プログラム静的解析法の効率化とフレームワーク構築に関する研究

大畑 文明

平成 14 年 2 月

内容梗概

ソフトウェアの品質改善，開発作業における生産性向上を目指し，さまざまな研究活動がなされている．その一つに，プログラムからその性質やふるまいを抽出し，それを提供することで開発者を支援する，プログラム解析がある．

これまでに数多くのプログラム解析手法が提案されている．これらの手法は，解析方針及び解析対象の分類による組合せを考えることで，それぞれを特徴付けすることができる．解析の方針による分類には，静的解析と動的解析がある．解析の対象による分類には，データフロー，制御フロー，エイリアス，手続き呼び出し，クラス階層，抽象構文木，意味情報などが代表的なものとして挙げられる．

しかし，既存のプログラム解析手法は，解析精度の向上，特定の解析対象に特化した解析アルゴリズムの提案及びその実装を重視してきた．そのため，(a) 解析の効率，(b) 解析コストと解析精度のトレードオフ制御，(c) 解析情報の二次利用に対する配慮の不足が問題となっている．

まずここで，これらの問題を解決するためのより具体的な目標を列挙する．

(a) 大規模化，複雑化するプログラムへの適用を実現するための，解析アルゴリズム及び解析手順の効率化

(b) ユーザの目的に応じて解析アルゴリズムが選択できる実装，及びコストと精度のトレードオフ制御ができる解析手法

(c) 二次利用を考慮し，かつその利用者に対する制約の少ない，解析情報データベース

本論文では，プログラムの静的解析に着目し，

(1) プログラムスライス（データフローと制御フローの組み合わせ）

(2) エイリアス

(3) 意味解析木（抽象構文木と意味情報の組み合わせ）

の抽出における (a) 解析の効率化手法の提案をそれぞれ行う．また，(b) 解析コストと解析精度のトレードオフ制御については (1)，(2) で，(c) 解析情報の二次利用に関しては (3) で議論する．

(1) では，プログラムスライスの抽出に利用するプログラム依存グラフに対して節点集約を適用し，節点数の減少による解析の効率化手法を提案した．これにより，精度の低下を抑えたコスト削減を得ることができる．提案手法は，解析コストと解析精度のトレードオフ制御を節点集約により実現したものであり，ユーザの要求に応じた使い分けが可能で

ある．また，提案手法を Pascal スライスシステム OSS に追加実装し，その有効性を評価した．

(2) では，エイリアスフローグラフを用いた，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した．これにより，ユーザの求めるエイリアスを効率よくかつ高い精度で抽出することができる．新たな解析アルゴリズムの定義や，既存の解析アルゴリズムの拡張も考慮されており，複数の解析アルゴリズムを容易に使い分けることができる．また，提案手法を Java エイリアス解析ツール JAAT として実装し，その有効性を評価した．

(3) では，XML による意味解析木のデータベース化手法を提案した．これにより，データベース化による解析手順の効率化に加え，XML を対象とするアプリケーションが数多く提供されていることから，解析情報の二次利用も容易である．また，提案手法の実装である意味解析木 XML データベースを JAAT に加えた，Java プログラム解析フレームワーク JAF を新たに構築し，提案手法の有効性を評価した．

論文一覧

主要論文

[1-1] Fumiaki OHATA, Akira NISHIMATSU, Katsuro INOUE, “Analyzing dependence locality for efficient construction of program dependence graph”, *Information and Software Technology*, vol.42, issue.13, pages.935–946, 2000, 学術雑誌 (論文)

[1-2] 大畑 文明, 近藤 和弘, 井上 克郎, “エイリアスフローグラフを用いたオブジェクト指向プログラムのエイリアス解析手法”, **電子情報通信学会論文誌**, vol.J84-D-I, no.5, pages.1–11, 2001, 学術雑誌 (論文)

[1-3] 大畑 文明, 横森 励士, 西松 顯, 井上 克郎, “スライス計算効率化のためのプログラム依存グラフの節点集約法”, **電子情報通信学会論文誌**, vol.J84-D-I, no.7, pages.24–32, 2001, 学術雑誌 (論文)

[1-4] Toshihiro KAMIYA, Fumiaki OHATA, Kazuhiro KONDO, Shinji KUSUMOTO, Katsuro INOUE, “Maintenance support tools for Java programs: CCFinder and JAAT”, *Proceedings of 23th International Conference on Software Engineering (ICSE2001)*, pages.837–838, May 16–28, 2001, Toronto, Canada, 国際会議 (フォーマル リサーチ デモンストレーション)

[1-5] Fumiaki OHATA, Kouya HIROSE, Masato FUJII and Katsuro INOUE, “A Slicing Method for Object-Oriented Programs Using Lightweight Dynamic Information”, *Proceedings of the 8th Asia Pacific Software Engineering Conference (APSEC2001)*, pages.273–280, December 4–7, 2001, Macau, China, 国際会議 (論文)

関連論文

[2-1] Toshihiro KAMIYA, Takuya UEMURA, Fumiaki OHATA, Shinji KUSUMOTO, Katsuro INOUE, “Osaka Slicing System”, 21th International Conference on Software Engineering (ICSE99), May 19–21, 1999, Los Angeles, California, 国際会議 (ポスターセッション)

[2-2] Yoshiyuki Ashida, Fumiaki Ohata, Katsuro Inoue, “Slicing Methods Using Static and Dynamic Information”, *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC’99)*, pages.344–350, December 7–10, 1999, Takamatsu, Japan, 国際会議 (論文)

[2-3] 誉田 謙二, 大畑 文明, 井上 克郎, “Java バイトコードにおけるデータ依存解析手法の提案と実現”, **コンピュータソフトウェア**, vol.18, no.3, pages.40–44, 2001, 学術雑誌 (ショートペーパー)

[2-4] Reishi Yokomori, Fumiaki Ohata, Yoshiaki Takata, Hiroyuki Seki and Katsuro Inoue, “Analysis and Implementation Method of Program to Detect Inappropriate Information Leak”, *Proceedings of the Second Asia-Pacific Conference on Quality Software (APAQS 2001)*, pages.5–12, December 10–11, 2001, Hong Kong, China, 国際会議 (論文)

[2-5] 高田 智規, 井上 克郎, 大畑 文明, 芦田 佳行, “制限された動的情報を用いたプログラムスライシング手法の提案”, **電子情報通信学会論文誌**, vol.J85-D-I, no.2, pages.1–8, 2002, 学術雑誌 (論文)

[2-6] 山中 祐介, 大畑 文明, 井上 克郎, “プログラム解析情報の XML データベース化 -提案と実現-”, **コンピュータソフトウェア**, vol.19, no.1, pages.39–43, 2002, 学術雑誌 (ショートペーパー)

謝辞

本研究の全般に関し，常日頃より適切な御指導を賜わりました，大阪大学大学院基礎工学研究科情報数理系専攻 井上克郎教授に，心から深く感謝申し上げます．

本論文を執筆するにあたり，適切な御助言と御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻 谷口健一教授，藤原融教授に，心から感謝いたします．

大阪大学大学院基礎工学研究科情報数理系専攻在籍中に，適切な御助言と御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻柏原敏伸教授，菊野亨教授，東野輝夫教授，今井正治教授，萩原兼一教授，村田正幸教授，増澤利光教授，宮原秀夫教授，橋本昭洋教授に感謝いたします．

本論文を執筆するにあたり，直接具体的な御指導を頂きました，大阪大学大学院基礎工学研究科情報数理系専攻 楠本真二助教授，松下誠助手に心より感謝いたします．

本研究を行うにあたり，御助言や御指導を頂きました，大阪大学基礎工学研究科情報数理系専攻（現 株式会社アールスリーインスティテュート） 西松顯氏に感謝いたします．

提案手法の実現にあたり，御協力を頂いた，大阪大学大学院基礎工学研究科情報数理系専攻 横森励士氏，近藤和弘氏，山中祐介氏に感謝いたします．

本研究を行うにあたり，御協力を頂いた，大阪大学大学院基礎工学研究科情報数理系専攻 高田智規氏，誉田謙二氏，藤井将人氏に感謝いたします．

本研究を行うにあたり，御協力を頂いた，大阪大学大学院基礎工学研究科情報数理系専攻（現 ＮＴＴソフトウェア株式会社） 芦田佳行氏に感謝いたします．

本研究を行うにあたり，御協力を頂いた，大阪大学大学院基礎工学研究科情報数理系専攻（現 ソニー株式会社） 廣瀬航也氏に感謝いたします．

最後に, 井上研究室の皆様の御助言, 御協力に御礼申し上げます．

目 次

図 目 次

表 目 次

第1章　まえがき

1.1　プログラム静的解析

コンピュータ上で動作するソフトウェアは大規模化の一途をたどっており，ソフトウェア開発はますます複雑なものとなってきている．また，ソフトウェアシステムが担う社会的役割もますます重要となっており，高品質なソフトウェアを効率良く開発することは，ソフトウェアに関する研究において重要なテーマとなっている [33].

ソフトウェアの品質改善，開発作業における生産性向上を目指し，さまざまな研究活動がなされている．その一つに，ソフトウェア開発工程において重要な要素の一つであるプログラム（または，そのソースコード）からその性質やふるまいを抽出し、それを開発者に提供することでソフトウェア開発を支援する，**プログラム解析**（*Program Analysis*）がある [28].

これまでに数多くのプログラム解析手法が提案されている．これらの手法は，解析方針及び解析対象の分類による組合せを考えることで，それぞれを特徴付けすることができる．ここでは，それぞれの分類に属する要素に関して，簡単な説明を交えながらその具体例を列挙する．

- 解析方針

  静的: 可能性のあるすべての実行において，一度でも起こる事象に関する情報をすべて収集する．**静的解析**（*Static Analysis*）と呼ばれる．

  動的: 特定の入力が与えられたある一つの実行において，そこで起こる事象に関する情報のみを収集する．**動的解析**（*Dynamic Analysis*）と呼ばれる．

- 解析対象

  データフロー: 変数を介した，データの流れ

  制御フロー: 制御構造による，制御（実行）の流れ

  エイリアス: 同一メモリ領域を指す可能性のある式の集合

  手続き呼び出し: 手続き間の呼び出し関係

  クラス階層: オブジェクト指向プログラムにおける，クラス間の継承関係

  抽象構文木: プログラムのソースコードを木構造で表現したもの

  意味情報: 識別子に関する，宣言と参照間の関係

  なお，プログラムにはこれら以外の解析対象も多く存在する．また，いくつかの解析対象を組み合わせることで，新たな解析対象が定義されることもある．

```
1:  readln(a);
2:  readln(b);
3:  max := a;
4:  min := b;
5:  if a < b then begin
6:     max := b;
7:     min := a
8:  end;
9:  writeln('Max: ', max);
10: writeln('Min: ', min).
```

図 1.1: (例) スライス

本論文では，プログラム解析の上記区分のうち，解析方針は静的，つまり静的解析に着目する．また解析対象には，

- データフローと制御フローを組み合わせた，プログラムスライス
- エイリアス
- 意味情報と抽象構文木を組み合わせた，意味解析木

に着目する．以降，プログラムスライス，エイリアス，意味解析木についてそれぞれ説明する．

1.2 プログラムスライス

プログラムスライス（*Program Slice*）は Weiser[44] により提案された．プログラムスライスは，大きく**静的スライス**（*Static Slice*）[44]，**動的スライス**（*Dynamic Slice*）[1, 29] に分けられる．前者は静的解析に基づき，入力データのすべての可能性を考慮した文間の依存関係が抽出される．後者は動的解析に基づき，その入力データによる実行系列間の依存関係が抽出される．本論文では主に静的スライスに着目し，以降，特に断りがない限り静的スライスを単にスライスと呼ぶ．

プログラムスライスとは，**スライス基準**（*Slicing Criterion*）（対 <s, v> で示され，s は文，v は s で定義若しくは参照される変数を表す）に影響を与える可能性のある文の集合を指す．図 1.1 に，サンプルプログラム及びスライス基準 <10, `min`>（太枠部）に対するスライス（網掛部）を示す．

プログラムスライスは，**プログラム理解**（*Program Understanding*），**プログラムデバッグ**（*Program Debugging*），**テスト**（*Testing*），**プログラム合成**（*Program Merging*）などに有効であるとされている [9, 8, 16, 27, 44]．我々の研究グループでも，プログラムデバッグ，プログラム保守におけるプログラムスライスの有効性に関して，実験による検証を行ってきた．これらの結果に基づき，我々はプログラムスライシングツールの実現における問題に取り組んでいる．

```
     ...
1:   b := 5;
2:   a := b + b;
3:   if a > 0 then
4:     c := a;
5:     d := b
     end;
     ...
```

図 1.2: (例) プログラム

文	定義変数	参照変数
1	b	
2	a	b
3		a
4	c	a
5	d	b

図 1.3: (例) 図 1.2 の各文の定義，参照変数

スライス計算にはさまざまな手法が存在するが，ここではPDGによるスライス抽出技法 [27] に着目する．これは，次の 4 フェーズで構成されており，

Phase 1: 定義，参照変数の抽出

Phase 2: 依存関係解析

Phase 3: PDG の構築

Phase 4: PDG によるスライスの抽出

以下，各フェーズを簡単に説明する．ただし，プログラムの構文解析，意味解析により**制御フローグラフ**（*Control Flow Graph, CFG*）[2] が得られているものとする．なお，ここで対象とする言語はポインタのない手続き型言語とする．ポインタを含むプログラムに対しては，既に提案されているポインタ解析手法 [34] を併用することで適用可能である．

□ Phase 1: 定義，参照変数の抽出

プログラムの各文で定義，参照される変数を抽出する．このフェーズはプログラム文の 1 回の走査で終わる．図 1.2 のサンプルプログラムにおける各文の定義変数，参照変数を図 1.3 に示す．

□ Phase 2: 依存関係解析

Phase 1 の結果を元に，プログラム文間の，次に示す 2 つの依存関係を抽出する．

表 1.1: (例) 図 1.2 のデータ依存関係と制御依存関係

データ依存関係	DD(1, b, 2), DD(2, a, 3)
	DD(1, b, 5), DD(2, a, 4)
制御依存関係	CD(3, 4), CD(3, 5)

プログラム中の 2 文 s，t に関して，以下の条件を満たすとき，s から t の間に**制御依存関係**（*Control Dependence Relation, CD Relation*）[27] が存在するという．

(1) s は条件文（節） かつ,

(2) t の実行は s の判定結果に依存する

この関係を $CD(s,t)$ と表す．また，以下の条件を満たすとき，s から t の間に変数 v に関する**データ依存関係**（*Data Dependence Relation, DD Relation*）[27] が存在するという．

(1) s は v を定義する かつ,

(2) t は v を参照する かつ,

(3) s から t への 1 つ以上の実行経路の中に，v の再定義が起こらない経路が少なくとも 1 つ存在する

この関係を $DD(s,v,t)$ と表す．表 1.1 に図 1.2 におけるデータ依存関係と制御依存関係を示す．制御依存関係の計算は Phase 1 の結果及び CFG から容易に行なうことができる．しかし，データ依存関係の条件 (3) の判定には，手続き呼び出しやポインタ等の解析が必要不可欠であり，繰り返し文，再帰呼び出しによる再帰的な依存関係も存在するため，スライス抽出過程の中でも最も計算量を必要とする．

□ Phase 3: PDG の構築

Phase 2 で抽出された依存関係を利用し，プログラム依存グラフを構築する．**プログラム依存グラフ**（*Program Dependence Graph, PDG*）[27] とは，プログラムに存在する文を節点，文間の依存関係（データ依存関係，制御依存関係）を辺で表現した有向グラフである．図 1.4 に図 1.2 のサンプルプログラムに対する PDG を示す．PDG は Phase 2 で抽出された依存関係から容易に構築可能である．

□ Phase 4: PDG によるスライスの抽出

スライス基準に対するスライスを抽出する．スライス基準 <s, v> に対するスライスとは，s に対応した PDG 中の節点 N_s から，逆方向に制御依存辺及びデータ依存辺を経て推移的に到達可能な節点集合に対応する文の集合をいう．図 1.5 に図 1.2 のスライス基準 <5, b> に対するスライス（網掛部，文 1，2，3，5）を示す．スライスの抽出は PDG を辿るのみであるため，依存関係解析に比べそれに要する計算量はわずかである．

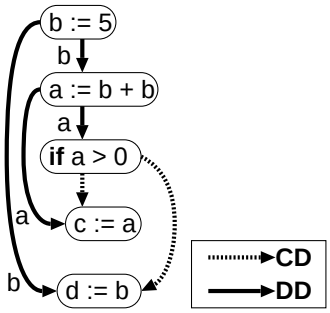


図 1.4: (例) 図 1.2 の PDG

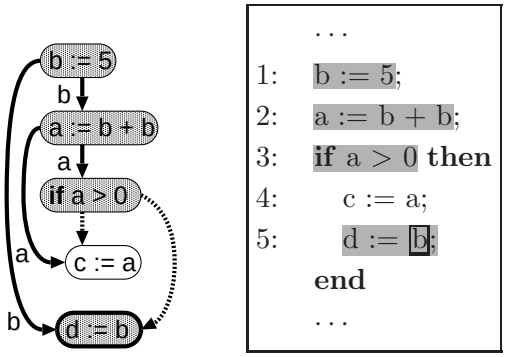


図 1.5: (例) 図 1.2 のスライス基準 <5, b> のスライス

1.3 エイリアス

1.3.1 エイリアス

プログラム上の式（または部分式）の対が同一のオブジェクト（メモリ領域）を指す場合，それらの式は**エイリアス関係**（*Alias Relation*）にあるという．エイリアス関係は，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によって生じる．エイリアス関係は同値関係であり，その同値類を**エイリアス集合**（*Alias Set*）と呼ぶ．また，プログラムを静的に解析しエイリアス集合を求めることを**エイリアス解析**（*Alias Analysis*）といい，コンパイラ最適化 [2, 17] や，プログラムスライスの計算に欠くことのできないものである．

□エイリアス解析とプログラムスライス

前節で述べたように，スライスの計算は

Phase 1: 定義，参照変数の抽出

Phase 2: 依存関係解析

Phase 3: PDG の構築

Phase 4: PDG によるスライスの抽出

という過程をたどるが，Phase 2 中のデータ依存関係解析において，各文でどの変数が定義，参照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ムに対する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不可能である．

□エイリアス解析とコンパイラ最適化

コンパイラ最適化でのエイリアス解析の利用例を図 1.6 を用いて示す．

```
1:   int a[], b[];
2:   void f(int i, int j) {
3:      int *p, *q;
4:      int x, y;
5:      p = &a[i];
6:      q = &b[j];
7:      x = *(q + 3);
8:      *p = 5;
9:      y = *(q + 3);
10:     g(x, y);
11:  }
```

(a) プログラム（最適化前）

```
1:   int a[], b[];
2:   void f(int i, int j) {
3:      int *p, *q;
4:      int x;
5:      p = &a[i];
6:      q = &b[j];
7:      x = *(q + 3);
8:      *p = 5;
9:
10:     g(x, x);
11:  }
```

(b) プログラム（最適化後）

図 1.6: (例) エイリアス解析とコンパイラ最適化

これは C 言語で書かれたサンプルプログラムであるが，解析によりポインタ変数 `p`，`q` はエイリアスを生成しないと判断でき，文 `y = *(q + 3)` の省略及び変数 `y` の変数 `x` への置き換えが可能となる．

□エイリアス解析とプログラム理解

また，オブジェクト指向言語の一つである JAVA[18] は C++[39] 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．

```
1:  class Employee {
2:    String name; int salary; Employee supervisor;
3:    Employee(String n, int s) {
4:      name = n; salary = s; supervisor = null;
5:    }
6:    void add_salary(int n) {
7:      salary += n;
8:    }
9:    void set_supervisor(Employee e) {
10:     supervisor = e;
11:   }
12:   void print() {
13:     System.out.println(name + " Salary:" + salary);
14:   }
15: }
16: class Manager extends Employee {
17:   Manager(String n, int s) {
18:     super(n, s);
19:   }
20:   void manage(Employee e) {
21:     e.set_supervisor(this); e.add_salary(200);
22:   }
23: }
24: class Office {
25:   public static void main(String args[]) {
26:     Employee Emp = new Employee("Emp", 750);
27:     Manager Mng = new Manager("Mng", 750);
28:     Mng.manage(Emp);
29:     Emp.print();
30:     Mng.print();
31:   }
32: }
```

(a) プログラム

```
% java Office
Emp Salary: 950
Mng Salary: 750
```

(b) (a) の実行結果（フォールトあり）

```
% java Office
Emp Salary: 750
Mng Salary: 950
```

(c) (a) の実行結果（フォールトなし）

図 1.7: (例) エイリアス解析とプログラム理解

図 1.7(a) にサンプルプログラムを，図 1.7(b) その実行結果を示す．このプログラムは従業員 Emp と経営者 Mng の給料を計算する．経営者の給料は従業員の給料よりも大きくなければならないが，このプログラムでは Emp に給料が上乗せされていることが，文 29 における出力異常から認識できる．ユーザはこの欠陥を発見すると，文 29 にある参照変数 Emp に関するエイリアスの抽出を試みる．本論文では，エイリアス解析の対象となる式を，**エイリアス基準**（*Alias Criterion*）（対 <s，e> で表され，s は文を e は s 中に存在する式を表す）．網掛部がエイリアス基準 <29，Emp>（太枠部）に対するエイリアス解析の結果である．なお，下線の引かれている式及び定義は，該当するエイリアスが呼び出す可能性のあるメソッドに対応している．このエイリアス結果を参照することで，文 21 の給料上乗せに関する式がフォールトの原因であることが突き止められる．この場合，`e.add_salary(200)` を `add_salary(200)` に変更することで，図 1.7(c) に表されるような正しい計算結果が得られるようになる．

1.3.2 エイリアス解析

エイリアス解析は，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，それぞれを図 1.8 を用いて簡単に説明する．

□ FI エイリアス解析

FI エイリアス解析 [36, 34] 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い，**エイリアスグラフ**（*Alias Graph*）を利用する．図 1.8(a) に変数 c（太枠部）の FI エイリアス（網掛部）を，図 1.8(c) にその計算に用いたエイリアスグラフを示す．エイリアスグラフは無向グラフであり，節点はメモリ領域を指しうる変数及び式を，辺は（代入や引数の参照渡しなどにより）節点間に直接のエイリアス関係があることを表す．文 7 の変数 c に関するエイリアスを求める場合，エイリアスグラフの c に対応する節点から到達可能な節点は {a, b, **new** `Integer(1)`, **new** `Integer(2)`)} であり，網掛部が求めるエイリアスとなる．

□ FS エイリアス解析

FS エイリアス解析 [13, 45] 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い，**到達エイリアス集合**（*Reaching Alias Set, RAset*）を利用する．図 1.8(b) に変数 c（太枠部）の FS エイリアス（網掛部）を，図 1.8(d) に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 $RA(s)$ の要素は 文 s の実行直前において成立し かつ 文 s において識別子を介して参照可能なエイリアス集合であり，その集合の各要素は (文番号，式) の組で表わされる．文 7 の変数 c に関するエイリアスを求める場合，$RA(7)$ 内の c を含むエイリアス集合を探索する．変数 c は集合 [(6, c), (2, a), (6, 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．

□ FI エイリアス解析と FS エイリアス解析

FS 解析はプログラム文の実行順を考慮しているため，FI 解析と比較し時間計算量，空間計算量を必要とするが，解析の精度は高く，実際に図 1.8 においてもその差は顕著に現

```
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) FI エイリアス解析

