Osaka University Knowledge Archive

Title	オブジェクト指向プログラムにおけるエイリアス解析について
Author(s)	大畑, 文明; 井上, 克郎
Citation	情報処理学会研究報告. ソフトウェア工学研究会報告. 2000, 2000-SE-126(25), p. 57-64
Version Type	VoR
URL	https://hdl.handle.net/11094/50194
rights	ここに掲載した著作物の利用に関する注意 本著作物の著作権は情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。
Note	

Osaka University Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

Osaka University

ソフトウェア工学 126－8
(２０００．３．９)

オブジェクト指向プログラムにおける エイリアス解析について

大畑文明† 井上克郎†‡

† 大阪大学大学院基礎工学研究科
〒560-8531 大阪府豊中市待兼山町 1-3

‡ 奈良先端科学技術大学院大学情報科学研究科
〒630-0101 奈良県生駒市高山町 8916-5

エイリアスとは，引数の参照渡し・参照変数・ポインタを介した間接参照などで生じる，識別子が異なるが同じメモリ領域を指す可能性のある変数および式の集合である．しかし，既存のエイリアス解析手法は，解析結果の再利用を考慮しておらず，オブジェクト指向プログラムの持つ再利用性がエイリアス解析に生かされていない．本研究では，エイリアス解析結果の再利用・モジュール化を考慮した，オブジェクト指向プログラムに対するエイリアス解析手法を提案する．

Alias analysis for object-oriented programs

Fumiaki Ohata† and Katsuro Inoue†‡

† Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan

‡ Nara Institute of Science and Technology Department of Informatio Science
8916-5 Takayama-cho, Ikoma-City,
Nara 630-0101, Japan

In this paper, we propose an alias analysis method for object-oriented programs, which takes reusability and modularity of its results into account. Alias is a set of variables and expressions which possibly refer to the same location during execution. Since existing alias analysis methods do not reuse own results, we can not make use of reusability of object-oriented programs on alias analyses.

1 まえがき

プログラムのデバッグ支援に有効な方法として**プログラムスライス** *(Prgram Slice)*[1] がある．プログラムスライスとは，プログラム P 中のある文 s に対して，s で参照しているある変数 v の内容に影響を与えうる文の集合 Q のことである．現在では，デバッグ支援だけでなくテスト，保守，プログラム合成，プログラム理解などにも利用されている．このプログラムスライスの計算は，

Phase 1: データ依存解析
Phase 2: 制御依存解析
Phase 3: プログラム依存グラフ (PDG) の構築
Phase 4: PDG を利用しスライスを計算

という過程 [12] をたどるが，**Phase 1: データ依存解析**においては，各文でどの変数が定義・使用されているかが判明していなければならない．そのため，ポインタ (参照) が存在するプログラミング言語のデータ依存解析では，エイリアスの解析が前提となっている．

また，現在のプログラム開発環境において，C などの手続き型言語だけでなく，JAVA[8]・C++[9] 等いわゆるオブジェクト指向言語の利用が高まっている．その理由としては，オブジェクト指向モデルが持つ，拡張容易性・抽象化・カプセル化・モジュール性・再利用性・階層などが挙げられる．オブジェクト指向言語には，従来の手続き型言語にはないクラス・継承・動的束縛・ポリモルフィズムなど新しい概念が導入されており，それらに対応したエイリアス解析手法がいくつか提案されている [7].

しかし，既存のエイリアス解析手法は，解析アルゴリズムをオブジェクト指向言語に対応させたに過ぎず，解析結果そのものの再利用性・モジュール性は満たされていない．プログラムの大規模化・プログラムの再利用への感心が高まる現在，これらの特性を満たすエイリアス解析手法が望まれる．

本研究では，再利用性・モジュール性を考慮したエイリアスフローグラフによるオブジェクト指向プログラムに対するエイリアス解析手法の提案を行う．

以降，2 ではエイリアスについて簡単に紹介する．3 ではオブジェクト指向言語およびそれに対するエイリアス解析について紹介する．4 ではエイリアスフローグラフによるエイリアス解析手法の提案を行い，5 でまとめと今後の課題について述べる．

2 エイリアス

エイリアス *(Alias)* とは，引数の参照渡し・参照変数・ポインタを介した間接参照などで生じる，識別子が異なるが同じメモリ領域 (オブジェクト) を指す可能性のある変数および式の集合 (以降，単に**エイリアス集合** *(Alias Set)* と略す) である．本研究では，同一識別子で同じメモリ領域を指す変数・式もエイリアス集合に含めるとする．

