プログラム解析の効率化に関する研究

高田 智規

2002 年 8 月

内容梗概

ソフトウェア開発において，デバッグ・テスト・保守フェーズにおける比重はソフトウェアの大規模化に伴い増加している．これらのフェーズにおいてフォールト位置の特定は非常に困難な作業であり，この作業を効率的に行うことが生産性を向上させるために重要である．作業者はデバッガなどの CASE（Computer Aided Software Engineering）ツールと呼ばれる開発・デバッグ支援ツールを利用することが多いが，一般的な CASE ツールは変数の値などの限定された情報しか提供しない．そのため，作業者はフォールト位置の特定に多大な労力を要する．

この作業を効率化するための 1 つのアプローチとして，プログラム中の文間の依存関係を解析する方法がある．文間の依存関係を求めることができれば，特定の文に関連のある文を抽出するが可能となるため，フォールト位置の特定のためにプログラム全体を確認する必要が無くなり，デバッグの効率が向上する．

しかし，一般に，プログラム解析には非常に多くの資源（時間・メモリ空間など）を要するため，実際のデバッグに適用することが難しい．したがって，プログラム解析を効率化し，解析に必要なオーバヘッドを小さくすることが重要である．

そのため，本論文では，以下の 3 つのプログラム解析の効率化手法の提案と評価を行う．

1. ソースプログラムが変更された場合の再解析オーバヘッド削減
2. 静的解析情報と動的解析情報を組み合わせた効率的な準動的解析手法
3. 準動的解析におけるより効率的な解析手法

1. では，ソースプログラム中の文に対して追加・削除・修正といった変更が行われた場合に，プログラム依存グラフのうち変更個所に関する部分のみを更新する手法を提案した．また，提案手法を Pascal スライスシステム OSS に追加実装し，評価実験を行った．この実験では，プログラムの変更が行われた際の PDG 再計算に要する時間を従来手法の約 1/8〜1/300 に短縮可能であった．提案手法を用いることで，プログラムに変更が加えられた場合に複雑な計算を省略することができ，インタラクティブにプログラムの変更・実行・解析を行うことが可能となる．

2. では，静的解析による制御依存関係と動的解析によるデータ依存関係を利用したスライシング手法を提案した．また，提案手法の有効性を OSS 上で検証した．スライスサイ

ズに関して，提案手法は静的スライスと比較し，約 33〜75% であり，動的スライスと比較し，約 300〜871% であった．実行時間に関しては，静的スライシングの約 102〜108%，動的スライシングの約 0.9〜29% であった．提案手法では，計算時間の短い静的スライシングと正確性の高い動的スライシングの中間的なスライシング手法を実現することで，効率的なデバッグを行うことが可能となっている．

3. では，静的解析と動的解析の中間的な手法である準動的スライシング手法において，複数の文からなるブロックを単位としてスライスを求める手法を提案した．提案手法では，プログラム依存グラフの節点・辺数を少なくすることが可能であるため，スライス計算に要する時間を短縮することができる．提案手法をコンパイラ型言語に適用した実験の結果，提案手法は依存キャッシュスライシングと比べ約 41〜52% の実行時間で計算可能であることを確認した．また，提案手法では，作業者が作業のフェーズに応じて自由にスライス粒度を変更することが可能である．

これらの手法を用いることにより，プログラム依存グラフを用いたプログラムスライシングについて効率的な解析が可能となる．これによってデバッグ作業における作業者の負担を軽減し，デバッグ作業の効率化，ひいてはソフトウェア開発の効率化を行うことができる．

主要論文一覧

1. 高田 智規, 佐藤 慎一, 飯田　元, 井上 克郎: “ソースコード解析ツール開発支援システムの試用”, 電子情報通信学会論文誌 D-I, Vol. J80-D-I, No. 3, pp. 317–318 (1997).
2. 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 D-I, Vol. J81-D-I, No. 3, pp. 253–260 (1998).
3. 高田 智規, 大畑 文明, 芦田 佳行, 井上 克郎: “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 D-I, Vol. J85-D-I, No. 2, pp. 228–235 (2002).
4. 高田 智規, 井上 克郎: “制限された動的情報を用いたブロック単位スライシング手法の提案”, 電子情報通信学会論文誌, (条件付採録).
5. Tomonori Takada, Fumiaki Ohata, and Katsuro Inoue: “Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information”, 10th IEEE International Workshop on Program Comprehension, pp. 169–177, June 2002, Paris, France (2002).

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，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 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，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 谷口 健一 教授，同情報ネットワーク学専攻 今瀬 真 教授に，心から感謝致します.

大阪大学大学院基礎工学研究科情報数理系専攻在席中に，適切なご助言とご指導を頂きました，大阪大学大学院情報科学研究科宮原 秀夫 教授，柏原 敏伸 教授，菊野 亨 教授，萩原 兼一 教授，今井 正治 教授，東野 輝夫 教授，藤原 融 教授，村田 正幸 教授，増澤 利光 教授，松田 秀雄 教授に感謝致します.

本研究を行うにあたり，常日頃より適切なご助言とご指導を賜りました，奈良先端科学技術大学院大学 鳥居 宏次 学長に心から感謝致します.

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，大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本 真二 助教授，松下 誠 助手に心より感謝致します.

本研究に関して，直接具体的な御指導を頂きました，奈良先端科学技術大学院大学情報科学センター 飯田 元 助教授に，心より感謝致します.

本研究を行うにあたり，ご助言やご指導を頂きました，株式会社 NTT データ 佐藤 慎一 氏，株式会社東芝 大畑 文明 氏に感謝致します.

本研究を行うにあたり，御協力を頂いた，NTT ソフトウェア株式会社 芦田 佳行 氏に感謝致します.

最後に，井上研究室の皆様，日本電信電話株式会社 サイバーソリューション研究所の皆様の御助言, 御協力に御礼申し上げます.

目次

第 1 章　まえがき **1**

1.1 はじめに 1

1.2 プログラムスライシング 1

1.2.1 静的スライシング 4

1.2.2 動的スライシング 7

1.2.3 静的スライシングと動的スライシングの特徴 7

1.2.4 準動的解析 9

1.3 本論文の概要 9

第 2 章　プログラム依存グラフの効率的な更新手法 **11**

2.1 導入 11

2.2 PDG 更新における諸定義 12

2.2.1 入力言語 12

2.2.2 到達定義 12

2.2.3 依存関係 12

2.2.4 プログラム依存グラフ 13

2.2.5 前定義節点 13

2.2.6 プログラムの変更 15

2.3 PDG 更新アルゴリズムの概要 15

2.4 単一関数内解析アルゴリズム 16

2.4.1 前定義節点の発見 16

2.4.2 削除 17

2.4.3 挿入 17

2.4.4 修正 18

2.5 複数関数間解析アルゴリズム 19

2.5.1 対象とする PDG 19
2.5.2 関数間解析 22
2.6 PDG 更新アルゴリズムの正当性 22
2.6.1 削除 23
2.6.2 挿入 23
2.6.3 修正 24
2.6.4 関数境界を越える場合 24
2.7 PDG 更新の実行例 24
2.7.1 削除 24
2.7.2 挿入 25
2.8 PDG 更新アルゴリズムの実行効率 27
2.8.1 実装 27
2.8.2 計算量 34
2.8.3 考察 34
2.9 既存手法との比較 35
2.9.1 PDG 計算アルゴリズム 35
2.9.2 既存の更新アルゴリズム 39
2.10 まとめ 42

第 3 章 準動的解析を用いたプログラムスライシング手法 43

3.1 導入 43
3.2 準動的スライシング手法の分類 43
3.2.1 実行経路の収集に着目した手法 44
3.2.2 データ依存関係の収集に着目した手法 44
3.3 依存キャッシュスライシング 45
3.3.1 概要 45
3.3.2 データ依存関係収集アルゴリズム 46
3.4 依存キャッシュスライシングの実行例 48
3.4.1 実行例 1 48
3.4.2 実行例 2 52
3.4.3 実行例に関する考察 54

3.5 依存キャッシュスライシングの評価実験 56
3.5.1 Osaka Slicing System の概要 56
3.5.2 サンプルプログラムの実行 56
3.6 考察 59
3.6.1 実験データの解釈 59
3.6.2 依存キャッシュスライスの適用領域と限界 60
3.6.3 他手法との関連 61
3.7 まとめ 61

第 4 章 準動的解析を用いたブロック単位スライシング手法 63
4.1 導入 63
4.2 ブロック単位スライシング 63
4.2.1 概要 63
4.2.2 ブロック化アルゴリズム 65
4.2.3 データ依存関係収集アルゴリズム 67
4.2.4 ブロック単位スライシングの例 69
4.3 ブロック単位スライシングの拡張 72
4.4 ブロック単位スライシングの評価実験 72
4.4.1 概要 72
4.4.2 考察 74
4.5 まとめ 75