```
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);
```

(b) FS エイリアス解析

(c) (a) のエイリアスグラフ

文 (s)	到達エイリアス集合 ($RA(s)$)
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), (4, b), (3, **new** Integer(2))]}
6	{[(2, a), (2, **new** Integer(1)), [(4, c), (5, c), (3, b), (4, b), (3, **new** Integer(2))]}
7	{[(6, c)], [(2, a)], [(6, a)], [(2, **new** Integer(1))], [(3, b), (4, b), (3, **new** Integer(2))]}

(d) (b) の到達エイリアス集合

図 1.8: (例) FI エイリアス解析と FS エイリアス解析

れている．両者の実験的比較は [22, 23, 24, 49] でなされており，本論文では解析精度を重視していることから特に FS 解析に着目する．

1.4 意味解析木

一般にプログラムのソースコードはテキストで書かれており，プログラミング言語は言語固有の文法を持っている．**抽象構文木**（*Abstract Syntax Tree, AST*）とは，この文法に従い記述されたソースコードを木構造で表現したものである．抽象構文木は，演算子，識別子名，修飾子などを表現する**字句情報**（*Lexical Information*）と，文の構造を表現する**構文情報**（*Syntax Information*）を持つ．

また，プログラムには，**意味情報**（*Semantic Information*）と呼ばれる，変数名や型名などの識別子に関する宣言と参照間の関係が存在する．

そして，これら抽象構文木と意味情報とを組み合わせたものを**意味解析木**（*Semantic Tree*）と呼び，多くのプログラム解析手法において，その存在は前提のものとなっている．

1.5 既存のプログラム解析における問題点

既存のプログラム解析に関する研究においては，これまで述べたようにさまざまな解析手法が提案されてきた．しかし，解析精度の向上，特定の解析対象に特化した解析アルゴリズムの提案及びその実装を重視してきたため，以下の点に対する配慮不足が問題となっている．

(a) 解析の効率

解析精度を向上させるにはより細粒度の解析が必要であり，例えば，解析の単位を文から式に変更する，また，構造体要素を区別するなどが考えられる．

しかし，プログラミング言語の高級化，プログラムの大規模化に伴い，解析に要するコストも増大している．これは，同一の解析方針を持つ手法であっても，対象言語の文法が複雑になるにつれ解析アルゴリズムも複雑になり，また，プログラムの規模が大きくなるほど解析コストも増大することによる．

そのため，高級言語で記述された大規模プログラムに対してプログラム解析を適用する際には，高い解析精度もさることながら，解析アルゴリズム及び解析手順の効率化が重要となる．

(b) 解析コストと解析精度のトレードオフ制御

解析アルゴリズムの性能評価に利用する尺度に，解析に要する計算機資源の大きさと解析結果のきめ細かさがある．前者を**解析コスト**（*Analysis Cost*），後者を**解析精度**（*Analysis Precision*）という．

一般に，これら二つの評価値は**トレードオフ関係**（*Trade-off Relation*）にあり，解析コストを抑えるためには解析精度の低下は避けられず，解析精度の向上のためには解析コストの増大は避けられない．解析アルゴリズムを効率化することで解析精度を保ったまま解析コストを抑えることも可能であるが，効率化にも限界がある．そのため，各解析手法が持つ解析精度，解析コストに関する特徴を把握し，ユーザの目的に見合った解析手法が選択が必要となる．

しかし，既存の解析手法においては，コストと精度のトレードオフ関係に異なる特徴を持つものにはその基本概念に共通性が少なく，解析アルゴリズムの共有が困難であることが多い．そのため，複数の解析手法の実装に手間がかかる．また，コストと精度のトレードオフの制御を前提とした解析手法についてはほとんど議論されていない．

そのため，ユーザの目的に応じて解析アルゴリズムを選択できる実装，及びコストと精度のトレードオフが制御できる解析手法が望まれる．

(c) 解析情報の二次利用

これまでに多くの解析手法が実装されてきたが，解析により得られた情報はメモリ上にのみ記憶されるのが一般的である．つまり，別の解析での利用を全く考慮していな

かったり，例え考慮していたとしても，特定のプログラミング言語による API を強いられることが多い．

そのため，二次利用を考慮し，かつその利用者への制約が少ない，解析情報データベースが望まれる．

1.6 本論文の概要

本論文では，静的解析に基づく，(1) プログラムスライス，(2) エイリアス，(3) 意味解析木，以上 3 つの抽出に対する (a) 解析の効率化手法の提案をそれぞれ行う．また，(b) 解析精度と解析コストのトレードオフ制御については (1)，(2) で，(c) 解析情報の二次利用に関しては (3) で考察する．

(1) プログラム依存グラフの節点集約によるスライス計算の効率化

プログラムスライスの抽出に利用するプログラム依存グラフにおいて，通常各文の依存情報はグラフの対応する 1 節点に保持させるが，複数文の依存情報を 1 節点で保持させる節点集約を利用し，節点数の減少による解析の効率化手法をを 2 つ提案した．一つは精度の低下を抑えた空間コスト，時間コスト削減手法，もう一つは空間コストが若干増加するが精度の低下のない時間コスト削減手法である．提案手法は，解析コストと解析精度のトレードオフ制御を節点集約により実現したものであり，ユーザの要求に応じた使い分けが可能である．

提案手法を Pascal スライスシステム OSS に追加実装し，その有効性を検証した．

(2) エイリアス情報のモジュール化によるエイリアス解析の効率化

エイリアスフローグラフを用いた，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した．これにより，ユーザの求めるエイリアスを効率よくかつ高い精度で抽出することができる．新たな解析アルゴリズムの定義や，既存の解析アルゴリズムの拡張も考慮されており，複数の解析アルゴリズムを容易に使い分けることができる．また，実利用を考慮したエイリアス解析システムの構築にも有効であるといえる．

提案手法を Java エイリアス解析ツール JAAT として実装し，その有効性を検証した．JAAT は，解析部，ユーザインターフェース部から構成されている．解析部は，JDK 附属クラスライブラリのような大規模プログラムに対しても効率よく解析できる．ユーザインターフェース部は，プログラム保守，プログラム理解を支援するための視覚的なエイリアス表示が可能で，適用事例を交えながらその有効性についても考察した．

(3) XML データベースを利用したプログラム解析の効率化

解析の効率化，解析情報の二次利用の容易性の向上を目的とする，XML によるプログラム解析情報データベース化手法を提案した．データベース化の対象としては，プログラム解析での利用頻度が非常に高い意味解析木を採用した．これにより，データベース化による解析手順の効率化に加え，XML を対象とするアプリケーションが数多く提供されていることから，解析情報の二次利用も容易に行うことができる．

提案手法の実装である意味解析木 XML データベースを JAAT に加え，Java プログラム解析フレームワーク JAF を新たに構築し，解析手順の効率化に対する有効性を検証した．また，XML データベースを扱う応用アプリケーションをいくつか試作し，二次利用の容易性についても考察した．

以下，2 章では，プログラム依存グラフの節点集約によるスライス計算の効率化について述べる．3 章では，エイリアスフローグラフを利用したエイリアス関係のモジュール化による効率化について述べる．4 章では，XML データベースを利用したプログラム解析の効率化について述べる．最後に 5 章で本論文の研究について纏め，今後の研究の方針について述べる．

第2章　プログラム依存グラフの節点集約による スライス計算の効率化

2.1　導入

スライス抽出における依存関係解析の効率に関して，我々は以下の 2 点に着目している．

- 依存関係解析のコスト

  一般に，スライス抽出に必要となるプログラムの依存関係解析は多くの時間，空間を消費する．例えば，28,000 行の C プログラムに対し 2 時間を要したとされる報告もある [41]．我々はスライスを用いた対話形式のプログラム解析ツールを求めており，依存関係解析のコストの増大はその実現の妨げとなる．

- 依存関係解析の精度

  スライス抽出はプログラム文間の依存関係に基づいて行われるが，依存関係の抽出精度を高めるにはその解析コストの増大は避けられない．加えて，現在主流となっているポインタ，配列を含む言語で記述されたプログラムの解析は更に困難となる．

依存関係解析におけるコストと精度のトレードオフに関しては，多くの研究がなされている [5, 15]．我々もこれまでに，スライシングシステムの実利用の観点から，スライスに関するさまざまな問題に取り組んできており，具体的なものとしては，スライシングシステムの構築 [51]，データフロー計算の効率化 [52]，PDG の効率的な更新 [55]，軽量な動的情報を利用したスライス精度の改善 [32] などがある．

ここでは，PDG を用いてコストと精度のトレードオフを考える．PDG の構築はスライス抽出に必要不可欠なものであり，その中でも，プログラム文間の依存関係把握のためのデータフロー計算に多くの時間，空間を要する．

解析効率を高めるには様々な手法が考えられる．その一つとして，通常各文の依存情報は PDG の対応する 1 節点に保持させるが，複数文の依存情報を 1 つの PDG 節点に保持させる節点集約による手法が考えられる．この節点集約を用いることにより，PDG 節点数は減少し解析コストの削減が期待できる．

本章では，節点集約によるコスト削減手法を目的別に 2 つ提案する．一つは精度の低下を抑えた空間コスト，時間コスト削減手法（手法 1），もう一つは空間コストが若干増加するが精度の低下のない時間コスト削減手法（手法 2）である．また，既存の Pascal スライスシステム OSS に提案手法を追加実装しその有効性を評価した．いくつかのサンプルプログラムに対する検証の結果，空間コスト 10 – 40%，時間コスト 5 – 60%の削減が得られた．

以降，2.2 では節点集約について述べ，2.3 で手法 1 を，2.4 で手法 2 をそれぞれ提案する．2.5 で提案手法の実現について述べる．2.6 で提案手法の評価を行い，2.7 でまとめと今後の課題について述べる．

2.2 節点集約

2.2.1 方針

通常 Phase 2 に最も手間がかかる．我々は，この依存関係解析の手間を小さくするための手法として，**節点集約**（*Node Merging*）（以降，単に集約と呼ぶ）を提案する．本手法を用いることにより，情報の共有による情報量（空間コスト）削減や，節点数の減少による計算量（時間コスト）削減が期待できる．

集約の方針を次に示す．なお，**非集約 PDG**（*Non-merged PDG*）とは従来手法で用いられる集約のない PDG である．一方，**集約 PDG**（*Merged PDG*）とは今回提案する集約が行われた PDG である．また，各 PDG を用いて抽出されるスライスをそれぞれ，**非集約スライス**（*Non-merged Slice*），**集約スライス**（*Merged Slice*）と呼び，集約された節点，集約のされていない節点を，それぞれ**集約節点**（*Merged Node*），**非集約節点**（*Non-merged Node*）と呼ぶ．

方針 1: 集約は Phase 1 及び Phase 2 と独立したフェーズで行い，Phase 1，Phase 2 の変更は最小限に抑える．

方針 2: Phase 1 及び CFG から得られる情報のみを用いて集約を行い Phase 2 に渡す．

方針 3: 方針 1，2 に関連するが，集約は後述する集約条件が成り立つ連続文に限定する．また，手続き呼び出し文及びそれを内包する制御文は集約対象としない．

方針 4: 同じスライス基準に対し，非集約 PDG を用いて抽出されたスライス $\mathcal{A}$ には含まれなかった文が，集約 PDG を用いて抽出されたスライス $\mathcal{B}$ に含まれること（精度の低下）があるが，$\mathcal{A}$ に含まれる文は必ず $\mathcal{B}$ に含まれる．

方針 5: 局所依存関係という，データ依存関係，制御依存関係を組み合わせた依存関係を新たに定義し節点集約に利用する．局所依存関係については次節で述べる．

2.2.2 局所依存関係

以下の条件のいずれかを満たす場合，文 s は文 t に対し変数 α に関する**局所依存関係**（*Local Dependence Relation, LD Relation*）が成り立つといい，$LD(s, \alpha, t)$ と表す．なお，前述のとおり，局所依存関係はデータ依存関係及び制御依存関係を組み合せたものであり，α を定義する s 及びそれを参照する t 間には，α の再定義が起らない経路が少なくとも 1 つ存在することが前提となる．

- s で α が定義され，t で α が参照される．(図 2.1(a))

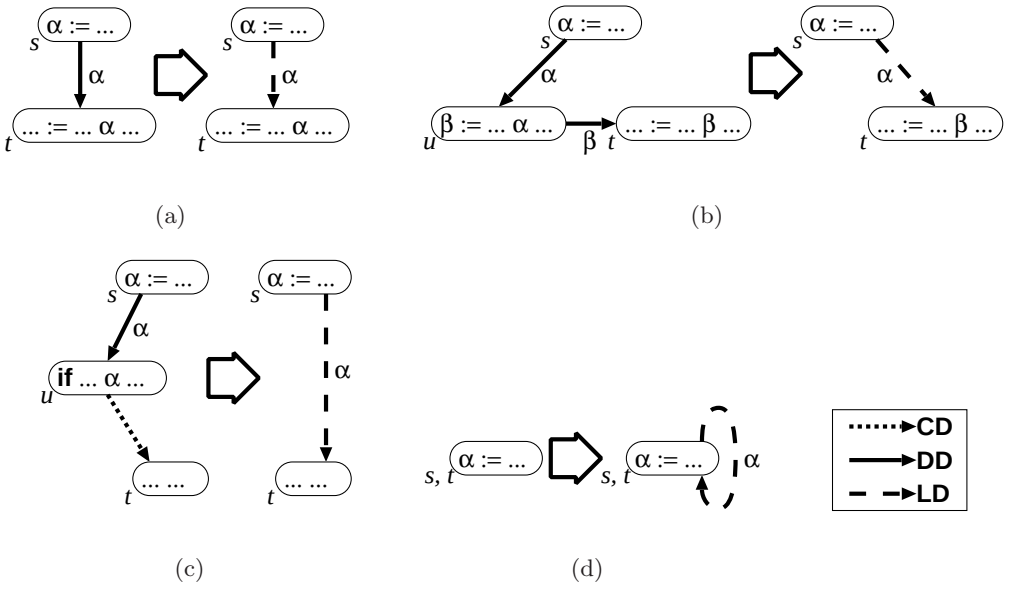


図 2.1: (定義) 局所依存関係

- α を参照し β を定義する文 u が存在し，かつ s が α を定義し t が β を参照する．(図 2.1(b))
- α を参照する条件文（繰り返し文）u が存在し，t はその分岐節（繰り返し節）であり（$CD(u,t)$），かつ s は α を定義する．(図 2.1(c))
  なお，このような α を，t を**支配**（*Control*）する変数という．
- s と t は同じ文であり，かつ t は α を定義する．(図 2.1(d))

局所依存関係の抽出は Phase 1 及び CFG から得られる情報を用いることで可能であり，一段の間接依存まで調べている．多段の間接依存を認めることも可能であるが，計算コストの増大が予想されるため行っていない．また，局所依存関係は後述する集約可否判定にのみ用いられ，PDG 辺として存在することはない．

2.2.3 節点集約

集約はプログラム中の 2 つ以上のとなりあう文（定義は後述）を対象としており，ここでは連続する 2 文の集約に関してのみ述べる．この手法を繰り返し適用することで 3 文以上の集約もできる．なお，離れた 2 文にも同様の方法は適用可能であるが，Phase 2 – 4 の実装への変更が必要であり対象としていない．

□となりあう文

プログラム中の文 s と文 t が**となりあう文**（*Adjacent Statements*）であるとは，以下の条件をすべて満たすときをいう．

- s と t は構文的に隣接している（ただし，s が `goto` 文であり かつ t が対応する飛び先ラベルを持つ場合は除く）
- s と t の集約がプログラム中の制御構造を破壊しない（例えば，s が `if` 文 u の `then` 節の最後の文 かつ t が u の次に実行される文であるとき，u と t はとなりあう文ではない）
- s，t いずれも手続き呼び出し文ではない

なお，この関係は，PDG 上での**となりあう節点**（*Adjacent Node*）という関係に置き換えることができ，非集約節点と集約節点との集約の判定にも利用される．反対に，非集約節点，集約節点は，それぞれ，プログラム上における単一の文，集約された文集合とみなすことができる．例えば，`then` 節，`else` 節いずれも 1 節点に集約できれば，これらはとなりあう節点となる．

□集約条件

となりあう 2 文 x，y について，x は変数 $X_1, X_2, \cdots, X_m$ を参照，y は変数 $Y_1, Y_2, \cdots, Y_n$ を参照しているとする．いま変数集合 $V \equiv \{X_1, X_2, \cdots, X_m, Y_1, Y_2, \cdots, Y_n\}$ に属する各変数 v に関して，$LD(s, v, x) \wedge LD(s, v, y)$ を満たす文 s がそれぞれ存在する場合，あるいは s が存在しないような変数がある場合でもその個数が $limit$ 個以下であれば，x，y を 1 節点に集約する．

□集約の制御

集約の程度は $limit$ 値（$limit \geq 0$）により制御され，$limit$ を大きくすることで集約は進み，$limit$ を小さくすることで集約は抑えられる．コストと精度はトレードオフ関係にあり，目的に応じた $limit$ 値の選定が必要となる．本章では，その中でも特徴的な $limit$ 値に着目し，先に述べた節点集約手法を拡張した 2 つの集約方法を提案する．

手法 1: 精度の低下を抑えた時間コスト，空間コスト削減のための，依存関係の局所性を利用した節点集約法（$limit \ll \infty$）

手法 2: 空間コストは若干増加するが精度の低下のない時間コスト削減のための，節点分解を伴う節点集約法（$limit = \infty$）

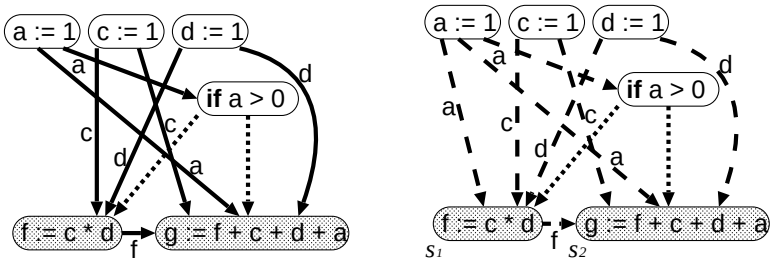
その他の $limit$ 値に関しては，特性を見い出せなかったため今回は扱っていない．また $limit \ll \infty$ の具体的な値として，2.6 では 0, 1, 2 を評価対象としている．

2.2.4 適用例

図 2.2 のサンプルプログラムの文 7，8 に対する $limit = 0$ での集約を例に挙げる．図 2.3(a) は DD，CD による依存グラフ，図 2.3(b) は LD，CD による依存グラフである（ただしいずれのグラフも，文 7，8 に関連する依存辺及びその辺に接する節点で構成された部分グラフである）．変数 `c`，`d` は文 7 で，変数 `f`，`c`，`d`，`a` は文 8 で参照されている．`c` に関して $LD(3, \texttt{c}, 7) \wedge LD(3, \texttt{c}, 8)$，`d` に関しても同様に $LD(4, \texttt{d}, 7) \wedge LD(4, \texttt{d}, 8)$ が