エイリアス集合を導き出す**エイリアス解析** *(Alias Analysis)* は，大きく *FI* **エイリアス解析** *(Flow-Insensitive Alias Analysis)*(以降，FI 解析と略す)・*FS* **エイリアス解析** *(Flow-Sensitive Alias Analysis)*(以降，FS 解析と略す) の 2 つに分けることができる．以下，FS 解析・FI 解析を簡単に説明する．

FS エイリアス解析

FS エイリアス解析 [2, 3] とは，プログラム文の実行順を考慮したエイリアス解析手法をいう．一般に**到達エイリアス集合** *(Reaching Alias Set)* を利用して解析を行う．図 1 に変数 c(太枠部) の FS エイリアス (網掛部) とその計算に用いた到達エイリアス集合を示す．到達エイリアス集合は，エイリアス関係の成り立つ組の集合であり，文を解析するたびに更新される．各エイリアス組の要素は (文, オブジェクトへの参照) で構成されており，例えば文 2: a = **new** Integer(1) により，エイリアス組 {(2, a), (2, **new** Integer(1))} が生成される．また，文 6: c = a により，エイリアス組 {(4, c), (3, b), (3, **new** Integer(2))} が {(6, c), (2, a), (2, **new** Integer(1))} 変更されているのが分かる．文 7 の変数 c に関するエイリアスを求める場合，文 7 における到達定義集合から識別子 c を含むエイリアス組を探す．変数 c は {(6, c), (2, a), (2, **new** Integer(1))} に含まれており，網掛部が求めるエイリアスとなる．

```
1: Integer a, b, c;
2: a = new Integer (1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(a) プログラム

文	到達エイリアス集合
1	ϕ
2	ϕ
3	{(2, a), (2, **new** Integer(1))}
4	{(2, a), (2, **new** Integer(1))}, {(3, b), (3, **new** Integer(2))}
5	{(2, a), (2, **new** Integer(1))}, {(4, c), (3, b), (3, **new** Integer(2))}
6	〃
7	{(6, c), (2, a), (2, **new** Integer(1))}, {(3, b), (3, **new** Integer(2))}
...	〃

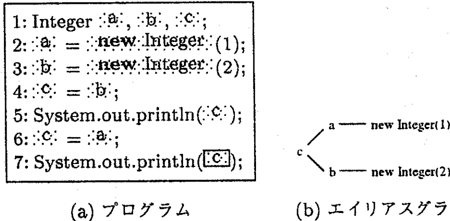
(b) 到達エイリアス集合

図 1: FS エイリアス解析

FI エイリアス解析

FI エイリアス解析 [4, 5] とは，プログラム文の実行順を考慮しないエイリアス解析手法をいう．一般にエイリアス (または，Point-to) グラフを利用して解析を行う．図 2 に変数 c(太枠部) の FI エイリアス (網掛部) およびエイリアスグラフを示す．エイリアスグラフの節点はメモリ領域を指しうる変数および式 (動的・静的いずれも含む) である．辺はエイリアス関係を表現している．また，複数辺による経路は節点間の間接のエイリアス関係を表わしている．例えば，文 4: c = b・文 6: c = a により，c〜b 間・c〜a 間に辺が引かれるため，a〜b 間

に経路が生じるため，{a, b, c} はエイリアス組と解釈される．c からの到達可能な節点が，文 7 の変数 c に関するエイリアスである．エイリアスグラフより，{a, b, **new** Integer(1), **new** Integer(2)} が求めるエイリアスとなる．



```
1: Integer a, b, c;
2: a = new Integer(1);
3: b = new Integer(2);
4: c = b;
5: System.out.println(c);
6: c = a;
7: System.out.println(c);
```

(a) プログラム

a —— new Integer(1)
c
b —— new Integer(2)

(b) エイリアスグラフ

図 2: FI エイリアス解析

FS 解析と FI 解析の比較

正確性: FS 解析はプログラム文の実行順を考慮しているため，FI 解析よりも正確性は高い

> **正確性** *(accuracy)* とは,「少なくとも実在するエイリアス集合は含まれているとして，どれだけ実在しないエイリアス集合を取り除くことができるかの程度」を表す．

コスト: FS 解析は FI 解析よりも時間計算量・空間計算量を必要とする

文献 [6] において，両者の詳細な比較がなされている．

Java とエイリアス

オブジェクト指向言語の 1 つである JAVA は C++ とは異なり，ポインタはなく参照変数のみ存在する．そのため解析結果が直接プログラム理解に結びつきやすく，プログラムスライスやコンパイラ最適化のためだけでなく，デバッグや保守においてもエイリアス解析の利用が期待できる．

図 3 に JAVA プログラムとその実行結果を示す．ユーザは文 A.print() の出力異常を認識し，参照変数 A に関するエイリアスを抽出を試みる．網掛部が文 A.print() の A に関するエイリアス，下線部がそのエイリアスが呼び出したメソッドである．この場合，文 e.add_salary(50) を add_salary(50) に変更することで期待動作を行うようになる．

3 オブジェクト指向プログラムにおけるエイリアス解析

3.1 オブジェクト指向言語

オブジェクト指向 *(Object-Oriented)* プログラミングでは，プログラムはオブジェクトの協調し合う集りとして構成され，そのオブジェクトはそれぞれ何らかのクラスのインスタンスであり，そのクラスは継承関係でまとめられたクラス階層の要素となる [10]．

