，12 行目で呼び出している method `getParameter` の戻り値 object が，23 行目の method `unsafe` に至る object flow path が得られる．一方，すべての class 参照を解決した結果，入力 program は 1,512 個の class からなる大きなものとなり，生成される *event* の総数（object flow graph の edge 数）は 598,371 個，解析時間は 835.7 秒と大幅に増加している．

表の 3 行目（closed+lib+fsc）は，5 章で述べた full subtype completion に基づく型の filtering を加えた結果である．生成される *event* の総数は 13,013 個（約 98%減），解析時間は 513.4 秒（約 39%減）と減少し，型の filtering の有効性が示されている．

表の 4 行目（open）は，4 章で述べた open program に対する解析を，servlet program に対して適用した結果である．外部 class である `ArrayList` class に関する作用を近似することで，前述した closed+lib の場合と同様に，12 行目から 23 行目に至る object flow path が得られる．解析時間は 0.6 秒で，servlet program に対して closed program に対する解析を適用した場合とほぼ同程度であり，上記の標準 library 等を用いた場合に比べるときわめて効率的である．

表の 5 行目（open+fsc）は，open program に対する解析に full subtype completion に基づく型の filtering を加えた結果である．生成される *event* の総

表 1 評価結果
Table 1 Evaluation result.

analysis	#class	#pred	time(s)	NEW	AS	PF	GF	PS	GS	CALL	RET	OF	IF	path
closed	7	1,548	0.6	1	0	0	0	0	0	1	1	0	0	no
closed+lib	1,512	1,231,184	835.7	1	420	44,422	362,722	13	44	131,977	58,722	0	0	yes
closed+lib+fsc	1,512	1,231,184	513.4	1	121	2,241	2,900	2	10	7,155	583	0	0	yes
open	7	1,548	0.6	1	0	0	0	0	0	10	2	9	16	yes
open+fsc	7	1,548	0.6	1	0	0	0	0	0	2	2	5	6	yes

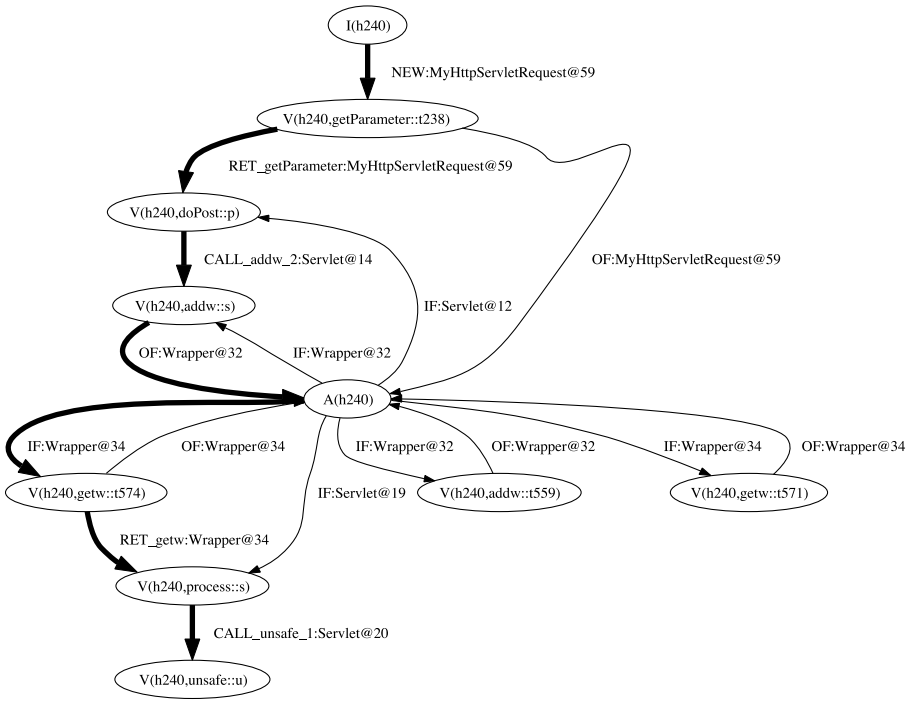

図 20 Servlet program に対する Object flow graph
Fig. 20 Object flow graph for servlet program.