第 5 章 むすび 77
5.1 まとめ 77
5.2 今後の研究方針 77

図 目 次

1.1 サンプルプログラム 3
1.2 PDG の例 4
1.3 静的スライシング基準 (24, d) に対する静的スライス 6
1.4 動的スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対する動的スライス . . 8

2.1 更新の対象とする PDG の例 14
2.2 プログラム proc 20
2.3 プログラム proc の PDG 20
2.4 プログラム max 25
2.5 プログラム max の PDG 26
2.6 削除–Step1 26
2.7 削除–Step2 27
2.8 削除終了 28
2.9 挿入前の PDG 29
2.10 $s4$ を作成した段階 29
2.11 挿入–Step1 30
2.12 挿入–Step2 30
2.13 挿入–Step3 31
2.14 挿入後の PDG 32
2.15 ツール実行画面 33
2.16 手続き定義の概略 36

3.1 配列変数に関するデータ依存 45
3.2 データ依存関係収集アルゴリズム 47
3.3 データ依存関係収集アルゴリズムによって生成される PDG 50

3.4 依存キャッシュスライス計算結果 50
3.5 静的スライス計算時に生成される PDG 51
3.6 サンプルプログラム 2 52
3.7 PDG 初期状態 53
3.8 依存キャッシュスライシングによって生成される PDG 55
3.9 サンプルプログラム 2 の実行履歴 55
3.10 サンプルプログラム 2 に対する動的スライス 56
3.11 スライスサイズ (行) 57
3.12 実行前解析時間 (ms) 58
3.13 実行時解析時間 (ms) 58
3.14 スライス計算時間 (ms) 59

4.1 ブロック化アルゴリズム 65
4.2 ブロック化サンプルプログラム 66
4.3 データ依存関係収集アルゴリズム 68
4.4 サンプルプログラム 70
4.5 スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対するブロック単位スライス ($N = 2$) 71
4.6 スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対するブロック単位スライス (基本ブロック単位でブロック化) 73

表目次

2.1　PDG に対する集合 15
2.2　特殊節点 21
2.3　実行時間（単位は秒） 32
2.4　PDG 再計算・更新にかかわる要素 34
2.5　PDG 更新アルゴリズムの計算量 34

4.1　平均実行時間 (秒) 74

第1章　まえがき

1.1　はじめに

ソフトウェア開発において，デバッグ・テスト・保守フェーズにおける比重はソフトウェアの大規模化に伴い増加している．これらのフェーズにおいてフォールト位置の特定は非常に困難な作業であり，この作業を効率的に行うことが生産性を向上させるために重要である [22, 30].

ソースプログラム全体を見るのではなくフォールトに関連する部分のみ着目することができれば，作業効率を改善することができる．このようなフォールト位置特定を行う手法の一つに，**プログラムスライシング**（*Program Slicing*，以降，スライシングと略す）[33] がある．**プログラムスライス**（*Program Slice*，以降，スライスと略す）とは，直観的には，関心のある文に含まれる変数に影響を与える文の集合を指す．作業者はスライスに含まれる文のみに着目すればよく，デバッグを効率的に行うことができる [16, 47, 48].

1.2　プログラムスライシング

プログラムスライシング技術とは，プログラム中の文間の依存関係に注目し，特定の文と依存関係のある文の集合（プログラムスライス）を抽出する技術である．注目した文に影響を与える文の集合を**バックワードスライス**（*Backward Slice*），注目した文が影響を与える文の集合を**フォワードスライス**（*Forward Slice*）と呼ぶ．一般に，フォワードスライスと比べバックワードスライスが利用されることが多いため，バックワードスライスを単にスライスと呼ぶことも多い．

ソフトウェア開発における様々なフェーズにおいて，スライスは以下のような用途に利用可能である [3, 8, 10, 38, 40, 45].

- **フォールト位置特定**

  スライスを利用することで，作業者の注意をソースプログラム全体ではなく，注目

した文に関連のある部分のみに絞ることができる．これはフォールト位置の特定に非常に有効である [8, 16, 40, 48].

- **プログラム再利用**

  スライスに含まれる部分に注目することで，プログラム全体から特定の機能や部分を抽出することができる．これを 1 つの部品とすることでプログラムの再利用に活用することができる．

- **プログラム理解**

  スライスを用いることにより，プログラム中から余分な文を省き関連のある部分のみを作業者に提示することができる．これによって保守者がプログラムの持つ機能を理解したり，プログラミング初心者が学習を行うといった用途に活用できる [3, 10].

- **プログラム合成**

  複数のプログラムを合成する際に，単純に合成を行うと他のプログラムに影響を与え正常に動作しなくなる可能性がある．スライスを用いてその影響を調べることでこれを避けることができる [40].

- **プログラム編集**

  デバッグ作業において，フォールトの発見や仕様変更などの原因によってプログラムを編集することは頻繁に行われる．しかし，1 つの文の修正が予想しない部分に影響を与え，新たなバグが発生することがある．スライスを用いて，変更によって生じる影響を調べることでこれを回避することができる [40].

- **プログラム簡素化**

  プログラム中から注目した機能にとって不要な部分を削除することでプログラムの簡素化を行うことができる．プログラムが簡素になることで不要なバグの発生要因を減らすとともに，再利用や理解が行いやすくなる [38, 45].

- **プログラム差分解析**

  複数のプログラムに対して差分を求める際に，単純なテキスト差分では変数名の相違など，意味的に同義である部分も差分に含まれてしまう．また，文自体のは同じでも他の文との依存関係意味的には異なっている場合がある．このような差分を求

める際にスライス及びスライスを求める際に求められるプログラム依存グラフを活用することができる [17, 34].

スライシング技術は大きく**静的スライシング**（*Static Slicing*）と**動的スライシング**（*Dynamic Slicing*）の 2 つに分類される．Weiser[33] によって提案された**静的スライス**（*Static Slice*）は，特定のプログラム文中のある変数の値に影響を与える可能性のある文の集合である．Agrawal ら [1, 15] によって提案された**動的スライス**（*Dynamic Slice*）は，注目した文中の変数の値に実際に影響を与えた実行文の集合である．

以降，静的スライシングと動的スライシングに関し，その抽出過程を中心に述べ，その具体例として，図 1.1 のプログラムに対する静的スライス及び動的スライスを求める．

```
1   program Square_Cube(input, output);
2   var a, b, c, d : integer;
3   function Square(x : integer) : integer;
4   begin
5     Square := x * x
6   end;
7   function Cube(x : integer) : integer;
8   begin
9     Cube := x * x * x
10  end;
11  begin
12    writeln("Squared Value ?");
13    readln(a);
14    writeln("Cubed Value ?");
15    readln(b);
16    writeln("Select Feature! Square: 0 Cube: 1");
17    readln(c);
18    if c = 0 then
19      d := Square(a)
20    else
21      d := Cube(b);
22    if d < 0 then
23      d := −1 * d;
24    writeln(d)
25  end.
```

図 1.1: サンプルプログラム

1.2.1 静的スライシング

ソースプログラム p 中の文 s_1 と s_2 について考える．

s_1 が制御文であり，s_1 の結果によって s_2 が実行されるかどうかが決定されるとき，文 s_1 から s_2 に対し**制御依存関係**（*Control Dependence, CD*）があるといい，$s_1 \dashrightarrow s_2$ と記述する．

s_1 から s_2 へ変数 v を再定義しない実行経路が少なくとも 1 つ存在し，s_1 において v が定義され，s_2 において v が参照されるとき，文 s_1 から s_2 に対し変数 v に関する**データ依存関係**（*Data Dependence, DD*）があるといい，$s_1 \xrightarrow{v} s_2$ と記述する．

プログラム依存グラフ（*Program Dependence Graph, PDG*）は辺が文間の依存関係（制御依存・データ依存）を表し，節点が制御文・代入文などの文を表す有向グラフである．また，関数間の依存関係を解析するための特殊節点を持つ．PDG の例を図 1.2 に示す．

