**修士学位論文**

題目

# オブジェクト指向プログラムにおける<br>エイリアス解析手法の提案と実現

指導教官
井上 克郎 教授

報告者
大畑 文明

平成 12 年 2 月 15 日

大阪大学 大学院基礎工学研究科
情報数理系専攻 ソフトウェア科学分野

平成 12 年度 修士学位論文

# オブジェクト指向プログラムにおける エイリアス解析手法の提案と実現

大畑 文明

## 内容梗概

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，同じメモリ領域を指す可能性のある式間の同値関係を表し，プログラム依存関係解析・コンパイラ最適化・プログラム理解などに用いられる．

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・C++などいわゆるオブジェクト指向言語の利用が高まっている．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必要である．

しかし，今まで提案されてい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は，既存の手続き型言語の解析法を単純に拡張しているだけで，プログラムの一部分のみの変更でライブラリを含めた全体の再解析が必要など，その方法に問題がある．また，詳細なエイリアス解析を行なうには，解析コストの増大は避けられない．

本研究では，Flow-Sensitive 解析に基づくエイリアスフローグラフ (Alias Flow Graph, AFG) 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を提案する．クラス・メソッド単位で AFG を構築しそれらを必要に応じ利用することで，正確性向上 および解析結果の再利用・モジュール化を実現する．また，本手法を Java を対象とするエイリアス解析ツールとして実装を行い，その有効性を確認する．

### 主な用語

エイリアス (Alias)
オブジェクト指向プログラム (Object-oriented program)
Java

# 目 次

## 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)*[15] 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 $\mathcal{Q}$ 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:** データ依存関係解析

**Phase 2:** 制御依存関係解析

**Phase 3:** プログラム依存グラフ (Program DependenceGraph, PDG) の構築

**Phase 4:** PDG を利用しスライスを計算

という過程 [19] をたどるが，**Phase 1: データ依存関係解析**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関係解析では，エイリアスの解析が前提となっている．エイリアスとは同じ メモリ領域 (オブジェクト) を指す可能性のある式間の同値関係であり，このエイリアスの存在により，プログラムの異なるスコープ中の異なる識別子が同じメモリ領域を指す可能性があるため，エイリアス解析なしにデータ依存関係を抽出することはできない．

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[7]・C++[3] 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 [16].

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 (Alias Flow Graph, AFG) によるエイリアス解析手法 および オブジェクト指向言語に対するその拡張 の提案を行う．また，提案手法を Java 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．

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 および オブジェクト指向言語に対する提案手法の拡張を行う．5でJavaを対象としたエイリアス解析ツールの紹介を行い，6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，同じメモリ領域 (オブジェクト) を指す可能性のある式間の同値関係である．

エイリアス解析の利用分野としては，まえがきで述べたプログラムスライスのほかに，コンパイラの最適化技法 [17] がある．図 1 は C 言語で書かれたプログラム例であるが，解析

```
int a[], b[];
void f(int i, int j) {
  int *p, *q;
  int x, y;
  p = &a[i];
  q = &b[j];
  x = *(q + 3);
  *p = 5;
  y = *(q + 3);
  g(x, y);
}
```

```
int a[], b[];
void f(int i, int j) {
  int *p, *q;
  int x;
  p = &a[i];
  q = &b[j];
  x = *(q + 3);
  *p = 5;
  g(x, x);
}
```

図 1: コンパイラの最適化

によりポインタ p・q はエイリアスを生成しないと判断でき，文 y = *(q + 3) の省略 および 変数 y の変数 x への置き換えが可能となる．

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 [10, 18] 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 2 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** Interger(1) により，エイリアス組 {(2, a), (2, **new** Interger(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** Interger(2))} が {(6, c), (2, a), (2, **new** Interger(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** Interger(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 Interger(1);
3: b = new Interger(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 | $\phi$ |
| 2 | $\phi$ |
| 3 | {(2, a), (2, **new** Interger(1))} |
| 4 | {(2, a), (2, **new** Interger(1))}, |
| | {(3, b), (3, **new** Interger(2))} |
| 5 | {(2, a), (2, **new** Interger(1))}, |
| | {(4, c), (4, b), (3, b), (3, **new** Interger(2))} |
| 6 | {(2, a), (2, **new** Interger(1))}, |
| | {(4, c), (5, c), (4, b), (3, b), (3, **new** Interger(2))} |
| 7 | {(6, c), (6, a), (2, a), (2, **new** Interger(1))}, |
| | {(4, b), (3, b), (3, **new** Interger(2))} |
| ... | {(7, c), (6, c), (6, a), (2, a), (2, **new** Interger(1))}, |
| | {(4, b), (3, b), (3, **new** Interger(2))} |

(b) 到達エイリアス集合

図 2: FS エイリアス解析

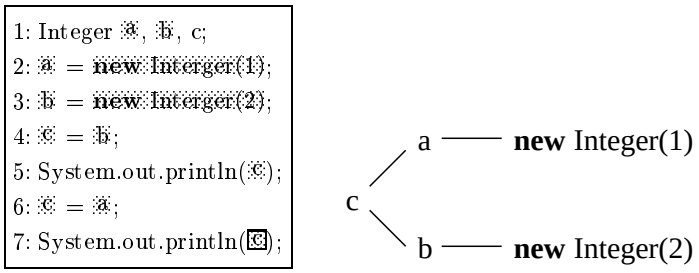
**FI エイリアス解析**

FI エイリアス解析 [2, 14] 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 3 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) で構成されており．辺はエイリアス関係を表わしている．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．到達可能な節点が文 7 の変数 c に関するエイリアスがある．エイリアスグラフより，{a, b, **new** Interger(1), **new** Interger(2))} が求めるエイリアスとなる．

**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.




(a) プログラム　　(b) エイリアスグラフ

図 3: FI エイリアス解析

**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

先に説明した例では，いずれの解析も線形時間で終了する．しかし，一般的なプログラムには手続き (関数) 呼び出し・繰り返し・再帰等が存在するため，プログラムの実行順を考慮する FS 解析はプログラム文 1 回の解析だけエイリアスを求めることはできない．一方，実行順を考慮しない FI 解析はプログラム文 1 回の解析でエイリアスを求めることができる．

文献 [11, 20] 等に，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 4 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．

```
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", 750);
    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 4: JAVA プログラム

## 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 [6].

オブジェクト指向言語には，オブジェクト生成機能・**メッセージパッシング** *(Message Passing)* 機能・**クラス** *(Class)* 機能・**継承** *(Inheritance)* などの特徴がある．

すべてのオブジェクトはクラスのインスタンスである．メッセージへの反応として，あるオブジェクトによって起動されるメソッドは，この受け手オブジェクトのクラスによって決定される．ある与えられたクラスのすべてのオブジェクトは，同じメッセージに対する反応として同じメソッドを使う．

クラスは**階層** *(Hierarchy)* を構築できる．サブクラスは，その継承階層の上位にあたるスーパークラスから状態と振舞いを継承する．メッセージがオブジェクトに送られると，対応するメソッドの検索が，そのオブジェクトのクラスから最上位のスーパークラスまで行われる．

**動的束縛** *(Dynamic Binding)* が提供されている場合，継承により**ポリモルフィズム** *(Polymorphism)* が起こる．サブクラスごとに異なる振舞いを定義することで，オブジェクトに送られた同一メッセージに対し，実体のクラス型に基づいて実行時に異なる解釈をさせることができる．

### 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 [16] 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．プログラム変更による再解析範囲をプログラム全体でなく変更の影響が及ぶ領域に限定し再解析コストを削減する手法が提案されているが [8]，1ヶ所の変更であっても全体解析コストの 25〜50%程度を必要とする結果となっている．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いが，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．エイリアス解析結果のモジュール化によるコスト削減手法としては，[13] があるが，手続き型言語でのエイリアスを対象としており，オブジェクト指向言語を考慮したものとなっていない．また，到達エイリアス集合を利用したモジュール化であるため，モジュールに影響するエイリアスに変化が起ったときその変化をモジュール内に伝播する必要がある．

**■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 5 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

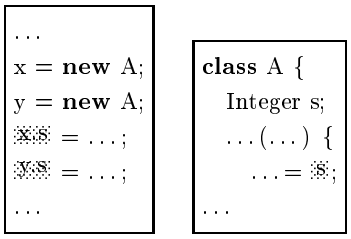



図 5: オブジェクト指向プログラム

- 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．

特定インスタンスに限定した解析手法として，**オブジェクト スライシング (Object Slicing)**[5] がある．これは**システム依存グラフ (System Dependence Graph, SDG)**[19] をオブジェクト指向言語に対応させ，特定のオブジェクトに関する情報を元にスライスを抽出する手法である．この考え方はエイリアス解析にも有効であると考えている．

**■解析結果の正確性向上**　複数の手続きで構成されているプログラムでは．**手続き呼び出し経路** *(Calling-Context)* を考慮することでより正確なエイリアス解析を行うことができる．図 6 にその例を示す．文 6 の変数 c(太枠部) に関し，網掛部は手続き呼び出し経路を考慮したときのエイリアスを表している．呼び出し経路を考慮しない場合，下線部もエイリアスに含まれることになる．

手続き B は異なる引数 b・c で手続き A を呼び出している．手続き呼び出し経路を考慮しない場合，手続き A 内では手続き呼び出し A(b)・A(c) の区別をしない．具体的には，到達エイリアス集合の計算の際，文 4: System.out.println(a) の解析後，

1. 手続き呼び出し 文 5: A(c) の解析
2. 手続き A の解析
3. 文 4: System.out.println(a)，文 6: System.out.println(a)