数は 16 個（約 58%減）で，より compact な object flow graph が得られている（**図 20**．太線は最も正確な object flow path）．一方，full subtype completion でなく DCA subtype completion を用いても，結果は変化しなかった．これは，解析対象 program が class の継承関係をほとんど持たないためである．複雑な継承関係を持つ実際的な program では DCA subtype completion の効果が現れると予想されるが，詳細な評価は今後の課題である．

なお，open program に対する解析を適用する場合，stub の記述はより簡略化することができる．すなわち，stub 中で解析対象の servlet class を生成し，その `doPost` method を呼び出す記述は不要であり，servlet の解析に関連する object を生成し，それを escape させる（たとえば static field へ代入する）だけでよい．これは stub が解析対象 class に依存しないことを意味しており，解析を自動化するうえで有用である．

8. 関 連 研 究

本章では，提案した object flow 解析と関連の深い分野の従来研究について述べる．

Java program に対する pointer 解析や escape 解析については，これまで数多くの研究がなされてきている[6),8),27),31),32)]．ここでは，本論文と特に関連の深い研究について述べる．本論文で示した pointer 解析の特徴は，open program の解析が可能であること，および deductive system により定式化されていることの 2 点である．

本研究に最も影響を与えているのは，Whaley らの研究[32)] である．Whaley らは，Java program に対する pointer 解析を datalog program の形式で定式化している．datalog program は ROBDD（Reduced

Ordered Binary Decision Diagram)[7] 上の演算に変換し，効率的に実行することが可能である．Whaley らは，ROBDD への変換を自動的に行う library bddbddb を開発し公開している[24]．本論文で一貫して用いた deductive system は datalog と密接な関連があり，すべての解析を datalog program として記述し，実行することは容易である．

Whaley らの pointer 解析は context-sensitive なものであり，本論文の pointer 解析に比べて精度は高い．一方，彼らの解析は closed program を対象としており，open program に対する健全な解析ではない．たとえば，図 19 の servlet program において，23 行目の仮引数 `u` は 12 行目で呼び出す method `getParameter` の返す object を指示するが，彼らの解析では program 外部との間の object の流出および流入を考慮しないため，この事実を求めることができない．また，彼らの解析も型情報に基づく type filtering を実施しているが，class の継承関係に関する完全な情報が存在していることを前提としているため，open program を解析する際に誤った filtering を行ってしまう可能性がある．たとえば，class `A`，`B`，`C` があり，`A` が `B`，`B` が `C` を継承している program において，解析対象に `B` が含まれていない場合，`A` が `C` を間接的に継承しているという関係が認識されず，`C` 型の変数が `A` 型の object を指示する可能性が排除されてしまう．最後に，彼らの解析では「ある pointer がある object を指示しうる」という結果は得られるが，我々の解析のように「その pointer がどういう経緯でその object を指示するようになったか」という過程に関する情報を得ることはできない．この過程情報は，たとえば 6 章で述べたような応用を可能とする点で有用である．

context-sensitive な解析が行えるという Whaley らの利点と，open program を解析でき，object の流れを求めることができるという我々の利点はほぼ直交しており，これらをともに備えるような解析の設計は今後の課題である．

Livshits らは，taint 解析を用いて Java program の脆弱性を検出する手法を示している[20]．彼らの解析は，PQL（Program Query Language）と呼ばれる記述言語[21] に基づき定式化されている．PQL は Java program の抽象的な実行 trace に対する問合せ言語である．実行 trace は event の系列であり，各 event は method 呼び出しおよび復帰，object の生成，field への代入等を表す．event を用いた定式化は本論文と共通しているが，PQL では semantic value にのみ着目し program 中の変数は捨象されているため[⋆1]，本論文のように object を指示する主体の変化に基づき object flow を定義することはできない．また Livshits らの解析は，Whaley らと同様，closed program を対象としており，open program に対する健全な解析ではない．本論文で定義した object flow graph に対して，PQL と同様の query language を定義することは可能であり，object flow を様々な用途で利用するのに有用だと思われる．

Besson らは，Java の subset に対する modular な class 解析を定式化している[5]．modular な解析とは，program を module 単位で解析し，その結果をもとに program 全体の解析を行うもので，大規模な program 全体を 1 度に解析するのに比べて解析の計算量を抑えることができる．彼らの解析は，Cousot らの示した modular な解析の一般的な framework[10] を Java の class 解析という文脈に適用したものになっている．各 module は class の集合であり，本論文での open program に相当する．class 解析とは，program 中の式が実行時に持ちうる型（class）の集合を求めるもので，本論文の解析では，各変数の指示先となる object の型の集合を求めることに相当する．

彼らの解析は，本質的には pointer 解析であり，object の流れを求めるものではない．また，最終的に program 全体を解析することが目的のため，open program からの object の流出・流入の概念はない．

彼らの定式化の大きな特徴は，program の syntactic な要素がすべて独立した predicate として表現されていることである．たとえば，program 中の変数 x に対して，その指示先を表す predicate $c.m.x$ が定義される．ここで c，m は x の属する class および method である．この表現に基づき，open program の解析における外部の近似は，「外部で値を定義しうるすべての predicate は，その operand の値によらずつねに真になる」という形で行われる．本論文における近似との精度比較は，表現が大きく異なるため難しく，今後の課題である．

program から，ある program 点における変数の値に影響を及ぼす部分を抽出する技術は program slicing と呼ばれている[30]．本論文で提案した object flow graph は，特定の object に関するある種の program slice を定義しているとみることもできる．たとえば，図 1 の program から生成した object flow graph（図 11）に現