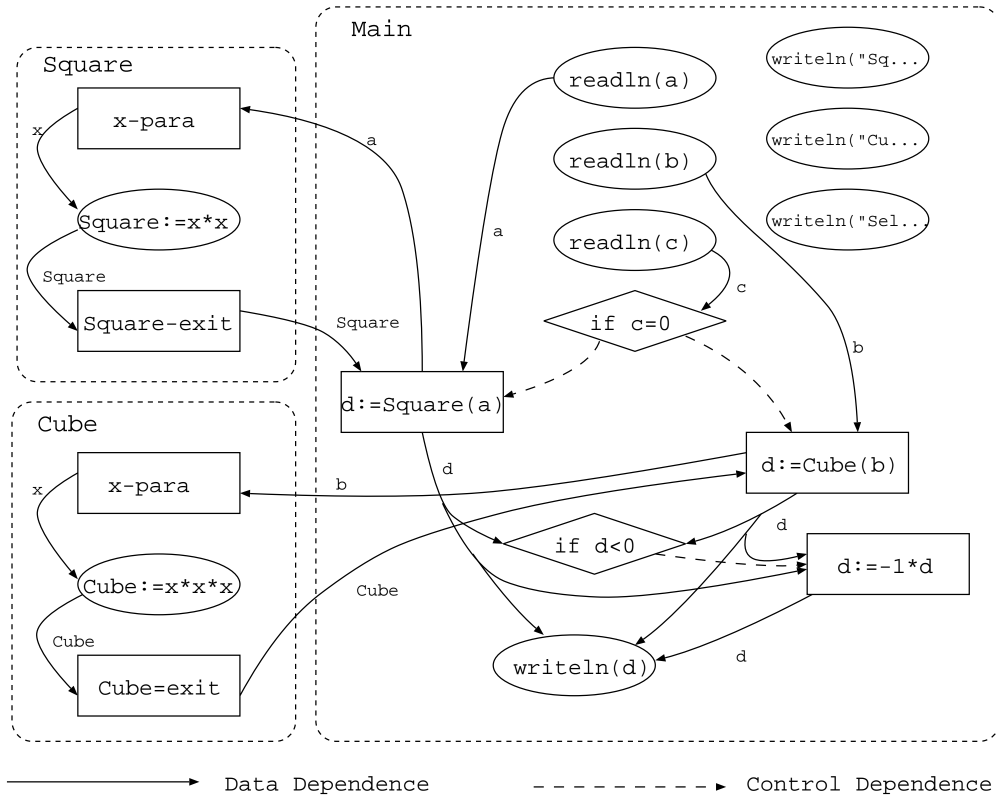


図 1.2: PDG の例

文 s における変数 v に関する静的スライス（静的バックワードスライス）とは，文 s に対応する PDG 節点から PDG 辺を逆向きに（制御依存辺は無条件に，データ依存辺は最

初は着目している変数名に対応するもののみで以降無条件に）辿ることで到達可能な節点集合に対応する文の集合である．なお，組 (s, v) を**静的スライシング基準**（*Static Slicing Criteria*）または単にスライシング基準と呼ぶ.

静的スライシングの解析手法は，解析方針によって，以下のように分類できる.

- **制御フロー解析の有無**（*Flow Sensitiveness / Insensitiveness*）

  ほとんどの静的スライシングアルゴリズムでは，様々なコンパイル最適化技術 [2] と同様，最初にコントロールフローが解析され，次にこの結果がデータフロー解析に用いられる．このようなスライシングアルゴリズムは *flow sensitive* な手法と呼ばれ，一般的に使用される.

  コントロールフローを解析しない手法は *flow insensitive* な手法と呼ばれ，解析オーバヘッドを削減できる．しかし，プログラムの実行順序に関係無く，すべての定義-参照関係についてデータ依存関係が存在すると仮定するため，結果のスライスサイズが増大する.

- **関数・手続き解析時に実際の引数・大域変数情報を用いるかどうか**（*Context Sensitiveness / Insensitiveness*）

  対象ソースプログラム中の複数関数/手続きの解析のために，関数/手続きの呼出文の各インスタンスに対し，実際のパラメータと大域変数の情報を用いて関数/手続きの本体が解析される手法は *context sensitive* 解析と呼ばれる [13]．この手法では，複数の呼出文について，関数/手続きの解析を繰り返さなければならない.

  実際の呼出文から関数/手続きが独立して解析される手法は *context insensitive* 解析と呼ばれる．この手法は context sensitive 手法より単純であるが，解析の正確さは低くなり，context sensitive 手法よりも一般的に大きなスライス結果を返す.

本論文では，特に明示しなければ，制御フロー解析を行い（Context Sensitive）かつ関数・手続き解析時に実際の引数・大域変数情報を用いない（Flow Insensitive）手法を採用する.

図 1.1 のサンプルプログラムに対する，静的スライシング基準 $(24, d)$ に関する静的スライスを図 1.3 に示す.

```
1   program Square_Cube(input, output);
2   var a, b, c, d : integer;
3   function Square(x : integer) : integer;
4   begin
5     Square := x * x
6   end;
7   function Cube(x : integer) : integer;
8   begin
9     Cube := x * x * x
10  end;
11  begin
12
13    readln(a);
14
15    readln(b);
16
17    readln(c);
18    if c = 0 then
19      d := Square(a)
20    else
21      d := Cube(b);
22    if d < 0 then
23      d := −1 * d;
24    writeln(d)
25  end.
```

図 1.3: 静的スライシング基準 (24, d) に対する静的スライス

1.2.2　動的スライシング

静的スライシングではソースプログラムを対象に依存関係を解析し，スライスを抽出していたが，動的スライシングでは，依存関係の抽出対象は**実行系列**（*Execution Trace*）になる．実行系列とは，ある入力を与えプログラムを実行したときの，実際に実行された文の列をいう．また，実行系列中の r 番目の文の実行のことを実行時点 r と呼ぶ.

実行系列 e 中の実行時点 r_1，r_2 について考える．

r_1 が制御文であり，r_1 の結果によって r_2 が実行されるかどうかが決定されるとき，実行時点 r_1 から r_2 に対し**動的制御依存関係**（*Dynamic Control Dependence, DCD*）があるという．

r_1 から r_2 へ変数 v を再定義する実行時点がなく，r_1 において v が定義され，r_2 において v が参照されるとき，実行時点 r_1 から r_2 に対し変数 v に関する**動的データ依存関係**（*Dynamic Data Dependence, DDD*）があるという．

そして，これらの動的依存関係を基に**動的依存グラフ**（*Dynamic Dependence Graph, DDG*）を構築する．

入力値の組 $\mathcal{X}$ が与えられたときの実行時点 r における変数 v に関する動的スライスとは，実行時点 r に対応する DDG 節点から DDG 辺を逆向きに辿ることで到達可能な節点集合に対応する文の集合である．なお，組 $(\mathcal{X}, r, v)$ を**動的スライシング基準**（*Dynamic Slicing Criteria*）と呼ぶ.

図 1.1 のサンプルプログラムに対する，動的スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に関する動的スライスを図 1.4 に示す.

1.2.3　静的スライシングと動的スライシングの特徴

静的スライシングはプログラムの実行を伴わず，入力データを必要としないため，一般に小さいオーバヘッド（解析時間・記憶領域など）で解析可能である．これにより，大量のプログラムを解析する場合や入力値に依存しない解析を行う場合に有用である．しかし，静的スライスは一般にスライスサイズが大きく，極端な場合，ソースプログラム全体がスライスとして抽出される．これは静的スライシングが，すべての入力データ，つまりすべての実行経路パターンを想定しているためである．また，静的スライシングにおいて，変数名の**別名**（*Aliasing*）の判定，配列及び構造体要素の区別，ポインタ変数の参照先の追

```
1     program Square_Cube(input, output);
2     var a, b, c, d : integer;
3     function Square(x : integer) : integer;
4     begin
5        Square := x * x
6     end;
7
8
9
10
11    begin
12
13       readln(a);
14
15
16
17       readln(c);
18       if c = 0 then
19          d := Square(a)
20
21
22
23
24       writeln(d)
25    end.
```

図 1.4: 動的スライシング基準 ({a = 2, b = 3, c = 0}, 24, d) に対する動的スライス

跡などの解析は容易ではない．さらにオブジェクト指向プログラムでは，識別子に対応するメソッドがプログラム実行時に決定される点も無視できない．

一方，動的スライスは特定の入力データに基づいて導出されるため，ソースプログラムのうち実行されなかった部分は自動的に除かれる．このため，スライスサイズは静的スライスと比べ一般的に小さくなり，フォールト位置特定には好ましい．しかし，動的スライシングは動的にプログラム文間の依存関係を追跡する必要があるため，一般に実行時のオーバヘッドが非常に大きくなる．プログラムの規模や入力値によっては，現実のデバッグ作業にとって現実的でないオーバヘッドとなる場合も少なくない．

1.2.4 準動的解析

1.2.3 で述べたように，動的スライシングは静的スライシングと比べ正確な（サイズの小さな）スライスを抽出することができるため，デバッグ作業において有効である．しかし，実行時及び解析時に非常に大きなオーバヘッドを必要とする．そこで，動的スライシングの実行時オーバヘッドを削減することが必要とされている．

静的解析の軽量さと動的解析の正確さというそれぞれの長所を組み合わせることによって，解析の正確性と実行時オーバヘッドのトレードオフを実現することができる．このような手法を本論文では，**準動的解析**（*Semi-Dynamic Analysis*）と呼ぶ．

1.3 本論文の概要

本論文では，プログラム依存グラフを用いたプログラムスライスの抽出に基づく，(1) プログラム依存グラフのインタラクティブな更新方法の提案，(2) 静的解析情報と動的解析情報を組み合わせた準動的解析方法の提案，(3) 準動的解析の効率化，について行う．