```
class Employee {
  String name; int salary; Employee boss;
  Employee(String n, int s) {
    name = n; salary = s; boss = null;
  }
  void add_salary(int n) {salary += n; }
  void set_boss(Employee e) { boss = e; }
  void print() {
    System.out.println(name + " Salary:" + salary);
  }
}
class Manager extends Employee {
  Manager(String n, int s) {super(n, s);}
  void manage(Employee e) {
    e.set_boss(this); e.add_salary(50);
  }
}
class Office {
  public static void main(String args[]) {
    Employee A = new Employee("Mr.A", 700);
    Manager B = new Manager("Mr.B", 850);
    B.manage(A); A.print(); B.print();
  }
}
```

```
% java Office
Mr.A Salary: 800
Mr.B Salary: 850
```

図 3: JAVA プログラム

3.2 オブジェクト指向言語におけるエイリアス解析

既存のオブジェクト指向プログラムにおけるエイリアス解析は，表 1 に挙げた概念変更を含め，既に提案されている手続き型言語のエイリアス解析手法の拡張の形態となっている．以降，既存のエイリアス解析手法に関して，いくつかの視点から考察を行う．

表 1: オブジェクト指向言語への拡張

手続き型言語	オブジェクト指向言語
構造体	クラス
構造体の要素	属性
関数 (手続き)	メソッド
ポインタ変数	オブジェクトへの参照

■ FS 解析の問題　手続き (関数) が存在するプログラムの場合，到達エイリアス集合を用いた FS 解析 [7] では，すべての実行 (手続き呼び出し) 経路をたどりながら到達エイリアス集合を計算する．その際には，呼び出し側でのエイリアス集合を，手続き側で解釈可能なように識別子の付け換え (参照渡しの場合，エイリアス組に属する実引数を仮引数に置き換えるのもその 1 つである) をその都度行わねばならない．また，再帰呼び出しが存在する場合，エイリアス集合が収束するまで再帰経路を解析し続ける必要がある．これは，解析結果がエイリアス組の形式で保持されていることに起因している．つまり，ある時点でエイリアス組に変化が起ったとき，その変化の影響を受ける到達エイリアス集合を持つ文すべてにその変化を伝播

させ，到達エイリアス集合を更新しなければならないためである．

■**解析結果の再利用**　ある文sの到達エイリアス集合は，その文が含まれるプログラム全体の解析により導出されたものであるため，他の文tが変更されたとき，文sも再解析しなければならない場合が多く存在する．これは，エイリアス解析結果のモジュール性・独立性が満されていないことに起因する．解析結果のモジュール化による再利用は，頻繁に変更されるプログラムや再利用の可能性の低い単一プログラムでは直接の効果が現れにくく，更新頻度の少なく再利用性の高いライブラリに対する解析にのみ有効であると考えていた．しかし，このライブラリが持つ特性はオブジェクト指向プログラムにも共通している．オブジェクト指向プログラムでは，継承機能など，言語自身が再利用を考慮したものとなっているため，記述されたクラスの利用範囲が特定のプログラムにとどまらない．また，クラス階層の上位に存在するクラスは属性・メソッドが汎化されているため一度定義されると変更されることは少ない．そのため，クラスやメソッド単位で解析結果をモジュール化することにより，再計算コストの削減が期待できる．

■**同一クラスの異なるインスタンス属性の取り扱い**　オブジェクト指向プログラムでは，異なるオブジェクト間での状態(属性)と振舞い(メソッド)は独立であるため，図4の例では{x.s, A::s}，{y.s, A::s}はエイリアス組と言えるが，同一クラスのインスタンスがその内部情報を共有する(具体的には，{x.s, y.s, A::s}がエイリアス組となる)ことは正確性の低下につながる．このため，インスタン

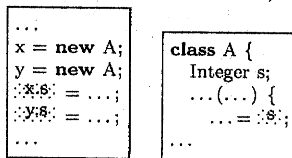


図 4: オブジェクト指向プログラム

スごとにクラス内部のエイリアス情報を持たせる手法が考えられるが，単純にインスタンスの数だけエイリアス情報を生成すると，多大な空間コストを要する．そこで，存在するエイリアス関係を

- 外部からの影響を受けないもの ...**内部エイリアス関係** *(Inner Alias Relation)*

  内部エイリアス関係は，例として，手続き型言語の手続き・関数，オブジェクト指向言語のメソッド・クラス内で成立する．

- 外部からの影響を受けるもの ...**外部エイリアス関係** *(Outer Alias Relation)*

に分け，内部エイリアス関係のみ前もって解析し情報を蓄え，外部エイリアス関係はエイリアス計算時に逐次導出する手法が考えられる．

3.3　提案する手法

そこで本研究では，先に述べた

- 再解析コストの軽減
- エイリアス解析結果のモジュール化 および それに伴う再利用性向上
- 内部エイリアス関係の表現

を目的とする，

- FS解析に基づくエイリアスフローグラフ(AFG)
- AFGを用いたエイリアス計算

の提案を行う．クラス・メソッド単位で到達エイリアス集合によるエイリアス解析を行い，クラスAFG・メソッドAFGをそれぞれ構築する．各AFGは独立したモジュールとして存在し，対応するクラス・メソッドが更新されない限り不変であり，二次媒体への記憶およびその再利用が可能である．AFGはクラス・メソッド内部のエイリアスに関する情報のみを保持しており，インスタンス独自のエイリアス関係やインスタンス間をまたぐエイリアス関係はユーザからの要求があった時点で逐次解析する．

4　エイリアスフローグラフ(AFG)

エイリアスフローグラフ *(Alias Flow Graph, AFG)* は，FSエイリアス関係を無向グラフで表現したものであり，FSエイリアス計算をグラフの到達問題に置き換える．AFGによるエイリアス解析手法は，大きく以下の5段階で構成させる．

Phase 1: オブジェクトへの参照の抽出
(AFG節点の生成)
Phase 2: 到達エイリアス集合を利用し，直接のエイリアス関係の抽出
(AFG辺の生成)
Phase 3: AFG構築
Phase 4: メソッド呼び出しグラフ(MFG)構築
Phase 5: AFGによるエイリアス計算

本節では，AFG構築・MFG構築・AFGによるエイリアス計算に分け，それぞれ説明する．今回提案するアルゴリズムはJAVAの参照変数によるエイリアスを対象としているが，ポインタ変数によるエイリアスにも適応可能であり，本節の最後で簡単に述べる．