⋆1 PQL の query 中の変数は，program 変数ではなく，semantic value を表す logic 変数である．

れる行番号の集合 $\{6, 8, 9, 15, 22, 23\}$ は，6 行目の変数 x からの forward slice と 15 行目の class field f からの backward slice の intersection（program chop [14),22)] とも呼ばれる）に一致する．ただし，提案手法は object の流れのみを扱うため，整数型等の基本型の field や変数の参照については，program model 生成の段階で捨象されており，slice としても現れない[⋆1]．

Fortran 等の手続き型言語で書かれた program から slice を抽出する方法としては，手続き内の各文の依存関係を表す program dependence graph（PDG）および手続き間にまたがる依存関係を表す system dependence graph（SDG）に基づく手法がよく知られている[13)]．また，これらを C++や Java のような object 指向言語に拡張する方法も多数提案されている[15),18),28)]．これらの graph が構文要素（主に文）を node とし，構文要素間の関係に着目するのに対し，object flow graph は，pointer 変数や（field を介した）object 間の指示関係という動的な関係に着目する点で観点が異なっている．SDG に基づく slicing では，事前に別名解析，依存解析，class 解析，callgraph の生成，method の副作用解析を行ったうえで，静的に参照先の型の決まらない変数（subclass を持つ class 型の変数や interface 型の変数）や呼び出し先の決まらない method 呼び出し（instance method や interface method の呼び出し）をすべての可能な場合に展開し，Java の動的な言語機能を静的な形で graph 上に表現していく処理が必要になる．これに対し，提案手法では object の流れに特化することで，着目する object に関する部分 slice を抽出する軽量な（実装の容易な）手段を提供している．object flow graph では別名の問題が自然に解決されるほか，slice とともに object の流れが状態遷移の系列として得られるという利点を持つ．また，open program の外部への object の流出および外部からの流入についても，4 章で示したように，対応する状態を導入することで容易に表現可能である．

9. おわりに

本論文では，Java program における object の流れの表現方法およびその解析方法を提案した．

提案手法により，program 中の object がどういう過程を経て生成点から使用点にたどりつくのかを統一的に表現することが可能となる．解析は deductive system によって構成されており，datalog 等の論理型言語を用いて容易に実現可能である．また，open program を解析することが可能であり，着目する object の生成点が含まれている場合，健全な解析結果を得るために stub を必要としない．これらは，解析を実用化する際の大きな利点である．

本論文では，object flow およびその解析の基本的な概念を示すにとどまった．今後追求すべき主な研究課題を以下にあげる．

Java 特有の言語機能への対応 本論文では，method をまたがる例外，thread，class の動的 load，reflection 等の言語機能の扱いについて示していない．このうち，例外については，method が正常値と例外値の 2 値を返すと見なすことで，通常の method 呼び出しと同様に扱うことができる．そのほかについては，object flow の定義の拡張や環境に対する前提の強化が必要になると思われる．

解析精度の向上 object flow graph は，pointer 解析の精度に大きく影響される．本論文では context insensitive な pointer 解析を用いたが，context sensitive な解析を用いることで，より正確な object flow graph を得ることが可能になる．

評価 本論文の 6 章では，提案手法を Java の taint 解析に応用する方法について示したが，その実装や実 program における評価は行っていない．また，5 章で示した解析精度の改善手法についても，その実用上の効果の評価が必要である．

object flow graph の応用 提案した object flow graph は object の振舞いに関する豊富な情報を保持している．ここから文法圧縮[16)] 等の手法を用いて，program の bug 検出や最適化可能な振舞い pattern を抽出する手法の研究は興味深い課題である．圧縮は大きな object flow graph の要約にも役立つ．

謝辞 非常に有益なコメントをくださった査読者の方々に感謝いたします．また本研究に関して議論いただいたシステム開発研究所の西山博泰氏に感謝いたします．

⋆1 object を箱，その（基本型）field の値を中身と考えると，提案する object flow 解析は，中身ではなく箱の流れの解析といえる．

参考文献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, techniques and tools*, Addison-Wesley (1986).
2) Apache Tomcat (1999). http://tomcat.apache.org/
3) Apt, K.R., Blair, H.A. and Walker, A.: Towards a theory of declarative knowledge, *Foundations of deductive databases and logic pro-*