```
     ...
1:   a := 1;
2:   b := 1;
3:   c := 1;
4:   d := 1;
5:   if a > 0 then
6:     e := b;
7:     f := c * d;
8:     g := f + c + d + a
     end;
9:   h := g;
10:  i := e;
     ...
```

図 2.2: (例) プログラム



(a) DD，CD による依存グラフ（PDG の一部）

(b) LD，CD による依存グラフ

図 2.3: (例) 図 2.2 の節点集約

成り立つ．f，a それぞれについても，$LD(7, \mathtt{f}, 7) \wedge LD(7, \mathtt{f}, 8)$，$LD(1, \mathtt{a}, 7) \wedge LD(1, \mathtt{a}, 8)$ が成り立つ．それゆえ，2 文 7，8 は 1 節点に集約される．

ここでは説明のためにグラフを用いたが，集約対象をとなりあう文に限定していることで $LD(s, \alpha, t)$ における文 s に該当する文を把握する必要はなくなり，集約判定は変数の集合演算に置き換えることができる．これは，集約対象文で同一識別子の変数が参照されている場合，その変数を定義した文は同じであることによる．ただし，集約対象文内でその変数が再定義される可能性があるが，その把握は容易である．

2.2.5 アルゴリズム

集約の判定には，集約対象文に関する以下のような情報が必要であるが，

- 定義，参照される変数名

- 分岐節（繰り返し節）内の文かどうか
- となりあう文（節点）の情報

これらの情報は Phase 1 及び CFG から取得できる．

集約はプログラム構造に依存するため，連続文，ブロック，分岐文，繰り返し文の集約アルゴリズム（集約判定及び集約手続き）をそれぞれ考えた．これらは後述するスライスシステムに Phase 1.5（Phase 1 – 2 間）として実装されている．

- 連続文 [$\cdots$; $\cdots$] (AlgorithmSequencial，図 2.5(a))
- ブロック [`begin` $\cdots$ `end`] (AlgorithmBlock，図 2.5(b))
- 分岐文（`else` 節なし）[`if` $\cdots$ `then` $\cdots$](AlgorithmSelection，図 2.5(c))
- 分岐文（`else` 節あり）[`if` $\cdots$ `then` $\cdots$ `else` $\cdots$](AlgorithmSelection', 図2.5(d))
- 繰り返し文 [`while` $\cdots$ `do` $\cdots$](AlgorithmIteration，図 2.5(e))

図 2.4 に節点集約に関わる要素定義を，図 2.5 に各構文要素ごとの集約アルゴリズムを示す．図 2.4 において，節点 N は自身に関する 4 つの集合（CTL(N)，USE(N)，DEF(N)，poDEF(N)）を持つ．節点集約フェーズによる Phase 1 や Phase 2 への影響を防ぐため，集約節点と非集約節点は同一構造をなす．図 2.5 の各**判定・手続き**では，**入力**節点集合が集約可能であるかを判定し，もし集約可能と判定されれば，**出力**節点に関する変数集合を計算する．また，**関連**は対応する構文要素での集約において満たされる関係式を表わしている．

N: （集約 または 非集約）節点

CTL(N): 節点 N を支配する変数の集合
$(\mathrm{CTL}(N_S) = \{v \mid {}^{\exists}N_T(v \in \mathrm{USE}(N_T) \wedge \mathrm{CD}(N_T, N_S))\})$

USE(N): 節点 N で参照される変数の集合

DEF(N): 節点 N で確実に定義される変数の集合

poDEF(N): 節点 N で定義される可能性のある変数の集合
$(\mathrm{DEF}(N) \subseteq \mathrm{poDEF}(N))$

図 2.4: (要素定義) 節点集約

入力: 節点 N_A，節点 N_B

出力: 集約節点 N_S（集約可能な場合）

判定・手続き:

if $|\ \mathrm{USE}(N_A) \cup \mathrm{USE}(N_B) - \mathrm{USE}(N_A) \cap \mathrm{USE}(N_B) - \mathrm{CTL}(N_S) - \mathrm{poDEF}(N_A) \cap \mathrm{USE}(N_B)\ | \leq limit$ **then**

begin

$\mathrm{USE}(N_S) = \mathrm{USE}(N_A) \cup (\mathrm{USE}(N_B) - \mathrm{DEF}(N_A))$;

$\mathrm{DEF}(N_S) = \mathrm{DEF}(N_A) \cup \mathrm{DEF}(N_B)$;

$\mathrm{poDEF}(N_S) = \mathrm{poDEF}(N_A) \cup \mathrm{poDEF}(N_B)$

end

関連: $\mathrm{CTL}(N_S) = \mathrm{CTL}(N_A) = \mathrm{USE}(N_B)$

(a) AlgorithmSequential

関連: AlgorithmSequential の再帰的適用による

(b) AlgorithmBlock

入力: 条件節節点 N_A，`then` 節節点 N_B

出力: 集約節点 N_S（集約可能な場合）

判定・手続き:

if $|\ \mathrm{USE}(N_B) - \mathrm{USE}(N_A)\ | \leq limit$ **then**

begin

$\mathrm{USE}(N_S) = \mathrm{USE}(N_A) \cup \mathrm{USE}(N_B)$;

$\mathrm{DEF}(N_S) = \emptyset$;

$\mathrm{poDEF}(N_S) = \mathrm{poDEF}(N_B)$

end

関連: $\mathrm{CTL}(N_B) = \mathrm{USE}(N_A)$

(c) AlgorithmSelection

図 2.5: (アルゴリズム) 節点集約

入力: 条件節節点 N_A，`then` 節節点 N_B，`else` 節節点 N_C

出力: 集約節点 N_S（集約可能な場合）

判定・手続き:

if $|\ \mathrm{USE}(N_B) \cup \mathrm{USE}(N_C) - \mathrm{USE}(N_B) \cap \mathrm{USE}(N_C) - \mathrm{USE}(N_A)\ | \leq limit$ **then**
begin
$\mathrm{USE}(N_S) = \mathrm{USE}(N_A) \cup \mathrm{USE}(N_B) \cup \mathrm{USE}(N_C))$;
$\mathrm{DEF}(N_S) = \mathrm{DEF}(N_B) \cap \mathrm{DEF}(N_C)$;
$\mathrm{poDEF}(N_S) = \mathrm{poDEF}(N_B) \cup \mathrm{poDEF}(N_C)$
end

関連: $\mathrm{CTL}(N_B) = \mathrm{CTL}(N_C) = \mathrm{USE}(N_A)$

(d) AlgorithmSelection'

入力: 条件節節点 N_A，繰り返し節節点 N_B

出力: 集約節点 N_S（集約可能な場合）

判定・手続き:

if $\mathrm{poDEF}(N_B) \cap \mathrm{USE}(N_A) \neq \emptyset$ **or** $|\ \mathrm{USE}(N_B) - \mathrm{USE}(N_A)\ | \leq limit$ **then**
begin
$\mathrm{USE}(N_S) = \mathrm{USE}(N_A) \cup \mathrm{USE}(N_B)$;
$\mathrm{DEF}(N_S) = \emptyset$;
$\mathrm{poDEF}(N_S) = \mathrm{poDEF}(N_B)$
end

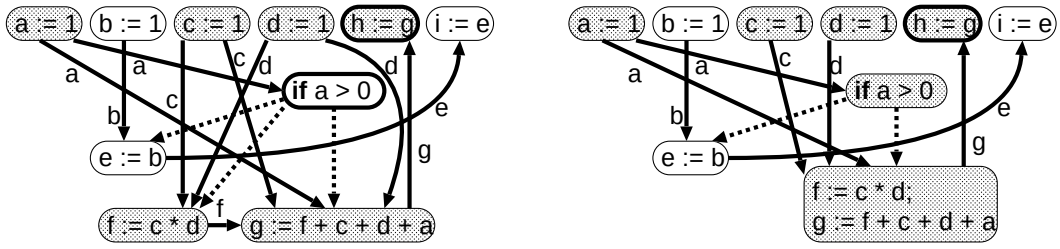
関連: $\mathrm{CTL}(N_B) = \mathrm{USE}(N_A)$

(e) AlgorithmIteration

図 2.5: (アルゴリズム) 節点集約（続き）

集約 PDG に対するスライス計算は，一般的な PDG 探索を適用することで容易に行うことができる．また，方針で述べたように，節点集約は**誤った**スライス結果をもたらすことはない（ここでいう**誤った**とは，同一スライス基準に対し，非集約スライスに含まれる文が集約スライスに含まれないことを指す.）．それは以下の理由による．

> 2 文 s，t が集約節点 N に集約されたとき，図 2.5 に示すアルゴリズムに基づき，s，t が保持していた定義変数及び参照変数の集合は N に委譲される．データ依存関係は変数情報から生成されるものであり，となりあう文は同じ制御依存関係を持つことから，s，t に関する依存情報が集約により失なわれるこ



(a) 非集約 PDG　　(b) 集約 PDG

図 2.6: (例) 図 2.2 のスライス基準 <9, g> のスライス

とはない.

2.3 依存関係の局所性を利用した節点集約法（手法 1）

本節では，依存関係の局所性を利用した節点集約法について述べる．本手法により，精度の低下を抑えた時間コスト，空間コストの削減を得ることができる．

2.3.1 依存関係の局所性

依存関係の局所性（*Dependency Locality*）とは，直観的には対象となる複数文が持つ依存関係の類似性を表したものであり，$limit \ll \infty$ で集約可能な文にその性質が現れる．複数文を 1 節点に集約することを考えたとき，集約対象文に関するすべての依存関係を集約節点に委譲させる必要があるが，単純に連続文を集約した PDG をスライス抽出に利用するとスライスの精度は著しく低下してしまう．しかし，$limit = 0$ のとき集約可能な 2 文 s，t は**ともに同じ文集合 $\mathcal{U}$ に局所依存している**ため，いいかえると，同一の $\mathcal{U}$ に依存する s，t を集約しているため，集約による精度の低下を抑えることができる．

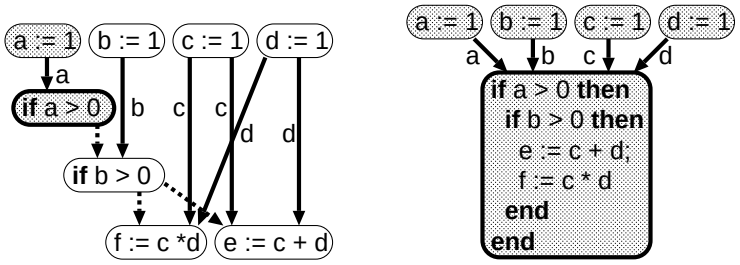
$limit = 0$ の集約で精度の低下が起こる状況は 2 つ存在する．一つは，文 s，t のいずれかのみ非集約スライスに含まれるようなスライス基準に対し集約スライスを抽出するときである．このとき，集約スライスは s，t いずれも含むことになる．もう一つは，集約節点で参照のみされる変数をスライス基準としたときである．こちら関しては 2.3.3 で述べる．

2.3.2 適用例

図 2.2 のサンプルプログラムに対する，非集約 PDG，集約 PDG（文 7，8 を $limit = 0$ で集約）を図 2.6(a)，(b) にそれぞれ示す．スライス基準 <9, g> に対するスライスを非集約 PDG 及び集約 PDG を用いて抽出したとき，文 7，8 が依存関係の局所性を有することから，非集約 PDG によるスライス（文 1，3，4，5，7，8，9）と同じスライスが集約 PDG からも抽出されていることが分かる．