```
1: Integer a = new Integer(0);
2: Integer b, c;
3: b = a;
4: c = b;
```

図 5: サンプルプログラム

Phase 1: AFG節点の生成

AFG標準節点

AFG節点 (AFG Node) は，文とオブジェクトへの参照の組である．ここでいう**オブジェクトへの参照**は，

(1) インスタンス生成式
(2) 参照変数

(3) ポインタ変数による間接参照式

のいずれかである．ただし (3) は，C や C++など ポインタを有する言語で用いる．また，これらの節点を特に**標準節点** *(Normal Node)* と呼ぶ．

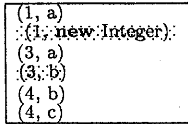

(1, a)
(1, **new** Integer)
(3, a)
(3, b)
(4, b)
(4, c)

図 6: AFG 節点

図 5 にサンプルプログラムを，図 6 に図 5 の AFG 節点を示す．文 3: b = a において，b は参照変数でありオブジェクトへの参照に該当する．よって，AFG 節点 (3, b) が生成される．同様に，文 1: Integer a = **new** Integer(0) においても，**new** Integer はインスタンス生成式であり，AFG 節点 (1, **new** Integer) が生成される．

AFG 特殊節点

手続きの存在しない単一プログラムにおいては，AFG 標準節点のみ用いることで AFG を構築することができる．しかし，オブジェクト指向プログラムではメソッド・属性が存在するため，

- 引数渡しによる，メソッド〜メソッド呼び出し元
- メソッドの戻り値による，メソッド〜メソッド呼び出し元
- 属性による，同一クラス (インスタンス) のメソッド〜メソッド

でのエイリアスの受け渡しを考慮しなければならない．そのため．表 2 に挙げる *AFG* **特殊節点** *(AFG Special Node)* を新たに追加する．

Phase 2: AFG 辺の生成

AFG **辺** *(AFG Edge)* は，2 つの AFG 節点間の，同一変数参照，代入文やパラメータの対応による直接のエイリアス関係を表す．複数の AFG 辺で構成された経路は，2 節点間の，推移的に成り立つ間接のエイリアス関係を表す．AFG 辺は到達エイリアス集合を利用して引かれる．

到達エイリアス集合

本手法で用いる到達エイリアス集合は，2 節の FS エイリアス解析で用いた到達エイリアス集合とは以下の点で異なる．

- 要素は，エイリアス組ではなく AFG 節点である．本手法では，エイリアス関係を AFG 辺および AFG 経路で表現するため，到達エイリアス集合内でエイリアス組を保持する必要はなく，後述する AFG 節点でよい
- 到達エイリアス集合は各文で保持されるのではなく，AFG 辺を引くためにのみ使われる

到達エイリアス集合の計算は 2 節の FS エイリアスの抽出アルゴリズムを基本とし，分岐文・繰り返し文・連続文など それぞれアルゴリズムを定義した．

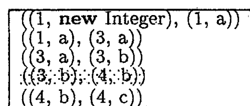

((1, **new** Integer), (1, a))
((1, a), (3, a))
((3, a), (3, b))
((3, b), (4, b))
((4, b), (4, c))

図 7: AFG 辺

図 7 に，図 5 の AFG 辺を示す．AFG 辺は先に述べた到達エイリアス集合を用いて引かれる．例えば 文 4: c = b においては，到達エイリアス集合は {(1, a), (3, b)} となっており．AFG 節点 (3, b)〜(4, b) 間に AFG 辺が引かれる．文 4 の解析後，到達エイリアス集合が更新され，新たに AFG 節点 (4, c) が追加され，{(1, a), (3, b), (4, c)} となる．

```
public class Calc {
  Integer i;
  public Calc() {
    i = new Integer(0);
  }
  public void inc() {
    i = new Integer
      (i.intValue() + 1);
  }
  public void add(int c) {
    i = new Integer
      (i.intValue() + c);
  }
  public Integer result() {
    return(i);
  }
}
```