gramming, San Francisco, CA, USA, pp.89–148, Morgan Kaufmann Publishers Inc. (1988).
4) Ball, T., Naik, M. and Rajamani, S.K.: From Symptom to Cause: Localizing Errors in Counterexample Traces (2003).
5) Besson, F., Jensen, T. and Spoto, F.: Modular Class Analysis with Datalog, *Proc. 10th International Symposium on Static Analysis* (2003).
6) Blanchet, B.: Escape analysis for Java: Theory and practice, *ACM Trans. Prog. Lang. Syst.*, Vol.25, No.6, pp.713–775 (2003).
7) Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation, *IEEE Trans. Comput.*, Vol.C-35, No.8, pp.677–691 (1986).
8) Choi, J.D., Gupta, M., Serrano, M.J., Sreedhar, V.C. and Midkiff, S.P.: Stack Allocation and Synchronization Optimizations for Java using Escape Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.25, No.6, pp.876–910 (2003).
9) Clarke, E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, The MIT Press (2000).
10) Cousot, P. and Cousot, R.: Modular Static Program Analysis, *Proc. Conference on Compiler Construction*, pp.159–178 (2002).
11) Eluard, M. and Jensen, T.: Secure Object Flow Analysis for Java Card, *Proc. 5th Smart Card Research and Advanced Application Conference*, pp.97–110 (2002).
12) Hasti, R. and Horwitz, S.: Using Static Single Assignment Form to Improve Flow-insensitive Pointer Analysis, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.97–105 (1998).
13) Horwitz, S., Reps, T. and Binkley, D.: Interprocedural Slicing using Dependence Graphs, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.35–46 (1988).
14) Jackson, D. and Rollins, E.J.: Chopping: A Generalization of Slicing, Technical ReportCS-94-169, Carnegie Mellon University (1994).
15) Larsen, L. and Harrold, M.J.: Slicing Object-Oriented Software, *International Conference on Software Engineering*, pp.495–505 (1996).
16) Larus, J.R.: Whole Program Paths, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.259–269 (1999).
17) Leroy, X.: Java bytecode verification: algorithms and formalizations, *Journal of Automated Reasoning*, Vol.30, No.3–4, pp.235–269 (2003).
18) Liang, D. and Harrold, M.J.: Slicing Objects using System Dependence Graphs, *International Conference on Software Maintenance*, pp.358–367 (1998).
19) Lindholm, T. and Yellin, F.: *The Java Virtual Machine Specification, 2nd Edition*, Addison Wesley (1999).
20) Livshits, V.B. and Lam, M.S.: Finding Security Errors in Java Programs with Static Analysis, *Proc. 14th USENIX Security Symposium*, pp.271–286 (2005).
21) Martin, M., Livshits, V.B. and Lam, M.S.: Finding Application Errors and Security Flaws Using PQL: A Program Query Language, *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.365–383 (2005).
22) Reps, T. and Rosay, G.: Precise Interprocedural Chopping, *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.41–52 (1995).
23) Salcianu, A.: Pointer Analysis and its Applications for Java Programs, SM thesis, Massachusetts Institute of Technology (2001).
24) Stanford University (2004). http://suif.stanford.edu/bddbddb
25) Stark, R.F., Borger, E. and Shmid, J.: *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag (2001).
26) Ullman, J.D.: *Principles of Database and Knowledge-base Systems: Volume II*, Computer Science Press (1989).
27) Vivien, F. and Rinard, M.C.: Incrementalized Pointer and Escape Analysis, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.35–46 (2001).
28) Walkinshaw, M., Roper, M. and Wood, M.: The Java system dependence graph, *IEEE International Workshop on Source Code Analysis and Manipulation*, pp.55–64 (2003).
29) Weihl, W.E.: Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables, *Symposium on Principles of Programming Languages*, pp.83–94 (1980).
30) Weiser, M.: Program Slicing, *International Conference on Software Engineering*, pp.439–449 (1981).
31) Whaley, J. and Lam, M.S.: An Efficient Inclusion-based Points-to Analysis for Strictly-typed Languages, *International Static Analysis Symposium*, pp.180–195 (2002).
32) Whaley, J. and Lam, M.S.: Cloning-based Context-sensitive Pointer Analysis Using Binary Decision Diagrams, *Proc. ACM SIGPLAN Conference on Programming Lan-*

guage Design and Implementation, pp.131–144 (2004).
33) 情報処理推進機構：情報セキュリティ白書 (2006). http://www.ipa.go.jp/security/vuln/documents/2005/ISwhitepaper2006.pdf
34) 森下真一：知識と推論，共立出版 (1994).

付　　録

A.1 補題 4.2 の証明

証明 $M(E[P]) \vdash ptv(v,h)$, $M(E[P]) \vdash pth(h_1,fn,h_2)$, $M(E[P]) \vdash calling(i,m)$ の導出に関する帰納法により示す.

Case $P1$

規則の形より

A1. $M(E[P]) \vdash ptv(v,h)$.

A2. $M(E[P]) \vdash new(i,v,h)$.

が成り立つ.

(a) の証明.

1. $v \in Var(P)$
   (a) の前提
2. $new(i,v,h) \in M(P)$
   A2，1，定義 4.3
3. $M(P) \vdash ptv(v,h)$
   $P1$，2
4. $h \in Obj(P)$
   2
5. $M(P) \vdash ptv(v,\beta_h(P)(h))$
   2，3，定義 4.4

(d) の証明.