```
1: Interger x; Interger b = new Interger(1); Interger c = new Interger(2);
   ...
   void A(Integer x) {
2:    a = x;
   }
   void B() {
3:    A(b);
4:    System.out.println(a);
5:    A(c);
6:    System.out.println(a);
   }
```

図 6: 手続き呼び出し経路 (Calling-Context)

というように，手続き A からすべての呼び出し元に制御が移行する．手続き呼び出し経路を考慮する場合，手続き A から 文 6: System.out.println(a) にのみ制御が移行する．手続き呼び出し経路を考慮することで，計算量は増加するが解析精度を向上させることができる．

プログラム解析の視点 (図 7 参照) から見た場合，オブジェクト指向言語のクラスは手続き型言語のプログラムに相当する．つまり，オブジェクト指向言語でエイリアス解析の正確性向上を計るには，

- メソッド呼び出し経路
- 各インスタンスの内部情報

を考慮しなければならず，さらに上記以外の解析アルゴリズムの適用もありうる．しかし，これらすべてを前もって解析することは容易ではない．また，特定の実行経路・オブジェクトに限定したエイリアス解析においては，要求に応じ逐次解析する手法がより有効であると考えている．

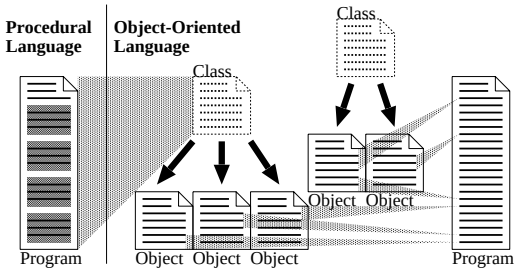



図 7: プログラム解析の視点から見た概念の相違

### 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
- エイリアス解析アルゴリズムの制御

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
- AFG のオブジェクト指向プログラムへの拡張

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することになる．各 AFG は対応するクラス・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的に解析する．

# 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 を用いたエイリアス解析手法は，大きく以下の 4 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**: AFG によるエイリアス計算

本節では，

**手続き内 AFG**　手続きのないプログラムの AFG

**手続き間 AFG**　手続きの存在するプログラムにおいて，手続きをまたぐエイリアスを考慮した AFG

**オブジェクト指向 AFG**　クラス・メソッドを考慮した AFG

の順に，各 AFG の構成・AFG の構築手法・AFG を用いたエイリアス計算について説明する．

## 4.1　AFG による手続き内 FS 解析

### 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に対し用いる．また，これらの節点を特に**標準節点** *(Normal Node)* と呼ぶ．図 8 に，サンプルプログラムおよびその 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(0) はインスタンス生成式であり，AFG 節点 (1, **new** Integer(0)) が生成される．

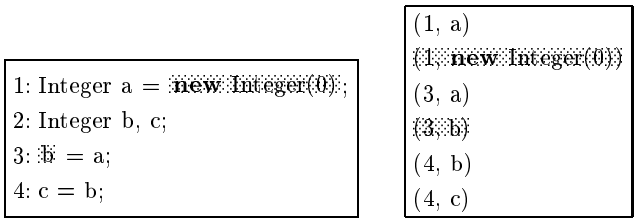



図 8: AFG 節点

### 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，エイリアス関係の成り立つ組ではなく，AFG 節点である．本手法では，エイリアス関係を AFG 辺 および 複数の AFG 辺で構成される 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．今回我々が実装した JAVA エイリアス解析ツールにおいても，if・while・do・try・for・synchronized・switch・catch・break・continue・return・throw・式 それぞれのアルゴリズムを定めており，本論文に付録として添付した．

### AFG 辺

*AFG* **辺** *(AFG Edge)* は，2 つの AFG 標準節点間の，同一変数参照，代入文やパラメータの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，推移的に成り立つ 2 節点間の間接のエイリアス関係を表す．図 9 に，サンプルプログラムおよび 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．

### 手続き内 AFG

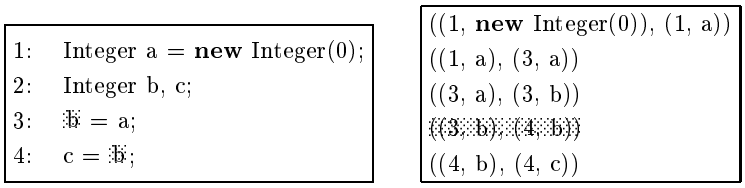



図 9: AFG 辺

**手続き内** *AFG(Intra-Procedural AFG)* は，AFG 節点・AFG 辺で構成される．図 10 に，サンプルプログラムおよびその手続き内 AFG を示す.

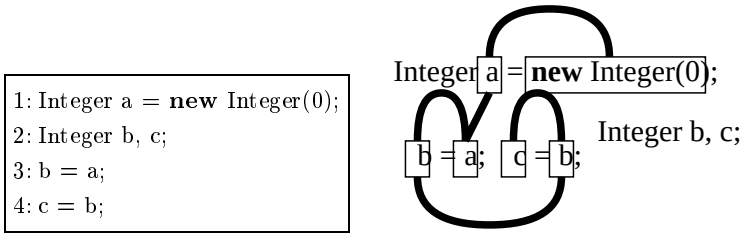



図 10: 手続き内 AFG

ALGORITHM 1: **手続き内 AFG によるエイリアス計算**

AFG を利用したエイリアス計算は以下の過程をたどる．

**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．

**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．

図 11 に，オブジェクトへの参照 c(太枠部) に対するエイリアス (網掛部) を示す．文 4: c = b の c に対応する AFG 節点は (4, c) であり，その節点から AFG 辺をたどる．この例では，AFG の全節点 {(1, a), (1, **new** Integer(0)), (3, a), (3, b), (4, b)} に到達可能であり，この到達可能節点に対応するオブジェクトへの参照が求める c のエイリアスである．

### 4.2 AFG による手続き間 FS 解析

AFG による手続き間 FS 解析では，手続き および 手続き呼び出しに対応させるため手続き内 FS 解析を拡張する．その拡張は以下の通りである．

- 手続き呼び出し (どの手続きがどの手続きを呼び出すか?) の把握

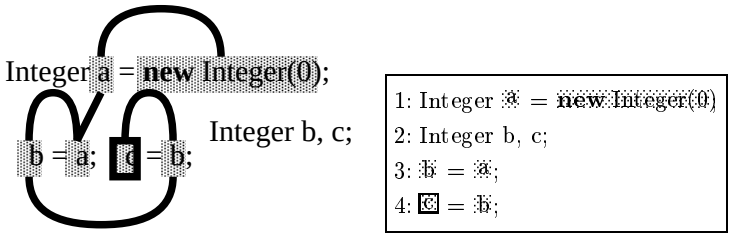



図 11: 手続き内 AFG による FS エイリアス解析

- 手続きの引数・戻り値によるエイリアス受け渡しの制御

前者は手続き呼び出しグラフにより，後者は新たな AFG 節点の追加とエイリアス計算アルゴリズムの拡張により行う．また，手続き呼び出しグラフの構築のため以下のフェーズが新たに追加される．

**Phase 3.5**: 手続き呼び出しグラフ構築

以下，それぞれについて説明する．

### 手続き呼び出しグラフ (CFG)

**手続き呼び出しグラフ** *(Call Flow Graph，CFG)* とは，プログラム中に存在する手続き (関数) 間の呼び出し関係を有向グラフで表現したものである．*CFG* **節点** *(CFG Node)* は手続きを表わし，手続き A が手続き B を呼ぶ場合，*CFG* **辺** *(CFG Edge)* を節点 *A* から節点 *B* に引く．図 12 にサンプルプログラムおよびその CFG の例を示す．手続き A は手続き B を，B は C を，C は A をそれぞれ呼び出しているため，右のような CFG が構築される．

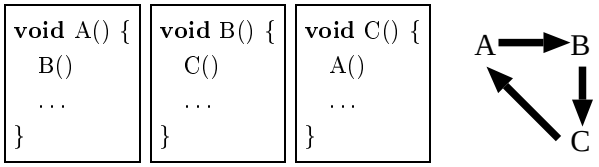



図 12: 手続き呼び出しグラフ (CFG)

### AFG 特殊節点 (手続き間 AFG)

手続きの引数・戻り値によるエイリアス受け渡しのため，手続き内 AFG で使用した標準節点に加え，表 2 の**特殊節点** *(Special Node)* が追加される．AA-in(out) 節点は，Procedure(Function) 節点の子節点として保持される．

### 手続き間 AFG

表 2: 特殊節点 (手続き間 AFG)

| 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，手続き側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，手続き側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，手続き呼び出し側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，手続き呼び出し側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，手続き呼び出し側に渡されるエイリアス) |