(1) プログラム依存グラフの効率的な更新手法

ソースプログラムに対して文の追加・削除・修正といった変更が行われた場合に，プログラム依存グラフのうち変更個所から影響を受ける部分のみを更新する手法を提案した．提案手法を用いることで，複雑な PDG 再計算を省略することができ，インタラクティブにプログラムの変更や実行を行い，効率的なデバッグを実現できる．また，提案手法を Pascal スライスシステム OSS に追加実装し，その有効性を検証するとともに，計算量に関する考察を行った．

(2) 準動的情報を用いたプログラムスライシング手法

静的解析による制御依存関係と動的解析によるデータ依存関係を利用したスライシング手法を提案した．提案手法では，計算時間の短い静的スライシングと正確性の高い動的スライシングの中間的なスライシング手法を実現することで，効率的なデバッグを行うことが可能となっている．提案手法の有効性を OSS 上で検証するとともに，コンパイラ型言語に適用した場合の考察を行った．

(3) 準動的解析を用いたブロック単位スライシング手法

静的解析と動的解析の中間的な手法である準動的スライシング手法において，複数の文からなるブロック単位でスライスを求める手法を提案した．提案手法では，プログラム依存グラフの構築及びスライスの抽出の対象を少なくすることができ，スライス計算に要する時間を短縮することが可能である．また，作業者が作業のフェーズに応じて自由にスライス粒度を変更することが可能である．コンパイラ型言語に提案手法を適用した場合の有効性を確認した．

以下，2 章では，プログラム依存グラフの効率的な更新手法について述べる．3 章では準動的解析手法を用いたプログラムスライシング手法について，4 章では準動的解析手法をさらに効率化させたブロック単位スライシング手法について述べる．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効率的な更新手法

2.1　導入

プログラム依存グラフ（Program Dependence Graph，PDG）[13, 14, 35] は，プログラムの各文における変数間の依存関係を表す有向グラフである．PDG の各節点はプログラム中の各文・条件式を表し，有向辺は依存関係（データ依存・制御依存）を表す．

PDG の有効辺を辿ることにより，プログラムスライス [6, 7, 8, 13, 19, 29, 40, 45, 46] と呼ばれる，プログラム中のある文に影響を受ける，または影響を与える文の集合を抽出することができる．プログラムスライスはデバッグやテスト，保守，プログラム合成などに利用されている．

これまでに，プログラムスライスを抽出しデバッグを効率良く進めるためのシステム [39] を開発した．このシステムは抽出したプログラムスライスを対象としてデバッグを行う機能を持っている．このシステムではプログラムに変更を加えるたびに PDG 全体を変更されたソースプログラムの全体から再計算していたが（これをここでは **PDG 再計算**と呼ぶ），PDG 再計算は複雑であり，それに多くの時間を費していた．これはインタラクティブにプログラムの修正や実行を行い，デバッグを効率良く行う際の大きな障害となっていた．

PDG のうちプログラムの変更箇所に関する部分だけを更新することができれば，PDG 再計算に要していた時間が短縮され，作業効率の大幅な向上が期待できる．プログラム内の部分的な変更がその他の文に与える影響を調べるアルゴリズムは既に提案されている [27, 37]．しかし，これらのアルゴリズムは関数内部の依存関係の変化を求めることはできるが，関数境界を越えるような変更の際に正しく依存関係を求められない．また，文献 [27] のアルゴリズムは独自のグラフを用いているため，PDG に適用することができず，文献 [37] のアルゴリズムでは保持する必要のある情報が非常に多くなるという問題がある．

そこで，プログラムの部分的変更が行われた時に関数境界を越える依存関係を正しく計

算し，効率良く PDG を再計算する手法を提案する．また，前述のデバッグ支援システムに提案する手法を実装し，PDG の再計算と本手法との実行時間を比較し，その有効性を確認する．

以降，2.2 では諸定義を行い，2.3 で提案アルゴリズムの概要，2.4 で関数内解析アルゴリズムの詳細，2.5 で関数境界を越える場合のアルゴリズムの詳細を述べる．次に，2.7 では実行例を示し提案アルゴリズムの動作を説明する．また，2.8 で実行効率に関する考察を行う．

2.2 PDG 更新における諸定義

2.2.1 入力言語

本研究では以下のような言語を入力言語として考える．この言語には文として条件文（if 文），代入文，繰返し文（while 文），入力文（readln 文），出力文（writeln 文），手続き呼出し文，複合文（begin–end）がある．変数の型としてはスカラ型のみを考える．プログラムは，大域変数宣言，手続き（関数）定義，メインプログラムからなり，ブロック構造はない．手続き内では内部で宣言された局所変数と仮引数変数及び大域変数のみが参照可能で，他の手続き内の局所変数は参照できない．手続きは，自己再帰的及び相互再帰的に定義可能であり，その引数は，値渡しで扱われる．

2.2.2 到達定義

到達定義とは，ある文における変数の値に影響を与える可能性のある文と変数の組を表す．

文 s における変数 x（の値）の定義が文 t に到達するとは，文 s が変数 x を定義し，かつ，s から t に至る実行パス $\underline{s, u_1, u_2, \cdots, u_k, t}$ 中に変数 x を再定義しないような実行パス $\underline{u_1, u_2, \cdots, u_k}$ が存在する場合を言う．

文 n における変数 v の定義が文 s に到達する場合，文 s には到達定義 $\langle n, v \rangle$ が存在するという．到達定義集合とは，文 s に到達するすべての到達定義からなる集合である．

2.2.3 依存関係

プログラム中の文間の依存関係として以下の二種類を考える．

1. **データ依存関係（Data Dependence）**

   変数 v を参照している文 t において到達定義集合中に到達定義 $\langle s, v \rangle$ が存在するとき，文 s から文 t に対して変数 v に関するデータ依存関係があるという．

2. **制御依存関係（Control Dependence）**

   文 s が条件文または繰り返し文の条件式であり，文 s の条件判定の結果によって文 t を実行するか否かが直接決まる時，文 s から文 t への制御依存関係があるという．

2.2.4　プログラム依存グラフ

プログラム依存グラフ（Program Dependence Graph，PDG）は，プログラム内の各文間に存在する依存関係を有向グラフで表したものである．本研究では，文献 [35] のアルゴリズムを用いて求めた PDG を対象とする．

節点として，プログラム内の文または条件式（条件文・繰り返し文の条件判定部分）に対応した節点（文節点）及び，手続き境界を越える依存関係の解析などに用いる特殊節点がある．

辺として，データ依存関係を表すものをデータ依存辺といい，制御依存関係を表すものを制御依存辺という．以下，節点（文）s から節点 t への変数 v に関するデータ依存辺を $s \xrightarrow{v} t$ ，節点 s から節点 t への制御依存辺を $s \dashrightarrow t$ と表す．

これらの依存関係を表す辺のほかに，フロー辺と呼ばれる実行順序を表す辺がある[1]．文 s から文 t に（他の文の実行無しに）直接制御が移る可能性がある時，節点 s から節点 t へのフロー辺が存在し， $s \rightarrow t$ と表す．

パス $\underline{s, \cdots, t}$ が存在するとは，節点 s からフロー辺を順方向にたどり節点 t へ到達するような経路が存在することをいう．

図 2.1 に PDG の例を示す，図中の DD はデータ依存辺，CD は制御依存辺，flow はフロー辺をそれぞれ表す．また，PDG に対して，表 2.1 で示す集合を定義する．

2.2.5　前定義節点

節点 r における変数 v の定義が節点 s の入口に到達している場合，節点 r を，節点 s の変数 v に関する前定義節点と呼ぶ．また，前定義節点の集合を前定義節点集合と呼ぶ．例

[1] フロー辺を PDG に含めない場合が多いが，ここではこのグラフを PDG と呼ぶ

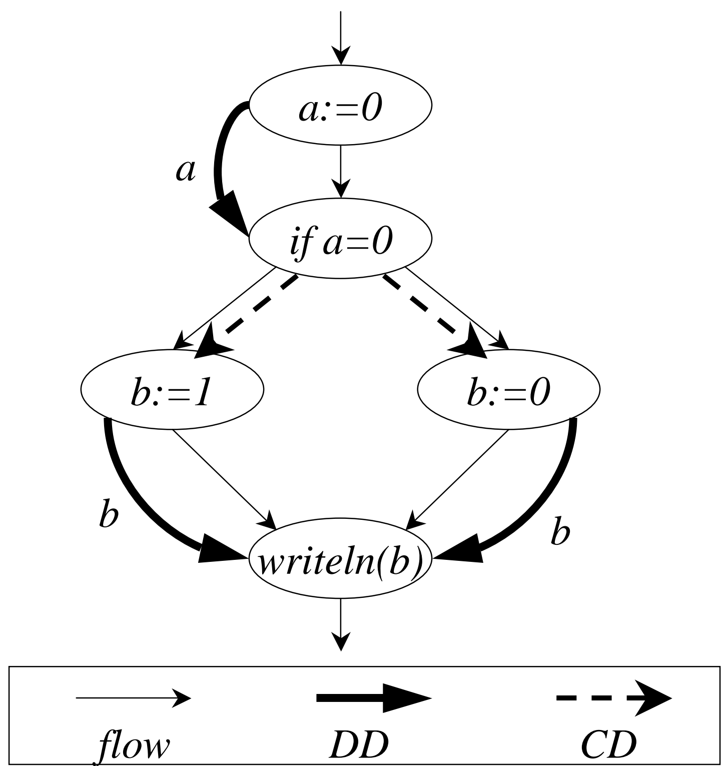


図 2.1: 更新の対象とする PDG の例

表 2.1: PDG に対する集合

$source(c, v)$	節点 c から変数 v に関するデータ依存辺を遡って到達できる（有向辺を逆方向にたどった）節点の集合を表す（c 自身を含めない．以下同様）．
$target(c, v)$	節点 c から変数 v に関するデータ依存辺を辿って到達できる（有向辺を順方向にたどった）節点の集合を表す.
$prev(c)$	節点 c からフロー辺を遡って到達できる節点の集合を表す.
$next(c)$	節点 c からフロー辺を辿って到達できる節点の集合を表す.
DD	データ依存辺の集合.
CD	制御依存辺の集合.
$FLOW$	フロー辺の集合.