```
class Test {
  Calc a, b;
  Integer c;
  Test() {
    a = new Calc();
    b = new Calc();
    a.inc();
    b.add(1);
    c = b.result();
  }
}
```

図 8: サンプルプログラム

インスタンス属性・インスタンスメソッドの表現

オブジェクト指向プログラムでは，インスタンス属性やインスタンスメソッドを表現するため「a.b.c」「d.e()」といった表記がなされる．このため，AFG 節点の拡張が必要となる．そのため，Method 節点・標準節点に，親節点・子節点に関する情報を追加する．具体的な例は図 9 に示す．

Phase 3: AFG の構築

AFG は，**Phase 1:**・**Phase 2:**で生成される AFG 節点・AFG 辺を用い，メソッド・クラスごとに構築される．各メソッドを解析して構築される**メソッド** *AFG(Method AFG)* は，メソッドが定義されたクラスの**クラス** *AFG(Class AFG)* に属する．図 8 にサンプルプログラムを，図 9 にその AFG を示す．実線の無向辺は 2 節点間のエイリアス関係を，破線の有向辺は節点の親子関係 (has-a 関係) を表している．

Phase 4: メソッド呼び出しグラフの構築

メソッド呼び出しグラフ *(Method Flow Graph, MFG)* とは，クラス中に存在する (同一インスタンス) メソッド間の呼び出し関係を有向グラフで表現したものである．*MFG* **節点** *(MFG Node)* はメソッドを表し，メソッド A がメソッド B を呼ぶ場合，*MFG* **辺** *(MFG Edge)* を節点 *A* から節点 *B* に引く．図 10 にプログラムおよびその MFG を示す．B クラスではメソッド p は定義されておらず，親クラス A のメソッド p が利用される．B クラスの MFG 構築の際には，B クラスが継承したメソッド A::p はメソッド q を呼び出しているが，それはメソッド B::q でありメソッド A::q ではないことに注意しなければならない．

表 2: 特殊節点

特殊節点	概要
Actual-Alias-in(AA-in)	実エイリアス引数 (実引数により，メソッドに渡されるエイリアス)
Formal-Alias-in(FA-in)	仮エイリアス引数 (仮引数により，メソッドに渡されるエイリアス)
Actual-Alias-out(AA-out)	実エイリアス引数 (実引数により，メソッド呼び出し元に渡されるエイリアス)
Formal-Alias-out(FA-out)	仮エイリアス引数 (仮引数により，メソッド呼び出し元に渡されるエイリアス)
Method-Alias-out(MA-out)	戻りエイリアス値 (戻り値により，メソッド呼び出し元に渡されるエイリアス)
Method	メソッド呼び出し (メソッドの戻り値により，メソッド呼び出し元に渡されるエイリアス，AA-in(out) 節点の親節点)
Instance-Alias-in(IA-in)	エイリアス属性 (属性により，メソッド内に渡されるエイリアス)
Instance-Alias-out(IA-out)	エイリアス属性 (属性により，メソッド外に渡されるエイリアス)

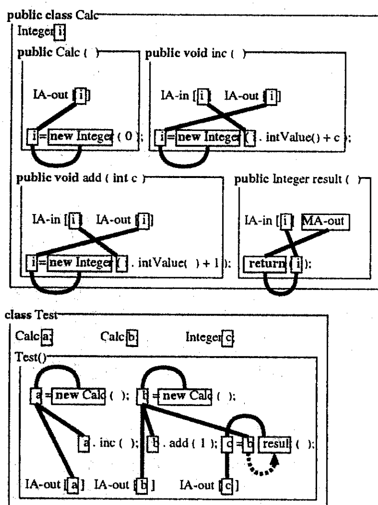


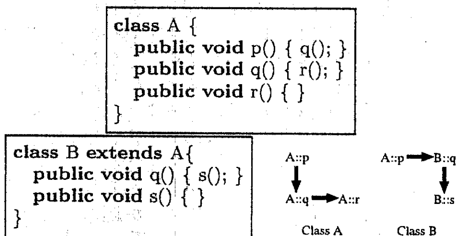
図 9: 図 8 の AFG

Phase 5: AFG を用いたエイリアス計算

AFG によるエイリアス計算は，AFG のグラフ到達問題に置き換えられる．あるオブジェクトへの参照 x のエイリアスを抽出する手順を基本方針とともに以下に示す．

基本方針 1: 親節点を持つ節点のエイリアスを導出する場合，まずその親節点のエイリアス計算問題を解決させる (これにより，エイリアス解析対象が特定インスタンスに限定される)

基本方針 2: MA-out や FA-in(out) 節点など，メソッド間をまたぐエイリアス関係の導出の際には，MFG を用いて対象 (呼び出し先・呼び出し元) メソッドを探索し，対応する Method や AA-in(out) 節点から AFG 辺をたどる