1. $v_1 \notin Var(P)$
   (d) の前提
2. $new(i,v,h) \notin M(P)$
   1，定義 4.3
3. $h \notin Obj(P)$
   2
4. $\beta_h(P)(h) = \mathrm{h}_0$
   3，定義 4.4
5. $M(P) \vdash escape(\beta_h(P)(h))$
   $E1$，4

Case $P2$

規則の形より

A1. $M(E[P]) \vdash ptv(v_1,h)$.

A2. $M(E[P]) \vdash assign(i,v_1,v_2)$.

A3. $M(E[P]) \vdash ptv(v_2,h)$.

が成り立つ.

(a) の証明.

1. $v_1 \in Var(P)$
   (a) の前提
2. $assign(i,v_1,v_2) \in M(P)$
   A2，1，定義 4.3
3. $v_2 \in Var(P)$
   2
4. $M(P) \vdash ptv(v_2,\beta_h(P)(h))$
   IH (a)，A3，3
5. $M(P) \vdash ptv(v_1,\beta_h(P)(h))$
   $P2$，2，4

(d) の証明.

1. $v_1 \notin Var(P)$
   (d) の前提
2. $assign(i,v_1,v_2) \notin M(P)$
   1，定義 4.3
3. $v_2 \notin Var(P)$
   2
4. $M(P) \vdash escape(\beta_h(P)(h))$
   IH (d)，3，A3

Case $P3$

規則の形より

A1. $M(E[P]) \vdash ptv(v_2,h_2)$.

A2. $M(E[P]) \vdash getfield(i,v_1,fn,v_2)$.

A3. $M(E[P]) \vdash ptv(v_1,h_1)$.

A4. $M(E[P]) \vdash pth(h_1,fn,h_2)$.

が成り立つ.

(a) の証明.

1. $v_2 \in Var(P)$
   (a) の前提
2. $getfield(i,v_1,fn,v_2) \in M(P)$
   A2，1，定義 4.3
3. $v_1 \in Var(P)$
   2
4. $M(P) \vdash ptv(v_1,\beta_h(P)(h_1))$
   IH (a)，A3，3
5. $M(P) \vdash pth(\beta_h(P)(h_1),fn,\beta_h(P)(h_2))$
   IH (b)，A4
6. $M(P) \vdash ptv(v_2,\beta_h(P)(h_2))$
   $P3$，2，4，5

(d) の証明.

1. $v_2 \notin Var(P)$
   (d) の前提
2. $getfield(i,v_1,fn,v_2) \notin M(P)$
   1，定義 4.3
3. $v_1 \notin Var(P)$
   2

4. $M(P) \vdash escape(\beta_h(P)(h_1))$
   IH（d），3，A3
5. $M(P) \vdash pth(\beta_h(P)(h_1), fn, \beta_h(P)(h_2))$
   IH（b），A4
6. $M(P) \vdash escape(\beta_h(P)(h_2))$
   $E2$，4，5

Case $P4$

規則の形より

A1. $M(E[P]) \vdash pth(h_1, fn, h_2)$.
A2. $M(E[P]) \vdash putfield(i, v_1, fn, v_2)$.
A3. $M(E[P]) \vdash ptv(v_1, h_1)$.
A4. $M(E[P]) \vdash ptv(v_2, h_2)$.

が成り立つ．

（a）の証明．$putfield(i, v_1, fn, v_2) \in M(P)$ かどうかで場合分けを行う．

（a1）

1. $putfield(i, v_1, fn, v_2) \in M(P)$
   case split
2. $v_1, v_2 \in Var(P)$
   1
3. $M(P) \vdash ptv(v_1, \beta_h(P)(h_1))$
   IH（a），A3，2
4. $M(P) \vdash ptv(v_2, \beta_h(P)(h_2))$
   IH（a），A4，2
5. $M(P) \vdash pth(\beta_h(P)(h_1), fn, \beta_h(P)(h_2))$
   $P4$，1，3，4

（a2）

1. $putfield(i, v_1, fn, v_2) \notin M(P)$
   case split
2. $v_1, v_2 \notin Var(P)$
   1
3. $M(P) \vdash escape(\beta_h(P)(h_1))$
   IH（d），2，A3
4. $M(P) \vdash escape(\beta_h(P)(h_2))$
   IH（d），2，A4
5. $M(P) \vdash pth(\beta_h(P)(h_1), fn, \beta_h(P)(h_2))$
   $E9$，3，4

Case $P5$

規則の形より

A1. $M(E[P]) \vdash ptv(v_1, h)$.
A2. $M(E[P]) \vdash getstatic(i_1, cn, fn, v_1)$.
A3. $M(E[P]) \vdash putstatic(i_2, cn, fn, v_2)$.
A4. $M(E[P]) \vdash ptv(v_2, h)$.