えば，図 2.1 において節点 $s5$ の変数 b に関する前定義節点集合は $\{s3, s4\}$ となる．

2.2.6　プログラムの変更

プログラムの変更として PDG の文節点一つに対し，以下の 3 種類の操作を考える．

- 文の削除

  PDG 上に存在する節点を削除する

- 文の挿入

  新たに節点を作り PDG に挿入する

- 文の修正

  節点はそのまま保存するが，その内容を修正する

2.3　PDG 更新アルゴリズムの概要

プログラムの変更による制御依存辺の変化は，変更される文の制御構造を調べることにより求めることができる．例えば，条件文の節中に文を挿入した場合，条件文から挿入文に対して制御依存辺を追加する．逆に，条件文の節中の文を削除した場合，制御依存辺を

削除する．このように制御依存辺の変化は，変更前の制御依存辺から容易に求めることができる．そこで，以降はデータ依存辺の変化について述べる．

文献 [37] では，プログラムの変更に伴う到達定義集合の変化をフロー辺を辿り，後の節点に伝えていくことにより，変更によって変化した依存関係を再計算していた．しかし，この方法では各節点ごとに計算した到達定義集合を保存しておかなければならず，実装する際に非常に多くのメモリを必要としていた[2]．

そこで，ここでは次のような工夫をした．PDG のデータ依存辺は解析時に到達定義集合を基にして引かれている．よって，変更前の PDG 中のデータ依存辺から，到達定義集合の一部を知ることができる．例えば，ある節点 s において $r \xrightarrow{v} s$ があったとき，変更前の s における到達定義集合の中に $\langle r, v \rangle$ が存在していたことになる．本手法ではプログラムが変更されたときに，到達定義集合を保存しておかなくても，このようなデータ依存辺の情報を利用し，依存関係の変化を求め，PDG の更新を行うことができる．

しかし，これだけでは関数境界を越える依存関係を正しく解析することはできない．そこで，関数内解析と関数間解析の二種類のアルゴリズムを考える．関数内解析アルゴリズムでは上記の手法により変更された文が含まれる関数内部のデータ依存辺を引き直す．関数間解析アルゴリズムでは変更による他の関数への影響を調べ，関数境界を越える依存関係を解析する．

2.4 単一関数内解析アルゴリズム

まず，文の削除・挿入・修正それぞれに共通な前定義節点の発見のためのアルゴリズムを示し，次に削除・挿入・修正のアルゴリズムを示す．

2.4.1 前定義節点の発見

このアルゴリズムではフロー辺を遡り，節点での定義・参照変数を調べることにより前定義節点を求める．このアルゴリズムは 2.4.2 節〜2.4.4 節において用いる．

アルゴリズム FindPreDef

input: 節点 s，変数 v

output: 前定義節点集合 $PreDef(s, v)$

---

[2]通常，到達定義集合はデータ依存辺を引いた後は必要無く，保存する必要は無い．

1. $PreDef(s, v) \leftarrow \phi$

2. $c \in prev(s)$ なる各 c に対して以下を順に実行.

   (a) c において v が定義されていれば,

   $$PreDef(s, v) \leftarrow PreDef(s, v) \cup \{c\}$$

   (b) c において v が参照されていれば,

   $$PreDef(s, v) \leftarrow PreDef(s, v) \ \cup \ source(c, v)$$

   (c) c において v が定義も参照もされていなければ，$c \leftarrow prev(c)$ として 2a. 以下を実行.

2.4.2　削除

節点を削除することにより，その文における変数の定義が無効となる．そのような変数に関するデータ依存辺を引き直し，その後節点を削除する．

アルゴリズム DELETEVERTEX

input:　削除する節点 s

output:　更新された PDG

1. s で定義されている各変数 v すべてについて，$DD \leftarrow DD \cup \{ r \xrightarrow{v} t \mid r \in PreDef(s, v), \ t \in target(s, v)\}$

2. $DD \leftarrow DD - \{ r \xrightarrow{v} s \mid r \in source(s, v)\} - \{ s \xrightarrow{v} t \mid t \in target(s, v)\}$

3. $FLOW \leftarrow FLOW \ \cup \ \{ p \rightarrow n \mid p \in prev(s) \ , n \in next(s)\} - \{ r \rightarrow s \mid r \in prev(s)\} - \{ s \rightarrow t \mid t \in next(s)\}$

4. 節点 s 自身を削除

2.4.3　挿入

このアルゴリズムではまず，フロー辺をつけかえる．次に，挿入する節点で定義する変数を参照する節点へデータ依存辺を引く．また，挿入によって依存関係の無くなるデータ依存辺を削除する．最後に，挿入する節点で参照する変数のデータ依存辺を引く．

アルゴリズム InsertVertex

input:　挿入する節点 s，$prev(s)$，$next(s)$

output:　更新された PDG

但し，手続き・関数・プログラムの入口においてすべての変数を定義したものとみなす.

1. $FLOW \leftarrow FLOW \cup \{ p \rightarrow s \mid p \in prev(s)\} \cup \{ s \rightarrow n \mid n \in next(s)\} - \{ p \rightarrow n \mid p \in prev(s)\ ,\ n \in next(s)\}$

2. s で定義している各変数 v すべてについて，以下を実行.

   (a) 各 $r \in PreDef(s,v), t \in target(r,v)$ に対して，v を定義しないようなパス $\underline{s,\cdots,t}$ が存在すれば，以下を実行.

      i. $DD \leftarrow DD \cup \{ s \xrightarrow{v} t \}$

      ii. 任意の $r \in PreDef(s,v)$ に対して v を定義しないようなパス $\underline{r,\cdots,t}$ が存在しなければ，

      $$DD \leftarrow DD - \{ r \xrightarrow{v} t \mid r \in PreDef(s,v)\}$$

3. s で参照している各変数 u について，

$$DD \leftarrow DD \cup \{ r \xrightarrow{u} s \mid r \in PreDef(s,u)\}$$

2.4.4 修正

このアルゴリズムは，削除アルゴリズムと挿入アルゴリズムを組み合わせることによって実現している.

変数集合 V を定義し変数集合 U を参照する節点 s を，変数集合 V' を定義し変数集合 U' を参照するように修正したとする.

アルゴリズム ModifyVertex

input:　節点 s，変更後の定義変数集合 V'，変更後の参照変数集合 U'

output:　更新された PDG

1. $FLOW \leftarrow FLOW$

2. $v \in V - V'$ に対して，

$DD \leftarrow DD \cup \{\ r \xrightarrow{v} t\ \mid r \in PreDef(s,v), t \in target(s,v)\} - \{\ s \xrightarrow{v} t\ \mid t \in target(s,v)\}$

3. $v' \in V' - V$ に対して，

   (a) 各 $r \in PreDef(s,v'), t \in target(r,v')$ に対して，v を定義しないようなパス $\underline{s,\cdots,t}$ が存在すれば以下を実行．

      i. $DD \leftarrow DD \cup \{\ s \xrightarrow{v'} t\ \}$

      ii. 任意の $r \in PreDef(s,v')$ に対して，v' を定義しないようなパス $\underline{r,\cdots,t}$ が存在しなければ，

$$DD \leftarrow DD - \{\ r \xrightarrow{v'} t\ \mid t \in PreDef(s,v)\}$$

4. $u \in U - U'$ に対して，

$$DD \leftarrow DD - \{\ r \xrightarrow{u} s\ \mid r \in PreDef(s,u)\}$$

5. $u' \in U' - U$ に対して，

$$DD \leftarrow DD \cup \{\ r \xrightarrow{u'} s\ \mid r \in PreDef(s,u')\}$$

2.5 複数関数間解析アルゴリズム

2.5.1 対象とする PDG

図 2.2 のプログラム `proc` は手続き inc の中で大域変数 a を参照・定義している．このプログラムの PDG は図 2.3 のようになる．関数（手続き）内部で関数外での大域変数を参照している場合は global-in と呼ばれる特殊節点を関数の入口に作り，関数内部で大域変数を定義している場合は global-out と呼ばれる特殊節点を関数の出口に作る．関数境界を越える依存関係がある場合はこれらの特殊節点を介してデータ依存辺を引く．