```
class A {
  public void p() { q(); }
  public void q() { r(); }
  public void r() { }
}
class B extends A{
  public void q() { s(); }
  public void s() { }
}
```


図 10: メソッド呼び出しグラフ (MFG)

Step 1: オブジェクトへの参照 x に対応する AFG 節点を X とする

Step 2: X を始点とし，新たな到達可能節点が見つからなくなるまで AFG 辺を推移的にたどる．その際，到達節点 C の種類に応じ，存在する AFG 辺をたどる以外に，以下のような処理を行う

親節点 P を持つ標準 (インスタンス属性) 節点:

(1) 節点 C に対応する属性を c とする
(2) 節点 P のエイリアス計算 (その結果を，$\mathcal{P}$ とする)
(3) $\mathcal{P}$ に存在するインスタンス生成式から，$\mathcal{P}$ の型を類推
(4) Object-Context($\mathcal{P}$) の計算
Object-Context($\mathcal{P}$) とは，$\mathcal{P}$ が子節点として持つ Method 節点の存在により導出される，$\mathcal{P}$ が直接もしくは間接的に呼び出す可能性のあるメソッド集合である．
直接呼び出されるメソッドとは，Method 節点に対応するメソッドを指す ($\mathcal{P}$ の類推型が複数クラスになる場合，対応するメソッドも複数になる可能性がある)．
間接的に呼び出されるメソッドとは，直接呼び出されるメソッドが内部で呼び出す同一インスタンスのメソッドのことを指し，MFG を用いて導出する．
(5) Object-Context($\mathcal{P}$) に属するメソッドの各メソッド AFG 内の IA-in[c]・IA-out[c] 節点から AFG 辺をたどる
(6) $\mathcal{P}$ の子節点のうち，識別子 c の標準節点から AFG 辺をたどる

親節点 P を持つ Method 節点:

(1) 節点 C に対応するメソッドを c とする
(2) 節点 P のエイリアス計算
(3) $\mathcal{P}$ に存在するインスタンス生成式から，$\mathcal{P}$ の型を類推
(4) Object-Context($\mathcal{P}$) の計算
(5) $\mathcal{P}$ の類推型が持つメソッド c に対応するメソッド AFG の MA-out 節点から AFG 辺をたどる

IA-in(out) 節点:

(1) 節点 C に対応する属性を c とする
(2) Object-Context(**this**) に属するメソッド

のメソッド AFG が持つ IA-out(in)[c] 節点から AFG 辺をたどる

AA-in(out) 節点:
(1) 節点 C の親節点である，Method 節点に対応するメソッドを P_A とする
(2) P_A に対応するメソッド AFG を G_A とする
(3) G_A の該当する FA-in(out) 節点から AFG 辺をたどる

FA-in(out) 節点:
(1) 節点 C を持つメソッド AFG を G_B，G_B に対応するメソッドを P_B とする
(2) Object-Context(**this**) に含まれており，かつ P_B を呼び出す メソッド P_A を MFG から抽出
(3) P_A に対応するメソッド AFG を G_A とする
(4) G_A の該当する AA-in(out) 節点から AFG 辺をたどる

MA-out 節点:
(1) 節点 C を持つメソッド AFG を G_B，G_B に対応するメソッドを P_B とする
(2) P_B を呼び出すメソッド P_A を MFG から抽出
(3) P_A に対応するメソッド AFG を G_A とする
(4) G_A が持つ (メソッド P_B を呼ぶ) メソッド節点から AFG 辺をたどる