が成り立つ．

（a）の証明．$putstatic(i_2, cn, fn, v_2) \in M(P)$ かどうかで場合分けを行う．

（a1）

1. $putstatic(i, cn, fn, v_2) \in M(P)$
   case split
2. $v_1 \in Var(P)$
   （a）の前提
3. $getstatic(i_1, cn, fn, v_1) \in M(P)$
   A2，2，定義 4.3
4. $v_2 \in Var(P)$
   1，定義 4.3
5. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH（a），A4，4
6. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   $P5$，1，3，5

（a2）

1. $putstatic(i, cn, fn, v_2) \notin M(P)$
   case split
2. $v_1 \in Var(P)$
   （a）の前提
3. $getstatic(i_1, cn, fn, v_1) \in M(P)$
   A2，2，定義 4.3
4. $v_2 \notin Var(P)$
   1
5. $M(P) \vdash escape(\beta_h(P)(h))$
   IH（d），A4，4
6. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   $E8$，3，5

（d）の証明．$putstatic(i_2, cn, fn, v_2) \in M(P)$ かどうかで場合分けを行う．

（d1）

1. $putstatic(i, cn, fn, v_2) \in M(P)$
   case split
2. $v_2 \in Var(P)$
   1
3. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH（a），2，A4
4. $M(P) \vdash escape(\beta_h(P)(h))$
   $E5$，1，3

（d2）

1. $putstatic(i, cn, fn, v_2) \notin M(P)$
   case split
2. $v_2 \notin Var(P)$
   1
3. $M(P) \vdash escape(\beta_h(P)(h))$
   IH（d），2，A4

Case $P6$

規則の形より

A1. $M(E[P]) \vdash calling(i, m)$.
A2. $M(E[P]) \vdash callstatic(i, cn, mn, dn)$.
A3. $M(E[P]) \vdash dispatch(m, cn, mn, dn)$.

が成り立つ.

(c) の証明.

1. $i \in PP(P)$
   (c) の前提
2. $m \in Method(P)$
   (c) の前提
3. $callstatic(i, cn, mn, dn) \in M(P)$
   1, A2, 定義 4.3
4. $M(P) \vdash dispatch(m, cn, mn, dn)$
   2, A3, 補題 4.1
5. $M(P) \vdash calling(i, m)$
   P6, 3, 4

(e) の証明.

1. $i \in PP(P)$
   (e) の前提
2. $m \notin Method(P)$
   (e) の前提
3. $callstatic(i, cn, mn, dn) \in M(P)$
   1, A2, 定義 4.3
4. $M(P) \vdash \neg dispatch(m, cn, mn, dn)$
   補題 4.1, 2, A3
5. $M(P) \vdash callaux(i)$
   E10, 3, 4

Case P7

規則の形より

A1. $M(E[P]) \vdash calling(i, m)$.
A2. $M(E[P]) \vdash calldynamic(i, cn_1, mn, dn)$.
A3. $M(E[P]) \vdash actual(i, 1, v)$.
A4. $M(E[P]) \vdash ptv(v, h)$.
A5. $M(E[P]) \vdash tyh(h, cn_2)$.
A6. $M(E[P]) \vdash dispatch(m, cn_2, mn, dn)$.

が成り立つ.

(c) の証明.

1. $i \in PP(P)$
   (c) の前提
2. $m \in Method(P)$
   (c) の前提
3. $calldynamic(i, cn_1, mn, dn) \in M(P)$
   1, A2, 定義 4.3
4. $M(P) \vdash dispatch(m, cn_2, mn, dn)$
   2, A6, 補題 4.1
5. $actual(i, 1, v) \in M(P)$
   1, A3
6. $v \in Var(P)$
   5
7. $M(P) \vdash ptv(v, \beta_h(P)(h))$
   IH (a), A4, 6
8. $M(P) \vdash tyh(\beta_h(P)(h), \beta_c(P)(cn_2))$
   A5, 補題 4.1
9. $cn_2 \in RefClass(P)$
   4, 補題 4.1
10. $\beta_c(P)(cn_2) = cn_2$
    9, 定義 4.4
11. $M(P) \vdash tyh(\beta_h(P)(h), cn_2)$
    8, 9
12. $M(P) \vdash calling(i, m)$
    P7, 3, 5, 7, 11, 4

(e) の証明.

1. $i \in PP(P)$
   (e) の前提
2. $m \notin Method(P)$
   (e) の前提
3. $calldynamic(i, cn, mn, dn) \in M(P)$
   1, A2, 定義 4.3
4. $actual(i, 1, v) \in M(P)$
   1, A3, 定義 4.3
5. $v \in Var(P)$
   4
6. $M(P) \vdash ptv(v, \beta_h(P)(h))$
   IH (a), A4, 5
7. $M(P) \vdash tyh(\beta_h(P)(h), \beta_c(P)(cn_2))$
   補題 4.1, A5
8. $M(P) \vdash \neg dispatch(m, cn_2, mn, dn)$
   補題 4.1, 2, A3
9. $cn_2 \in RefClass(P)$
   case split
10. $\beta_c(P)(cn_2) = cn_2$
    定義 4.4, 9
11. $M(P) \vdash \neg dispatch(m, \beta_c(P)(cn_2), mn, dn)$
    8, 10
12. $M(P) \vdash callaux(i)$
    E11, 3, 4, 6, 7, 11
13. $cn_2 \notin RefClass(P)$
    case split
14. $\beta_c(P)(cn_2) = \mathrm{cn}_0$
    定義 4.4, 13
15. $M(P) \vdash \neg dispatch(m, \beta_c(P)(cn_2), mn, dn)$
    補題 4.1, 14
16. $M(P) \vdash callaux(i)$