このような特殊節点を表 2.2 に示す．

関数（手続き）中の文に対して変更を行った場合，その関数内で参照・定義している変数が変更前と異なる場合がある．このような場合は既存の特殊節点の削除や新たな特殊節点の挿入が必要になる．

このような変更に対応するため，文献 [35] で PDG を計算する際に用いている確実定義集合，潜在定義集合，暗使用される変数集合を考える．

```
program proc(input,output);
var a: integer;


procedure inc;
begin
  a:=a+1
end;


begin
  readln(a);
  inc;
  writeln(a)
end.
```

図 2.2: プログラム proc

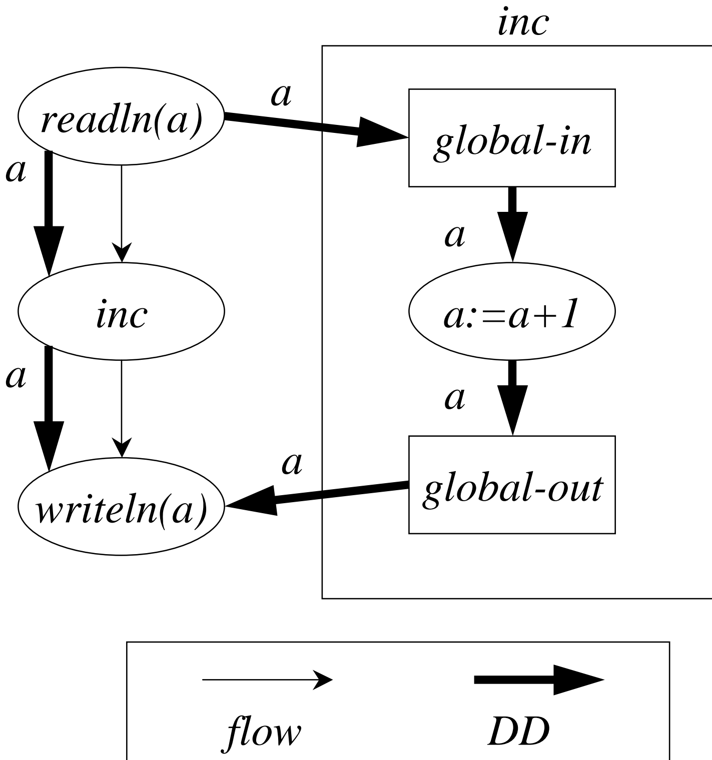


図 2.3: プログラム proc の PDG

表 2.2: 特殊節点

	特殊節点	表記例
entry 節点	プログラム及び手続き (関数) にひとつずつあり，そのプログラム (関数・手続き) の中のすべての文は entry 節点からの CD 関係がある	$f\text{–}Entry$
exit 節点	関数の戻り値を通して伝わる影響を検出するための節点で,各関数にひとつずつある	$f\text{–}exit$
global-in 節点	手続き外からの大域変数の影響を内部へ伝えるための節点で, 手続きに, 個々の大域変数に対して, ひとつずつある	$f_g\text{–}in$
global-out 節点	手続き内で定義された大域変数の影響をその外へ伝えるための節点で, 手続きに, 個々の大域変数に対して，ひとつずつある	$f_g\text{–}out$
paramter-in 節点	手続きの引数を通して伝わる影響を検出するための節点で,その引数それぞれにひとつずつある	$f_p\text{–}par$
parameter-out 節点	手続きの変数引数を通して伝わる影響を検出するための節点で，変数引数それぞれにひとつずつある	$f_p\text{–}par_{out}$

プログラムのある領域 S について，S を実行したときに必ずその値が定義される変数とその定義節点との組の集合を確実定義集合と呼び，$SuDEF(S)$ と表す．また，定義される可能性のある変数とその定義節点との組の集合を潜在定義集合と呼び，$PoDEF(S)$ と表す.

また，次に示す条件をすべて満たす変数 v を「手続き p で暗使用される変数」と呼び,その集合を $ImUSE(p)$ と表す.

- v は大域変数である
- p 内での v の参照地点に p 外の v の定義が到達する

2.5.2 関数間解析

関数（手続き）f 中の文 s に対して変更を行うことを考える．

1. s に関数内変更アルゴリズムを行う．
2. f を直接，または間接に呼び出す関数及び f 自身の集合を F とする．
3. $\forall g \in F$ に対して，

$$\begin{aligned} SuDEF(g) &\leftarrow \phi \\ PoDEF(g) &\leftarrow \phi \\ ImUSE(g) &\leftarrow \phi \end{aligned}$$

4. F 中各関数の $SuDEF, PoDEF, ImUSE$ を求める．
5. 4. を F 中のすべての関数の $SuDEF, PoDEF, ImUSE$ が変化しなくなるまで繰り返す．
6. 関数ごとに変更前後の $SuDEF, PoDEF, ImUSE$ を比較し，異なっていれば以下を実行する．
   (a) 新たに定義・参照するようになった変数に関する global-out, global-in 節点を作り，関数内部のデータ依存辺を引く．
   (b) 関数呼出し文に対して アルゴリズム ModifyVertex を用いてデータ依存辺を引き直す．その際，関数内で参照・定義する変数を呼び出し文で参照・定義する変数として用いる．
   (c) 呼び出し文におけるデータ依存辺を global-in, global-out でのデータ依存辺とする．

2.6 PDG 更新アルゴリズムの正当性

本アルゴリズムによってデータ依存辺が PDG 計算のアルゴリズムと同様に引かれることを示す．

2.6.1　削除

削除する節点 s で変数を定義していない場合，削除によって s 以降の文に与える影響はない．この場合，データ依存辺を新たに追加する必要は無く，s と関連する辺を削除すればよい．

削除する節点 s で変数 v を定義し，$RD_{in}(s)$ 中に到達定義 $\langle r, v\rangle$ が存在したとする．この場合，削除前の $RD_{out}(s)$ 中には $\langle r, v\rangle$ は存在せず，$\langle s, v\rangle$ が存在する．

s が無ければ，s 以降，v が再び定義されるまで到達定義集合中には $\langle s, v\rangle$ は存在せず，$\langle r, v\rangle$ が存在する．そのため，v を参照する節点 t でのデータ依存辺は $s \xrightarrow{v} t$ の代わりに $r \xrightarrow{v} t$ が存在する．

アルゴリズム DeleteVertex では $r \in PreDef(s, v)$, $t \in target(s, v)$ に対し $r \xrightarrow{v} t$ を追加する．ここで r, t は上記の節点 r, t に相当する，また，s に関するデータ依存辺を削除するため，$s \xrightarrow{v} t$ の代わりに $r \xrightarrow{v} t$ が新たに引き直されている．

2.6.2　挿入

挿入する節点 s で変数 v を定義し，変数 u を参照していたとする．

変数 v に関して，挿入アルゴリズムは削除アルゴリズムの逆を行う，つまり $r \xrightarrow{v} t$ を $s \xrightarrow{v} t$ にすることで PDG の更新が行える．但し，挿入を行う段階で s に関するデータ依存辺が存在しないため，到達定義集合に関する情報を復元するのにいくつかの問題がある．

一つめの問題は，変数 v に関する前定義節点 r，参照する節点 t としたときに $r..t..s$ という実行順序となる時である．この時，s が t 以前に実行されない可能性がある．

二つめの問題は，s の挿入によって r での v の定義が t に到達するかどうかがわからないことである．

これらを解決するため，アルゴリズム InsertVertex ではパス $\underline{s, \cdots, t}$ が存在する場合だけ処理を行う．また，すべてのパス $\underline{r, \cdots, t}$ に対して，パス中に v を定義する文が存在しているならば r での v の定義は t に到達しないとして，$r \xrightarrow{v} t$ を削除している．これにより，すべての場合において PDG 全体を再計算するのと同じ PDG が計算できる．

2.6.3　修正

文の修正アルゴリズムは削除・挿入アルゴリズムを組合せたものである． アルゴリズム ModifyVertex は変更によって定義・参照されなくなった変数に対して削除アルゴリズムを，新たに定義・参照されるようになった変数に対して挿入アルゴリズムを適用している．

2.6.4　関数境界を越える場合

まず，2.4.2 節〜2.4.4 節のアルゴリズムによって関数内部の解析を行う．次に，変更による他の関数への影響を調べる．

変更によって関数内部で定義・参照する変数が変化する可能性があり，これらが変化すればこの関数の呼び出し文で参照・定義する変数が異なってくる．そこで，変更を受けた関数を直接または間接に呼び出す可能性のある関数の集合に対して，$SuDEF$, $PoDEF$, $ImUSE$ を求め，これらの値が収束するまで計算を行う．この方法は PDG 計算アルゴリズムの関数間解析と同じ方法であるため，これによって PDG 全体を再計算するのと同じ PDG が計算できる．

2.7　PDG 更新の実行例

2.7.1　削除

2.4.2 節で示した アルゴリズム DeleteVertex を用いて図 2.4 のプログラム中の文 $s4$ を削除する様子を示す.