| Procedure | 手続き呼び出し |
| Function | 関数呼び出し (関数呼び出しの戻り値により，関数呼び出し側に渡されるエイリアス) |

手続き呼び出しを含むプログラムに対応した**手続き間** *AFG(Inter-Procedural AFG)* は，各手続き (関数) ごとに生成され，手続き間をまたぐエイリアス関係はエイリアス計算時に導出する．図 13 にサンプルプログラムおよびその手続き間 AFG の例を示す．破線の有向辺は，AA-in[x] 節点が Function 節点 (4, A(a)) の子節点であることを表している．

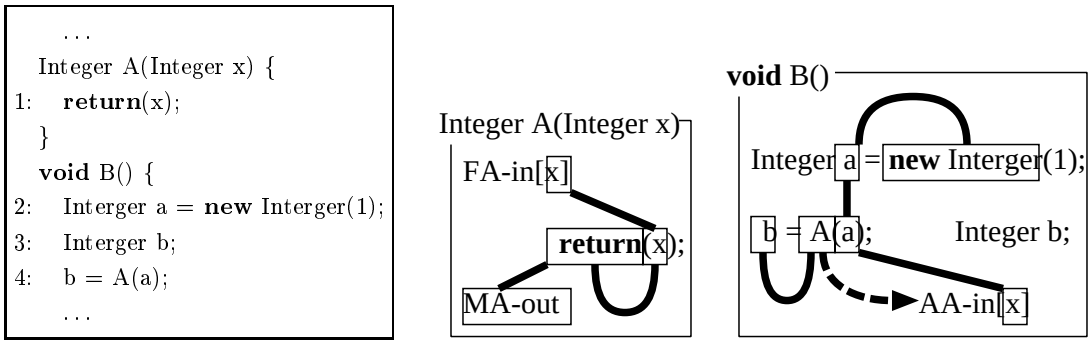



図 13: 手続き間 AFG

ALGORITHM 2: **手続き間 AFG によるエイリアス計算**

手続き間 FS 解析のため，ALGORITHM 1 の**Step 2** を拡張する．到達エイリアス集合のみによる FS エイリアス解析では，手続き間をまたぐエイリアス関係も前もって解析されている．本手法では，手続き内のエイリアス関係のみ前もって解析し (**Phase 1**〜**Phase 3**)，手続き間をまたぐエイリアス関係はエイリアス計算時に導出する (**Phase 4**)．

**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，以下の処理を行う．

**AA-in(out) 節点:**

1.節点 $C$ の親節点である，Procedure(Function) 節点に対応する手続き (関数) を $P_A$ とする

2.$P_A$ に対応する手続き間 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 $C$ を持つ手続き間 AFG を $G_B$，$G_B$ に対応する手続き (関数) を $P_B$ とする

2.$P_B$ を呼び出す手続き (関数)$P_A$ を CFG から抽出

3.$P_A$ に対応する手続き間 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 $C$ を持つ手続き間 AFG を $G_B$，$G_B$ に対応する関数を $P_B$ とする

2.$P_B$ を呼び出す手続き (関数)$P_A$ を CFG から抽出

3.$P_A$ に対応する手続き間 AFG を $G_A$ とする

4.$G_A$ が持つ (関数 $P_B$ を呼ぶ)Function 節点から AFG 辺をたどる

**Function 節点:**

1.節点 $C$ に対応する関数 $P_A$ の手続き間 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

**標準節点:**

**Step 2:**

図 14 にオブジェクトへの参照 b(太枠部) に対するエイリアス (網掛部) を示す．文 4: b = A(a) の b に対応する AFG 節点は (4, b) であり，その節点から AFG 辺をたどり Function 節点 (4, A(a)) に到達する．そこでAlgorithm 2:に従い関数 A() の MA-out 節点に移動する．次に MA-out 節点から AFG 辺をたどり FA-in[x] まで到達する．そしてAlgorithm 2:により，CFG から関数 A の呼び出し元である手続き B の AA-in[x] 節点に移動する．最後に

AA-in[x] 節点から AFG 辺を順にたどり標準節点 (2, **new** Integer) まで到達する．

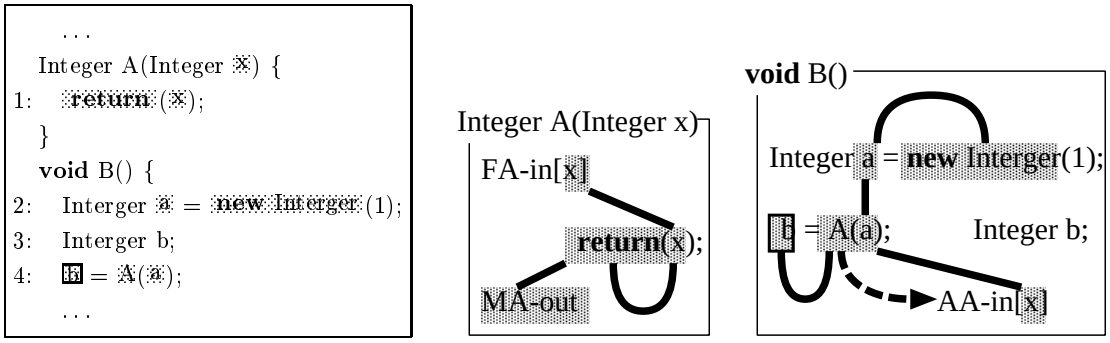



図 14: 手続き間 AFG による FS エイリアス解析

### 4.3 AFG によるオブジェクト指向プログラムのエイリアス解析

AFG によるオブジェクト指向プログラムのエイリアス解析では，オブジェクト指向プログラムに対応させるため手続き間 FS 解析を拡張する．その拡張は以下の通りである．

- クラス内での (継承を考慮した) メソッド呼び出しの把握
- クラス・属性への対応
- インスタンス内部のエイリアス計算

メソッド呼び出しの把握はメソッド呼び出しグラフにより，クラス・属性への対応/インスタンス内部のエイリアス計算は 新たな AFG 節点の追加とエイリアス計算アルゴリズムの拡張により行う．また，メソッド呼び出しグラフの構築のため以下のフェーズが新たに追加される．

**Phase 3.5**: メソッド呼び出しグラフ構築

以下，それぞれについて説明する．

#### メソッド呼び出しグラフ (MFG)

**メソッド呼び出しグラフ** *(Method Flow Graph，MFG)* は，CFG をクラス・継承に対応させたものである．クラス内でメソッドが呼び出されるとき，そのメソッドがクラス内で定義されていなくとも，スーパークラスで定義された呼び出し可能な同一シグネチャのメソッドが存在すればばよい．MFG は，

- スーパークラスから継承されるメソッド

- サブクラスで再定義されるメソッド
- 明示的に指定されるスーパークラスのメソッド

に留意しながらクラス単位に生成される．図 15 にサンプル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，スーパークラス A のメソッド p が利用される．B クラスの MFG の構築の際には，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．

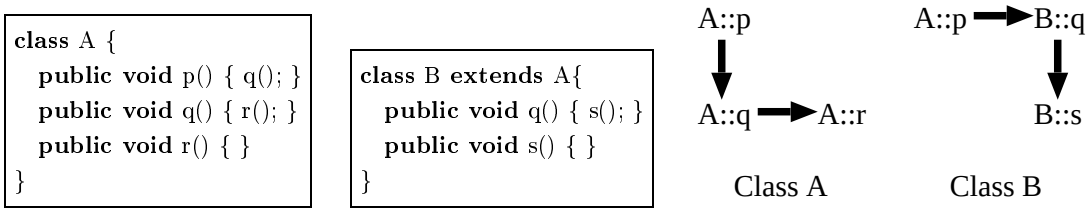



図 15: メソッド呼び出しグラフ (MFG)

**AFG 特殊節点 (オブジェクト指向 AFG)**

属性によるエイリアス受け渡しのため，手続き間 AFG で使用した標準節点・特殊節点に加え，表 3 の特殊節点が追加される．

表 3: 特殊節点 (オブジェクト指向 AFG)

| 特殊節点 | 概要 |
|---|---|
| Instance-Alias-in(IA-in) 節点 | 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) 節点 | 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) |

**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，Procedure(Function) 節点・標準節点に，親節点・子節点に関する情報を追加する．図 16 にその例を示す．破線の有向辺が節点の親子関係 (has-a 関係) を表している．ここで注意しなければならないのは，オブジェクト指向 AFG では，AFG 構築時にすべての AFG 節点に AFG 辺を引くことはできないことである．AFG の独立性・モジュール性を達成させるため，AFG 構築はそれ自身もしくはその親 AFG が持つ情報のみを利用する．図

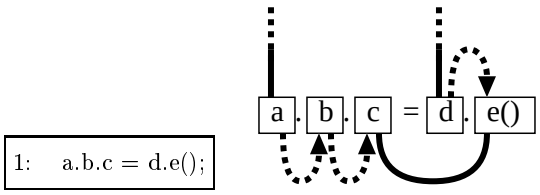



図 16: AFG 節点の親子関係

16 においても，明確に引くことのできる AFG 辺 (到達エイリアス集合を用いて導出可能なエイリアス関係) は，((1, c), (1, e())) ・ (· · ·, (1, a)) ・ (· · ·, (1, d)) にすぎない．AFG 構築時 (**Phase 1**〜**Phase 3**) に解決できなかったエイリアス関係は，エイリアス計算 (**Phase 4**) 時に導出する．

### オブジェクト指向 AFG

オブジェクト指向プログラムに対応した**オブジェクト指向** *AFG(Object-Oriented AFG)* では，メソッドを解析し**メソッド** *AFG(Method AFG)* を，クラスを解析し**クラス** *AFG(Class AFG)* を構築する．メソッド AFG は，メソッドが定義されたクラスのクラス AFG に属する．図 17 にサンプルプログラムおよびそのオブジェクト指向 AFG を示す．

### Algorithm 3: オブジェクト指向 AFG によるエイリアス計算

オブジェクト指向プログラムのエイリアス解析のため，Algorithm 2 の**Step 2'** を拡張する．先に述べたように，オブジェクト指向 AFG ではすべての節点のエイリアス関係が既に求まってはいない．そのため，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．