$E11$, 3, 4, 6, 7, 15

Case $P8$

規則の形より

A1. $M(E[P]) \vdash ptv(v_1, h)$.
A2. $M(E[P]) \vdash calling(i, m)$.
A3. $M(E[P]) \vdash incoming(i, v_1)$.
A4. $M(E[P]) \vdash outgoing(m, p, v_2)$.
A5. $M(E[P]) \vdash ptv(v_2, h)$.

が成り立つ.

(a) の証明. $v_2 \in Var(P)$ かどうかで場合分けを行う.

(a1)

1. $v_2 \in Var(P)$
   case split
2. $v_1 \in Var(P)$
   (a) の前提
3. $incoming(i, v_1) \in M(P)$
   A3, 2, 定義 4.3
4. $outgoing(m, p, v_2) \in M(P)$
   A4, 1, 定義 4.3
5. $m \in Method(P)$
   4
6. $i \in PP(P)$
   3
7. $M(P) \vdash calling(i, m)$
   IH (c), A2, 6, 5
8. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH (a), A5, 1
9. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   $P8$, 7, 3, 4, 8

(a2)

1. $v_2 \notin Var(P)$
   case split
2. $outgoing(m, p, v_2) \notin M(P)$
   1
3. $m \notin Method(P)$
   2
4. $v_1 \in Var(P)$
   (a) の前提
5. $incoming(i, v_1) \in M(P)$
   A3, 4, 定義 4.3
6. $i \in PP(P)$
   5
7. $M(P) \vdash callaux(i)$
   IH (e), 6, 3
8. $M(P) \vdash escape(\beta_h(P)(h))$
   IH (d), A5, 1
9. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   $E7$, 8, 5, 7

(d) の証明. $outgoing(m, p, v_2) \in M(P)$ かどうかで場合分けを行う.

(d1)

1. $outgoing(m, p, v_2) \in M(P)$
   case split
2. $v_2 \in Var(P)$
   1
3. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH (a), 2, A5
4. $M(P) \vdash escape(\beta_h(P)(h))$
   $E3$, 1, 3

(d2)

1. $outgoing(m, p, v_2) \notin M(P)$
   case split
2. $v_2 \notin Var(P)$
   1
3. $M(P) \vdash escape(\beta_h(P)(h))$
   IH (d), 2, A5

Case $P9$

規則の形より

A1. $M(E[P]) \vdash ptv(v_1, h)$.
A2. $M(E[P]) \vdash calling(i, m)$.
A3. $M(E[P]) \vdash formal(m, k, v_1)$.
A4. $M(E[P]) \vdash actual(i, k, v_2)$.
A5. $M(E[P]) \vdash ptv(v_2, h)$.

が成り立つ.

(a) の証明. $v_2 \in Var(P)$ かどうかで場合分けを行う.

(a1)

1. $v_2 \in Var(P)$
   case split
2. $v_1 \in Var(P)$
   (a) の前提
3. $formal(m, k, v_1) \in M(P)$
   A3, 2, 定義 4.3
4. $m \in Method(P)$
   3
5. $actual(i, k, v_2) \in M(P)$
   A4, 1, 定義 4.3
6. $i \in PP(P)$
   5
7. $M(P) \vdash calling(i, m)$
   IH (c), A2, 6, 4

8. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH（a），A5，1
9. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   P9，7，3，5，8

（a2）

1. $v_2 \notin Var(P)$
   case split
2. $v_1 \in Var(P)$
   （a）の前提
3. $formal(m, k, v_1) \in M(P)$
   A3，2，定義 4.3
4. $M(P) \vdash escape(\beta_h(P)(h))$
   IH（d），1
5. $M(P) \vdash ptv(v_1, \beta_h(P)(h))$
   E6，3，4

（d）の証明．$actual(i, k, v_2) \in M(P)$ かどうかで場合分けを行う．

（d1）

1. $actual(i, k, v_2) \in M(P)$
   case split
2. $v_1 \notin Var(P)$
   （d）の前提
3. $formal(m, k, v_1) \notin M(P)$
   2，定義 4.3
4. $m \notin Method(P)$
   3
5. $i \in PP(P)$
   1
6. $M(P) \vdash callaux(i)$
   IH（e），4，5
7. $v_2 \in Var(P)$
   1
8. $M(P) \vdash ptv(v_2, \beta_h(P)(h))$
   IH（a），7，A5
9. $M(P) \vdash escape(\beta_h(P)(h))$
   E4，8，6