```
     ...
1:   a := 1;
2:   b := 1;
3:   c := 1;
4:   d := 1;
5:   if a > 0 then
6:     if b > 0 then
7:       e := c + d;
8:       f := c * d
       end
     end;
     ...
```

図 2.7: (例) プログラム



(a) 非集約 PDG　　(b) 集約 PDG

図 2.8: (例) 図 2.7 のスライス基準 <5, a> のスライス

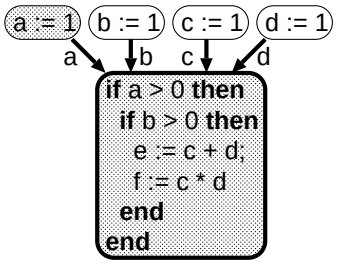
2.3.3 支配表

$limit \ll \infty$ の集約により，スライスの精度低下を抑えた集約を行なうことができる．しかし集約節点で参照のみされる変数をスライス基準としたとき，得られるスライスの精度が著しく低下する可能性がある（集約節点で定義される変数をスライス基準としたときや，PDG 探索中に集約節点に到達したときには，このような問題は生じない）．集約により集約前節点間に存在した制御依存に関する情報は失われ，定義変数は当然であるが参照変数であっても，すべての参照変数に依存していると解釈せざるを得ない．例えば，図 2.7 のサンプルプログラムのスライス基準 <5, a> に対するスライスにおいては，集約によりスライスサイズが 2（図 2.8(a)）から 8（図 2.8(b)）に増加している．

この問題を解決するため，同一集約節点に存在する参照変数間の支配関係を記憶する支配表を集約節点に持たせた．**支配関係**（*Control Relation*）とは，変数 α が条件節で参照され変数 β が対応する述部で参照されるとき，α と β の間に存在する関係をいう．また，α が β を支配するともいう．前述のとおり，集約節点内の制御依存に関する情報は破棄されるため，この支配関係でその役割を補う．また，**支配表**（*Control Table*）とは n 行 n 列

支配 \ 被支配	a	b	c	d
a		√	√	
b			√	√	√
c				√	
d					√

(a) 支配表



(b) 集約 PDG

図 2.9: (例) 支配表を利用した図 2.7 のスライス基準 <5, a> のスライス

(n は各集約節点の参照変数の数) の表であり，支配関係の存在する組には √ で，存在しない組は空欄で表す．支配表は集約時に逐次更新され，集約節点の参照のみされる変数 a がスライス基準となったとき，支配表を参照することで a が真に依存している参照変数を把握できる．

図 2.8 の集約節点の支配表及びそれに基づき抽出された集約節点（太枠部）の変数 a のスライス（網掛部）を，図 2.9(a)，図 2.9(b) にそれぞれ示す．支配表から集約節点の変数 a はそれ自身にのみ依存していることが分かり，スライスサイズは 8（図 2.8(b)）から 5（図 2.9(b)）に減少している．

2.4 節点分解を伴う節点集約法（手法 2）

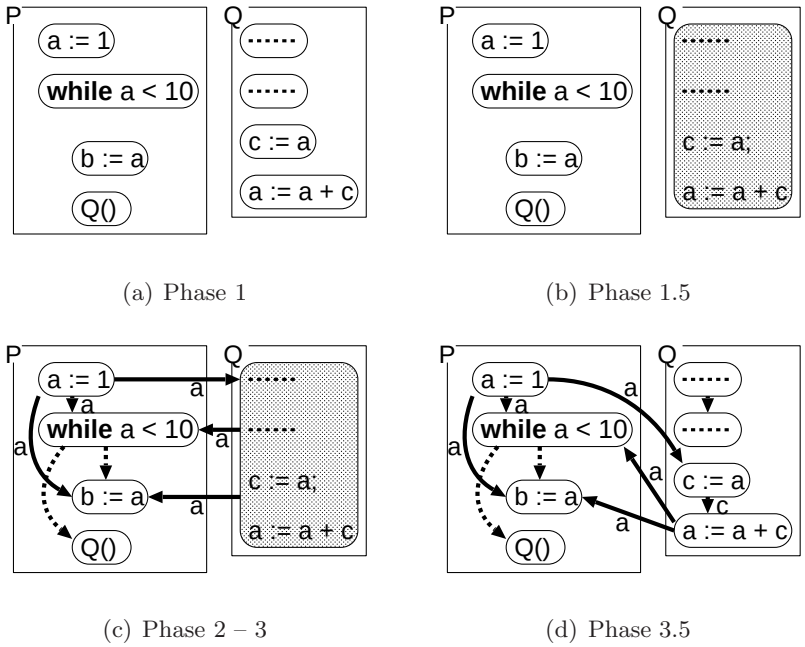
2.3 では，依存関係の局所性を有する $limit \ll \infty$ 集約による，精度の低下を抑えた節点集約法を提案した．依存関係の局所性を利用した集約では，空間コスト，時間コストの削減を行うことができる．一方 $limit = \infty$ とすると，コスト削減は十分に期待できるが，精度の大幅な低下は避けられない．そこで，$limit = \infty$ の集約による PDG を構築後，その集約節点を分解する手法が考えられる．この手法は，節点の分解を考慮する必要があるため空間コストが若干増加するが，精度の低下のない時間コスト削減を行うことができる．また $limit = \infty$ では局所依存関係を抽出する必要はなく，集約可能な文（手続き呼び出し文及びそれを内包する制御文以外）はすべて集約される．

2.4.1 節点分解

節点分解（*Node Decomposition*）とは，Phase 1.5 を経て構築された集約 PDG の集約節点を分解し，その内部の依存関係のみ解析することで非集約 PDG を再構成することをいう．

ここで，データ依存関係を**大域的な依存関係**（*Global Dependence Relarion*）と**局所的な依存関係**（*Local Dependence Relation*）に分け，それぞれを以下のように定義する．

大域的な依存関係: 手続きをまたぐ依存関係や再帰的な依存関係などを指し，その抽出に要する解析範囲が大きい



(a) Phase 1　(b) Phase 1.5

(c) Phase 2 – 3　(d) Phase 3.5

図 2.10: (例) 節点分解を伴う節点集約

局所的な依存関係: 手続き内で閉じた依存関係や非再帰的な依存関係などを指し，その抽出に要する解析範囲が小さい

前述のとおり，スライス抽出過程の中でも最も時間計算量を要するのはデータ依存関係の抽出であるが，それは大域的な依存関係の存在によるものが多い．そこで，$limit = \infty$ での集約により大域的な依存関係の抽出対象となる節点を可能な限り減らし，その抽出コストを削減させる．一方，集約により 1 つの集約節点にまとめられていた節点間の依存関係は局所的な依存関係に該当し，分解時にそれら節点間の依存関係解析のみ行うことで抽出可能である．

2.4.2 適用例

図 2.10(a) は Phase 1 終了後の手続き `P`，`Q` を表しており，Phase 1.5 の $limit = \infty$ の節点集約により図 2.10(b) となる．その後 Phase 2 – 3 により図 2.10(c) の集約 PDG が構築される．`Q` は `P` から呼び出されており，かつその呼び出し文は `while` 文の繰り返し節に存在するため，`Q` 内の依存関係解析を少なくとも 2 回行わなければならない．しかし，`Q` 内を $limit = \infty$ で集約することでデータ依存関係解析の対象節点が減り，`P` から `Q` におよぶ大域的な依存関係解析のコスト削減が得られる．最後に集約節点内の局所的な依存関係解析を行い，図 2.10(d) の非集約 PDG が再構成される．

ここでは説明のため比較的単純な例を用いたが，`Q` がより大規模である場合や，再帰呼び出し経路に含まれている場合，集約によるコスト削減の効果はより大きなものとなる．

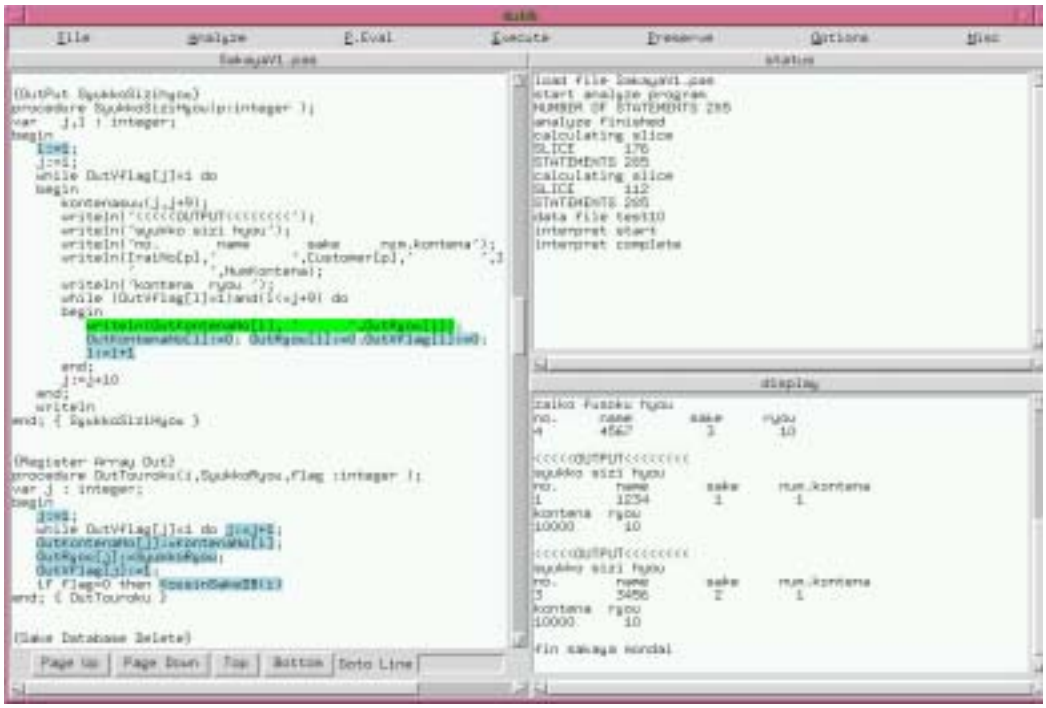


図 2.11: (実装) Pascal スライスシステム

2.4.3 アルゴリズム

$limit = \infty$ の節点集約により構築された集約 PDG に対し，各集約節点内部で分解アルゴリズムを適用し非集約 PDG を再構成する．分解アルゴリズムは，Phase 2 の依存関係解析アルゴリズムとほぼ同じものとなっている．異なるのは，集約前節点に関する依存関係は対応する集約節点が代わりに保持しているため，集約節点内部の依存関係解析後，集約節点に関する依存関係を分解後（集約前）の節点に分散（図 2.10(c) – 図 2.10(d)）させなければならない点である．依存関係の分散が終わると，これらの依存関係を元に非集約 PDG の再構成を行う．分解アルゴリズムは Phase 3.5（Phase 3 – 4 間）として実装されている．

2.5 Pascal スライスシステム Osaka Slicing System (OSS)

提案手法を我々の研究グループで開発した Pascal プログラムを対象とするスライスシステム [51] 上に実現した．節点集約の実装は前節で述べた方法に基づいて行った．

システム，**解析部**（*Analysis Subsystem*）と**ユーザーインタフェース部**（*UI Subsystem*）で構成されており，画面スナップを図 2.11 に示す．以降，解析部及びユーザーインターフェース部をそれぞれ簡単に説明する．

表 2.1: (定義) 節点集約による PDG 構築手法に関わる要素

記号	概要
N	文の総数
V	変数の総数

2.5.1 解析部

解析部は C で記述されており，アルゴリズムは [52] による．Pascal プログラムのソースコードは抽象構文木に変換され，メモリ上に記憶される．抽象構文木はユーザからの入力により解析され PDG に変換される．スライスは指定されたスライス基準により PDG 上で計算される．プログラムや静的スライスは，インタプリタにより実行することもできる．

2.5.2 ユーザーインターフェース部

ユーザーインターフェース部は C で記述されており，Tcl/Tk[40] ツールキットを使用している．ユーザは対話的で視覚的なエディタによりプログラム編集が可能である．また，トレース，ブレークポイントの設定などの機能もある．

2.6 評価

2.6.1 アルゴリズムの複雑さ

節点集約による PDG 構築手法の複雑さについて述べる．ここでは，節点集約，PDG 構築，節点分解に要するコストに限定している．表 2.1 にその関連する要素を挙げる．

□節点集約（Phase 1.5）に要するコスト

節点集約は，各文を 1 度走査することで終る．各文で実際に集約を行うには，集約後の節点における参照変数，定義変数，支配表を計算する必要があるため，$O(V^2)$ のコストがかかる．よって，時間コストは $O(N \cdot V^2)$ となる．また，空間コストに関して，支配表のセル数は $O(N \cdot V^2)$ となる．

□ PDG 構築（Phase 1，2，3）に要するコスト

PDG 構築では，新たな依存関係が抽出されなくなるまで文を繰り返し解析しなければならない．依存関係の数は $O(N^2)$ で抑えられる．依存関係を抽出するためには，各文における到達定義集合を計算しなければならないが，その演算回数は $O(N)$ で抑えられる．よって，最悪時の時間コストは $O(N^3)$ となる． また，節点数は $O(N)$ で，辺数は $O(N^2)$ であることから，空間コストは $O(N^2)$ で抑えられる．

つまり，節点集約により N の数が減るため（集約 PDG と非集約 PDG における解析コストのオーダは同じである），我々は全体の解析コストの削減を得られるのである．

□節点分解（Phase 3.5）に要するコスト

表 2.2: (統計データ) 評価用プログラム

プログラム	行	手続き	概要
P_1	333	14	チケット予約
P_2	429	18	酒屋問題
P_3	449	30	小計算問題の集合
P_4	831	22	ソーティング

節点分解では，各集約節点において PDG 構築と同等の解析を行い，非集約 PDG を構築する．そのため，時間コストは $O(N^3)$，空間コストは $O(N^2)$ で抑えられる．

2.6.2 実験

実験は以下に示す種類の PDG の比較を行った．

N: 集約なし

L_0: 依存関係の局所性を利用した節点集約（手法 1，$limit = 0$）

L_1: 依存関係の局所性を利用した節点集約（手法 1，$limit = 1$）

L_2: 依存関係の局所性を利用した節点集約（手法 1，$limit = 2$）

C: 節点分解を伴う節点集約（手法 2）

今回使用したプログラムの統計データを表 2.2 に，PDG 節点数を表 2.3 に（括弧内は集約節点数を表す），PDG 辺数を表 2.4 に，支配表のセル数を表 2.5 に，節点数，辺数，支配表セル数の単純和を表 2.6 に，PDG 構築までの時間（Phase 1 – 3 の合計時間，集約，分解を行う場合は Phase 1.5，Phase 3.5 もそれぞれ含まれる）を表 2.7 に，平均スライスサイズ（評価用プログラムの各手続きで最後に参照される変数をスライス基準として選び，それらのスライスに含まれる文数の平均）を表 2.8 に示す．

2.6.3 考察

□空間コスト（PDG 節点数，PDG 辺数，支配表セル数）

依存関係の局所性を利用した節点集約に関して，PDG 節点数は 8.85 – 39.05%（表 2.3），PDG 辺数は 6.70 – 18.15%（表 2.4）の削減が得られた．節点数に比べて辺数の削減率が小さいが，これは，

- 辺の始点と終点に対応する 2 節点間で集約が行われる，または
- ある 2 辺に関して，始点（終点）に対応する節点が共通で，終点（始点）に対応する異なる 2 節点間で集約が行われる

表 2.3: (実験結果) PDG 節点数 [個]

	非集約	集約			分解
プログラム	**N**	$\mathbf{L_0}$ (limit = 0)	$\mathbf{L_1}$ (limit = 1)	$\mathbf{L_2}$ (limit = 2)	**C**
P_1	169	119(26) (−29.59%)	114(27) (−32.54%)	103(34) (−39.05%)	180(11) (+6.51%)
P_2	211	166(22) (−21.33%)	153(27) (−27.49%)	136(28) (−35.55%)	231(20) (+9.48%)
P_3	243	199(19) (−18.11%)	187(24) (−23.05%)	177(28) (−27.16%)	270(27) (+11.11%)
P_4	503	459(124) (−8.75%)	419(150) (−16.70%)	409(165) (−18.69%)	547(44) (+8.75%)

表 2.4: (実験結果) PDG 辺数 [本]

	非集約	集約			分解
プログラム	**N**	$\mathbf{L_0}$ (limit = 0)	$\mathbf{L_1}$ (limit = 1)	$\mathbf{L_2}$ (limit = 2)	**C**
P_1	935	833 (−10.91%)	817 (−12.62%)	774 (−17.22%)	935
P_2	1,487	1,387 (−6.72%)	1,336 (−10.15%)	1,290 (−13.25%)	1,487
P_3	1,092	980 (−10.26%)	951 (−12.91%)	912 (−16.48%)	1,092
P_4	3,360	3,135 (−6.70%)	2,791 (−16.93%)	2,750 (−18.15%)	3,360

ときに限り，辺数が削減されるためである．L_0 – L_2 には今回新たに定義した支配表が導入されているが（表 2.5），支配表の各セルは真偽の 2 値のみ保持し，辺，節点に必要な情報量に比較すると十分に小さく，ビット演算による実現も可能である．

節点分解を伴う節点集約に関して，表 2.3 に示すように集約前節点の保存のため節点数が増加している（N 及び L_1 と比較すると，それぞれ約 10%，約 40%の増加となっている）．

□時間コスト（PDG 構築時間，Phase 1，(1.5)，2，3，(3.5)）

依存関係の局所性を利用した節点集約に関して，解析時間は 4.14 – 27.38%の削減が得られた（表 2.7）．L_0 と L_1 間での大幅な時間削減に比べ，L_1 と L_2 間では節点数，辺数は削減されるものの時間削減は少ない．このことから，PDG 中で最も解析時間の要する節点集合は $limit \leq 1$ で集約されていることが推測される．一方，$limit = 2$ で集約された節点集合は，解析時間を必要としない，依存関係が非再帰形をなす部分に存在したといえる．また，集約（Phase 1.5）に要する時間はいずれのプログラムに対しても 10ms 以下で

表 2.5: (実験結果) 支配表セル数 [個]

プログラム	非集約	集約			分解
	N	$\mathbf{L}_0$ (limit = 0)	$\mathbf{L}_1$ (limit = 1)	$\mathbf{L}_2$ (limit = 2)	**C**
P_1	0	59	87	176	0
P_2	0	81	125	183	0
P_3	0	26	45	102	0
P_4	0	150	212	237	0

表 2.6: (実験結果) 節点，辺数，支配表セル数の和

プログラム	非集約	集約			分解
	N	$\mathbf{L}_0$ (limit = 0)	$\mathbf{L}_1$ (limit = 1)	$\mathbf{L}_2$ (limit = 2)	**C**
P_1	1104	1011	1018	1053	915
		(−8.42%)	(−7.79%)	(−4.62%)	(−17.12%)
P_2	1698	1634	1614	1609	1704
		(−3.77%)	(−4.95%)	(−5.24%)	(+0.45%)
P_3	1335	1205	1183	1191	1081
		(−9.74%)	(−11.39%)	(−10.79%)	(−19.03%)
P_4	3863	3744	3442	3396	1811
		(−3.08%)	(−11.42%)	(−12.09%)	(−53.12%)

あった．

節点分解を伴う節点集約に関して，解析時間は 16.04 – 61.90%の削減が得られた（表 2.7）．$limit = \infty$ の集約により，解析時間の大半を占める大域的な依存関係の解析時間が削減されたためである．一方，局所的な依存関係の解析に要する計算コストは少なく，分解に要する時間は P_4 で 70ms であった．また，分解（Phase 3.5）に要する時間はいずれのプログラムに対しても 10ms 以下であった．

□精度（スライスの平均文数）

依存関係の局所性を利用した節点集約に関して，非集約スライスと比べ集約スライスのサイズは多少大きくなる．相互に依存する 2 文が集約された場合，集約によるスライスサイズの増加はないが，それ以外の場合は一般にスライスサイズは増加する．しかし，依存関係の局所性を有する文のみ集約するため，スライスの精度が大きく低下することはない．表 2.8 で示されているように，L_0，L_1 とも 1 – 3%程度のスライスサイズ増加で抑えられている．P_4 の L_2 に関して，平均スライスサイズは N と比較して 12.18%まで増加している．これは，$limit = 2$ による集約により，あるスライス基準に対するスライスサイズが 6 から 134 に大きく増加したものが存在したためである．この問題は，$limit = 2$ のような，多少の依存関係の違いを許容する方針の場合に起きうる．しかし，$limit$ 値を制御するこ

表 2.7: (実験結果) PDG 構築時間（Phase 1，(1.5)，2，3，(3.5)）[ms]

	非集約	集約			分解
プログラム	**N**	$\mathbf{L_0}$ (limit = 0)	$\mathbf{L_1}$ (limit = 1)	$\mathbf{L_2}$ (limit = 2)	**C**
P_1	102.53	80.789 (−21.21%)	80.573 (−21.42%)	74.48 (−27.38%)	69.258 (−32.45%)
P_2	191.01	161.792 (−15.30%)	157.557 (−17.51%)	150.603 (−21.15%)	129.192 (−32.36%)
P_3	187.84	161.494 (−14.03%)	156.074 (−16.91%)	147.365 (−21.55%)	157.712 (−16.04%)
P_4	739.172	708.597 (−4.14%)	607.979 (−17.75%)	585.679 (−20.77%)	281.638 (−61.90%)

Celeron-450MHz-128MB(FreeBSD)

表 2.8: (実験結果) スライスの平均文数 [文]

	非集約	集約			分解
プログラム	**N**	$\mathbf{L_0}$ (limit = 0)	$\mathbf{L_1}$ (limit = 1)	$\mathbf{L_2}$ (limit = 2)	**C**
P_1	94.21	95.21 (+1.06%)	95.21 (+1.06%)	95.21 (+1.06%)	94.21
P_2	143.22	145.00 (+1.24%)	147.44 (+2.95%)	148.28 (+3.53%)	143.22
P_3	46.80	48.27 (+3.13%)	48.27 (+3.13%)	52.50 (+12.18%)	46.80
P_4	173.43	178.52 (+2.94%)	178.57 (+2.97%)	178.57 (+2.97%)	173.43

とでその影響範囲を制限することが可能である．実際，$limit = 1$ とすることで，平均スライスサイズの増加は 3.13%に抑えられた．

節点分解を伴う節点集約に関して，定義からも分かるように解析精度への影響はなく，平均スライスサイズは非集約 PDG によるスライスと同じである（表 2.8）．

□推奨する *limit* 値

これらの実験結果から，依存関係の局所性を利用した節点集約では $limit = 1$ が最も有効であると考えられる．

limit 値が 0，1，2 と大きくなるにつれ，解析時間の短縮は期待できるが反対にスライスサイズは増加する．L_0 と L_1 間ではスライスサイズに影響をほとんど及ぼすことなく解析時間の短縮が得られているが，L_2 では P_6 のスライスサイズの大幅な増加が確認されている．一方，節点数，辺数，支配表セル数の単純和（表 2.6）を考えたとき，$limit = 1$ ま

で減少傾向であったものが $limit = 2$ で増加傾向に変化している．これら精度，時間コスト，空間コストの 3 点を考慮すると，$limit = 1$ を妥当な値として導き出すことができる．なお，$limit \geq 3$ についても検証を行ったが，20 – 40%程度の平均スライスサイズの増加は避けられず，有効な結果は見い出せなかった．

今回は実装の言語制約のため，比較的規模の小さいプログラムに対する実験のみであった．残念ながら，現時点では存在するすべてのプログラムに対して $limit = 1$ が有効であると断定することはできないが，今後，大規模プログラムに対する検証も行いたいと考えている．

2.6.4 関連研究

スライスに関してさまざまな研究がなされている．コストと精度とのトレードオフに関するものとしては [5, 15] がある．

[5] では，ユーザが解析コストとスライス精度を操作できるシステムを開発しているが，手続き間解析における**呼び出し元情報**（*Calling Context*）の考慮の有無によりトレードオフを制御している．我々は PDG 上の節点集約により制御を行っている．

[15] では，3 段階（制御フローの考慮の有無，呼び出し元情報の考慮の有無 の組み合わせ）の解析精度をユーザが選択可能なシステムを構築している．このアプローチは，実利用を考慮した解析ツールの実現に有効なアイディアを提供しているが，節点集約については議論されていない．

[41] では相互依存する節点のみの集約を行っている．ここでは，PDG 節点中の強連結成分を 1 節点に集約するアプローチを採用しているが，どの段階でこの集約が行われるかは不明である．我々の手法は，相互依存しなくとも同じ依存関係を持つ節点であれば集約を行う．また，*limit* 値を変更することでコストと精度のトレードオフが制御可能である．

2.7 まとめ

節点集約による PDG 構築コスト削減手法を 2 つ提案し，その有効性を実験により評価した．依存関係の局所性を利用した節点集約法は，スライスサイズが大きくなる副作用が存在するが，その増加はわずかであり，空間コスト，時間コストの削減が可能である．コストと精度のトレードオフは *limit* 値により制御可能で，実験結果から $limit = 1$ が最も有効であると思われる．節点分解を伴う節点集約法は，空間コストが増加するが精度の低下のない時間コスト削減を行うことができる．これらの手法はそれぞれ特徴を持っており，ユーザの要求に応じた使い分けが可能である．例えば，限られた計算機資源で大規模プログラムを解析する場合，依存関係の局所性を利用した節点集約法が有効であろう．提案したアルゴリズムはポインタ変数を扱っていないが，ポインタ変数の存在するプログラミング言語に対するアルゴリズムの拡張も可能である．

本研究をふまえ，Java を対象言語とする新しいスライスシステム及びプログラム理解ツールの開発を計画している．また，ソフトウェア開発工程でのスライスシステムの有効性をこのシステムを用いて評価したいと考えている．また，大規模プログラムに対する有効性検証も行いたい．

謝辞

本研究は，一部文部省科学研究費補助金特定領域研究 (A)(2)(課題番号:10139223) の補助を受けた.

第3章　エイリアス情報のモジュール化によるエイリアス解析の効率化 - Javaエイリアス解析ツールの開発 -

3.1　導入

エイリアス解析は，ポインタ変数に関する静的解析手法として，C や Pascal などの手続き型言語に対して提案された [36, 13, 34, 45]．しかし現在のプログラム開発環境において，手続き型言語だけでなく，Java や C++などの**オブジェクト指向言語**（*Object-Oriented Languages*）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**（*Class*），**オブジェクト**（*Object*），**継承**（*Inheritance*），**動的束縛**（*Dynamic Binding*），**多態**（*Polymorphism*）などの新しい概念が導入されており [10]，それらに対応したエイリアス解析手法が提案されている [43, 11]．しかし，これまでのエイリアス解析手法は，解析アルゴリズムに焦点が置かれ，解析ツールとしての実用性，スケーラビリティに関してはほとんど考慮されていない．我々は，Java などのオブジェクト指向言語に対する，実利用を考慮したエイリアス解析ツールの実現を目指しているが，既存手法には [25] に述べられているように，以下の問題がある．

スケーラビリティ: プログラムは大規模化の一途をたどっており，また，そのプログラムが利用するクラスライブラリ自身も大規模化，複雑化し続けていることから，プログラム解析は大規模プログラム及び関連するクラスライブラリが扱えるよう，スケーラビリティを満たさなければならない．しかし，多くの解析手法において，そのエイリアス解析の実装が不十分であるがゆえに，有益な解析結果を導出できない事例が多く，さらなる研究を必要としている [25]．

解析結果の利用と解析手法: 既存の解析手法では，エイリアス解析の適用範囲として，コンパイラ最適化，データフロー解析の前処理に焦点を当てていた．これらの用途においては，対象プログラム中に存在するすべてのエイリアス関係を抽出する必要があった．我々は，エイリアス解析はプログラム保守やプログラム理解においても有効である [25] と考えており，このような用途では，すべてのエイリアス関係を一度に抽出する必要はなく，ユーザの要求したエイリアス関係のみを即座に抽出できることが望まれる．そのためには，**オンデマンド解析**（*On-demand Analysis*）に基づくエイリアス解析アルゴリズムが必要になる．

本章では，上記の問題に対処するため，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．提案手法は 2 フェーズで構成されている．Phase 1 で，すべてのプログラム及びクラスライブラリに対し，モジュール内のエイリアス解析を行う．Phase

2 で，ユーザからの要求に応じてモジュール間のエイリアス解析を行う．この 2 フェーズ構成によるエイリアス解析により，解析時間の大幅な短縮を得ることができる．

提案手法では解析精度にも注目する．プログラム解析において，プログラム実行順の考慮の有無は解析精度や解析コストに対して大きな影響を与える [22, 23, 24, 49]．プログラム実行順を考慮しない FI 解析は一般に実装が容易であるが，解析精度が低い．また，オブジェクト指向プログラムでは，同一クラスのインスタンスを区別する，インスタンス単位での解析を行うことで，クラス単位での解析に比べ，高い解析精度を得ることができる．しかし，その実現のためには多大な解析コストを必要とする．提案手法では，許容可能な解析コストの範囲内でプログラムの実行順を考慮し，かつインスタンス単位での解析の実現を目指す．

2 フェーズ構成での解析手法は，その解析アルゴリズムの拡張に対しても有効である．これは，各フェーズにおいて，既存アルゴリズムを解析精度と解析コストに関して異なる特性を持つものに変更可能であることによる．

また，提案手法を Java エイリアス解析ツール JAAT として実装し，その有効性を評価した．JAAT は，大規模プログラムに対しても実用的な時間で解析できる特徴を持つ．例えば，119,564 行の JDK（Java Development Kit）附属クラスライブラリを利用する 32,037 行の Java プログラムに対し，Phase 1 で 168sec，Phase 2 で 1ms 以下の解析時間であった．さらに，インスタンス単位の解析により，エイリアス集合の平均要素数は 4.42 – 15.37 個に抑えられおり，これは，クラス単位の解析によるものの 32 – 63%程度の要素数である．さらに，プログラム保守，プログラム理解を支援するための，視覚的なエイリアス表示機能を持つ GUI も保持している．

以降，3.2 では既存のオブジェクト指向プログラムに対するエイリアス解析について考察する．3.3 で手法を提案し，3.4 でその実現について述べる．3.5 で評価及び考察を行い，最後に 3.6 でまとめと今後の課題について述べる．

3.2 オブジェクト指向プログラム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手法は，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 [13, 21, 34, 45, 49] をオブジェクト指向言語に拡張したものとなっているが，オブジェクト指向言語独自のものも含めいくつかの未解決の問題が残っており，それらについて考察する．

エイリアス解析に要する解析コスト: 実用的な解析ツールの実現において，そのツールが必要とする解析コストの大きさは最も考慮すべき要素の一つである．手続きを含むプログラムを解析する場合，到達エイリアス集合を用いた FS 解析では，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しなければならない．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集合の形で保持されていることに起因している．つまり，文 s における到達エイリアス集合 $RA(s)$ にある時点で変化が起こったとき，$RA(s)$ の影響を受ける可能性のあ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，各文の到達エイリアス集合を再計算しなければならないためである．

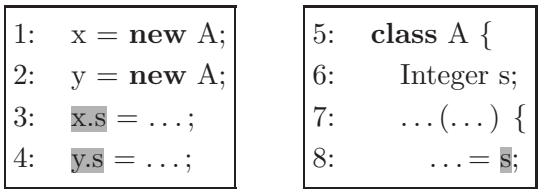


図 3.1: (例) インスタンスを区別しない解析によるエイリアス

解析結果の再利用: 到達エイリアス集合 $RA(s)$ は文 s が存在するプログラム全体を解析することにより導出されたものであるため，別の文 t が変更されたとき，$RA(s)$ を再計算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．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，上位クラスであるほどその属性やメソッドは汎化されたものとなり，一度定義されると変更されることは少ない．そのため，解析結果のクラスやメソッド単位でのモジュール化は，再計算コストの削減に有効であると考えている．

インスタンスを区別した解析による解析精度の向上: 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 3.1 の例では [(3, `x.s`), (8, `s`)]，[(4, {`y.s`), (8, `s`)] がそれぞれエイリアス集合といえるが，同一クラスのインスタンスがその内部のエイリアス情報を共有する場合，[(3, `x.s`), (4, `y.s`), (8, `s`)] がエイリアス集合とみなされるため，解析精度の低下につながる．

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．

ここではインスタンス属性に関してのみ述べた．インスタンスメソッドにも同様の問題があるが，それに関しては次節で述べる．

3.3 エイリアスフローグラフを利用したエイリアス解析手法

3.3.1 方針

□解析の効率化に関する方針

方針 1: メソッド，クラスといったモジュール単位で，モジュール内エイリアス関係を抽出しておく．

方針 2: モジュール間エイリアス関係はオンデマンド計算により抽出する．

この 2 つの方針を採用することでエイリアス解析のモジュール性及び独立性が満たされ，解析の効率化が可能となる．

とりわけ，この方針はオブジェクト指向プログラミングにおいて有効である．オブジェクト指向プログラムを解析する場合，対象プログラムはもちろんであるがそれが利用してるクラスライブラリも解析しなければならない．通常，クラスライブラリは膨大で，その解析コストは非常に大きい．モジュール性を満たすアルゴリズムが利用できれば，これらのクラスライブラリを繰り返し解析する必要はなくなる．

□解析精度の向上に関する方針

プログラム中に存在するエイリアス関係を，**内部エイリアス関係**（*Inner Alias Relation*），**外部エイリアス関係**（*Outer Alias Relation*）に分類する．それぞれの定義は以下のとおりである．

内部エイリアス関係: 同一クラスのインスタンス間で共通なエイリアス関係

外部エイリアス関係: 同一クラスのインスタンス間で独立なエイリアス関係
（外部エイリアス関係は外部オブジェクトの影響により生成される）

方針 3: 同一クラスの異なるインスタンスの外部エイリアス関係を区別するため，内部エイリアス関係のみ前もって抽出し，外部エイリアス関係はオンデマンド解析により抽出する．

方針 3 により，インスタンスを区別した解析を効率的に行うことができる．

□オブジェクトコンテキスト

本手法では，インスタンス属性と同様，インスタンスメソッドの呼び出し情報もインスタンスごとに区別する．これを**オブジェクトコンテキスト**（*Object Context*）と定義し，各インスタンスの外部エイリアス関係を抽出する際に利用される．

エイリアス集合 $\mathcal{A}$ に関するオブジェクトコンテキスト $OC(\mathcal{A})$ とは，$\mathcal{A}$ に属するオブジェクトに関して，直接若しくは間接的に呼び出される可能性のあるメソッドの集合である．直接呼び出されるメソッドとは，式 $a.m(\ldots)$ における参照変数 a が $\mathcal{A}$ に含まれていたとき，$a.m(\ldots)$ により呼び出されるメソッドを指す[1]．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そのメソッド本体から呼び出される可能性のある同一インスタンスのメソッドを指す．

また，オブジェクトコンテキストは，**FS オブジェクトコンテキスト**（*Flow Sensitive Object Context*），**FI オブジェクトコンテキスト**（*Flow Insensitive Object Context*）に分けられる．前者はメソッド呼び出し順を考慮するが，後者は考慮しない．そのため，FS オブジェクトコンテキストを用いることで，FI オブジェクトコンテキストよりも正確なエイリアス計算を行うことができる．

例として，図 3.2 において `Calc.Calc()`，`Calc.add()`，`Calc.result()` が順に呼ばれたときのエイリアス基準 <13, `return(i)`> に対するエイリアス計算を挙げる．図 3.2(b)

[1] メソッドのオーバーライドが行われた場合，$\mathcal{A}$ の型が唯一に特定できなければ同一シグニチャに対してメソッドが複数存在する可能性がある

はメソッド呼び出し順を考慮しているため，図 3.2(a) と比較してそのエイリアス集合は小さく，`Calc.Calc()` メソッド内の 2 式 `i`，**new** `Integer` が除外されているのが分かる．

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer(i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer(i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

(a) FI オブジェクトコンテキスト

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer(i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer(i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

(b) FS オブジェクトコンテキスト

図 3.2: (例) オブジェクトコンテキスト

メソッド呼び出し順を考慮するには解析コストの増加が避けられず，解析精度と解析コストはトレードオフの関係である．なお，本論文では FI オブジェクトコンテキストを採用している．

□その他のオブジェクト指向言語特有の概念に対する方針

JAVA などのオブジェクト指向言語が持つ，クラス，オブジェクト以外の特性への対応について簡単に述べる．

継承: 継承は，**メソッドオーバーライド**（*Method Overriding*），動的束縛など，他のオブジェクト指向言語の特性をもたらしている．また，仮想関数呼び出しを考慮するため，継承関係に配慮しながらエイリアス解析を行わねばならない．この詳細は後述する．

メソッドオーバーライド，動的束縛: メソッドオーバーライドは，あるクラス階層内に同一**シグニチャ**（*Signature*）を持つ複数のメソッドの存在を認める．呼び出しメソッドの選択は，メッセージを受け取ったオブジェクトの参照型に依存するため，実際に呼び出されるメソッドを静的に決定することは難しい．これは，オブジェクトの型はプログラム実行時に決まることによる．しかし，エイリアス解析を利用することで，そのオブジェクトの型をより正確に推測することができる．

例えば，$a.b(\cdots)$ により呼び出されるメソッドを導出しなければならないとき，a のエイリアス情報を利用することで，a により参照されているインスタンスの型を類推できる．a の型が類推できれば，そのクラス内に存在する $b(\cdots)$ と同一シグニチャのメソッドを探すだけでよい．

スレッド，例外: 本章では，繰り返し文，分岐文，メソッド呼び出しなどの一般的な制御フローのみ取り扱っており，**スレッド**（*Thread*）や**例外**（*Exception*）は扱っていない．これらに関しては，後述する．

3.3.2 アルゴリズム

前節で述べた方針に基づき，以下の 2 フェーズで構成されるエイリアス解析手法を提案する．

Phase 1(a): メソッド内解析による，AFG の構築

Phase 1(b): クラス内及びメソッド間解析による，MFG の構築

Phase 2: クラス間及びメソッド間解析による，AFG 及び MFG によるエイリアス計算

本手法は JAVA を前提としたものとなっているが，C，C++プログラムに存在するポインタ変数の間接参照によるエイリアスにも拡張可能であり，3.5.5 で述べる．

□ Phase 1(a): AFG の構築

エイリアスフローグラフ（*Alias Flow Graph*，*AFG*）とは，単一メソッド内の FS エイリアス関係を無向グラフで表現したものである．**AFG 節点**（*AFG Node*）は，文番号

表 3.1: (定義) AFG 特殊節点

特殊節点	概要	位置
Actual-Alias-in (AA-in)	実引数により呼び出し先に渡されるエイリアス	呼び出し元
Formal-Alias-in (FA-in)	仮引数により呼び出し先に渡されるエイリアス	呼び出し先
Actual-Alias-out (AA-out)	実引数により呼び出し元に渡されるエイリアス	呼び出し元
Formal-Alias-out (FA-out)	仮引数により呼び出し元に渡されるエイリアス	呼び出し先
Method-Alias-out (MA-out)	`return` 文により呼び出し元に渡されるエイリアス	呼び出し先
Method-Invocation (MI)	`return` 文により呼び出し元に渡されるエイリアス（AA-in，AA-out 節点の親節点）	呼び出し元
Instance-Alias-in (IA-in)	属性によりメソッドに渡されるエイリアス	-
Instance-Alias-out (IA-out)	属性によりメソッドから渡されるエイリアス	-

とオブジェクトへの参照式の組である．ここでいう**オブジェクトへの参照式**（*Referring Expression to Objects*）とは，参照型の式のことを指す（参照変数，インスタンス生成式，参照型を返すメソッド呼び出し文がこれに含まれる）．このような節点を特に **AFG 標準節点**（*AFG Normal Node*）と呼ぶことにする．加えて，オブジェクト指向プログラムにはメソッド引数やインスタンス（クラス）属性が存在するため，これらを介したエイリアスの受け渡しも考慮しなければならない．そこで，表 3.1 に挙げる **AFG 特殊節点**（*AFG Special Node*）も用意しておく．

AFG 辺（*AFG Edge*）は，2 つの AFG 節点間で，同一変数の参照や代入文，パラメータの対応などによる**直接のエイリアス関係**（*Direct Alias Relation*）が存在するとき引かれる．また，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**（*Indirect Alias Relation*）を表す．

ここで，AFG をオブジェクト指向プログラムに適用する際，$a.b$ や $c.d()$ のような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に，インスタンス及び属性（メソッド）を表す AFG 節点間に親子関係を定義する．例えば $a.b$ に関して，a，b に対応する AFG 節点をそれぞれ N_a，N_b とすると，N_a は N_b の**親節点**（*Parent Node*），N_b は N_a の**子節点**（*Child Node*）となる．この関係はエイリアス計算時（Phase 2）に利用される．このような節点間の親子関係は MI 節点と AA-in（out）節点間にも存在し，これも同様に利用される．

□ Phase 1(b): MFG 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（または，同一インスタンス内における）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 の各節点を **MFG 節点**（*MFG Node*）と呼び，メソッドを表す．メソッド A がメソッド B を呼ぶ場合，A，B に対応する MFG 節点をそれぞれ M_A，M_B とすると，有向辺である **MFG 辺**（*MFG Edge*）が節点 M_A から節点 M_B に引かれる．

メソッド呼び出し関係の抽出においては，継承及びそれによるメソッドオーバーライドを考慮する必要がある．そのため，MFG 構築の際には，クラスの継承関係の解析も並行して行われている．

□ Phase 2: AFG 及び MFG によるエイリアス計算

オブジェクトへの参照式 X のエイリアス $\mathcal{A}(X)$ の計算は AFG の到達問題に置き換えられる．$\mathcal{A}(X)$ は X と同じメモリ領域を指す可能性のある式の集合であり，X 自身も $\mathcal{A}(X)$ に含まれる．

AFG によるエイリアス計算は以下の方針に基づく．

方針 1: 親節点 P を持つ節点 C のエイリアスを導出する場合，まず P のエイリアス $\mathcal{A}(P)$ を求める．次に，$\mathcal{A}(P)$ に関する情報，具体的には，

- $\mathcal{A}(P)$ に含まれるインスタンスの型
- $\mathrm{OC}(\mathcal{A}(P))$

を計算する．

これは，$\mathcal{A}(P)$ 及び $\mathcal{A}(P)$ に含まれるインスタンスの型を推定することなく，C のエイリアスを計算することは困難であるためである．仮に，$\mathcal{A}(P)$ の計算を省略した場合，P の型を親に持つすべてのクラスのインスタンスを P が指すとみなさなければならないため，$\mathcal{A}(C)$ は非常に大きくなってしまう．

方針 2: AFG の探索時，MI，MA-out，FA-in，FA-out，AA-in，AA-out 節点に到達した際には，MFG を用いて呼び出し先（呼び出し元）メソッドを探索し，対応する MA-out，MI，AA-in，AA-out，FA-in，FA-out 節点から探索を続ける．

探索アルゴリズムの詳細に関しては，関連する式定義を図 3.3 に，アルゴリズムを図 3.4 に示す．

$\varphi(\mathbf{x})$**:** 式 x から AFG 節点 X への射影

$\varphi^{-1}(\mathbf{X})$**:** AFG 節点 X から式 x への射影

$\mathcal{A}(\mathbf{X})$**:** AFG 節点 X のエイリアス

$\mathbf{N_P(X)}$**:** AFG 節点 X の親節点

$\psi(\mathbf{m})$**:** メソッド m からメソッド AFG M への射影

$\psi^{-1}(\mathbf{M})$**:** メソッド AFG M からメソッド m への射影

$\mathbf{e} =_{\mathbf{id}} \mathbf{e'}$**:** 式 e と式 e' が同じ識別子であるとき**真**，さもなくば**偽**

図 3.3: (要素定義) エイリアス計算

Step 1: エイリアス基準 e の指定

$E := \varphi(e)$

Step 2: E 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

加えて，到達節点 C の種類に応じて以下のような処理も行う

[Cond.1] 親節点を持つ標準（インスタンス属性）節点：

1. $c := \varphi^{-1}(C)$，$P := N_P(C)$
2. $\mathcal{A}(P)$ の計算
3. $\mathcal{A}(P)$ に含まれるインスタンス生成式から $\mathcal{A}(P)$ の型を類推
4. $\mathrm{OC}(\mathcal{A}(P))$ の計算
5. $\{G_M \mid \psi^{-1}(G_M) \in \mathrm{OC}(\mathcal{A}(P))\}$ に含まれる IA-in$[c]$ 及び IA-out$[c]$ 節点から AFG 辺をたどる

   $\{N \mid N_P(N) \in \mathcal{A}(P)$ **and** $\varphi^{-1}(N) =_{id} c\}$ から AFG 辺をたどる

[Cond.2] 親節点を持つ MI 節点：

1. $c := \varphi^{-1}(C)$，$P := N_P(C)$
2. $\mathcal{A}(P)$ の計算
3. $\mathcal{A}(P)$ に含まれるインスタンス生成式から $\mathcal{A}(P)$ の型を類推
4. $\mathrm{OC}(\mathcal{A}(P))$ の計算
5. $\{M \mid \psi^{-1}(M) \in \mathrm{OC}(\mathcal{A}(P))\}$ に含まれる MA-out 節点から AFG 辺をたどる

[Cond.3] IA-in または IA-out 節点：

1. $c := \varphi^{-1}(C)$
2. $\{M \mid \psi^{-1}(M) \in \mathrm{OC}(\mathbf{this})\}$ に含まれる IA-out$[c]$ または IA-in$[c]$ 節点から AFG 辺をたどる

[Cond.4] AA-in または AA-out 節点：

1. MI 節点 $N_P(C)$ が呼ぶメソッド m を計算
2. $\psi(m)$ に含まれる FA-in または FA-out 節点から AFG 辺をたどる

[Cond.5] FA-in または FA-out 節点：

1. $M_B := \{M \mid C \in M\}$, $m_B := \psi^{-1}(M_B)$
2. $m_A := \{m \mid m \in \mathrm{OC}(\mathbf{this})$ **and** m が m_B を呼ぶ $\}$（MFG より）
3. $M_A := \psi(m_A)$
4. M_A に含まれる AA-in または AA-out 節点から AFG 辺をたどる

[Cond.6] MA-out 節点：

1. $M_B := \{M \mid C \in M\}$，$m_B := \psi^{-1}(M_B)$
2. $m_A := \{m \mid m$ が m_B を呼ぶ $\}$（MFG より）
3. $M_A := \psi(m_A)$
4. M_A に含まれる（m_B を呼ぶ）MI 節点から AFG 辺をたどる

[Cond.7] MI 節点：

1. C に対応するメソッド呼び出し文から呼ばれるメソッドを m_A とする
2. $M_A := \psi(m_A)$
3. M_A に含まれる 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式が，求める e のエイリアス

図 3.4: (アルゴリズム) エイリアス計算