```
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();
  }
}
```

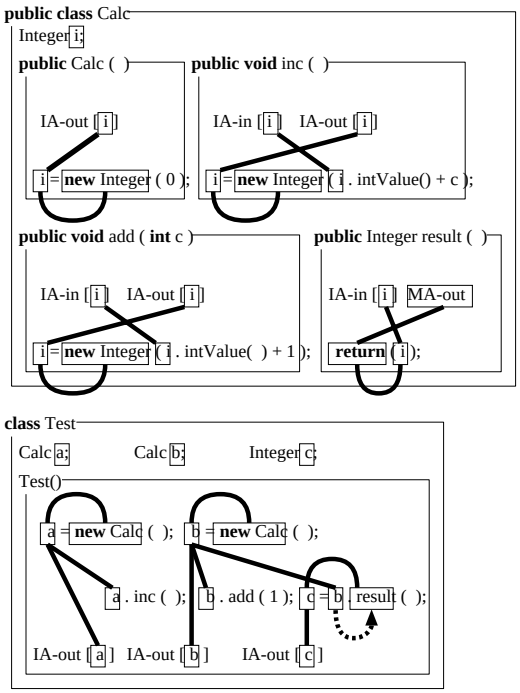



図 17: オブジェクト指向 AFG

**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 Procedure(Function) 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，Procedure(Function) 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

**親節点 $P$ を持つ Function 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.$\mathcal{P}$ から類推される型のメソッド c のメソッド AFG が持つ 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:**

節点 $C$ の親節点である，Procedure(Function) 節点を $Q$ とする

節点 $Q$ の種類に応じ，以下のような処理を行う

**親節点 $P$ を持つ Procedure(Function) 節点:**

1.節点 $Q$ に対応するメソッドを q 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.$\mathcal{P}$ から類推される型のメソッド q のメソッド AFG を $G_Q$ とする

6.$G_Q$ の該当する FA-in(out) 節点から AFG 辺をたどる

**親節点を持たない Procedure(Function) 節点:**

1.**Step 2':**

**上記以外の節点:**

1.**Step 2':**

**オブジェクト コンテキスト**

**オブジェクト コンテキスト** *(Object-Context)* とは，オブジェクトへの参照 $c$ のエイリアス集合 $\mathcal{C}$ が直接もしくは間接的に呼び出しうるメソッド集合を指し，Object-Context($\mathcal{C}$) と表記する．このオブジェクトコンテキストには，

**Flow-Insensitive(FI):** メソッド呼び出し順を考慮しない

**Flow-Sensitive(FS):** メソッド呼び出し順を考慮する

の 2 種類が存在する．Algorithm 3: は FI オブジェクトコンテキストであり，後述するエイリアス解析ツールも FI オブジェクトコンテキストを採用している．

このオブジェクトコンテキストを用いることで，存在する可能性のないエイリアスを排除することができる．図 18(a)・18(b) に，FI(FS) オブジェクトコンテキスト (下線部) の例を示す．Object-Context(b) は {Calc::Calc()，Calc::add(**int** c)，Calc::result()} となるため，図 18(a) では Calc::inc() の i が c のエイリアスとなっていない．さらに図 18(b) では，Calc::Calc()・Calc::add(**int** c)・Calc::result() の順にメソッドが呼ばれたことから，Calc::Calc()・Calc::add(**int** c) からエイリアスが除外されている．