プログラム max の PDG は図 2.5 の様になる．灰色の網掛けで示した節点が $s4$ でこの節点を削除することを考える．

まず， アルゴリズム DeleteVertex の 1. を実行するために $PreDef(s4, max)$ 及び $target(s4, max)$ を求める．$PreDef(s4, max) = \{s2\}$, $target(s4, v) = \{s6\}$ となるので，データ依存辺 $s2 \xrightarrow{max} s6$ を追加する（図 2.6）.

2. 以降，順に，$s4$ に関連するデータ依存辺の削除，$s4$ に関連するフロー辺の削除，フロー辺 $s3 \rightarrow s6$ の追加，及び $s4$ 自身の削除が行われる．図 2.8 の薄く示された辺・節点が削除されたものを表し，図 2.7 にフロー辺が追加されたものを表す.

以上から，文 $s4$ を削除することにより， $s2 \xrightarrow{max} s6$ が新たに追加され，依存関係が正

しく解析されていることがわかる．

```
   program max(input,output);
   var x, y, max : integer;
   begin
s1   readln(x,y);
s2   max := x;
s3   if x > y then

s4      max := x

     else
s5      max := y;
s6   writeln(max)
   end.
```

図 2.4: プログラム max

2.7.2 挿入

2.4.3 節で示した アルゴリズム InsertVertex を用いて，2.7.1 節で削除した文 $s4$ を挿入する様子を示す.

挿入前の PDG は図 2.9 の様になる．まず，挿入する文 $s4$ を作成し，制御依存辺を引く（図 2.10）.

アルゴリズム InsertVertex の 1. により， $s3 \rightarrow s4$ ，$s4 \rightarrow s6$ を追加し， $s3 \rightarrow s6$ を削除する（図 2.11）．但し，$prev(s4) = \{s3\}$ 及び $next(s4) = \{s6\}$ は $s4$ を作成した時点で与えられるものとする.

2a. を実行するために $PreDef(s4)$ を求めると，$PreDef(s4, max) = \{s2\}$ となる．そして，$target(s2, max) = \{s6\}$ となる．ここで，パス $\underline{s4, \cdots, s6}$ が存在するので 2(a)i.,2(a)ii. を実行する.

2(a)i. では $s4 \xrightarrow{max} s6$ を追加する（図 2.12．次に，2(a)ii. で $s2$ から $s6$ へのパスを調べる．このようなパスは $\underline{s2, s3, s4, s6}$, $\underline{s2, s3, s5, s6}$ の二つ存在する．この内，パス中で max を定義しないパスが存在しない（$s4, s5$ で max を定義）ので， $s2 \xrightarrow{max} s6$ を削除する（図 2.13）.

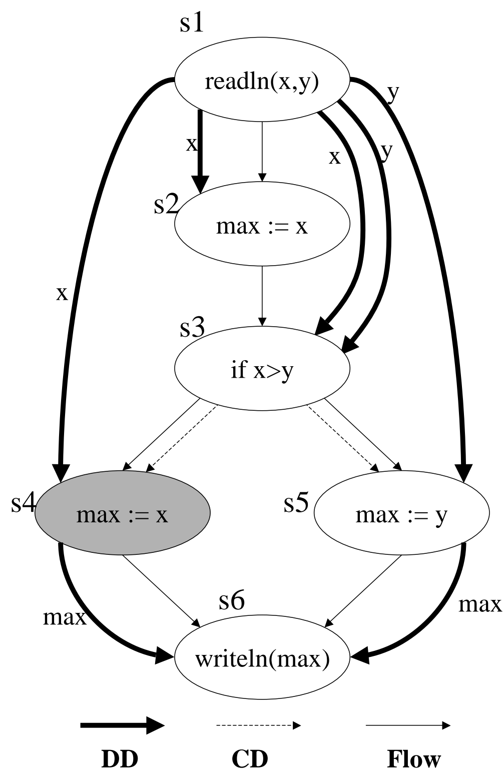


図 2.5: プログラム max の PDG

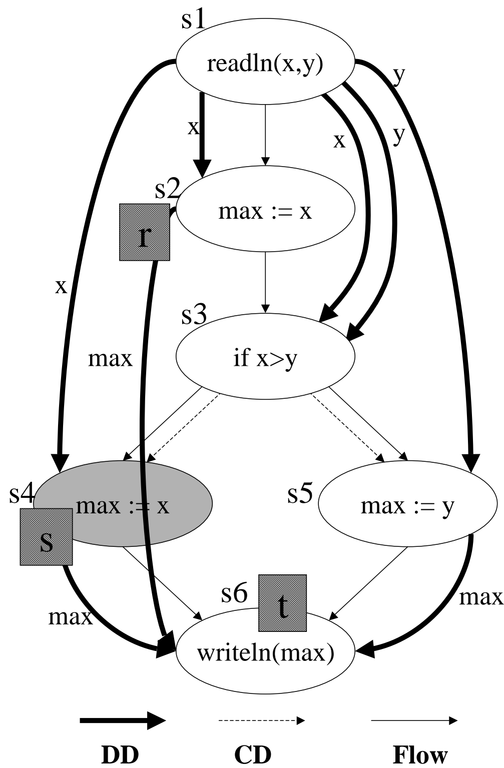


図 2.6: 削除–Step1

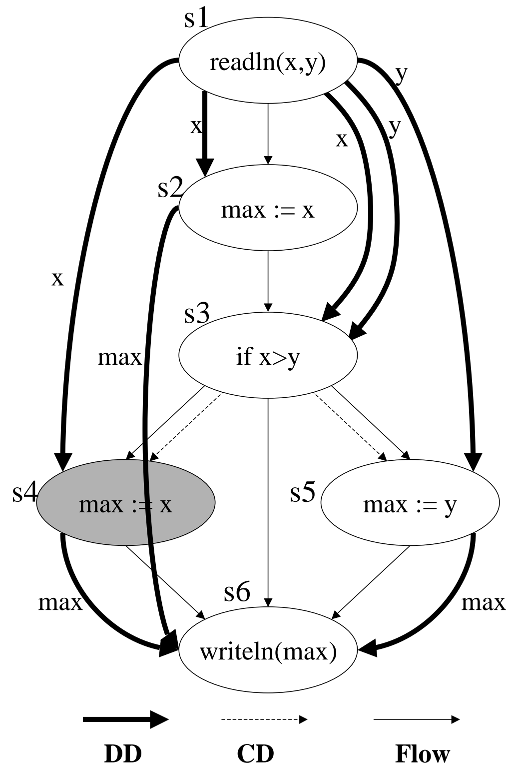


図 2.7: 削除–Step2

最後に 3. で，参照している変数 x について $PreDef(s4, x)$ を求め，$PreDef(s4, x) = \{s1\}$ となるので $s1 \xrightarrow{x} s4$ を追加する（図 2.14）.

これは，削除以前の PDG（図 2.5）と同じグラフとなり，正しく挿入が行われたことがわかる.

2.8　PDG 更新アルゴリズムの実行効率

2.8.1　実装

本アルゴリズムのうち，削除・挿入・修正のアルゴリズムを Osaka Slicing System(OSS) と呼ぶプログラムデバッグシステム文献 [39] に組み込んだ．ソースコードは C で記述し，ユーザインタフェースには Tcl/Tk を用いた．本アルゴリズムを組み込むにあたり，フロー辺の追加や仕様変更，機能拡張などを行った．削除アルゴリズムは約 1060 行，挿入アルゴリズムは約 570 行である．また，システム全体では約 13500 行である．