```
1:  public class Calc {
2:      Integer i;
3:      public Calc() {
4:          i = new Integer(0);
5:      }
6:      public void inc() {
7:          i = new Integer(i.intValue() + 1);
8:      }
9:      public void add(int c) {
10:         i = new Integer(i.intValue() + c);
11:     }
12:     public Integer result() {
13:         return(i);
14:     }
15: }
```

```
16: class Test {
17:     Calc a, b;
18:     Integer c;
19:     Test() {
20:         a = new Calc();
21:         b = new Calc();
22:         a.inc();
23:         b.add(1);
24:         c = b.result();
25:     }
26: }
```

図 3.5: (例) プログラム

AFG 探索は，MFG の情報を利用しながら，プログラム全体若しくは複数クラスをまたいで行われる．プログラムに再帰経路が存在する場合，AFG 探索はエイリアス集合の増加が収束するまで続けられる．

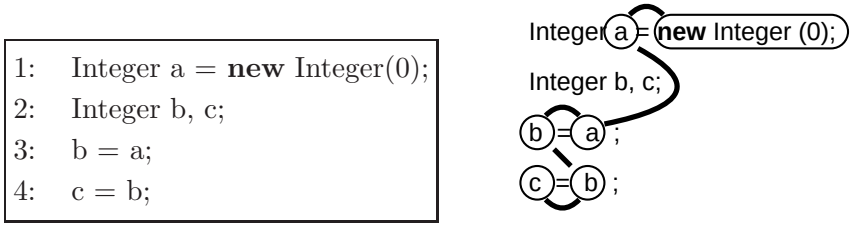
AFG 探索アルゴリズムの停止性に関して，探索停止を検出するための方針を以下に挙げる．

方針 1: オブジェクトへの参照式 c に関するエイリアス計算において，AFG 探索が AFG 節点 $\varphi(e)$ に到達したとき，$\varphi(e)$ を c に関する**到達節点リスト**（*Reached Nodes List, RNlist*）である $RNlist(c)$ に追加する．AFG 探索中に RNlist が持つ AFG 節点に到達した場合は，それ以降の探索を中止する．

方針 2: 到達節点リストは，同一メモリ領域を指す可能性のあるエイリアス集合ごとに生成される．例えば，$a.b$ がエイリアス基準であれば，$RNlist(a)$ 及び $RNlist(b)$ が生成される．

方針 3: $a.b.c$ がエイリアス基準であり，a に関するエイリアス計算のための AFG 探索中に AFG 節点 N に到達したとき，N が $RNlist(a)$ に含まれているかを調べるだけでなく，$N \in RNlist(b)$ 及び $N \in RNlist(c)$ を満たさないことも調べる．

これらのことから，各到達節点リストの大きさは E 以下であり，RNlist の個数は k で抑えられるため，無限の到達節点リストの生成は回避できる．すなわち，AFG 探索は必ず停止する．



(a) プログラム　　(b) (a) の AFG

図 3.6: (例) AFG

3.3.3 適用例

□ Phase 1(a): AFG の構築

図 3.6 にサンプルプログラム及びその AFG を示す．AFG 節点は式を円で囲んだ形で，AFG 辺は実線で示されている．節点内の式以外の文字列（`Integer`，`=`，`Integer b, c;` など）は，元のソースコードとの対応を分かりやすくするために残している．図 3.6(b) において，AFG 節点 (1, `a`) と (1, **new** `Integer(0)`) が AFG 辺で結ばれており，これは直接のエイリアス関係を表す．このエイリアス関係は，ソースコード上において **new** `Integer(0)` から `a` への代入が行われているためである．

図 3.5 の 2 つのクラス定義を含むサンプルプログラムに対する AFG を図 3.7 に示す．文 7 及び文 10 の右辺に現れる変数 `i` は参照型のインスタンス属性であり，AFG 特殊節点である IA-in[`i`] 及び IA-out[`i`] が，`Calc` クラス内の各メソッドに用意される．また，文 13 の `return(i)` が返す値はオブジェクトへの参照であるため，MA-out 節点が `Calc.result()` メソッドに用意される．また，文 24 の `result()` に関して，`result()` は `b` の子節点，`b` は `result()` の親節点となる．

□ Phase 1(b): MFG の構築

図 3.8 にサンプルプログラム及びその MFG を示す．`B` クラスではメソッド `p` は定義されておらず，シグニチャ`p()` に対するメソッドは `A.p()` となる．`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()` はシグニチャ`q()` のメソッドを呼び出しているが，それは `B.q()` でありメソッド `A.q()` ではないことに留意する必要がある．

図 3.5 の `Calc` クラス及び `Test` クラスの MFG を図 3.9 に示す．いずれもクラスにもクラス内メソッド呼び出しがないため，MFG 辺は存在しない．

□ Phase 2: AFG 及び MFG によるエイリアス計算

図 3.5 のエイリアス基準 <24, `c`>（太枠部）のエイリアス計算を考える．

1. AFG 節点 φ(`c`) から AFG 辺をたどり，MI 節点 φ(`result()`) に到達（図 3.11）
2. φ(`result()`) は親節点 φ(`b`) を持つため，まず `b` のエイリアス計算を行い，φ(`result()`) のエイリアス計算に関与するインスタンスを特定する

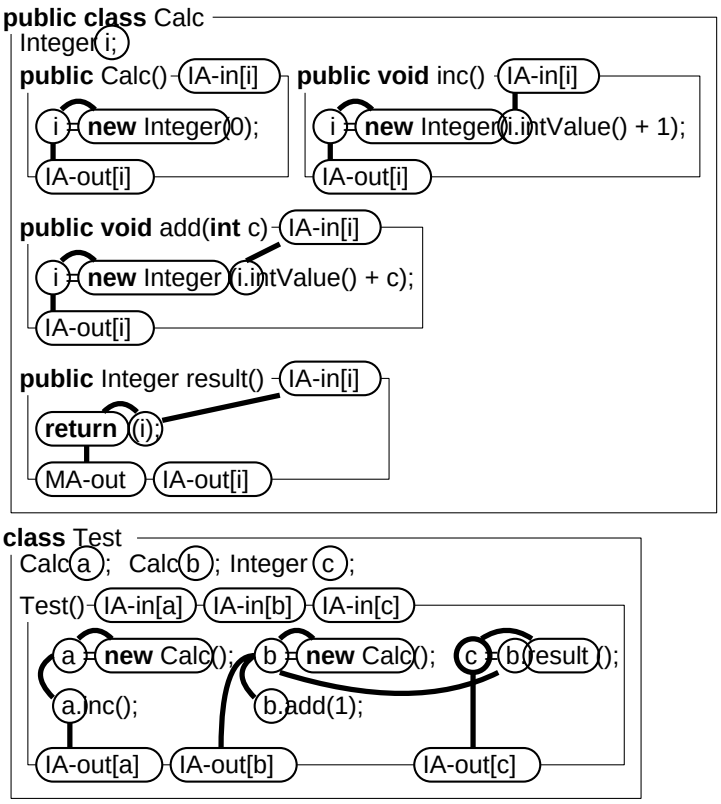


図 3.7: (例) 図 3.5 の AFG

(a) $\mathcal{A}(\varphi(\mathtt{b}))$ を計算（図 3.11）

(b) $\mathcal{A}(\varphi(\mathtt{b}))$ に存在するインスタンス生成式からの $\mathcal{A}(B)$ 型を特定（結果は，`Calc` クラスとなる）

(c) $\mathrm{OC}(\mathcal{A}(\varphi(\mathtt{b})))$ を計算
（結果は，{`Calc.Calc()`, `Calc.add(`**int** `c)`, `Calc.result()`} となる）

3. `b` のエイリアス結果より，`b` は `Calc` クラスのインスタンスへの参照であることから，ψ(`Calc.result()`) の MA-out 節点から AFG 辺をたどる（図 3.12）

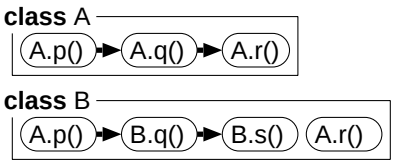
(a) AFG 辺をたどり，ψ(`Calc.result()`) 内の IA-in[`i`] に到達

(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 $\mathrm{OC}(\mathbf{this}) = \mathrm{OC}(\mathcal{A}(\varphi(\mathtt{b})))$ の IA-out[`i`] から AFG 辺をたどる
（`Calc.Calc()`, `Calc.add()` も同様）

図 3.10 に，図 3.5 のエイリアス基準 <24, `c`> のエイリアス（網掛部）及び $\mathrm{OC}(\mathcal{A}(\varphi(\mathtt{b})))$（下線部）を示す．$\mathrm{OC}(\mathcal{A}(\varphi(\mathtt{b})))$ に含まれない `Calc.inc()` はエイリアス計算対象から除外されているのが分かる．

```
1:  class A {
2:      void p() { q(); }
3:      void q() { r(); }
4:      void r() {}
5:  }
6:  class B extends A{
7:      void q() { s(); }
8:      void s() {}
9:  }
```

(a) プログラム



(b) (a) の MFG

図 3.8: (例) MFG

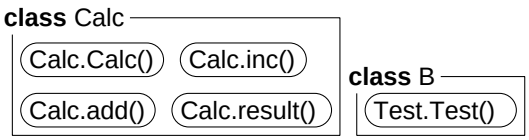


図 3.9: (例) 図 3.5 の MFG

3.4 Java エイリアス解析ツール Java Alias Analysis Tool (JAAT)

提案手法を JAVA を対象言語とするエイリアス解析ツールとして実装し，本手法の有効性を評価した．解析アルゴリズムとして，FI オブジェクトコンテキストを採用している．

ツールは**解析部**（*Analysis Subsystem*）と**ユーザーインタフェース部**（*UI Subsystem*）で構成されており，その構成を図 3.13 に示す．以降，解析部及びユーザーインターフェース部をそれぞれ説明する．

3.4.1 解析部

解析部は C++で記述されており，`libANTLR`[2]，`libjavamm`，`libAFG` の 3 つのライブラリで構成されている．

□ **Syntax Analyzer**（`libANTLR`）

JAVA プログラムのソースコードを読み込み，**字句解析**（*Lexical Analysis*）及び**構文解析**（*Syntax Analysis*）を行い，抽象構文木を構築する．

□ **Semantic Analyzer**（`libjavamm`）

[2]ANTLR[3] 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++若しくは JAVA プログラムとして生成するツールである．

```
1:  public class Calc {
2:    Integer i;
3:    public Calc() {
4:      i = new Integer(0);
5:    }
6:    public void inc() {
7:      i = new Integer(i.intValue() + 1);
8:    }
9:    public void add(int c) {
10:     i = new Integer(i.intValue() + c);
11:   }
12:   public Integer result() {
13:     return(i);
14:   }
15: }
```