```
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();
  }
}
```

(a) Flow-Insensitive

```
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[1]();
    a.inc();
    b.add[2](1);
    c = b.result[3]();
  }
}
```

(b) Flow-Sensitive

図 18: オブジェクトコンテキスト

### エイリアス計算例

エイリアス計算の例として，図18(a)に示したプログラムのc(太枠部)のエイリアス(網掛部)の抽出手順を説明する．アルゴリズムは，Flow-Insensitive Object-ContextであるALGORITHM 3: に基づく．

1. 参照変数cに対応するAFG節点からAFG辺をたどり，Function節点result()に到達
2. Function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，Function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
   (a) 節点bのエイリアス$\mathcal{B}$を計算(図19)
   (b) $\mathcal{B}$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]
   (c) Object-Context($\mathcal{B}$)を計算[{Calc::Calc()，Calc::add(**int** c)，Calc::result()}]
3. 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20)
   (a) AFG辺をたどり，IA-in[i]に到達
   (b) 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)

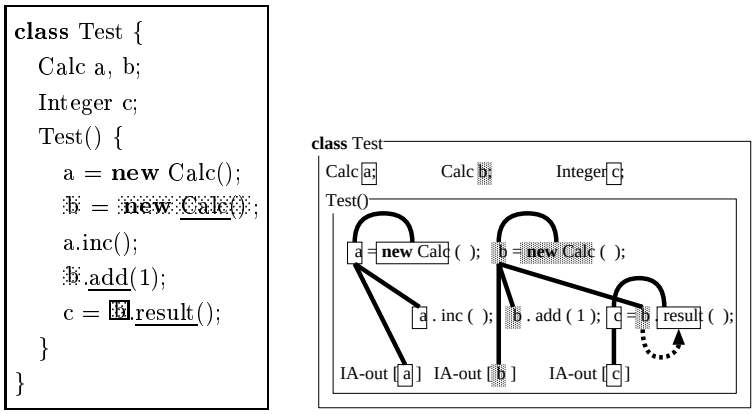



図 19: 図18のbに関するエイリアス(網掛部)