Method 節点:
(1) 節点 C に対応するメソッド P_A のメソッド AFG を G_A とする
(2) G_A が持つ MA-out 節点から AFG 辺をたどる

Step 3: 到達可能な AFG 節点集合に対応するオブジェクトへの参照が，求める x のエイリアス

4.1 エイリアス計算例

エイリアス計算の例として，図 8 の参照変数 c(太枠部) のエイリアスの抽出手順を説明する．

(1) 参照変数 c に対応する AFG 節点から AFG 辺をたどり，Method 節点 result() に到達
(2) Method 節点 result() は親節点 b を持つため，親節点 b のエイリアス計算を行い，Method 節点のエイリアス計算に関与するインスタンスを特定する
  (1) 節点 b のエイリアス $\mathcal{B}$ を計算 (図 11)
  (2) $\mathcal{B}$ に存在するインスタンス生成式から節点 b の型を類推 [Calc クラス]
  (3) Object-Context($\mathcal{B}$) を計算 [{Calc::Calc(), Calc::add(**int** c)，Calc::result()}]
(4) 節点 b の型類推より，節点 b は Calc クラスのインスタンスへの参照であることから，Calc::result() の MA-out 節点から AFG 辺をたどる (図 12)
  (1) AFG 辺をたどり，IA-in[i] に到達
  (2) 属性 i のエイリアスに影響を与えうるメソッド，すなわち Object-Context(**this**)(= Object-Context($\mathcal{B}$)) の IA-out[i] 節点からエイリアス計算 (Object-Context($\mathcal{B}$) に含まれない Calc::inc() は対象から除外される)

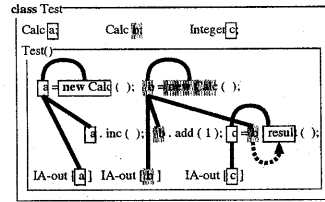


図 11: 図 8 の b に関するエイリアス (網掛部)

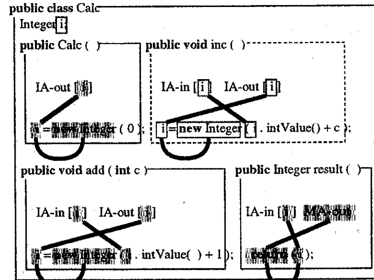


図 12: 図 8 の b.result() に関するエイリアス (網掛部)

図 13 に，c のエイリアス (網掛部) および Object-Context($\mathcal{B}$)(下線部) を示す．

4.2 ポインタ変数を持つ言語でのエイリアス解析

これまで，参照変数を仮定したアルゴリズムを記述したが，C・C++などポインタ変数を持つ言語では，ポインタ (特に n 階 ($n \geq 2$) ポインタ) による間接参照を考慮する必要がある．引数として n 階ポインタ変数を使用することで，たとえ値渡しであっても，手続き内で呼び出し元のエイリアス関係を変更可能であるためである．

そこで，参照変数では 1 変数あたり 1 個のエイリアス情報であったが，n 階ポインタ変数では 1 変数あたり n 個のエイリアス情報を用意する．図 14 は，C 言語で記述されたプログラムおよびそこから抽出されたエイリアス辺の集合を示している．手続き assign(**char** **y, **char** *x) は 2 階のポインタ変数 y を使用しており，実引数である&b について&b および b に関するエイリアス情報が AA-in として，仮引数 y について y および*y に関するエイリアス情報が FA-in として，手続き main()・手続き assign(**char** **y, **char** *x) にそれぞれ用意されている．また，b(*y) に関するエイリアス情報を呼び出し先に反映するため，AA-out(FA-out) が追加されている．

5 まとめと今後の課題

本研究では，オブジェクト指向言語に対する，再利用性・モジュール性を考慮したエイリアスフローグラフによるエイリアス解析手法 の提案を行った．また，今回紹介できなかったが，提案手法を JAVA