```
16: class Test {
17:   Calc a, b;
18:   Integer c;
19:   Test() {
20:     a = new Calc();
21:     b = new Calc();
22:     a.inc();
23:     b.add(1);
24:     c = b.result();
25:   }
26: }
```

図 3.10: (例) 図 3.5 のエイリアス基準 <24, c> のエイリアス

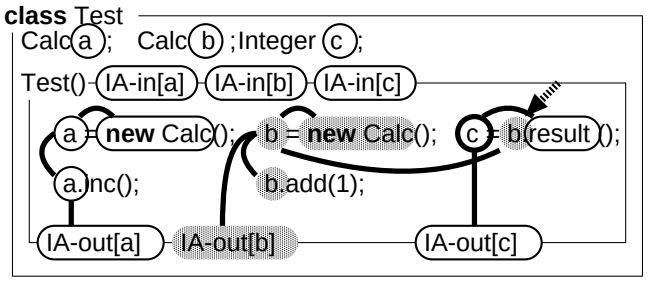


図 3.11: (例) 図 3.5 のエイリアス基準 <24, b> のエイリアス

抽象構文木を読み込み，**意味解析**（*Semantic Analysis*）（識別子表を作成し，識別子に関する宣言と参照間の関係を抽出する）を行い，意味解析木を構築する．

□ **Alias Analyzer**（`libAFG`）

意味解析木を読み込み，AFG 及び MFG の構築（提案手法の Phase 1 に相当）を行う．また，ユーザの要求に応じたエイリアス計算（提案手法の Phase 2 に相当）もこれにより行われる．

3.4.2 ユーザーインターフェース部

ユーザーインターフェース部は C++で記述されており，Gtk−−[20] 及び GTK+[19] ツールキットを使用している．その機能には，**テキストウインドウ**（*Text Window*）（図 3.14(a)）及び**エイリアスツリーウインドウ**（*Alias Tree Window*）（図 3.14(b)）によるエ

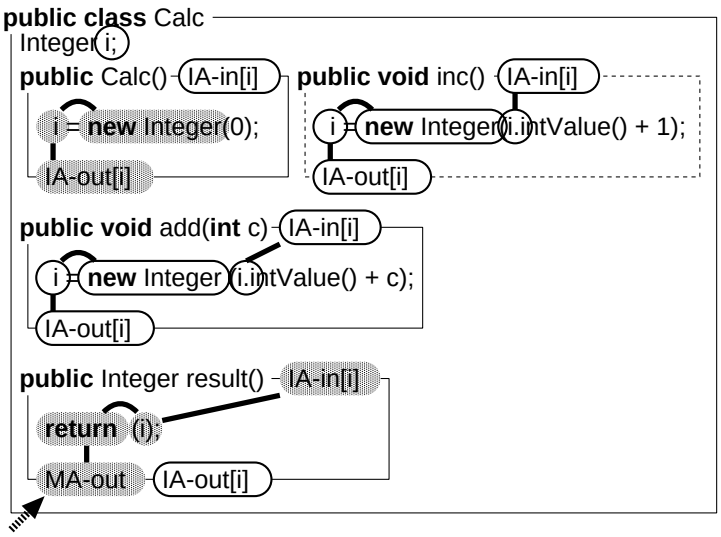


図 3.12: (例) 図 3.5 のエイリアス基準 <24, `result()`> のエイリアス

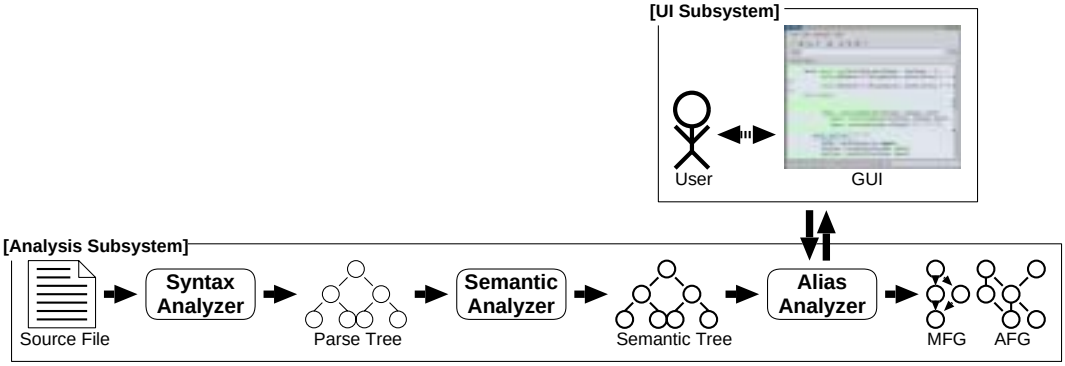


図 3.13: (実装) JAVA エイリアス解析ツール（構成）

イリアス表示，プログラム編集がある．ユーザがあるオブジェクトへの参照式に関するエイリアスの抽出を指示すると，ツールはテキストウィンドウ及びエイリアスツリーウィンドウ上にその解析結果を表示する．

ユーザインターフェース部の作成にあたり，[57] を参考にした．[57] には，情報の視覚化において必要な技法として以下のものが挙げられている．

(1) 必要なもののみ表示する技法

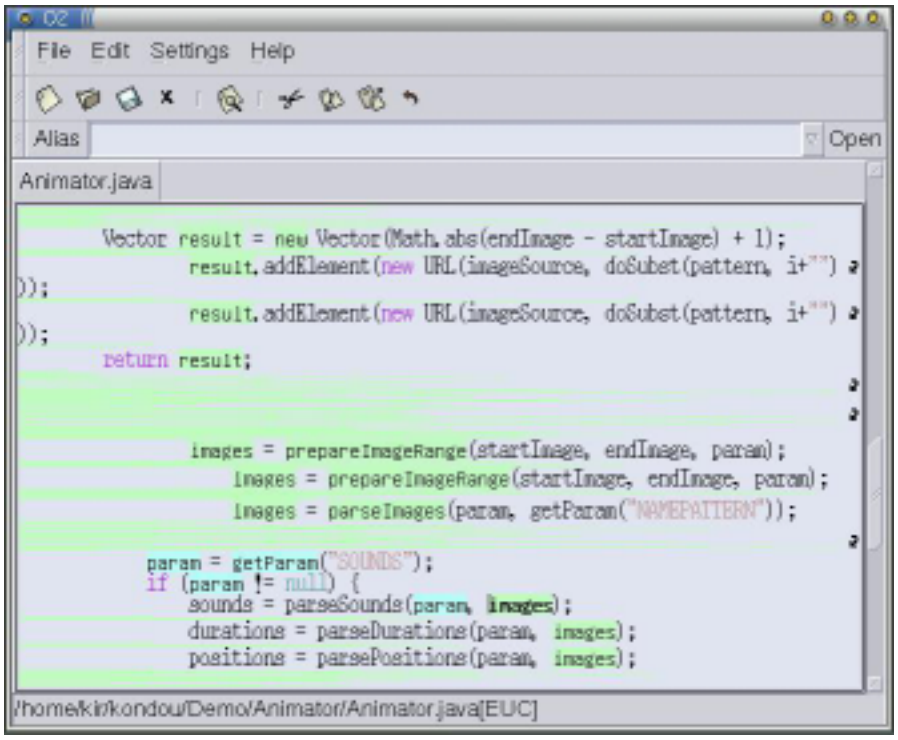
(2) 全体と詳細を同時に見る方法

(3) 抽象的データの画面へのマッピング法

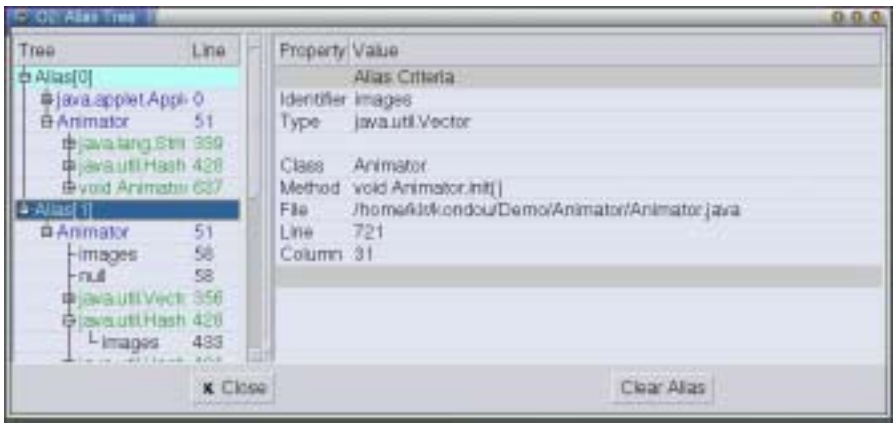
(4) 自動レイアウト手法

(5) 操作しやすいインタフェース

今回，ユーザインターフェース部の構築にあたって，テキストウィンドウ，エイリアスツリーウィンドウは，(1)，(2) それぞれを「エイリアス集合の効果的な表示」,「エイリアス集合全体像の把握と理解の簡易化」と解釈しそれらを実現している．



(a) テキストウィンドウ

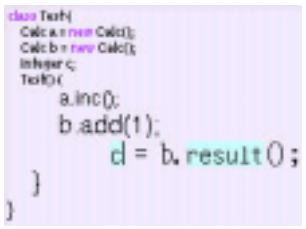


(b) エイリアスツリーウィンドウ

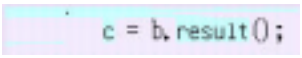
図 3.14: (実装) JAVA エイリアス解析ツール（ユーザインターフェース）

□テキストウィンドウ

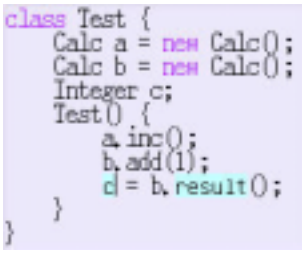
エイリアス集合の導出により，プログラムはエイリアス集合に含まれる**エイリアス部分**

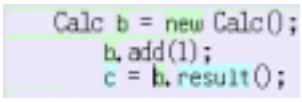


(a) 縮小（可変）



(b) 縮小（線分化）





(c) 保存（左が通常，右は (b) の適用後）

図 3.15: (例) 非エイリアス部分の表示方法

（*Alias part*）と，エイリアス集合に含まれない**非エイリアス部分**（*Non-alias part*）に区分される．テキストウィンドウでは，ユーザの視点をエイリアス部分に着目させなければならないが，エイリアス部分と非エイリアス部分の差別化の手段はユーザの目的によって様々である．とりわけ，エイリアスの性質上プログラム中に占める割合は非エイリアス部分が圧倒的に多く，その視覚化に重点を置く必要がある．ここで，エイリアス部分，非エイリアス部分の視覚化についてそれぞれ述べる．

エイリアス部分: 字体及び背景色の変更により差別化を試みる．異なるエイリアス集合では別の背景色を使用する．

非エイリアス部分: 目的に応じて選択可能な 3 つの表示方法を実現した．

縮小（可変）: エイリアス部分との距離に比例して字体の大きさを縮小する．これにより，大まかな制御構造を把握しながらエイリアス部分に着目することができる（図 3.15(a)）．

縮小（線分化）: 字体の大きさを線状にまで縮小する．ソースコードの表示領域が削減されるため，エイリアス部分にのみ着目したいとき有効である．また，対象となる文のインデント位置及びその長さは保存されるため，非エイリアス部分の大まかな構造も把握できる（図 3.15(b)）．

保存: 字体の大きさを維持する．エイリアス表示後もプログラム全体のソースコードが必要な場合に用いる（図 3.15(c)）．

図 3.14(a) では，エイリアス部分を背景色の変更により，非エイリアス部分を線分化により差別化を行っている．

表 3.2: (定義) エイリアス解析手法に関わる要素

記号	概要
C	クラスの総数
A	1 クラスでの属性数の最大値（継承を含む）
M	1 クラスでのメソッド数の最大値（継承を含む）
L	1 メソッドでの局所変数，仮引数の数の最大値
E	式の総数
k	親子関係の最大連鎖長 （連鎖が再帰型をなす場合，既に訪れている式は数えない）

□エイリアスツリーウィンドウ

エイリアス部分は複数メソッドにわたって点在することが多く，複数ファイルにおよぶことも少なくない．エイリアスツリーウィンドウは，テキストウィンドウでは困難なエイリアス部分全体の把握を支援する．エイリアスツリーの各節点は，クラス名，メソッド名，オブジェクトへの参照式を表わす．

図 3.14(b) では，2 つのエイリアス集合（`ALIAS[0]`，`ALIAS[1]`）のエイリアスツリーが示されており，ツリーの各節点を選択することで対応する要素の名前，型，位置などの情報を得ることができる．

3.5 評価

3.5.1 アルゴリズムの複雑さ

提案したエイリアス解析手法の複雑さについて述べる．表 2.1 にその関連する要素を挙げる．

□ AFG の構築（Phase 1(a)）に要するコスト

プログラムに繰り返し文が存在する場合，最悪 E^2 回の式の解析を行わなければならない．一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，一つの文の解析時間は，A と L の和に比例する．よって，最悪時の計算コストは $O((A+L)\cdot E^2)$ となる．また，節点数は $O(E)$，辺数は $O(E^2)$ であることから，空間コストは $O(E^2)$ で抑えられる．

□ MFG の構築（Phase 1(b)）に要するコスト

MFG 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 $O(E)$ 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 M で抑えられることから，節点数は $O(C\cdot M)$，辺数は $O(C\cdot M^2)$ となる．

□ AFG 及び MFG によるエイリアス計算（Phase 2）に要するコスト

表 3.3: (統計データ) 評価用プログラム

プログラム	サンプルプログラム		関連するクラスライブラリ	
[概要]	ファイル	行	ファイル	行
TextEditor	1	915	802	114,887
[テキストエディタ]	(0.1%)	(0.8%)	(99.9%)	(99.2%)
WeirdX	47	16,703	815	115,977
[X サーバ]	(5.5%)	(12.6%)	(94.5%)	(87.4%)
ANTLR	129	18,775	267	33,847
[構文解析生成]	(32.6%)	(35.7%)	(67.4%)	(64.3%)
DynamicJava	242	32,037	825	119,564
[Java インタプリタ]	(22.7%)	(21.1%)	(77.3%)	(78.9%)

PentiumIII-667MHz-512MB(Linux)

表 3.4: (実験結果) AFG の構築時間（Phase 1(a)）[ms]

プログラム	サンプルプログラム	関連するクラスライブラリ
TextWditor	-	99,980
WeirdX	14,220	100,540
ANTLR	12,830	23,480
DynamicJava	56,260	110,150

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．最悪の場合，時間コスト，空間コストとも $O(E^k)$ になるが，このような状況はまれで，後述する実験においては k の値は平均 3 程度であった．

3.5.2 実験

JAAT を用いて本手法の有効性を評価した．

評価用プログラムの統計データを表 3.3 に示す．サンプルプログラムを解析する際には，それ自身はもちろんのこと，モジュール間解析のためにそれが利用する JDK 附属クラスライブラリも解析しなければならない．例えば，`TextEditor` は 915 行の単一プログラムであるが，直接若しくは間接的にクラスライブラリを利用しており，それは 802 ファイル，114,887 行 (全体ソースコードの 99.2%を占める) にもおよぶ．このことから，解析コストの大半は，対象プログラムではなく，関連するクラスライブラリで消費されているといえる．

表 3.4，表 3.5 に，サンプルプログラム及び関連するクラスライブラリの AFG 構築時間，MFG 構築時間をそれぞれ示す．また，表 3.6，表 3.7 に，各サンプルプログラム中の最大クラスに存在するすべての AFG 標準節点をエイリアス基準としたときの，平均 AFG 探索時間，エイリアス集合の平均要素数をそれぞれ示す．

表 3.5: (実験結果) MFG の構築時間（Phase 1(b)）[ms]

プログラム	サンプルプログラム	関連するクラスライブラリ
TextEditor	10	390
WeirdX	20	390
ANTLR	100	80
DynamicJava	960	450

表 3.6: (実験結果) AFG 及び MFG によるエイリアス計算時間（Phase 2）[ms]

プログラム	
TextEditor	0.65
WeirdX	0.29
ANTLR	0.17
DynamicJava	0.07

3.5.3 考察

□時間コスト（AFG 構築時間，Phase 1(a)）

解析のモジュール化は，対象プログラムのソースコードを変更した際，それ自身のみを再解析すればよいことに利点がある．到達エイリアス集合のみによるエイリアス解析 [13, 45] では，モジュール単位でエイリアス解析結果を保持することができないため，モジュール M が変更されると M が利用する他のモジュール M' も再解析の対象にせざるを得ない．とりわけ，Java プログラムの開発においては，サンプルプログラムのソースコードが頻繁に変更されることはあっても，関連するクラスライブラリが変更されることはまれである．

クラスライブラリの解析結果は AFG として保存されており，サンプルプログラムのみを解析すればよいため，解析時間は大幅に削減される．例えば，`TextEditor` 自身の解析時間は 100ms であり，関連するクラスライブラリの解析時間は 99,980ms である．`TextEditor` を変更した際には，クラスライブラリを再解析する必要はなく，`TextEditor` のみを再解析すればよい（表 3.4）．

□時間コスト（MFG 構築時間，Phase 1(b)）

MFG 構築時間は，対象プログラムの大きさではなく，クラス内のメソッド呼び出し回

表 3.7: (実験結果) エイリアス集合の平均要素数 [個]

プログラム	インスタンス単位	クラス単位
TextEditor	4.42	8.31
WeirdX	15.37	24.54
ANTLR	5.94	18.77
DynamicJava	9.16	17.19

数に依存する．そのため，サンプルプログラムの MFG 構築時間が，関連するクラスライブラリの MFG 構築時間よりも常に小さいとは限らない．DynamicJava の場合，自身の MFG 構築時間は 960ms であるが，関連するクラスライブラリの MFG 構築時間は 450ms であった．

ただし，MFG 構築時間は AFG 構築時間と比べ十分に小さく，解析時間全体に占める比率はごく僅かである．

□時間コスト（AFG 及び MFG によるエイリアス計算時間，Phase 2）

AFG 構築時間（Phase 1(a)）と比較した場合，エイリアス計算（Phase 2）に要する時間コストは十分に小さい．TextEditor の場合，自身と関連するクラスライブラリの AFG 構築時間（100,080ms = 100ms + 99,980ms）であるが，平均エイリアス計算時間は 0.65ms である（表 3.6）．

オンデマンド解析は，プログラム全体のエイリアス情報を必要とするような，コンパイラ最適化，データフロー解析の前処理には不向きであるが，求めるエイリアスが特定のものに限られている場合やユーザとの対話形式でエイリアスを求める解析ツールの実現においては，実用に十分に足るものであると考えている．

□精度（エイリアス集合の平均要素数）

提案手法は，同一クラスの異なるインスタンスの外部エイリアス関係を区別する，**インスタンス単位**（*Instance-based*）方式を採用している．一方，同一クラスの異なるインスタンスの外部エイリアス関係を区別しない，つまり，外部エイリアス関係を共有する，**クラス単位**（*Class-based*）方式を採用した場合，解析精度は低下する．既存のエイリアス解析手法には，インスタンス単位方式はなく，クラス単位方式のみである．

TextEditor の場合，インスタンス単位方式（平均 4.32）はクラス単位方式（平均 8.31）に比べ十分な解析精度が得られている．エイリアス集合の平均要素数は，クラス単位方式よりもずっと小さく，インスタンス単位方式は実用的な値をもたらした（表 3.7）．

ここでは，インスタンス属性に関してのみインスタンス単位方式の有効性のみを検証しており，インスタンスメソッドに関する両者の比較は行っていない．これは，今後の課題の一つである．

3.5.4 プログラム保守工程における JAAT の適用

JAAT の適用範囲として，プログラム保守，プログラム理解を考えている．しかし，現時点ではプログラム保守工程におけるツール及びエイリアス解析の有効性に関する実験的評価を行っていない．そこで，デバッグフェーズでのフォールト位置特定をシミュレートすることで，有効性の存在確認をまず行う．

SpreadSheet.java（約 1,000 行）は JDK に附属する表計算アプレットである．このアプレットの実行の際，図 3.16 のようなフォールトが発生したと仮定する．表中のセル C1（カーソル部）は fA1*B1，つまりセル A1（10.0）とセル B1（500.0）の積が表示されなければならない．ところが，セル C1 は 10.0 と出力されており，期待される値 5000.0 となっていない．

図 3.16: (適用事例) フォールトを含んだ実行結果

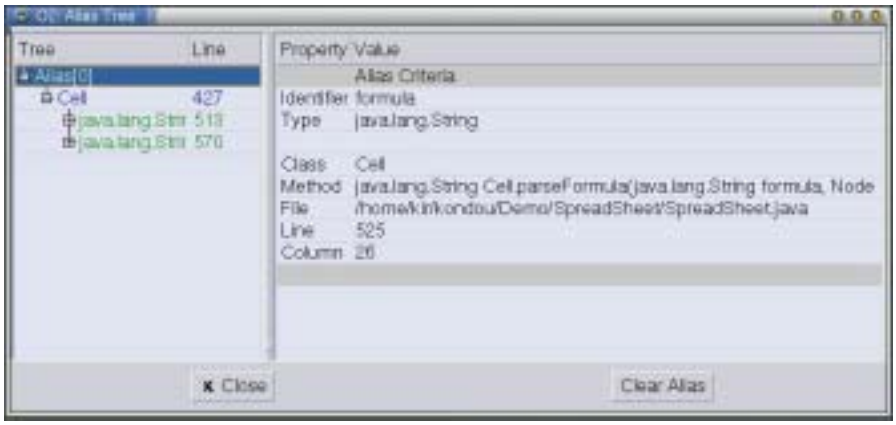


図 3.17: (適用事例) 参照変数 formula のエイリアス（エイリアスツリーウィンドウ）

そこで，以下に示すような手順でフォールト位置特定を試みる．

1. parseFormula() メソッドの第 1 仮引数である，String 型参照変数 formula のエイリアスを計算する．parseFormula() はその名が示すように，式の解析を行うメソッドである．

2. エイリアスツリーウィンドウ（図3.17）から，formula のエイリアスは parseFormula()，parseValue() メソッドにのみ存在することが分かる．非エイリアス部分は線分化されているため，テキストウィンドウ上（図 3.18）において容易にエイリアス解析結果を把握することができる．この例では，非エイリアス部分の線分化により，ソースコードの表示領域は 10%程度にまで削減されている．

3. エイリアスツリーウィンドウを参照し，エイリアスツリー上の各式を調べていくと，parseValue() メソッドにある最後の return 文が formula を返していることが分かる．なお，該当するエイリアスツリー節点をクリックすることで（図 3.19），formula が仮引数であることも分かる．

4. return 文付近を調べ，正しくは restFormula を返すべきであることを類推し，実際に return 文の引数を restFormula に変更することで，SpreadSheet.java は正しい実行結果をもたらすようになる（図 3.20）．

エイリアス解析結果の視覚化はプログラム保守活動の支援に有効であると考えられる．この適用例では，約 1,000 行の JAVA プログラムに対し，デバッグ対象を 20 行程度に絞る

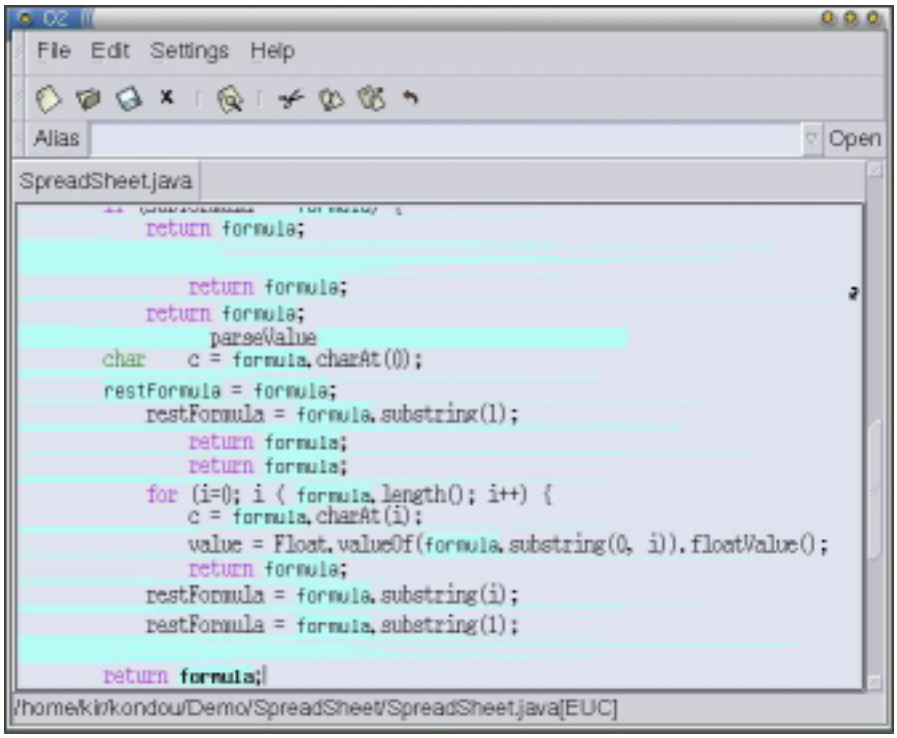


図 3.18: (適用事例) 参照変数 formula のエイリアス（テキストウィンドウ）

ことができた．ここではフォールト位置特定を例に用いたが，プログラム理解などにも応用可能である．

3.5.5 ポインタ変数を持つ言語でのエイリアス解析

3.3 で述べたアルゴリズムは JAVA の持つ参照型の式によるエイリアスを仮定したものであったが，C，C++などポインタ変数を持つ言語ではポインタ（特に高階ポインタ）による間接参照を考慮する必要がある．これは，引数として n($n \geq 2$) 階ポインタ変数を使用することで，例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ことによる．

提案手法をポインタに適用するため Phase 1(a)，Phase 2 を拡張する．以下にその拡張を簡潔に述べる．

□ Phase 1(a): AFG の構築

参照変数では 1 変数あたり 1 つ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 3.21 は，C 言語で記述されたサンプルプログラム及びそこから抽出された直接のエイリアス関係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` 及び b に関するエイリアス情報が AA-in 節点として，仮引数 y について y 及び `*y` に関するエイリアス情報が FA-in 節点として，手続き `main()`，手続き `assign(char`

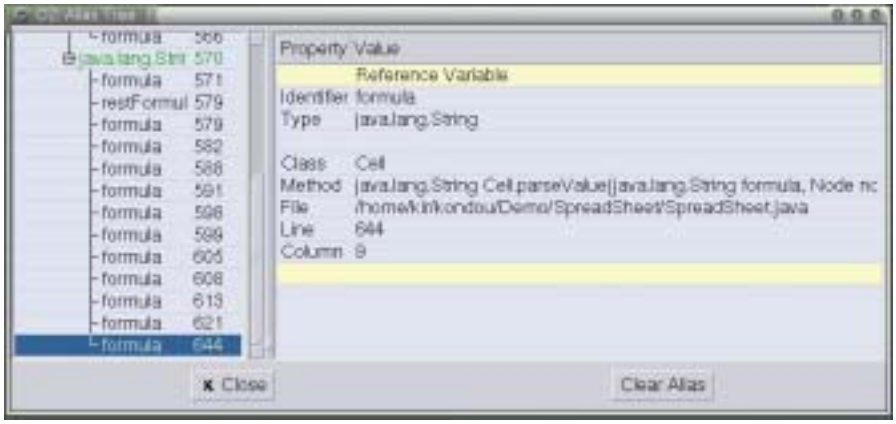


図 3.19: (適用事例) `parseValue()` メソッドにある最後の `return` 文の詳細

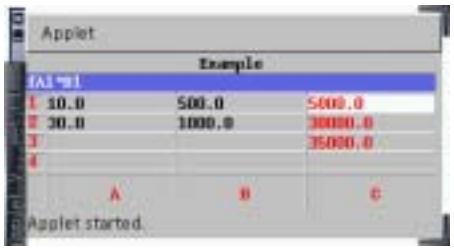


図 3.20: (適用事例) フォールトを除いた実行結果

`**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 節点がそれぞれ追加されている．

□ Phase 2: AFG 及び MFG によるエイリアス計算

ポインタに関する演算子としては，間接演算子'$*$' 及びアドレス演算子'&' がある．$*x$ は x が指す記憶領域の内容を，$\&x$ は x の記憶領域の位置を表す．エイリアス計算アルゴリズムをこれらの演算子に対応させるため，図 3.4 の **Step 2** を拡張する．詳細は，図 3.22 に示す．

3.5.6 関連研究

ここでは，関連研究について，いくつかの視点から本研究との比較を行う．

□ 2 フェーズ構成による解析

前もってモジュール内のエイリアス解析を行う手法としては [21, 11] があるが，これらの手法では各文の到達エイリアス集合の要素をエイリアス関係の成り立つ組 $\{(\alpha, \beta), (\gamma, \delta)\}$ で表現している．これは，α, β がエイリアス関係を満たすならば，(γ, δ) がエイリアスとして導出されることを意味する．このとき，(α, β) を (γ, δ) に対する制約条件という．この制約条件は，本論文でいう間接のエイリアス関係の成立条件に対応する．

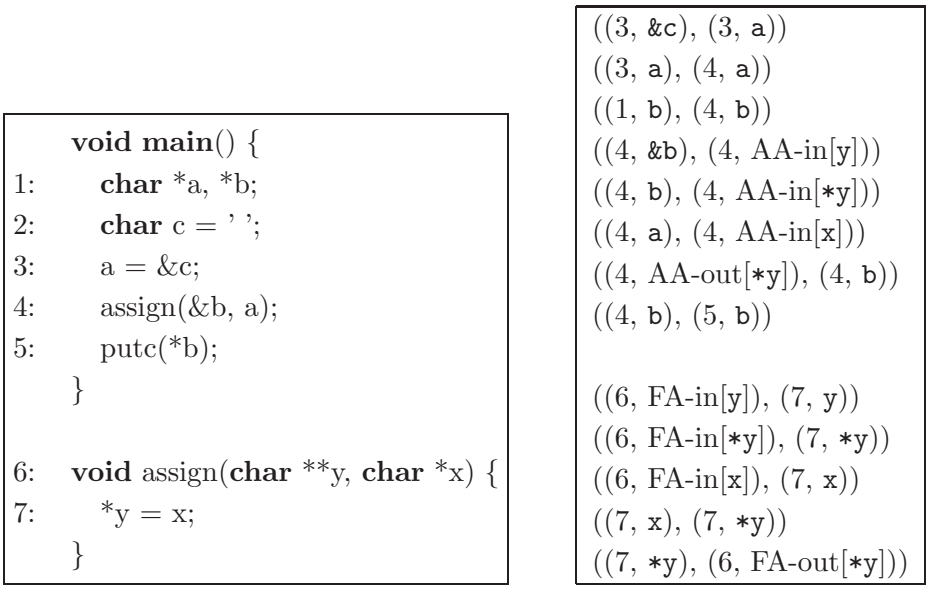


図 3.21: (例) ポインタ変数を持つ言語（C）でのエイリアス解析

しかし，これらの手法はエイリアス解析のモジュール化（呼び出し側モジュールの解析結果に依存しないエイリアス解析）を実現するため，各モジュール内でのエイリアス解析時に，対象モジュールから参照可能な外部変数（大域変数や仮引数など）のすべての組み合わせを (α, β) として与え，そのとき成立するエイリアスの組 (γ, δ) を抽出する必要がある．

我々の手法は直接のエイリアス関係のみを AFG 辺で表現し，間接のエイリアス関係は AFG 上での到達可能性を調べることで得るようにしている．したがって，モジュール内のエイリアス解析時（Phase 1(a)）には，外部変数すべての組み合わせを想定する必要はなく，直接のエイリアス関係を満たす節点間に AFG 辺を引くだけでよい．モジュール間をまたぐエイリアス解析（Phase 2）のとき，外部変数間のエイリアス関係は到達可能な AFG として与えられる．モジュール内解析ではエイリアスとして導出されなかった 2 つの式が（外部からの影響により）間接のエイリアス関係を満たすかどうかは，与えられた外部からの AFG を経由することで到達可能となるかを調べればよい．

[21, 11] のアルゴリズムは，対象プログラム中のすべてのエイリアス関係を必要とする，データフロー解析の前処理に向いている．我々はエイリアス情報を利用したプログラム保守，プログラム理解の支援に焦点を当てており，これらはより単純なエイリアス情報の表現方法を求める．今回提案した AFG 探索アルゴリズムは，ユーザにより指定されたエイリアス基準に対するエイリアスを計算する目的で定義されており，データフロー解析の前処理には不向きである．ただし，プログラム中のすべてのエイリアス関係を抽出することを目的とする新たな AFG 探索アルゴリズムも新たに定義可能である．

また，[21] は手続き型言語に関してのみ論じており，[11] は参照変数への代入により発生するエイリアスのみが対象でポインタ変数によるエイリアスには触れられていない．また，いずれの研究もプロトタイプ実装を行っただけで，実用的なシステム構築には至っていない．

Step 2: E 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
加えて，到達節点 C の種類に応じて以下のような処理も行う
[Cond.1] 親節点を持つ標準（インスタンス属性）節点 :
……
[Cond.8] '$*$' 演算子を持つ AFG 標準節点（$*x = \varphi^{-1}(C)$）:
1. $X := \varphi(x)$
2. $\mathcal{A}(X)$ の計算
3. $\{\varphi(y) \mid \varphi(\&y) \in \mathcal{A}(X)\}$ から AFG 辺をたどる
   $\{\varphi(*y) \mid \varphi(y) \in \mathcal{A}(X)\}$ から AFG 辺をたどる

[Cond.9] '&' 演算子を持つ AFG 標準節点（$\&x = \varphi^{-1}(C)$）:
1. $X := \varphi(x)$
2. $\mathcal{A}(X)$ の計算
3. $\{\varphi(y) \mid \varphi(*y) \in \mathcal{A}(X)\}$ から AFG 辺をたどる
   $\{\varphi(\&y) \mid \varphi(y) \in \mathcal{A}(X)\}$ から AFG 辺をたどる

図 3.22: (アルゴリズム) エイリアス計算（ポインタ）

□インスタンス単位の解析

インスタンス単位の解析方法は，**オブジェクトスライシング**（*Object Slicing*）[31] で提案されている．オブジェクトスライシングとは，オブジェクト指向プログラムを対象とする**システム依存グラフ**（*System Dependence Graph, SDG*）を利用し，特定オブジェクトに関するスライスを計算する手法である．しかしながら，この手法はエイリアス解析の存在が前提となっているが，その解析方法については触れられていない．実利用を考慮したエイリアス解析システムを構築するためには，我々が提案しているようなエイリアス解析アルゴリズムとインスタンス単位方式との統合が重要であると考えている．

□オンデマンド解析

今回，我々はオンデマンド解析によるエイリアス解析手法を提案した．これまで，エイリアス情報はコンパイラ最適化やデータフロー解析など，他の解析で利用されることを前提としていたが [2, 36, 13, 34, 43, 45]，Java プログラムの保守，理解活動においてはエイリアス情報そのものが有効である．これは，Java プログラムが参照型の式により生成されるエイリアスを多く含むためである．

プログラム保守，プログラム理解では，すべてのエイリアス情報は必要とせず，オンデマンド解析が有効であると思われる．

□アルゴリズムの拡張可能性

1 章において，FS 解析及び FI 解析について述べたが，これらのアルゴリズムは全く異なったものであり，独立に実装されるのが一般的である．前者は到達エイリアス集合を利用し，後者は point-to グラフ若しくはエイリアスグラフを利用する．これらのアルゴリズムは，解析精度，解析コストに関してそれぞれ長所，短所があり，実利用を考慮したエイ

リアス解析システムは両者とも実装すべきである．また，さまざまなエイリアス解析手法を比較するために多くの試作システムが実装されてきたが [22, 23, 24, 49]，解析アルゴリズムの拡張に関しては考慮されていない．

提案手法ではメソッド内のエイリアス情報を複数のエイリアス解析アルゴリズムで共有することが可能である．各 AFG は，メソッド内 FS エイリアス関係のみを保持している．一般に FS 解析は FI 解析よりも多くの解析時間を必要とするが，メソッド内の解析コストはメソッド間の解析コストに比べはるかに小さく，メソッド内エイリアス解析で FS 方式を採用するのは実用的である．

エイリアス解析手法は AFG 探索アルゴリズムとして記述可能されるため，探索アルゴリズムを変更することで解析精度と解析コストのトレードオフが制御できる．FS オブジェクトコンテキスト及び FI オブジェクトコンテキストは，拡張可能アルゴリズムの適用例の一つである．インスタンス単位方式及びクラス単位方式の解析アルゴリズムもまた，その一例である（これらの比較は表 3.7 に示されている）．

現在，図 3.4 のエイリアス計算アルゴリズムは，FI オブジェクトコンテキストを使用しているため，完全な FS 解析とはいえない．FS オブジェクトコンテキストを実現するためには，エイリアス情報の伝播経路を把握するために AFG 辺を有向辺にする必要がある．更に，エイリアス関係を，may エイリアス関係と must エイリアス関係に分類することで，より精度の高い AFG 探索アルゴリズムの実現及びさらなるユーザの理解支援が可能となるだろう．

□スレッド，例外

FS エイリアス解析はプログラム文の実行順を考慮しているため，その解析精度は制御フロー情報の精度に依存する．いいかえると，FS エイリアスは制御フローに従い抽出される．そのため，より正確な制御フロー情報を抽出することができれば，FS エイリアス解析はより正確なものとなる．

現在，JAAT では，スレッドや例外による制御フローを考慮してないため，それらを含むプログラムに対するエイリアス解析結果は不正確なものとなる．これまでに多くの研究者がスレッド，例外に対する制御フロー解析手法を提案しており [50, 35]，これらのアルゴリズムを適用することで JAAT の解析精度も改善される．

□解析情報の視覚化

プログラム解析情報の視覚化に関する研究としては，SeeSoft[37] を元に AT&T で開発された SeeSlice[7] がある．SeeSlice はプログラムスライスの視覚化を目的に開発され，(1) 必要なもののみ表示する技法に対しては,「スライス基準からの距離に応じた強調表示，スライスを含まない手続きに関する情報の非表示」として，(2) 全体と詳細を同時に見る方法に対しては,「プログラムの 3 階層（ファイル，手続き，文）への分類，局所的なソースコードの表示」として実現されている．SeeSlice は手続き型プログラムのスライスを対象としているが，エイリアス解析結果の視覚化においても有効な技法が多く述べられている．

3.6 まとめ

本章では，オブジェクト指向言語に対する，スケーラビリティを考慮したオンデマンド解析による，高い解析精度とアルゴリズムの拡張性を持つエイリアス解析手法の提案を行った．この手法を利用することで，実利用を考慮したエイリアス解析システムの構築が容易となる．

また，提案手法を Java エイリアス解析ツール JAAT として実装し，その有効性を検証した．JAAT は，解析部，ユーザインターフェース部から構成されている．解析部は，JDK 附属クラスライブラリのような大規模プログラムも解析可能である．ユーザインターフェース部は，プログラム保守，プログラム理解を目的に開発され，適用事例を交えながらその有効性についても考察した．

今後の課題としては，

- FS オブジェクトコンテキストの実装と，FI 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，スレッドへの対応

などが挙げられる．

謝辞

本研究は，栢森情報科学振興財団の補助を受けた．

第4章　XMLデータベースを利用したプログラム解析の効率化 - Javaプログラム解析フレームワークの構築 -

4.1　導入

これまで，様々なプログラミング言語に対するプログラム解析手法の提案及びツールの実装がなされてきた．このようなツールではプログラム解析情報はメモリ上にのみ記憶されるのが一般的である．そのため，解析情報を必要とする際にはその都度対象プログラムを解析しなければならず効率が悪い．解析情報データベースを構築しそれに対する *Application Programming Interface (API)* を定義することで，解析情報の再利用が可能となるツールも存在するが，データベースが独自形式であったり，API が特定のプログラミング言語を前提としているものが多く，解析情報の二次利用が容易であるとはいい難い．

本章では，XML を用いてプログラム解析情報のデータベース化を行う手法を提案する．本手法により，データベース化による解析の効率化はもちろんのこと，XML を対象とするアプリケーションが数多く提供されていることから，二次利用の容易性も期待できる．また，Java エイリアス解析ツール JAAT に対し提案手法の実装である意味解析木 XML データベースを追加実装した，Java プログラム解析フレームワーク JAF を構築し，その有効性を確認した．

以降，4.2 でプログラム解析情報の XML データベース化の提案を行う．4.3 で提案手法の実現について述べ，4.4. で手法の評価及び関連研究について考察する．最後に 4.5. でまとめと今後の課題について述べる．

4.2　意味解析木の XML データベース化

プログラム解析情報の例としては，抽象構文木，**手続き呼び出しグラフ**（*Procedure Call Graph*，*PCG*），制御フローグラフなどがあるが，ここでは意味解析木に着目する．意味解析木は多くのプログラム解析手法においてその存在が前提となる解析情報の一つであり，データベース化による効果は大きい．

以降，XML について簡単に説明したのち，Java を対象とする意味解析木の XML データベース化手法の提案を行う．

4.2.1 拡張可能マークアップ言語 eXtensible Markup Language (XML)

拡張可能マークアップ言語（*eXtensible Markup Language, XML*）[14] は，文書構造化言語の一つである．

XML 文書（*XML Document*）は，**要素**（*Element*）などのマーク付けにより構造化され，木構造を形成する．また，XML はユーザの目的に応じて要素名や**属性**（*Attribute*）などを自由に定義することができる．つまり，XML はデータ構造及びそれが持つ情報を容易にかつ的確に表現することができる．

更に，**文書型定義**（*Document Type Definition, DTD*）を用いることで，要素間の親子関係，属性として持つべき情報を厳密に定義することができる．これにより，XML 文書に潜む致命的な構文間違いを防ぐことが可能となり，データベースが持つべき特性の一つである**一貫性**（*Consistency*）の保証をすることができる．

また，XML 文書を扱うためのライブラリ，プログラムも数多く開発されていることから，情報を XML 文書化しておくことによる利便性向上が十分に期待できる．

4.2.2 方針

意味解析木の XML 表記に関して，以下 2 つの方針に基づき，Java ソースコードの意味解析木に対する DTD 記述を行った．

方針 1: 要素を用いて，意味解析木が持つ構文情報を表現する

方針 2: 属性を用いて，意味解析木が持つ字句情報及び意味情報を表現する

Java 意味解析木の XML 表記に関する，要素の BNF 表記による定義を図 4.1 に，属性の定義を表 4.1 に示す．