```
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);
  }
}
```

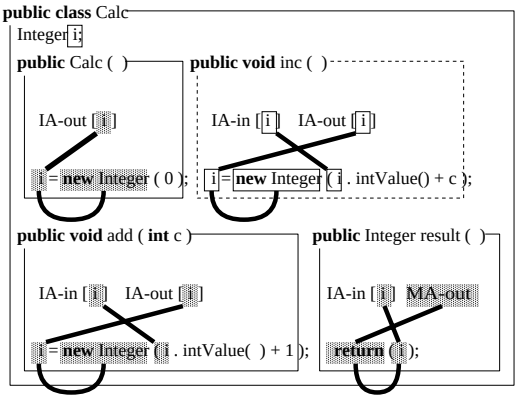



図 20: 図 18 の b.result() に関するエイリアス (網掛部)

### 4.4 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 21 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．

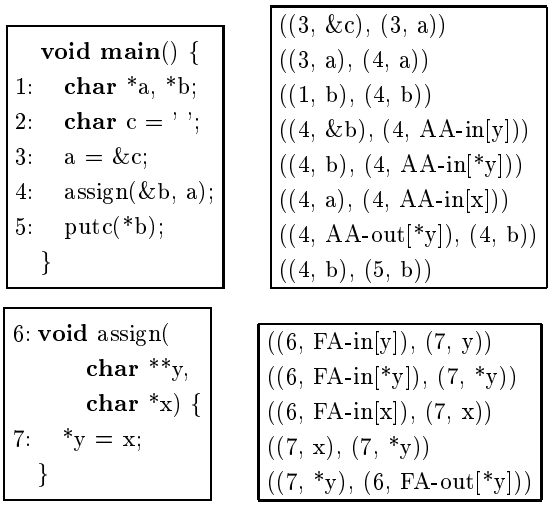



図 21: ポインタ変数を持つ言語 (C) での解析

### 4.5 計算量

提案した AFG によるエイリアス解析手法の複雑さについて述べる．ここでは，オブジェクト指向 AFG の構築・メソッド呼び出しグラフ・オブジェクト指向 AFG によるエイリアス計算 に要するコストに限定している．

#### オブジェクト指向 AFG の構築に要するコスト

オブジェクト指向 AFG の構築に関わる要素を表 4 のように定義する．

表 4: オブジェクト指向 AFG の構築に関わる要素

| 記号 | 概要 |
|---|---|
| $At$ | 1 クラスでの属性数の最大値 (継承を含む) |
| $Lo$ | 1 メソッドでの局所変数・仮引数の数の最大値 |
| $Ex$ | 式の総数 |

プログラムに繰り返し文が存在する場合，最悪 $Ex^2$ 回の式の解析を行わなければならない．ひとつの文の解析には定数回の集合演算が必要である．1 回の集合演算にかかる時間が集合に含まれる要素の数に比例すると仮定すると，ひとつの文の解析の時間は，$At$ と $Lo$ の和に比例する．よって，計算コスト $O(Ex^2(At + Lo))$ となる．空間コストに関して，AFG 節点数は $O(Ex)$，AFG 辺数は $O(Ex^2)$ となる．

#### メソッド呼び出しグラフの構築に要するコスト

メソッド呼び出しグラフの構築に関わる要素を表 5 のように定義する．

表 5: メソッド呼び出しグラフの構築に関わる要素

| 記号 | 概要 |
|---|---|
| $Cl$ | クラスの総数 |
| $Md$ | 1 クラスでのメソッド数の最大値 (継承を含む) |
| $Ex$ | 式の総数 |

メソッド呼び出しグラフはクラス単位に構築される．この計算には，すべての式を 1 回解析し (同じインスタンスの) メソッド呼び出しを探索する必要があるため，計算コストは $O(Ex)$ となる．また空間コストに関して，1 クラス中のメソッド数は $Md$ で抑えられることから，グラフの節点数は $O(Cl \cdot Md)$，辺数は $O(Cl \cdot Md \cdot Md)$ となる．

#### オブジェクト指向 AFG によるエイリアス計算に要するコスト

オブジェクト指向 AFG によるエイリアス計算に関わる要素を表 6 のように定義する．

表 6: オブジェクト指向 AFG によるエイリアス計算に関わる要素

| 記号 | 概要 |
|---|---|
| $Ex$ | 式の総数 |
| $k$ | 再帰解析上限 (k = 0 のとき，親節点を解析しない) |

全ての節点間でエイリアス関係が成り立つ場合に最も多くの計算量が必要になる．ただし，エイリアス関係である節点が親節点を持つ場合，その親節点の解析をしなければならない．これは無限ループになる可能性があるため，再帰解析上限 $k$ を用意する．よって，計算コストは $O(Ex^{k+1})$ である．

# 5　本手法の実現 (Java エイリアス解析ツール)

## 5.1　概要

今回我々は提案手法の実現のため，JAVA を対象言語とするエイリアス解析ツールを実装した．ツール全体の構成は図 22 に示す．ツール構成は [4] を参考にした．ツールは解析部とユーザインターフェース部で構成されており，それぞれを簡単に紹介する．

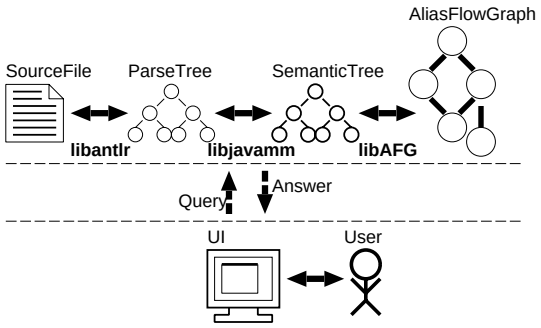



図 22: エイリアス解析ツール

### 解析部

解析部は C++で記述されており，libantlr・libjavamm・libAFG の 3 つのライブラリで構成されている．

**libantlr**

**入力:**　JAVA ソースファイル

**出力:**　JAVA 構文解析木

**概要:**　字句解析・構文解析を行う / 構文解析木へのインターフェースの提供
ANTLR[22] は，言語 $L$ の文法 $\mathcal{L}$ を与えることで $L$ の字句解析・構文解析ルーチンを C++もしくはJAVA で生成するツールである．

**行数:**　17,000[自動生成] + 1,200

**libjavamm**

**入力:**　JAVA 構文解析木

**出力:**　JAVA 意味解析木

**概要:**　意味解析を行う / 意味解析木へのインターフェースの提供

**行数:** 5,500

**libAFG**

**入力:** Java 意味解析木

**出力:** AFG

**概要:** AFG の生成 / AFG へのインターフェースの提供 / エイリアス解析

**行数:** 3,300

#### ユーザインターフェース部

ユーザインターフェース部は C++で記述されており，GUI ライブラリとして Gtk−−[23] (GTK+[24]) を使用した．ソースファイル編集・エイリアス解析・エイリアス表示の機能を有する．解析部とは独立した構成となっているため，他のインターフェースを実装することも容易である．

### 5.2 評価

Java エイリアス解析ツールを用いて本手法の有効性を評価した．ここでは，以下の点について比較を行う．

- 解析結果のモジュール化による効率化
  - 対象プログラムが利用する JDK 附属クラスライブラリが解析されている状態で，対象プログラムのみを解析する時間
  - 対象プログラム および それが利用する JDK 附属クラスライブラリの解析時間

  の比較．
- AFG によるエイリアス計算のオーバヘッド
- 同一クラスのインスタンス間でのエイリアス情報共有による正確性低下

対象プログラムの概要を表 7 に挙げる．対象プログラムは，新規プログラム欄に記述された大きさであり，その新規プログラム内で，再利用可能プログラム欄に記述された規模のクラスおよびインターフェースが利用される．表 8 に，対象プログラムが利用する JDK 附属クラスライブラリが解析されている状態で，対象プログラムのみを解析する時間 (新規 プログラム欄)，対象プログラム および それが利用する JDK 附属クラスライブラリの解析時間 (新規+再利用可能 プログラム欄) を示す．表 9 に対象ファイルに存在する全 AFG 標準節点

からのエイリアス計算に要する平均時間を示す．平均計算時間に関して,「共有」欄の数値は，既にエイリアス計算によりいずれかのエイリアス集合に属する オブジェクトへの参照がエイリアス計算対象として与えられた時に，そのエイリアス計算を省略し再利用した時の値である.「非共有」欄はエイリアス計算結果の再利用を行わないときの値である．表 10 に，同一クラスのインスタンス間で属性に関するエイリアス情報を共有することによる正確性の低下の程度を示す.「非共有」欄が本手法による結果であり,「共有」欄が同一クラスの属性に関するエイリアス情報を共有した場合の結果である．

表 7: 対象プログラム

| プログラム [概要] | 新規プログラム ファイル | 新規プログラム 行 [コメント無] | 再利用可能プログラム (JDK1.1.8 付属) ファイル | 再利用可能プログラム (JDK1.1.8 付属) 行 [コメント無] | 新規+再利用可能 プログラム ファイル | 新規+再利用可能 プログラム 行 [コメント無] |
|---|---|---|---|---|---|---|
| TextEditor | 1(0.12%) | 1,125(0.39%) | 802 | 288,973 | 803 | 290,098 |
| [テキストエディタ] | | 915(0.79%) | | 114,887 | | 115,802 |
| WeirdX | 47(5.45%) | 19,701(6.32%) | 815 | 292,178 | 862 | 311,879 |
| [X サーバ] | | 16,703(12.59%) | | 115,977 | | 132,680 |
| ANTLR | 129(32.58%) | 25,277(21.08%) | 267 | 94,610 | 396 | 119,887 |
| [構文解析生成器] | | 18,775(35.68%) | | 33,847 | | 52,622 |
| DynamicJava | 242(22.68%) | 58,300(16.24%) | 825 | 300,626 | 1,067 | 358,926 |
| [Java インタプリタ] | | 32,037(21.13%) | | 119,564 | | 151,601 |

表 8: 解析結果のモジュール化による効率化

| プログラム | 新規 プログラム 解析時間 [ms] | 新規 プログラム AFG グラフ | 節点 | 辺 | 新規+再利用可能 プログラム 解析時間 [ms] | 新規+再利用可能 プログラム AFG グラフ | 節点 | 辺 |
|---|---|---|---|---|---|---|---|---|
| TextEditor | - | 11 | 1,325 | 2,261 | 99,980 | 906 | 166,327 | 303,509 |
| WeirdX | 14,220 | 70 | 16,989 | 34,069 | 114,760 | 978 | 185,270 | 337,772 |
| ANTLR | 12,830 | 129 | 22,040 | 44,270 | 36,310 | 427 | 66,380 | 125,450 |
| DynamicJava | 56,260 | 242 | 57,338 | 99,720 | 166,410 | 1,165 | 233,258 | 420,642 |

### 5.3 考察

表 8(解析結果のモジュール化による効率化) に関して，新たな変更が加えられず繰り返し再利用されるプログラムを解析し AFG を構築しておくことで，ユーザは新規に作成したプログラム部分のみを解析すればよいため大幅はコスト削減が得られる．

表 9(AFG によるエイリアス計算のオーバヘッド ) に関して，到達エイリアス集合のみによ

表 9: AFG によるエイリアス計算のオーバヘッド

| プログラム | 計算対象クラス | 平均計算時間 [ms] | | エイリアス | | |
|---|---|---|---|---|---|---|
| | | 共有 | 非共有 | 平均 | 最小 | 最大 |
| TextEditor | test.MyTextArea | 0.65 | 1.38 | 4.42 | 1 | 24 |
| WeirdX | com.jcraft com.jcraft.weirdx.Client | 0.29 | 1.71 | 15.37 | 1 | 46 |
| ANTLR | antlr.MakeGrammar | 0.17 | 0.81 | 5.94 | 1 | 18 |
| DynamicJava | koala.dynamicjava.interpreter.TypeChecker | 0.07 | 0.35 | 9.16 | 1 | 37 |

表 10: 同一クラスのインスタンス間でのエイリアス情報共有による正確性低下

| プログラム | 計算対象クラス | エイリアス | | | | | |
|---|---|---|---|---|---|---|---|
| | | 非共有 | | | 共有 | | |
| | | 平均 | 最小 | 最大 | 平均 | 最小 | 最大 |
| TextEditor | test.MyTextArea | 4.42 | 1 | 24 | 8.31 | 1 | 40 |
| WeirdX | com.jcraft com.jcraft.weirdx.Client | 15.37 | 1 | 46 | 24.54 | 1 | 47 |
| ANTLR | antlr.MakeGrammar | 5.94 | 1 | 18 | 18.77 | 1 | 76 |
| DynamicJava | koala.dynamicjava.interpreter.TypeChecker | 9.16 | 1 | 37 | 17.19 | 1 | 105 |

るエイリアス解析では，解析終了の時点ですべてのエイリアス関係が抽出されるため，AFG によるエイリアス計算に相当する時間は 0 である．そのため，一概に比較することはできないが，AFG 構築時間と比較して非常に短かい時間でエイリアス計算が終了しているのが分かる．また，計算結果の共有がエイリアス計算時間の短縮に大きな影響を与えているのが分かる．

表 10(同一クラスのインスタンス間でのエイリアス情報共有による正確性低下) に関して，本手法では，同一クラスのインスタンスに共通するエイリアス関係のみ AFG に保持され，インスタンス独自のエイリアス関係はエイリアス計算時に導出する．本手法の適応により，各インスタンス属性を区別することによる効果が得られていることが分かる．

### 5.4 Java に対する AFG 構築

ここでは，提案手法のオブジェクト指向言語Java への適応 (実装) にあたり，AFG 構築に必要となる

- 到達エイリアス集合
- Java ジャンプ命令に対する到達エイリアス集合の計算
- Java 非ジャンプ命令に対する到達エイリアス集合の計算

について説明する．

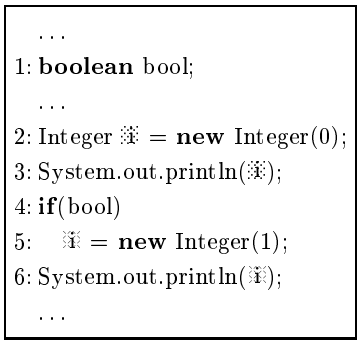



図 23: May エイリアスと Must エイリアス

### 5.4.1 到達エイリアス集合

到達エイリアス集合 $RA(s)$ とは，文 $s$ に到達可能である “何らかのオブジェクトを参照している (もしくは，何らかのオブジェクトとエイリアス関係にある) 変数もしくは式” をいう．到達エイリアス集合の各要素は

**AFG 節点へのポインタ**

エイリアスを生成する AFG 節点へのポインタ

**Must エイリアスフラグ (Must Alias Flag)**

対応する AFG 節点 (エイリアス) が “必ず到達する” とき**true**，“到達する可能性がある” (必ず到達するとも，必ず到達しないともいえない) とき**false**

ここで，“必ず到達する” エイリアスを *Must***エイリアス** *(Must Alias)*，“到達する可能性がある” エイリアスを *Must***エイリアス** *(Must Alias)* という．

> 図 23 に Must エイリアス・May エイリアスの例を示す．文 3: System.out.println(i) において，文 2: Integer i = **new** Integer(0) の i に関するエイリアス (網掛部) は文 2: に “必ず到達する” ため，Must エイリアスとなる．一方，文 6: System.out.println(i) においては，文 5: i = **new** Integer(1) の i に関するエイリアス (網掛部) は，**if** 文の分岐節内で定義されており，文 6: に “到達する可能性がある” にすぎないため，May エイリアスとなる．

で構成されている．到達エイリアス集合は命令のたびに計算され，AFG 辺はその情報を元に引かれる．

### 5.4.2 ジャンプ命令に対する解析

一般に，プログラミング言語には強制的に制御移行を行う命令が存在する．JAVA の場合，goto 文こそないが，break・continue・return 文が存在する．特に break・continue 文は，C 言語と違いラベル先の指定ができるため移行先の制限が緩和されている．その例を，図 24 に示す．このため，逐次的な到達エイリアス集合の計算だけでは対処できず，制御移行文に対処させた計算アルゴリズムが必要になる．

> ラベルのない break 文は，**ブレーク・ターゲット** *(break target)* と呼ぶ break を囲んでいる最も内側の switch，while，do，for 文への制御の移行を試み，その後即座に正常終了する．ラベルとしての**識別子** *(Identifier)* が付いた break 文は，**ブレーク・ターゲット**と呼ぶそのラベルと同一の**識別子**を保持したラベル付き文に制御の移行を試み，その後即座に正常終了する．この場合，ブレーク・ターゲットは while，do，for，switch 文である必要はない．
>
> 文献 [7] より

> ラベルのない continue 文は，**継続ターゲット** *(continue target)* と呼ぶ continue 文を囲んでいる最も内側の while，do，for 文への制御の移行を試み，現在の繰り返しを即座に終了後，新たな繰り返しを開始する．ラベルとしての**識別子** *(Identifier)* が付いた continue 文は，そのラベルとして同一の**識別子**が付いた continue 文を囲んでいる，**継続ターゲット** *(continue target)* と呼ぶラベル付き文へ制御の移行を試み，現在の繰り返しを即座に終了後，新たな繰り返しを開始する．継続ターゲットは，while，do，for 文である必要がある．
>
> 文献 [7] より

> **式** *(Expression)* のない return 文は，その文を含むメソッドや構築子の呼び出し元へ制御の移行を試みる．**式**のある return 文は，その文を含むメソッドの呼び出し元に制御の移行を試み，**式**の値はメソッド呼び出しの値となる．
>
> 文献 [7] より

このような制御移行文に対する解析法としては，

1. 制御移行文を考慮し，考えられうるすべての経路を導出
2. 各経路ごとに到達エイリアス集合を計算
3. 到達エイリアス集合の集約