```
public class Calc {
  Integer i;
  public Calc() {
    i = new Integer(0);
  }
  public void inc() {
    i = new Integer
      (i.intValue() + 1);
  }
  public void add(int c) {
    i = new Integer
      (i.intValue() + c);
  }
  public Integer result() {
    return(i);
  }
}
```

```
class Test {
  Calc a, b;
  Integer c;
  Test() {
    a = new Calc();
    b = new Calc();
    a.inc();
    b.add(1);
    c = b.result();
  }
}
```

図 13: 図 8 の c に関するエイリアス (網掛部)

```
  void main() {
1:  char *a, *b;
2:  char c = ' ';
3:  a = &c;
4:  assign(&b, a);
5:  putc(*b);
  }
```

```
((3, &c), (3, a))
((3, a), (4, a))
((1, b), (4, b))
((4, &b), (4, AA-in[y]))
((4, b), (4, AA-in[*y]))
((4, a), (4, AA-in[x]))
((4, AA-out[*y]), (4, b))
((4, b), (5, b))
```

```
6: void assign(
      char **y,
      char *x) {
7:  *y = x;
  }
```

```
((6, FA-in[y]), (7, y))
((6, FA-in[*y]), (7, *y))
((6, FA-in[x]), (7, x))
((7, x), (7, *y))
((7, *y), (6, FA-out[*y]))
```

図 14: ポインタ変数を持つ言語 (C) での解析

を対象としたエイリアス解析ツールとして実装を行った．今後の課題としては，

- AFG データベースの構築
- 例外処理・スレッドへの対応

などが挙げられる．また，エイリアス解析ツールによる本手法の評価も行いたいと考えている．

謝辞 本研究は，栢森情報科学振興財団の補助を受けている．

参考文献

[1] M. Weiser, "Program Slicing," in Proceedings of the 5th International Conference on Software Engineering, pp.439–449, 1981.

[2] M. Enami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis int the presence of function pointers." in SIGPLAN'94 Conference on Programming Language Design and Implementation, pp.242–256, 1994.

[3] R. P. Wilson, and M. S. Lam, "Efficient context-sensitive pointer analysis for C Programs," in SIGPLAN'95 Conference on Programming Language Design and Implementation, pp.1–12, 1995.

[4] B. Steensgaard, "Points-to analysis in almost linear time." in 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, pp.32-41, 1996.

[5] M. Shapiro, and S. Horwitz, "Fast and accurate flow-insensitive point-to analysis," in 24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages, 1997.

[6] M. Hind, and A. Pioli, "An Empritical Comparison of Interprocedural Pointer Alias Analysis," in IBM Research Report #21058, 1997.

[7] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo, "Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing," in Proceedings of the 19th International Conference on Software Engineeringm, pp.433-443, 1997.

[8] J. Gosling, B. Joy, and G. Steele, 村上 雅章 [訳], "The Java 言語仕様"

[9] B. Stroustrup, "The C++ Programming Language(Third edition)," Addison-Wesley, 1997.

[10] G. Booch, "Object-Oriented Design with Application," The Benjamin/Cummings Pubrishinh Company, Inc, 1991.

[11] 片山, 土居, 鳥居 [監訳], "ソフトウェア工学大事典," 朝倉書店.

[12] S. Horwitz, and T. Reps, "The Use of Program Dependence Graphs in Software Engineering," In Proceedings of the 14th International Conference on Software Engineering, pp 392–411, 1992.