（d2）

1. $actual(i, k, v_2) \notin M(P)$
   case split
2. $v_2 \notin Var(P)$
   1，A4
3. $M(P) \vdash escape(\beta_h(P)(h))$
   IH（d），2，A5

□

A.2 定理 4.1 の証明

証明 以下では $PP(event) \in PP(P)$ である $event$ を内部 event，内部 event でない $event$ を外部 event と呼ぶ．

h の object flow path 中の連続する遷移

$$(st_1, event_1, st_2), (st_2, event_2, st_3)$$

において，$event_1$ が内部 event，$event_2$ が外部 event ならば，ある $v \in Var(P)$ が存在し，$st_1 = (h, v)_V$ であることがいえる．同様に $event_1$ が外部 event，$event_2$ が内部 event ならば，ある $v \in Var(P)$ が存在し，$st_2 = (h, v)_V$ である．

定理の前提として与えられる object flow path は，最初と最後の event が内部 event のため，内部 event の列で始まり，外部 event の列と内部 event の列を 0 回以上繰り返したものとなる．補題 4.4 より，$evCALL$ および $evRET$ 以外の内部 event は，P のもとで導出可能である．また $evCALL$（$evRET$）については呼び先（戻り先）method がそれぞれ P に属するならば，P のもとで導出可能である．一方，$evCALL$（$evRET$）の呼び先（戻り先）が外部 method の場合，続く event は外部 event となる．このとき，$evCALL$（$evRET$）に対応する $evOF$ が導出できること，および，続く外部 event 列の最後の event（内部 event 列との境界となる外部 event）に対応する $evIF$ が導出できることが前段の結果を用いて示せる．すなわち，外部 event 列による状態遷移は，$(h)_A$ への遷移および $(h)_A$ からの遷移によってつねに模倣できる．よって定理は成り立つ． □

A.3 補題 5.1 の証明

証明 $M(E[P]) \vdash subtype(cn_1, cn_2)$ の導出に関する帰納法による．

Case T1

1. $M(E[P]) \vdash subtyped(cn_1, cn_2)$
   T1 の前提
2. $cn_1 \in DefClass(P)$
   case split.
3. $subtyped(cn_1, cn_2) \in M(P)$
   1，2，program model の前提
4. $M(P) \vdash subtype(cn_1, cn_2)$
   T1，3
5. $cn_1 \notin DefClass(P)$
   case split.
6. $M(P) \vdash auxclass(cn_1)$
   S1，5
7. $M(P) \vdash subtype(cn_1, cn_2)$

$S2a$，6

Case $T2$

1. $cn \in RefClass(P)$
   case split.
2. $M(P) \vdash subtype(cn, cn)$
   $T2$，1
3. $cn \notin RefClass(P)$
   case split.
4. $cn \notin DefClass(P)$
   3
5. $M(P) \vdash auxclass(cn)$
   $S1$，4
6. $M(P) \vdash subtype(cn, cn)$
   $S2a$，5

Case $T3$

1. $M(E[P]) \vdash subtyped(cn_1, cn_2)$
   $T3$ の前提
2. $M(E[P]) \vdash subtyped(cn_2, cn_3)$
   $T3$ の前提
3. $M(P) \vdash subtyped(cn_1, cn_2)$
   IH，1
4. $M(P) \vdash subtyped(cn_2, cn_3)$
   IH，2
5. $M(P) \vdash subtyped(cn_1, cn_3)$
   $T3$，3，4

□

A.4 補題 5.2 の証明

証明 $M(E[P]) \vdash subtype(cn_1, cn_2)$ の導出に関する帰納法による．

Case $T1$

1. $M(E[P]) \vdash subtyped(cn_1, cn_2)$
   $T1$ の前提
2. $cn_1 \in DefClass(P)$
   case split.
3. $subtyped(cn_1, cn_2) \in M(P)$
   1，2，program model の前提
4. $M(P) \vdash subtype(cn_1, cn_2)$
   $T1$，3
5. $cn_1 \notin DefClass(P)$
   case split.
6. $M(P) \vdash auxclass(cn_1)$
   $S1$，5
7. $cn_2 \in DefClass(P)$
   case split.
8. $impossible$
   1，6，7，DCA
9. $cn_2 \notin DefClass(P)$
   case split.
10. $M(P) \vdash auxclass(cn_2)$
    $S1$，9
11. $M(P) \vdash subtype(cn_1, cn_2)$
    $S2b$，6，10

Case $T2$，$T3$

補題 5.1 の証明と同様にして示すことができる．

□

(平成 19 年 7 月 7 日受付)
(平成 19 年 12 月 4 日採録)

千代英一郎（正会員）

1973 年生．1999 年東京大学大学院工学系研究科情報工学専攻修士課程修了．同年日立製作所（株）入社．システム開発研究所にてコンパイラの研究開発に従事．