```
   Integer a, b;
1: b = new Integer(0);
   ...
2: label1: while(x < 100) {
3:    label2: while(y < 100) {
           ...
4:       if(z > 9) {
5:          continue labell;
6:          a = b;
         }
           ...
7:       if(z <= 9)
8:          continue label2;
   }
}
```

図 24: 継続ターゲットのある continue 文

という手順が考えられる．しかし今回の実装においては，前もって経路導出を行わず各文の解析時に以下の過程を経るようにした．

1. 制御移行文のとき
   (a) 制御移行文の (仮想的な) エイリアス J を作成
   (b) その時点での到達エイリアス集合 $\mathcal{R}$ を J に登録
   (c) $\mathcal{R}$ に J を追加
2. 制御移行対象になりうる文 s のとき
   (a) 到達エイリアス集合 $\mathcal{R}'$ に，s が移行先となる制御移行文のエイリアス J' が存在するかを調べる
   (b) もし存在すれば，J' が持つエイリアス集合と $\mathcal{R}'$ の和集合演算を行い，$\mathcal{R}'$ に反映

このような手順を採用したのは，到達エイリアス集合から，現在解析対象となっている文が到達可能かどうかの判定が可能であるためである．例えば，図 24 の文 6 は直前の continue 文の存在により実行されることはない．この事実は，文 6 の解析時の到達エイリアス集合に continue 文のエイリアスが含まれていることから知ることができ，文 6 の実行により生成されるであろうエイリアスを防ぐことができる．

### 5.4.3 非ジャンプ命令に対する解析

JAVA における非ジャンプ命令に対する到達エイリアス集合の計算は，プログラムスライス計算 [21] に用いるデータフロー演算 [1] が利用できる．[21] では，サブセット Pascal における到達定義集合の計算アルゴリズムが記述されており，それを基に，JAVA 到達エイリアス集合の計算アルゴリズムを定義した．その詳細は付録として添付した．

## 6　まとめと今後の課題

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 (AFG) によるエイリアス解析手法 および オブジェクト指向言語に対するその拡張 の提案を行った．また，提案手法を Java を対象としたエイリアス解析ツールとして実装を行い，その有効性を評価した．

既存手法はエイリアス解析結果の再利用を考慮に入れていないため，その都度プログラム全体を解析しなければならなかった．また，正確性の向上のために同一クラスの異なるインスタンス属性を区別するには多大なコストが必要であった．

本手法では，クラス (メソッド) はそれぞれ独立に解析され，クラス AFG(メソッド AFG) を構築する．AFG 節点はオブジェクトへの参照を，AFG 辺は節点間の直接のエイリアス関係を表している．各 AFG は，一度構築されると対応するメソッド (クラス) およびそれらが依存するクラス (メソッド) が変更されない限り再構築する必要はない．また，エイリアス計算は通常グラフの到達問題として行われるが，インスタンス x の属性 (メソッド )y に関するエイリアス計算の場合，まず x に関するエイリアス計算を行い，その後 y 関するエイリアス計算を行う．特定インスタンスに限定した解析が行えるため，正確なエイリアス解析結果を得ることができる．

今後の課題としては，

- Flow-Sensitive オブジェクトコンテキストの実現 および Flow-Insensitive オブジェクトコンテキストとの比較
- 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．

## 謝辞

本論文の作成において，常に適切な御指導および御助言を頂きました 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上 克郎 教授に深く感謝致します.

本論文の作成において，常に適切な御指導を頂きました 同 楠本 真二 助教授に深く感謝致します.

本論文の作成において，常に適切な御助言を頂きました 同 松下 誠 助手に深く感謝致します.

本論文の作成において，ご協力を頂きました 同 近藤 和弘 君に深く感謝致します.

最後に，その他の面で様々な御指導，御助言を頂いた 大阪大学 大学院基礎工学研究科 情報数理系専攻 ソフトウェア科学分野 井上研究室の皆様に深く感謝致します.

## 参考文献


[1] A. V. Aho, S. Sethi and J. D. Ullman, “Compilers : Principles, Techniques, and Tools”, Addison-Weseley, 1986.

[2] 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.

[3] B. Stroustrup, “The C++ Programming Language(Third edition),” Addison-Wesley, 1997.

[4] D. C. Atkison and W. G. Griswold, “The Design of Whole-Program Analysis Tools”. In Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pages 16–27, 1996.

[5] D. Liang, and M. J. Harrold, “Slicing Objects Using System Dependence Graphs,” In Proceedings of the International Conference on Software Maintenance (ICSM’98), pp.358–367, 1998.

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

[7] J. Gosling, B. Joy, and G. Steele, 村上 雅章 [訳], “The Java 言語仕様”

[8] J. Yur, B. G. Ryder, and W. A. Landi, “An Incrimental Flow- and Context-sensitive Pointer Aliasing Analysis,” In Proceedings of the 21th International Conference on Software Engineering, pp.442–451, 1999.

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

[10] 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.

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

[12] M. Hind, and A. Pioli, “Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses,” The Fifth International Static Analysis Symposium (SAS’98), 1998.

[13] M. J. Harrold, and G. Rothermel, "Separate Computation of Alias Information for Reuse," In IEEE Transactions on Software Engineering, Special section of best papers of the 1996 International Symposium on Software Testing and Analysis, vol. 22, no. 7, pp.442–460, 1996.

[14] 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.

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

[16] 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.

[17] R. Ghiya, and L. J. Hendren, "Connection Analysis: A practical interprocedural heap analysis for C," in International Journal of Parallel Programming, 24(6), pp.547–578, 1996.

[18] 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.

[19] 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.

[20] S. Zhang, B. G. Ryder, and W. A. Landi, "Experiments with Combined Analysis for Pointer Aliasing," In Proceedings of PASTE'98, pp.11–18, 1998.

[21] 佐藤, 飯田, 井上 "プログラムの依存解析に基づくデバッグ支援ツールの試作", 情報処理学会論文誌, Vol. 37, No.4, pages 536–545, 1996.

[22] `http://www.ANTLR.org/`, "ANTLR Website."

[23] `http://lazy.ton.tut.fi/terop/iki/gtk/gtk--.html`, "Gtk−−."

[24] `http://www.gtk.org/`, "GTK+ − The GIMP Toolkit."

# 付録

## JavaAFG 構築アルゴリズム

Algorithm ClassAFG(**T**:Java クラス意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**C**= **new**ClassAFG, **G**.register(**C**)
<pre_Member(**T**, **C**, **RA**)>
初期化子・静的初期化子の解析 <Block(**T**.statements[i], **C**, **RA**)> ...
<post_Member(**T**, **C**, **RA**)>
内部参照型の解析 <ClassAFG(**T**.types[i], **C**, copySet(**RA**)), InterfaceAFG(**T**.types[i], **C**, copySet(**RA**))> ...
メソッドの解析 <MethodAFG(**T**.methods[i], **C**, copySet(**RA**))> ...

Algorithm InterfaceAFG(**T**:Java インターフェース意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**I**= **new**InterfaceAFG, **G**.register(**I**)
<pre_Member(**T**, **I**, **RA**)>
初期化子・静的初期化子の解析 <Block(**T**.statements[i], **I**, **RA**)> ...
<post_Member(**T**, **I**, **RA**)>
内部参照型の解析 <ClassAFG(**T**.types[i], **I**, copySet(**RA**)), InterfaceAFG(**T**.types[i], **I**, copySet(**RA**))> ...
抽象メソッドの解析 <MethodAFG(**T**.methods[i], **I**, copySet(**RA**))> ...

Algorithm MethodAFG(**T**:Java メソッド意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**M**= **new**MethodAFG, **G**.register(**M**)
<pre_Member(**T**.parent, **M**, **RA**)>
仮引数節点$\mathbf{n}^{\P}$ の作成, **M**.register(**n**), [**RA**= **RA**∪ **n**...]
メソッドブロックの解析 <Block(**T**.statements[0], **M**, **RA**)>
**RA**= **RA**− $\mathbf{n}^{\P}$ ...
<post_Member(**T**.parent, **M**, **RA**)>

Algorithm Block(ブロック)(**T**:Java ブロック意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

<pre_Local(**T**, **G**, **RA**)>
内部参照型の解析 <ClassAFG(**T**.types[i], **G**, copySet(**RA**)), InterfaceAFG(**T**.types[i], **G**, copySet(**RA**))> ...
各文の解析 <Block(**T**.statements[i], **G**, **RA**)> ...
<post_Break(**T**, **RA**)>
<post_Return(**T**, **RA**)>
<post_Local(**T**, **G**, **RA**)>

Algorithm LabeledStatement(ラベル付き文)(**T**:Java ラベル付き文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

文の解析 <Statement(**T**.statements[0], **G**, **RA**)>

Algorithm Statement(文)(**T**:Java 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

以下のいずれかが適用される.

BlockStatement(**T**, **G**, **RA**)
IfStatement(**T**, **G**, **RA**)
WhileStatement(**T**, **G**, **RA**)
DoStatement(**T**, **G**, **RA**)
ForStatement(**T**, **G**, **RA**)
SwitchStatement(**T**, **G**, **RA**)
SynchronizedStatement(**T**, **G**, **RA**)
TryStatement(**T**, **G**, **RA**)
LabeledStatement(**T**, **G**, **RA**)
BreakStatement(**T**, **G**, **RA**)
ContinueStatement(**T**, **G**, **RA**)
ReturnStatement(**T**, **G**, **RA**)
ThrowStatement(**T**, **G**, **RA**)
ExpressionStatement(**T**, **G**, **RA**)

Algorithm If(**T**:Java if 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

条件節の解析 <Expression(**T**.expression, **G**, **RA**)>
[**thenRA**= **elseRA**= **RA**]
then 節の解析 <Statement(**T**.statement[0], **G**, **thenRA**)>
else 節の解析 <Statement(**T**.statement[1], **G**, **elseRA**)>
then 節で定義されたエイリアスを抽出 [**thenRA**= **thenRA**− **RA**]
else 節で定義されたエイリアスを抽出 [**elseRA**= **elseRA**− **RA**]
**RA** から then 節・else 節で確実に定義されるエイリアスを削除
[**RA**= **RA**− selectMust(**thenRA**) ∩ selectMust(**elseRA**)]
then 節で定義されたエイリアスを may エイリアス化 [**thenRA**= toMay(**thenRA**)]
else 節で定義されたエイリアスを may エイリアス化 [**elseRA**= toMay(**elseRA**)]
[**RA**= **RA**∪ **thenRA**∪ **elseRA**]

Algorithm While(**T**:Java while 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

1 回目の条件節の解析 <Expression(**T**.expression, **G**, **RA**)>
[$\mathbf{recursiveRA}_1$ = **RA**]
1 回目の繰り返し節の解析 <Statement(**T**.statements[0], **G**, $\mathbf{recursiveRA}_1$)>
<post_Continue(**T**, $\mathbf{recursiveRA}_1$)>
[$\mathbf{recursiveRA}_n$ = $\mathbf{recursiveRA}_1$]
制御移行文による n(n > 1) 回目の繰り返しがない 以外,

n 回目の条件節の解析 <Expression(**T**.expression, **G**, $\mathbf{recursiveRA}_n$)>
n 回目の繰り返し節の解析 <Statement(**T**.statements[0], **G**, $\mathbf{recursiveRA}_n$)>
<post_Continue(**T**, $\mathbf{recursiveRA}_n$)>
n 回目の (条件節 + 繰り返し節) で定義されたエイリアスを抽出
[$\mathbf{recursiveRA}_n$ = $\mathbf{recursiveRA}_n$ − **RA**]
n 回目の (条件節 + 繰り返し節) で定義されたエイリアスを may エイリアス化
[$\mathbf{recursiveRA}_n$ = toMay($\mathbf{recursiveRA}_n$)]

1 回目の繰り返し節で定義されたエイリアスを抽出 [$\mathbf{recursiveRA}_1$ = $\mathbf{recursiveRA}_1$ − **RA**]
1 回目の繰り返し節で定義されたエイリアスを may エイリアス化

[$\mathbf{recursiveRA}_1$ = toMay($\mathbf{recursiveRA}_1$)]

[$\mathbf{RA}$= $\mathbf{RA}$∪ $\mathbf{recursiveRA}_1$ ∪ $\mathbf{recursiveRA}_n$]

<post_Break($\mathbf{T}$, $\mathbf{RA}$)>

Algorithm Do($\mathbf{T}$:Java while 文意味解析木, $\mathbf{G}$:AFG, $\mathbf{RA}$:到達エイリアス集合)

1 回目の繰り返し節の解析 <Statement($\mathbf{T}$.statements[0], $\mathbf{G}$, $\mathbf{RA}$)>

<post_Continue($\mathbf{T}$, $\mathbf{RA}$)>

1 回目の条件節の解析 <Expression($\mathbf{T}$.expression, $\mathbf{G}$, $\mathbf{RA}$)>

[$\mathbf{recursiveRA}_n$ = $\mathbf{RA}$]

制御移行文による n(n > 1) 回目の繰り返しがない 以外,

n 回目の繰り返し節の解析 <Statement($\mathbf{T}$.statements[0], $\mathbf{G}$, $\mathbf{recursiveRA}_n$)>

n 回目の条件節の解析 <Expression($\mathbf{T}$.expression, $\mathbf{G}$, $\mathbf{recursiveRA}_n$)>

<post_Continue($\mathbf{T}$, $\mathbf{recursiveRA}_n$)>

n 回目の (条件節 + 繰り返し節) で定義されたエイリアスを抽出

[$\mathbf{recursiveRA}_n$ = $\mathbf{recursiveRA}_n$ − $\mathbf{RA}$]