```
Java ::= CompilationUnit.
CompilationUnit ::= Package? Import* (ClassType | InterfaceType | FakeType)*.
ClassType ::= Extends? Implements?
              (Method | Constructor | Variable | ClassType |InterfaceType |
               FakeType | InstanceInitializer | StaticInitializer)*.
InterfaceType ::= Extends?
                  (Method | Variable | ClassType | InterfaceType | FakeType)*.
Extends ::= type+.
Implements ::= type+.
Method ::= type Variable* Exception? Block?.
Constructor ::= Variable* Exception? Block?.
Variable ::= type Expression_.
InstanceInitializer ::= Block.
StaticInitializer ::= Block.
Statement_ ::= Block | If | While | Do | Switch | For | Try | Synchronized |
               Break | Continue | Return | Throw | Labeled | Empty |
               Expression_.
Block ::= (Variable | ClassType | InterfaceType | FakeType | Statement_)*.
If ::= Expression_ Statement_ Statement_?.
While ::= Expression_ Statement_.
Do ::= Statement_ Expression_.
Switch ::= Expression_ SwitchBlock?.
SwitchBlock ::= (Case | Statement_)*
Case ::= Expression_?
For ::= ForInit? ForExp? ForRenew? Statement_?.
ForInit ::= Variable | Operation*.
ForExp ::= Expression_.
ForRenew ::= method | Operation*.
Try ::= Block Catch* Finally?.
Catch ::= Variable Block.
Finally ::= Block.
Synchronized ::= Expression_ Block.
Return ::= Expression_?.
Throw ::= Expression_?.
Labeled ::= Statement_.
Expression_ ::= variable | type | method | Literal | ClassInstanciation |
                ArrayInstanciation | ArrayInitializer | Operation.
method ::= Expression_*.
ClassInstanciation ::= type method.
ArrayInstanciation ::= type Expression_+.
ArrayInitializer ::= Expression_*.
Operation ::= type Expression_ Expression_?.
```

図 4.1: (定義) XML 要素

4.2.3 適用例

図 4.2 に示すサンプルプログラムの意味解析木に対する XML 表記を図 4.3 に示す．図 4.2 の代入演算 `b = a + 1` は，図 4.3 の <`Operation text="="`> に対応する．第一子要素 `type` は演算結果の型を，第二，第三子要素はそれぞれ代入演算の左被演算子，右被演

表 4.1: (定義) XML 属性

属性名	役割
`text`	識別子の限定名，または演算子
`text_`	識別子の単純名（単純名による参照が可能なときに限る）
`modifiers`	修飾子
`id`	識別子の宣言，制御移行先文（`break` 文，`continue` 文の飛び先）が持つ，ユニークな識別番号
`ref`	識別子の参照，制御移行元（`break` 文，`continue` 文）が持つ，識別番号[1]
`file`	識別子が定義された.xml ファイル名（定義が別ファイルのときに限る）

```
1:  class Sample {
2:      public static void main(String args[]) {
3:          int a = 1;
4:          int b;
5:          b = a + 1;
6:      }
7:  }
```

図 4.2: (例) プログラム

算子を表す.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Java SYSTEM "java.dtd">

<Java>
  <CompilationUnit>
    <Package text=""/>
    <Import text="java.lang.*"/>
    <ClassType modifiers="" text="Sam5" id="0x12">
      <Extends>
        <type text="Object" ref="0x18" file="java/lang/Object.xml"/>
      </Extends>
      <Method modifiers="public static" text="main" id="0x189b">
        <type text="void" ref="0x4"/>
        <Variable modifiers="" text="args" id="0x18a4">
          <type text="array" ref="0x20"/>
        </Variable>
        <Block>
          <Variable modifiers="" text="a" id="0x18a9">
            <type text="int" ref="0x7"/>
            <Literal text="1"/>
          </Variable>
          <Variable modifiers="" text="b" id="0x18ad">
            <type text="int" ref="0x7"/>
          </Variable>
          <Operation text="=">
            <type text="int" ref="0x7"/>
            <variable text="b" ref="0x18ad"/>
            <Operation text="+">
              <type text="int" ref="0x7"/>
              <variable text="a" ref="0x18a9"/>
              <Literal text="1"/>
            </Operation>
          </Operation>
        </Block>
      </Method>
    </ClassType>
  </CompilationUnit>
</Java>
```

図 4.3: (例) 図 4.2 の意味解析木の XML 表記

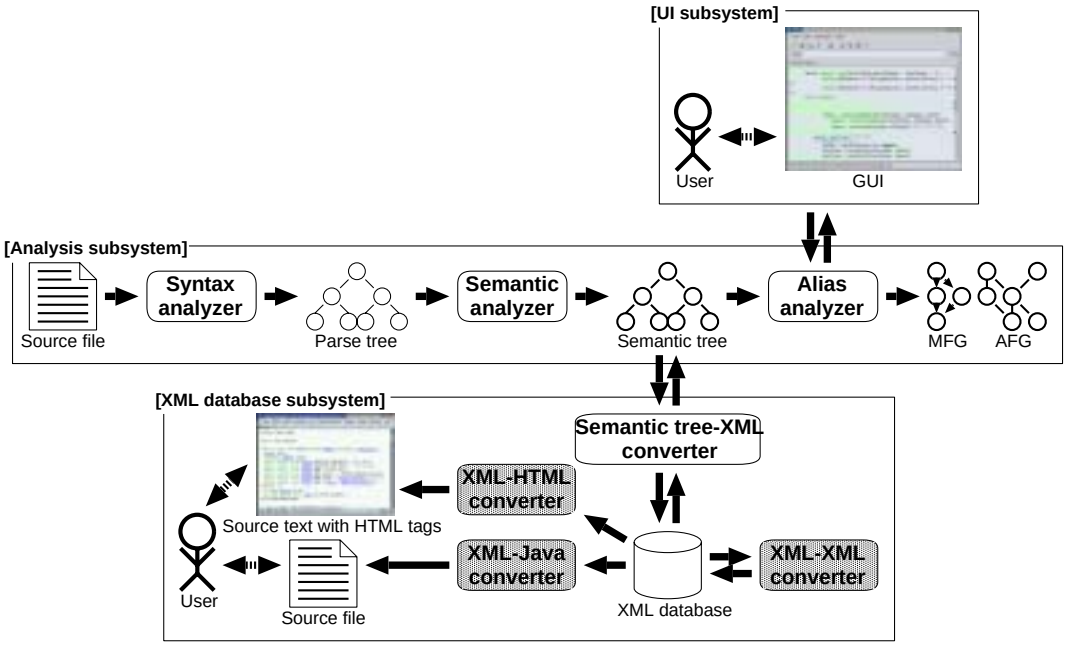


図 4.4: (実装) JAVA プログラム解析フレームワーク

4.3 Java プログラム解析フレームワーク Java program Analysis Framework (JAF)

これまでに我々の研究グループでは，JAVA エイリアス解析ツール JAAT を開発している．JAAT は，本来エイリアス解析及びエイリアス視覚化を目的とし開発されたものであるが，その設計においては，他のプログラム解析での利用を考慮したものとなっている．具体的には，解析部は，構文解析，意味解析，エイリアス解析の各モジュールはそれぞれ独立したライブラリとして，またユーザインターフェース部も独立したツールとして存在している．

今回，JAAT に対して提案手法を追加実装し，新たに **Java プログラム解析フレームワーク**（JAVA *program Analysis Framework, JAF*）を構築した．図 4.4 にその概要を示す．図 4.4 において，上段及び中段がそれぞれ，JAAT に組み込まれていた解析部（Analysis Subsystem）及びユーザインターフェース部（UI Subsystem）であり，下段が提案手法の実装である，**XML データベース部**（*XML Database Subsystem*）である．

以降，解析部，ユーザーインターフェース部，XML データベース部をそれぞれ説明する．なお，解析部及びユーザーインターフェース部に関しては，JAAT での説明との重複を抑えるため簡単な説明に留める．

4.3.1 解析部

□ **Syntax Analyzer**（`libANTLR`）

JAVA プログラムのソースコードを読み込み，字句解析及び構文解析を行い，抽象構文木を構築する．

□ **Semantic Analyzer**（`libjavamm`）

抽象構文木を読み込み，意味解析を行い，意味解析木を構築する．

□ **Alias Analyzer**（`libAFG`）

意味解析木を読み込み，AFG 及び MFG の構築を行う．また，ユーザの要求に応じたエイリアス計算もこれにより行われる．

4.3.2 ユーザーインターフェース部

テキストウインドウ及びエイリアスツリーウインドウによるエイリアス表示，プログラム編集を行う．

4.3.3 XML データベース部

XML データベース部は，提案手法の実装である意味解析木-XML 相互変換ライブラリ，変換ライブラリにより作成された XML 文書の集合である XML データベース，及び XML 文書を対象とする XML アプリケーション群で構成されている．

□ **SemanticTree-XML Converter**（意味解析木-XML 相互変換ライブラリ）

意味解析木-XML 相互変換ライブラリは C++で記述されており，

(1) 意味解析木を読み込み，変換された XML 文書を XML データベースに登録

(2) XML データベースから XML 文書を読み込み，意味解析木を復元

の 2 つの機能を提供している．

(2) の実現において，XML 文書解析のための **XML パーザ**（*XML Parser*）として，libxml[42] 及び XercesC++[47] を利用した．これらの XML パーザは *DOM (Document Object Model)*[12] と呼ばれる API を提供しており，XML パーザは XML 文書を読み込むと，それを一つの木構造とみなした **DOM ツリー**（*DOM Tree*）を構築する．そして，DOM ツリーを介して XML 文書内の各要素を参照しながら，対応する意味解析木要素が生成される．

DOM を提供する XML パーザはそのインターフェースが酷似しており，簡単なマクロ変更で様々な XML パーザを併用できるよう実装を行った．現時点では，ユーザの要求に応じて，上記 2 つのうちいずれかの XML パーザが選択できるようになっている．

□ **XML Database**（XML データベース）

表 4.2: (実験結果) 意味解析木構築コスト

構築元	構築時間 [sec]	データサイズ [MB]
ソースコード	37	25
XML データベース	24	62

XML データベースはファイルシステム上に保存された XML 文書の集合である．JAF 上でプログラムを解析する際，対応する XML 文書が既にこのデータベースに存在する場合，そのソースコードではなく XML 文書を読み込み意味解析木を構築する．

□ **XML Applications**（XML アプリケーション群）

XML データベースを利用する応用アプリケーション群である．XML-Java 変換，XML-HTML 変換，XML-XML 変換をそれぞれ行う．これらの詳細は後述する．

4.4 評価

4.1 において，XML データベース化により期待される効果として，解析効率の向上，二次利用の容易性を挙げた．本節では，それぞれの観点から提案手法の検証を行う．具体的には，前者はデータベースの有無による意味解析木構築時間の比較を行い，後者はいくつかの応用アプリケーションの試作を行ってみた

4.4.1 実験

はじめに，JDK 附属クラスライブラリの全ソースコードの意味解析木を XML 文書に変換し，XML データベースを構築した．この変換には 46sec を要した．

次に，意味解析木構築に関して，ソースコードからの構築時間と XML データベースからの再構築時間との比較を行なった（図 4.2）．ソースコードからの構築は 37sec，データベースからの再構築は 24sec[2]であり，約 40%削減されたことになる．また，構築された XML データベースの大きさは 62MB，すべてのソースコードの大きさの合計は 24MB であった．