n 回目の (条件節 + 繰り返し節) で定義されたエイリアスを may エイリアス化

[$\mathbf{recursiveRA}_n$ = toMay($\mathbf{recursiveRA}_n$)]

[$\mathbf{RA}$= $\mathbf{RA}$∪ $\mathbf{recursiveRA}_n$]

<post_Break($\mathbf{T}$, $\mathbf{RA}$)>

Algorithm For($\mathbf{T}$:Java for 文意味解析木, $\mathbf{G}$:AFG, $\mathbf{RA}$:到達エイリアス集合)

<pre_Local($\mathbf{T}$, $\mathbf{G}$, $\mathbf{RA}$)>

1 回目の初期化部の解析 <Statement($\mathbf{T}$.statement[i], $\mathbf{G}$, $\mathbf{RA}$)> ...

1 回目の条件節の解析 <Expression($\mathbf{T}$.expression, $\mathbf{G}$, $\mathbf{RA}$)>

[$\mathbf{recursiveRA}_1$ = $\mathbf{RA}$]

1 回目の繰り返し節の解析 <Statement($\mathbf{T}$.statements[i], $\mathbf{G}$, $\mathbf{recursiveRA}_1$)>

<post_Continue($\mathbf{T}$, $\mathbf{recursiveRA}_1$)>

制御移行文による n(n > 1) 回目の繰り返しがない 以外,

1 回目の更新部の解析 <Statement($\mathbf{T}$.statement[i], $\mathbf{G}$, $\mathbf{recursiveRA}_1$)> ...

[$\mathbf{recursiveRA}_n$ = $\mathbf{recursiveRA}_1$]

n 回目の条件節の解析 <Expression($\mathbf{T}$.expression, $\mathbf{G}$, $\mathbf{recursiveRA}_n$)>

n 回目の繰り返し節の解析 <Statement($\mathbf{T}$.statements[i], $\mathbf{G}$, $\mathbf{recursiveRA}_n$)>

<post_Continue($\mathbf{T}$, $\mathbf{recursiveRA}_n$)>

n 回目の更新部の解析 <Statement($\mathbf{T}$.statement[i], $\mathbf{G}$, $\mathbf{recursiveRA}_n$)> ...

n 回目の (条件節 + 繰り返し節 + 更新部) で定義されたエイリアスを抽出

[$\mathbf{recursiveRA}_n$ = $\mathbf{recursiveRA}_n$ − $\mathbf{RA}$]

n 回目の (条件節 + 繰り返し節 + 更新部) で定義されたエイリアスを may エイリアス化 [$\mathbf{recursiveRA}_n$ = toMay($\mathbf{recursiveRA}_n$)]

1 回目の (繰り返し節 + 更新部) で定義されたエイリアスを抽出

[$\mathbf{recursiveRA}_1$ = $\mathbf{recursiveRA}_1$ − $\mathbf{RA}$]

1 回目の (繰り返し節 + 更新部) で定義されたエイリアスを may エイリアス化 [$\mathbf{recursiveRA}_1$ = toMay($\mathbf{recursiveRA}_1$)]

[$\mathbf{RA}$= $\mathbf{RA}$∪ $\mathbf{recursiveRA}_1$ ∪ $\mathbf{recursiveRA}_n$]

<post_Break($\mathbf{T}$, $\mathbf{RA}$)>

<post_Local($\mathbf{T}$, $\mathbf{G}$, $\mathbf{RA}$)>

ALGORITHM SWITCH(**T**:JAVA for 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

式の解析 <EXPRESSION(**T**.expression, **G**, **RA**)>
[$\mathbf{caseRA}_1 = \ldots = \mathbf{caseRA}_N = \mathbf{RA}$]
**foreach** n in 1..N

n 個目の case ブロックから switch 文末まで解析
<STATEMENT(**T**.statement[i], **G**, $\mathbf{caseRA}_n$)> ...
n 個目の case ブロックから switch 文末までで定義されたエイリアスを抽出
[$\mathbf{caseRA}_n = \mathbf{caseRA}_n$ - **RA**]

**RA** から swicth ブロックで確実に定義されるエイリアスを削除
[**RA**= **RA**− selectMust($\mathbf{caseRA}_1$) ∩ ...∩ selectMust($\mathbf{caseRA}_N$)]
**foreach** n in 1..N

n 個目の case ブロックから switch 文末までで定義されたエイリアスを may エイリアス化 [$\mathbf{caseRA}_n$ = toMay($\mathbf{caseRA}_n$)]

[**RA**= **RA**∪ $\mathbf{caseRA}_1 \cup \ldots \cup \mathbf{caseRA}_N$]

ALGORITHM BREAK(**T**:JAVA break 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

break 節点**n** の作成, **G**.register(**n**), **n**.save(**RA**), [**RA**= **RA**∪ **n**]

ALGORITHM CONTINUE(**T**:JAVA continue 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

continue 節点**n** の作成, **G**.register(**n**), **n**.save(**RA**), [**RA**= **RA**∪ **n**]

ALGORITHM RETURN(**T**:JAVA return 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

式の解析 <EXPRESSION(**T**.expression, **G**, **RA**)>
return 節点**n** の作成, **G**.register(**n**), **n**.save(**RA**), [**RA**= **RA**∪ **n**]
MAout 節点**m** の作成, **G**.register(**m**)
**G**.line(**n**, **m**)

ALGORITHM SYNCHRONIZED(**T**:JAVA synchronized 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

式の解析 <EXPRESSION(**T**.expression, **G**, **RA**)>
**thenRA**= **elseRA**= **RA**
ブロックの解析 <STATEMENT(**T**.statement[0], **G**, **thenRA**)>

ALGORITHM THROW(**T**:JAVA throw 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

式の解析 <EXPRESSION(**T**.expression, **G**, **RA**)>

ALGORITHM TRY～CATCH～FINALLY(**T**:JAVA try 文意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**tryRA**= **RA**
try ブロックの解析 <STATEMENT(**T**.statement[0], **G**, **tryRA**)>
try ブロックで定義されたエイリアスを抽出 [**tryRA**= **tryRA**− **RA**]
try ブロックで定義されたエイリアスを may エイリアス化 [**tryRA**= toMay(**tryRA**)]
[**RA**= **RA**∪ **tryRA**]
**foreach** n in 1..N

[$\mathbf{catchRA}_n$ = **RA**]
n 個目の catch 節を解析 <Statement(**T**.statement[n], **G**, $\mathbf{catchRA}_n$)>
n 個目の catch 節で定義されたエイリアスを抽出
[$\mathbf{catchRA}_n$ = $\mathbf{catchRA}_n$ - **RA**]
n 個目の catch 節で定義されたエイリアスを may エイリアス化
[$\mathbf{catchRA}_n$ = toMay($\mathbf{catchRA}_n$)]

[**RA**= **RA**∪ $\mathbf{catchRA}_1 \cup \ldots \cup \mathbf{catchRA}_N$]
try ブロックの解析 <Statement(**T**.statement[i], **G**, **RA**)>
<post_Break(**T**, **RA**)>

Algorithm Expression(式)(**T**:Java 式意味解析木, **G**:AFG, **RA**:到達エイリアス集合):AFG 節点

Algorithm Operation(演算)(**T**, **G**, **RA**)

- "++", "−−", "−", "+", "−", " ", "!", "*", "/", "%", "++", "−−", ">>", ">>>", "<<", "<=", "<", ">=", ">", "&", "∧", "|", "||", "&&", "&=", "−=", "*=", "/=", "%=", ">>=", ">>>=", "<<=", "∧=", "|=", "==", "!=", "incetanceof"
  左辺の解析 <Statement(**T**.left, **G**, **RA**)>
  右辺の解析 <Statement(**T**.right, **G**, **RA**)>
  **return** $\phi$
- "+="
  左辺の解析 <$\mathbf{Node}_L$ = Statement(**T**.left, **G**, **RA**)>
  右辺の解析 <Statement(**T**.right, **G**, **RA**)>
  [**RA**= **RA**∪ $\mathbf{Node}_L$]
  **return** $\mathbf{Node}_L$
- "="
  左辺の解析 <$\mathbf{Node}_L$ = Statement(**T**.left, **G**, **RA**)>
  右辺の解析 <$\mathbf{Node}_R$ = Statement(**T**.right, **G**, **RA**)>
  [**RA**= **RA**∪ $\mathbf{Node}_L$]
  **G**.line($\mathbf{Node}_L$, $\mathbf{Node}_R$)
  **return** $\mathbf{Node}_L$
- "[]"
  配列の解析 <$\mathbf{Node}_L$ = Statement(**T**.left, **G**, **RA**)>
  添字の解析 <$\mathbf{Node}_R$ = Statement(**T**.right, **G**, **RA**)>
  [**RA**= **RA**∪ $\mathbf{Node}_L$]
  配列参照変数節点**n** の作成, $\mathbf{Node}_L$.register(**n**), **G**.register(**n**)
  **return n**
- "(キャスト式)"
  キャスト対象の解析 <$\mathbf{Node}_R$ = Statement(**T**.right, **G**, **RA**)>
  **return** $\mathbf{Node}_R$
- "?"
  左辺の解析 <Statement(**T**.left, **G**, **RA**)>
  右辺の解析 <$\mathbf{Node}_R$ = Statement(**T**.right, **G**, **RA**)>
  **return** $\mathbf{Node}_R$
- ":"
  左辺の解析 <$\mathbf{Node}_L$ = Statement(**T**.left, **G**, **RA**)>

右辺の解析 <**Node**$_R$ = Statement(**T**.right, **G**, **RA**)>
仲介節点**n** の作成, **G**.register(**n**)
**G**.line(**Node**$_L$, **n**)
**G**.line(**Node**$_R$, **n**)
**return n**

- ":"
  左辺の解析 <**Node**$_L$ = Statement(**T**.left, **G**, **RA**)>
  右辺の解析 <**Node**$_R$ = Statement(**T**.right, **G**, **RA**)>
  仲介節点**n** の作成, **G**.register(**n**)
  **return n**
- "." ... Q.Id
  Q の解析 <**Node**$_L$ = Statement(**T**.left, **G**, **RA**)>
  Id の解析 <**Node**$_R$ = Statement(**T**.right, **G**, **RA**)>
  **Node**$_L$.register(**n**)
  **return Node**$_R$

Algorithm VariableReference(変数参照)(**T**, **G**, **RA**)

変数参照節点**n** の作成, **G**.register(**n**)
**foreach m** in **RA**

**m** と **n** が同一識別子のとき,

**G**.line(**m**, **n**)

**return n**

Algorithm Literal(値)(**T**, **G**, **RA**)

リテラル節点**n** の作成, **G**.register(**n**)
**return n**

Algorithm ArrayInitializer(配列初期化子)(**T**, **G**, **RA**)

配列初期化子節点**n** の作成, **G**.register(**n**)
初期化子要素を解析 <Expression(**T**.elements[i], **G**, **RA**)> ...
**return n**

Algorithm ClassInstanciation(クラスインスタンス化)(**T**, **G**, **RA**)

クラスインスタンス化節点**n** の作成, **G**.register(**n**)
**foreach** i in 1..N

実引数を解析 <**Node**= Expression(**T**.actuals[i], **G**, **RA**)>
Actual-Alias-in 節点**m** の作成, **n**.register(**m**)
**G**.line(**Node**, **m**)

**return n**

Algorithm ArrayInstanciation(配列インスタンス化)(**T**, **G**, **RA**)

配列インスタンス化節点**n** の作成, **G**.register(**n**)
添字を解析 <Expression(**T**.actuals[i], **G**, **RA**)> ...
配列初期化子を解析 <$\mathbf{Node}_R$ = Expression(**T**.initializer, **G**, **RA**)>
**G**.line($\mathbf{Node}_R$, **n**) **return n**

Algorithm MethodInvocation(メソッド呼び出し)(**T**, **G**, **RA**)

メソッド呼び出し節点**n** の作成, **G**.register(**n**)
**foreach** i in 1..N

実引数を解析 <$\mathbf{Node}_R$ = Expression(**T**.actuals[i], **G**, **RA**)>
Actual-Alias-in 節点**m** の作成, **n**.register(**m**)
**G**.line($\mathbf{Node}_R$, **m**)

**return n**

Algorithm TypeReference(型参照)(**T**, **G**, **RA**)

型参照節点**n** の作成, **G**.register(**n**)
**return n**

Algorithm pre_Local(**T**:Java スコープ意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**foreach** i in **T**.variables

Local-Alias-in 節点**n** の作成, **G**.register(**n**)
[**RA**= **RA**$\cup$ $\mathbf{n}^{\P}$]

Algorithm post_Local(**T**:Java スコープ意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**foreach** i in **T**.variables

[**RA**= **RA**$-$ $\mathbf{n}^{\P}$]

Algorithm pre_Member(**T**:Java スコープ意味解析木, **G**:AFG, **RA**:到達エイリアス集合)

**foreach** i in **T**.variables

Instance-Alias-in 節点**n** の作成, **G**.register(**n**)
[**RA**= **RA**$\cup$ $\mathbf{n}^{\P}$]

Algorithm post_Member

**foreach** i in **T**.variables

Instance-Alias-out 節点**n** の作成, **G**.register(**n**)
**foreach m** in **RA**

**m** と **n** が同一識別子のとき,
**G**.line(**m**, **n**)

Algorithm post_Break(**T**:Java 文意味解析木, **RA**:到達エイリアス集合)

$\mathbf{RA}\_ = \phi$
**foreach n** in **RA**

**n** が break 節点で かつ ブレーク・ターゲットが**T** である場合,

[$\mathbf{RA\_} = \mathbf{RA\_} \cup \mathbf{n}$.load()]
[$\mathbf{RA} = \mathbf{RA} - \mathbf{n}$]

[$\mathbf{RA} = \mathbf{RA} \cup \mathbf{RA\_}$]

Algorithm post_Continue(**T**:Java 文意味解析木, **RA**:到達エイリアス集合)

$\mathbf{RA\_} = \phi$
**foreach n** in **RA**

**n** が continue 節点で かつ 継続ターゲットが**T** である場合,

[$\mathbf{RA\_} = \mathbf{RA\_} \cup \mathbf{n}$.load()]
[$\mathbf{RA} = \mathbf{RA} - \mathbf{n}$]

[$\mathbf{RA} = \mathbf{RA} \cup \mathbf{RA\_}$]

Algorithm post_Return(**T**:Java 文意味解析木, **RA**:到達エイリアス集合)

$\mathbf{RA\_} = \phi$
**foreach n** in **RA**

**n** が return 節点で かつ **T** がメソッドブロックである場合,

[$\mathbf{RA\_} = \mathbf{RA\_} \cup \mathbf{n}$.load()]
[$\mathbf{RA} = \mathbf{RA} - \mathbf{n}$]

[$\mathbf{RA} = \mathbf{RA} \cup \mathbf{RA\_}$]