4.4.2 応用アプリケーション

XML データベースが持つ解析情報の二次利用の例として，試作したいくつかの応用アプリケーションを紹介する．

□ **XML-Java 変換**（XML 文書をソースコードのテキスト表記に変換）

[2]再構築時間は XML パーザの性能に大きく依存する．そのため，より高速な XML パーザを利用することで再構築時間の短縮が期待される．

```
1:  import java.lang.*;
2:
3:  class Sample extends Object {
4:      public static void main(String[] args) {
5:          int a = 1;
6:          int b;
7:          b = a + 1;
8:      }
9:  }
```

図 4.5: (例) 図 4.3 に対する XML-JAVA 変換

意味解析木の持つ字句情報及び構文情報を利用し，コンパイル可能でかつ十分に閲覧可能なソースコードの復元を行う．これには libxml を利用した C++実装（約 1,500 行）と，XSLT による実装（約 2,000 行）がある．

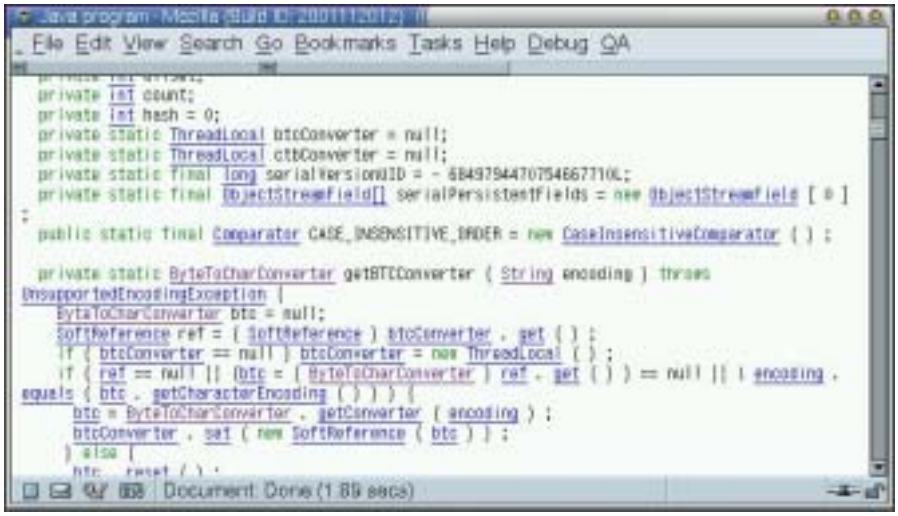
XSL 変換（*XSL Transformations, XSLT*）[48] とは，XML 文書のスタイル指定言語である**拡張可能スタイルシート言語**（*eXtensible Stylesheet Language, XSL*）の一部で，XML 文書の木構造に対し，各節点に一致する条件及びその条件に一致したときの処理を記述するものである．これを利用することで，XML 文書を他の XML 文書（HTML 文書及びプレーンテキスト文書を含む）に変換することができる．なお，実際に変換を行う XSLT プロセッサには Xalan[46] を利用した．

例として，図 4.3 に対する XML-JAVA 変換を図 4.5 に示す．

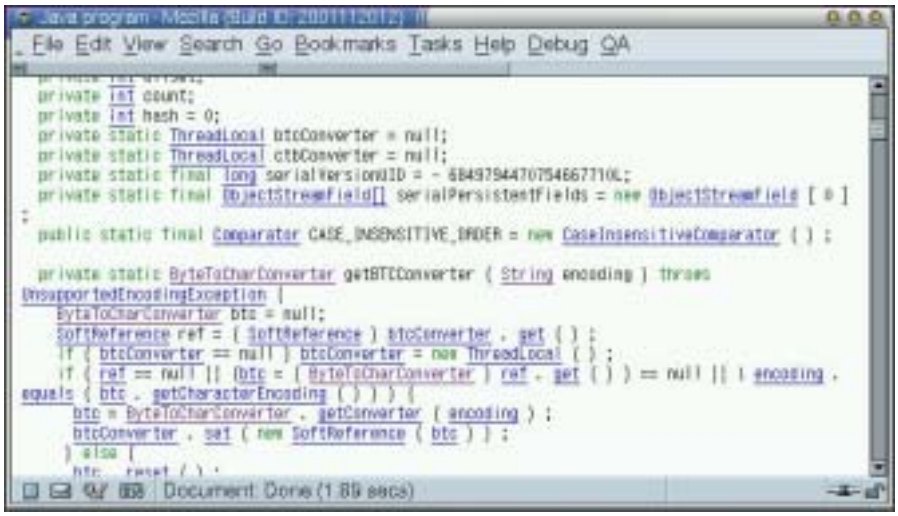
□ **XML-HTML 変換**（XML 文書をソースコードの HTML 表記に変換）

XML-JAVA 変換に HTML タグを埋め込みを追加することにより，ウェブブラウザを介したソースコード閲覧を可能にする．また，識別子（参照型，メソッド，属性，ローカル変数）の宣言と参照間の関係はリンクを用いて表現されている．これは XSLT により実装された（約 2,000 行）．

図 4.6 にその例を示す．この例では，`java.lang.String` クラス（図 4.6(a)）内において参照型 `ThreadLocal` が利用されており，そのリンク（下線部）をクリックすることで `ThreadLocal` を定義している `java.lang.ThreadLocal` クラス（図 4.6(b)）に対応する.html ファイルへ移動する．



(a) java.lang.String クラス



(b) java.lang.ThreadLocal クラス

図 4.6: (例) XML-HTML 変換

また，HTML への変換の際，指定した深さ以上のブロック構造に含まれるコード片を非表示にできるよう拡張したものも作成した．図 4.7 は，java.lang.String クラスの.html ファイルの作成において，深さ 2 以上の文を非表示にする制約を付加したものである．この例では，非表示部分は'Secret!' と置き換えられている．

□ **XML-XML 変換**（XML 文書中の識別子を置換）

識別子の置換は一般的なエディタでも可能であるが，置換操作を行う際に異なるスコープ上に存在する同一名の識別子を区別することはできない．そのため，誤操作によって意味的誤りを生じさせてしまう可能性がある．このアプリケーションでは，置換の際に識別

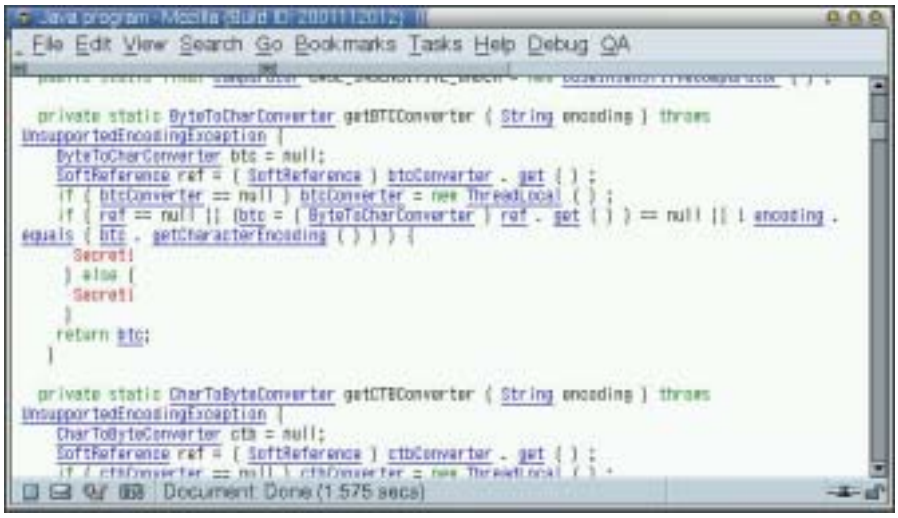


図 4.7: (例) XML-HTML 変換（深さ 2 以上の文を非表示）

子名と id 属性の値の組を与えることができるため，上記の問題を回避することができる．[3]これは XSLT を用いて実装されている（約 600 行）．

図 4.8 は，java.lang.String クラスにおいて，id 属性値に'0x34' を持つ変数名 value（図 4.8(a)）を after_value（図 4.8(b)）へ置換した例である．その際，id 属性値を制約として与えているため，異なるスコープ上に存在する変数 value（id 属性値'0x4d17'）は置換の対象から除外されていることが分かる．

4.4.3 考察

□時間コスト（意味解析木構築の時間）

初回に XML データベース構築が行われるが，それ以降，対象プログラムのソースコードが変更されない限りデータベースの再構築は必要なく，この構築時間を考慮したとしても，データベース化による恩恵は大きい．

□空間コスト（データの大きさ）

XML データベースはソースコードに比べ 2.5 倍の大きさとなった．しかし，XML データベースが意味情報を含むこと，またテキストデータベースであることを考慮すれば，この程度の大きさに抑えれたことは十分に評価できる．

□二次利用の容易性

既存の多くのデータベースが自身へのアクセスに際して特定の言語を要求することが多く（データベースが API を複数言語で提供することはその実現性から考えて難しい），二次利用の容易性を満たすのは一般に難しい．

[3]異なるスコープ上に存在する識別子は，同一名であってもその id 属性の値が異なる．

```
   ...
   <Variable modifiers="private" text="value" id="0x34">
     <type text="char[]" ref="0x9"/>
   </Variable>
   <Variable modifiers="private" text="offset" id="0x3b">
     <type text="int" ref="0x7"/>
   </Variable>
   ...
 <Constructor modifiers="public" text="String" id="0x34e">
   <Variable modifiers="" text="value" id="0x4d17">
     <type text="java.lang.String" text_="String" ref="0x22"/>
   </Variable>
```

(a) XML 表記（置換前）

```
  ...
  <Variable modifiers="private" text="after_value" id="0x34">
    <type text="char[]" ref="0x9"/>
  </Variable>
  <Variable modifiers="private" text="offset" id="0x3b">
    <type text="int" ref="0x7"/>
  </Variable>
  ...
<Constructor modifiers="public" text="String" id="0x34e">
  <Variable modifiers="" text="value" id="0x4d17">
    <type text="java.lang.String" text_="String" ref="0x22"/>
  </Variable>
```

(b) XML 表記（置換後）

図 4.8: (例) XML-XML 変換（id 属性値'0x34' を持つ変数名 value の after_value への置換）

我々のデータベースは XML を用いて構築されているため，XML の特徴がそのまま生かされる．現在のプログラム開発環境で利用されることの多いプログラミング言語では，XML に対する API が例外なく提供されている．そのため，C，C++，perl，XSLT など，様々な言語で応用アプリケーションを記述することができる．

4.4.4 関連研究

プログラム解析情報の XML 表記に関する研究として，JavaML[6] がある．JavaML は，Java プログラムのソースコードに対して XML タグを埋め込む形式であり，プログラム変換，プログラム理解を主な目的としている．しかし，意味情報が十分に含まれておらず，意味解析木データベースとしての利用は困難である．例えば，単一ファイル内での宣言と参照間の関係は存在するが，複数ファイル間におよぶ関係は考慮されていない．

プログラム解析情報のデータベース化を実現したシステムとしては，細粒度リポジトリに基づく CASE ツール・プラットフォームである Sapid[56] がある．Sapid は，**ソフトウェ**

アデータベース（*Software Database, SDB*），**アクセスルーチン**（*Access Routines, AR*），**ソフトウェア操作言語**（*Software Manipulating Language, SML*）から構成される．SDB は，我々の意味解析木 XML データベースが持つ構文，構造，意味の各情報以外にも，プリプロセッサ命令，メモリ領域及びその値などの情報も存在しており，この点で我々より優れている．しかしながら，SDB にアクセスするためには，AR が提供する API を利用した C プログラム，若しくは SML で記述されたプログラムが前提となる．SDB とそのテキスト表記を相互変換可能な StreamedSapid（以下，|Sapid|）[38] があるが，そのテキスト表記の構造は |Sapid| 独自形式である．

4.5 まとめ

一般に，プログラム解析ツールにより得られる解析情報はメモリ上にのみ存在し，解析情報を繰り返し利用することができないため，効率が悪い．また，データベースを利用することで解析情報の再利用を考慮したツールも存在するが，データベースが独自形式であるために他のアプリケーションでの二次利用が容易ではない．

本章では，これら 2 つの問題を解決する手法として，プログラム解析情報の一つである意味解析木の XML データベース化を提案した．また，実装及びいくつかの応用アプリケーションを試作し，解析効率の向上，二次利用の容易性を確認した．

今後の課題としては，意味解析木以外の解析情報，例えば，手続き呼び出しグラフ，制御フローグラフなどのデータベース化が考えられる．また，データベースの持つ情報量とその管理コストは比例関係にあるため，ユーザが必要とする情報からのみ構成されるデータベースを構築する機構の実現を考えている．

謝辞

本研究は，栢森情報科学振興財団の補助を受けた．

第5章　むすび

5.1　まとめ

本論文では，静的解析により求められるプログラム解析情報に関する，3つの効率化手法について述べた．まず，プログラム依存グラフの節点集約によるスライス計算の効率化を行った．次に，エイリアス情報のモジュール化によるエイリアス計算の効率化を行った．最後に，XMLデータベースを利用したプログラム解析の効率化を行った．

また，これらの提案手法は，解析の効率化に加え，解析に関するコストと精度のトレードオフ制御，解析情報の二次利用を考慮しており，実装により提案手法の有効性を確認した．

5.2　今後の研究方針

本論文では，様々なプログラム解析情報のうち，データフロー，制御フロー，エイリアス，意味解析木，それぞれを効率よく解析するための解析手法を提案した．

これらのうち，意味解析木はより上位に位置する情報取得のために利用されるのが一般的であり除外するが，データフロー，制御フロー，エイリアスなどには，自身の情報の抽出に他者の情報が必要となる，**情報抽出過程の相互依存**が発生する．通常，これらの解析手法においては，情報抽出過程の相互依存による解析の無限の繰り返しを防ぐため，当該部分において，再帰的な情報抽出を行う代わりに（情報抽出を必要としない）最悪事象の想定による情報を作為的に作成し，それを利用する方法が採用されている．

しかしながら，このように作為的に作成された情報に対し，解析中に

- より精度の高い情報に置き換えられるかどうかの判定
- 置き換えられると判定されたときの，当該値の計算
- 置き換えによる影響範囲と，その範囲内に存在する情報の再計算

を行い，解析精度を向上させる機構は既存のプログラム解析手法には存在しない．

今後，情報抽出過程の相互依存を考慮した，総合的なプログラム解析フレームワークを構築することで，さらなるプログラム解析の精度向上が図られるだろう．

これまで，本研究では静的解析によるプログラム解析フレームワークを中心に取り扱ってきた．しかし，我々の研究グループでは動的解析に基づく解析フレームワークの開発も並行して行なわれており，静的解析と動的解析の組み合わせによる新たなプログラム解析手法の提案及びその実現を目指している．このような組み合わせによる解析手法には[4]が既にあるが，手続き型言語を解析対象としており，現在のソフトウェア開発環境への貢

献は少ない．我々の解析フレームワークはオブジェクト指向言語 JAVA を対象としており，今後，オブジェクト指向言語に対する様々な解析手法の実現が期待できる．

参考文献

[1] Agrawal, H. and Horgan, J., "Dynamic Program Slicing", *SIGPLAN Notices*, vol.25, no.6, pages.246–256, 1990.

[2] Aho, A.V., Sethi, S. and Ullman, J.D., "Compilers : Principles, Techniques, and Tools", 1986.

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

[4] Ashida, Y., Ohata, F. and Inoue, K., "Slicing Methods Using Static and Dynamic Information", *Proceedings of the 6th Asia Pacific Software Engineering Conference (APSEC'99)*, pages.344–350, Takamatsu, Japan, 1999.

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

[6] Badros, G.J., "JavaML:a markup language for JAVA source code", *Computer Networks*, vol.33, pages.159–177, 2000.

[7] Ball, T. and Eick, S.G., "Visualizing Program Slices", *Proceedings of the 1994 IEEE Symposium on Visual Languages*, pages.288–295, St. Louis, Missouri, 1994.

[8] Bates, S. and Horwitz, S., "Incremental program testing using program dependence graphs", *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pages.384–396, Charleston, South Carolina, 1993.

[9] Beck, J. and D, Eichmann., "Program and interface slicing for reverse engineering", *Proceedings of the 15th International Conference on Software Engineering*, pages.509–518, Baltimore, Maryland, 1993.

[10] Booch, G.,"Object-Oriented Design with Application", 1991.

[11] Chatterjeem, R.K. and Ryder, B.G., "Modular Concrete Type-Inference for Statically Typed Object-Oriented Programming Languages", *Technical Report*, no.DCS-TR-349, Rutgers University, 1997.

[12] "Document Object Model (DOM)", `http://www.w3.org/DOM/`.

[13] Enami, M., Ghiya, R. and Hendren, L.J., "Context-sensitive interprocedural points-to analysis in the presence of function pointers", *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages.242–256, Orlando, Florida, 1994.

[14] "Extensible Markup Language (XML) 1.0 (Second Edition)", `http://www.w3.org/TR/2000/REC-xml-20001006`.

[15] Fiutem, R., Tonella, P., Antoniol, G. and Merlo, E., "Variable Precision Reaching Definitions Analysis for Software Maintenance", *Proceedings of the Euromicro Working Conf. on Soft. Maintenance and Reengineering*, pages.60–67, Berlin, Germany, 1997.

[16] Gallagher, K.B and Lyle, J.R., "Using program slicing in software maintenance", In *IEEE Transactions on Software Engineering*, vol.17, no.8, pages.751–761, 1991.

[17] Ghiya, R. and Hendren, L.J., "Connection Analysis: A practical interprocedural heap analysis for C", *International Journal of Parallel Programming*, vol.24, no.6, pages.547–578, 1996.

[18] Gosling, J., Joy, B. and Steele, G., "The JAVA™ Language Specification", 1996.

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

[20] "Gtk−−", `http://gtkmm.sourceforge.net/`.

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

[22] Hind, M. and Pioli, A., "An empirical comparison of interprocedural pointer alias analysis", *IBM Research Report*, no.21058, 1997.

[23] Hind, M. and Pioli, A., "Assessing the Effects of Flow-Sensitivity on Pointer Alias Analysis", *IBM Research Report*, no.21251, 1998.

[24] Hind, M. and Pioli, A., "Which Pointer Analysis Should I Use?", *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages.113–123, Portland, Oregon, 2000.

[25] Hind, M., "Pointer Analysis: Haven't We Solved This Problem Yet?", *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages.54–61, Snowbird, Utah, 2001.

[26] Horwitz, S., Reps, T. and Binkley, D., "Interprocedural slicing using dependence graphs", *ACM Transactions on Programming Languages and Systems*, vol.12, no.1, pages.26–60, 1990.

[27] Horwitz, S. and Reps, T., "The use of program dependence graphs in software engineering", *Proceedings of the 14th International Conference on Software Engineering*, pages.392–411, Melbourne, Australia, 1992.

[28] Jackson, D. and Rinard, M., "Software Analysis: A Roadmap", *The Future of Software Engineering*, pages.135–145, 2000.

[29] Korel, B. and Rilling, J., "Dynamic program slicing methods", *Information and Software Technology Special Issue on Program Slicing*, vol.40, pages.647–659, 1998.

[30] Kudo, H., Sugiyama, Y., Fujii, M. and Torii, K., "Quantifying a design process based on experiments", *Proceedings of the 21th International Conference on System Sciences*, vol.2, pages.285–292, Kailua-Kona, Hawaii, 1988.

[31] Liang, D. and Harrold, M.J., "Slicing Objects Using System Dependence Graphs", *Proceedings of the International Conference on Software Maintenance*, pages.358–367, Washington, D.C., 1998.

[32] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K., "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", *Proceedings of the 21th International Conference on Software Engineering*, pages.422-431, Los Angeles, California, 1999.

[33] Pressman, R.S., "Software Engineering A Practitioner's Approach, fourth edition", 1997.

[34] Shapiro, M. and Horwitz, S.,"Fast and accurate flow-insensitive point-to analysis", *Proceedings of the 24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages.1–14, Paris, France, 1997.

[35] Sinha, S. and Harrold, M. J., "Analysis of Programs With Exception-Handling Constructs", *Proceedings of the International Conference on Software Maintenance*, pages.358–367, Washington, D.C., 1998.

[36] Steensgaard, B., "Points-to analysis in almost linear time", *In Proceedings of the 23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages.32–41, Beach, Florida, 1996.

[37] Steffen, L.J., Eick, S.G. and Sumner Jr., E.E., " Seesoft - a tool for visualizing line-oriented software statistics", *IEEE Transactions on Software Engineering*, vol.18, no.11, pages.957–968, 1992.

[38] "|Sapid|", `http://streamed.sapid.org/`.

[39] Stroustrup, B., "The C++ Programming Language (Third edition)", 1997.

[40] "Tcl/Tk", `http://dev.scriptics.com/software/tcltk/`.

[41] "The Wisconsin Program-Slicing Tool 1.0, Reference Manual", University of Wisconsin-Madison, 1997.

[42] "The XML C library for Gnome", `http://xmlsoft.org/`.

[43] Tonella, P., Antoniol, G., Fiutem, R. and Merlo, E., "Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing", *Proceedings of the 19th International Conference on Software Engineering*, pages.433–443, Boston, Massachusetts, 1997.

[44] Weiser, M., "Program slicing", *Proceedings of the 5th International Conference on Software Engineering*, pages.439–449, San Diego, California, 1981.

[45] Wilson, R.P. and Lam, M.S., “Efficient context-sensitive pointer analysis for C programs”, *Proceedings of SIGPLAN’95 Conference on Programming Language Design and Implementation*, pages.1–12, La Jolla, California, 1995.

[46] “Xalan-C++ version 1.2”, `http://xml.apache.org/xalan-c/index.html`.

[47] “Xerces C++ Parser”, `http://xml.apache.org/xerces-c/index.html`.

[48] “XSL Transformations (XSLT)”, `http://www.w3.org/TR/xslt`.

[49] Zhang, S., Ryder, B.G. and Landi, W.A., “Experiments with Combined Analysis for Pointer Aliasing”, *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages.11–18, Montreal, Canada, 1998.

[50] Zhao, J., “Slicing Concurrent Java Programs”, *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, pages.126–133, Pittsburgh, Pennsylvania, 1999.

[51] 佐藤 慎一, 飯田 元, 井上 克郎, “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.

[52] 植田 良一, 練 林, 井上 克郎, 鳥居 宏次, “再帰を含むプログラムのスライス計算法”, **電子情報通信学会論文誌**, vol.J78-D-I, no.1, pages.11–22, 1995.

[53] 西松 顯, 西江 圭介, 楠本 真二, 井上 克郎, “フォールト位置特定におけるプログラムスライスの実験的評価”, **電子情報通信学会論文誌**, vol.J82-D-I, no.11, pages.1336–1344, 1999.

[54] 西松 顯, 楠本 真二, 井上 克郎, “保守プロセスに対するプログラムスライスの実験的評価”, **電子情報通信学会論文誌**, vol.J82-D-I, no.8, pages.1121–1123, 1999.

[55] 高田 智規, 佐藤 慎一, 井上 克郎, “プログラム依存グラフの効率的な更新手法”, **電子情報通信学会論文誌**, vol.J81-D-I, no.3, pages.253–260, 1998.

[56] 福安 直樹, 山本 晋一郎, 阿草 清滋, “細粒度ソフトウェア・リポジトリに基づいたCASEツール・プラットフォームSapid”, **情報処理学会論文誌**, vol.39, no.6, pages.1990–1998, 1998.

[57] 増井 俊之, “情報視覚化の最近の研究動向”, **第 *9* 回データ工学ワークショップ論文集**, 1998.