図 2.15 に OSS の実行画面を付す．ツールの左側のウィンドウはソースコード，右上側はツールの状態，右下側は実行画面をそれぞれ表示する．

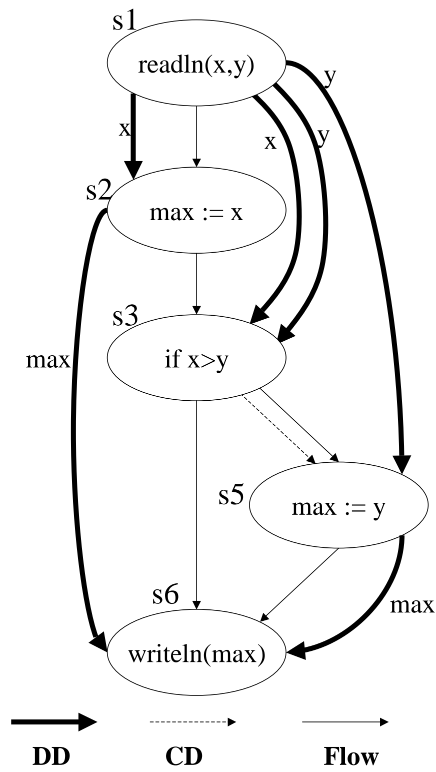


図 2.8: 削除終了

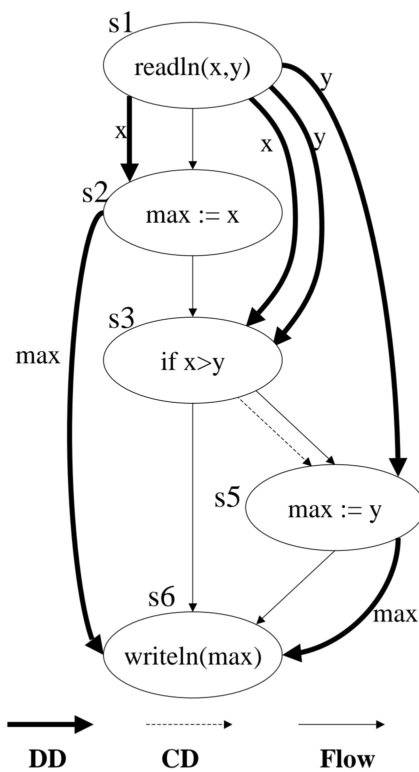


図 2.9: 挿入前の PDG

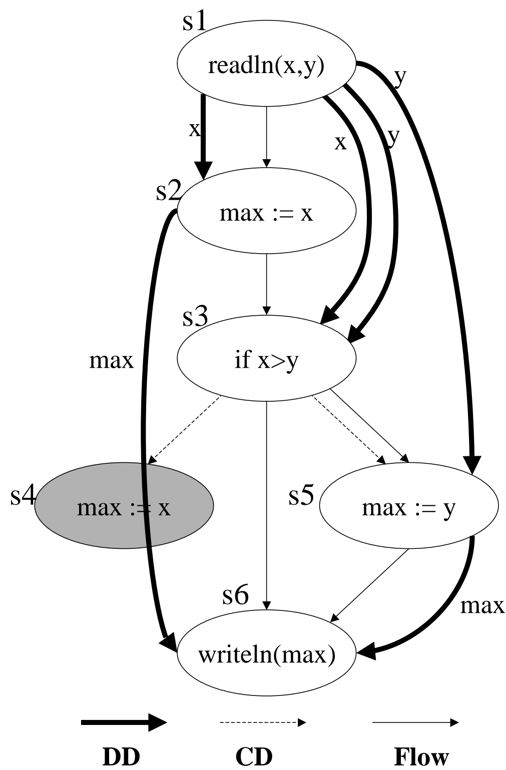


図 2.10: $s4$ を作成した段階

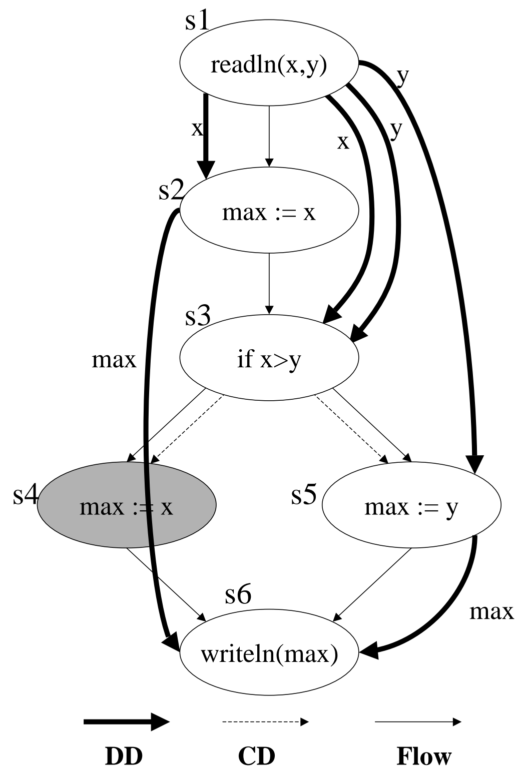


図 2.11: 挿入–Step1

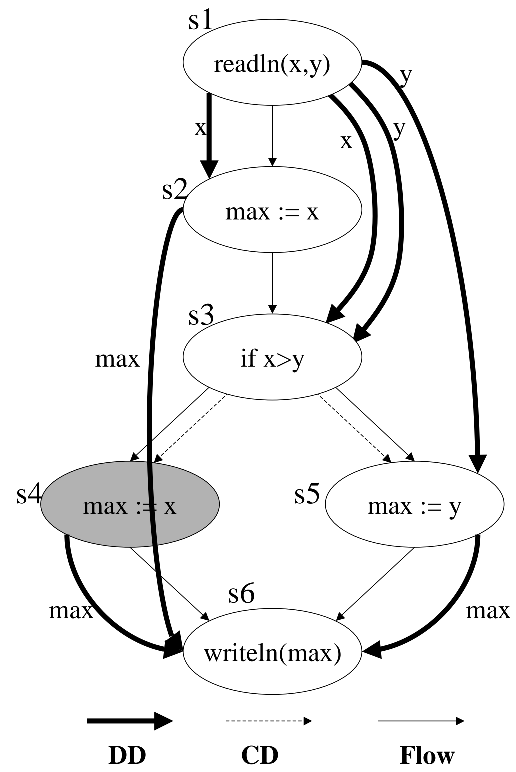


図 2.12: 挿入–Step2

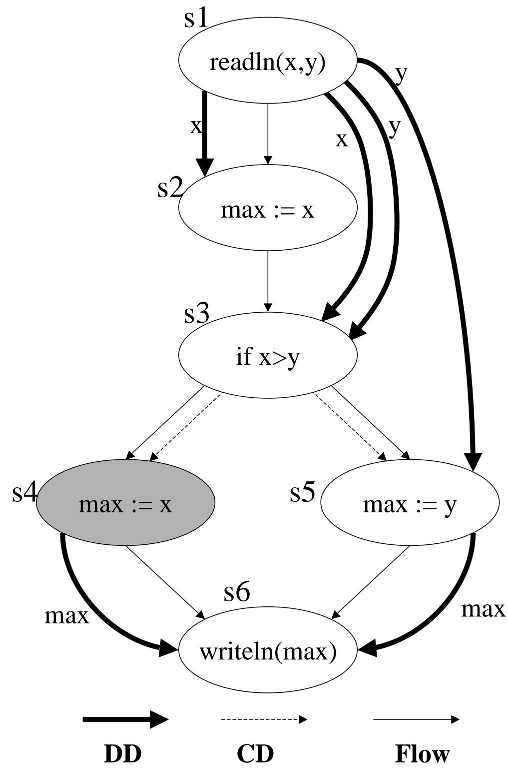


図 2.13: 挿入–Step3

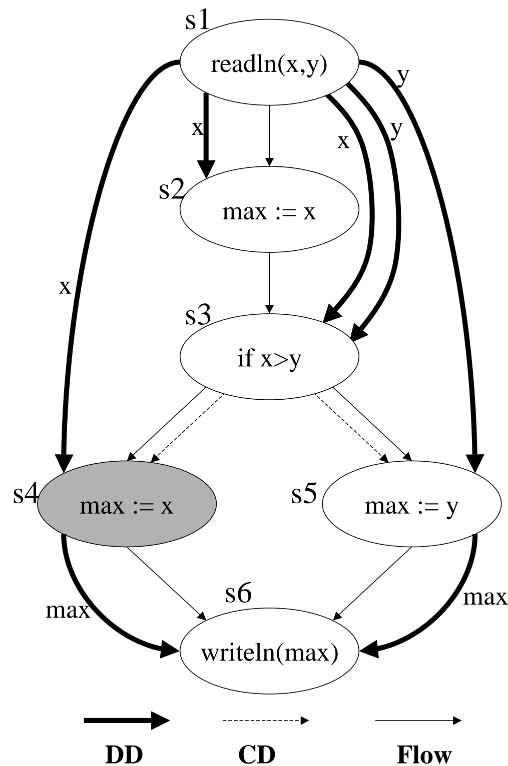


図 2.14: 挿入後の PDG

ソースコード中の 3 つ目の網掛け部分（濃い網掛け）がスライシング基準，2 つ目がブレイクポイント，その他の網掛けがスライスに含まれる文を表しており，実際の作業画面ではそれぞれ緑色・水色・黄色に表示され，作業者に視覚的にスライスを認識させることができる．

これらのアルゴリズムの実行時間を表 2.3 に示す（SPARCstation 20, Memory 64MB 上での実行時間）．但し，表 2.3 中の左側二つ（50 行・100 行）は単一関数，右側二つは複数関数のプログラムである．

表 2.3: 実行時間（単位は秒）

	50 行	100 行	250 行	430 行
PDG 再計算	0.08	0.35	1.42	16.61
削除	0.01 以下	0.01 以下	0.01	0.07
挿入・変更	0.01 以下	0.01 以下	0.01	0.05

これにより，本アルゴリズムによる PDG の変更がプログラムを変更した後に全体を再

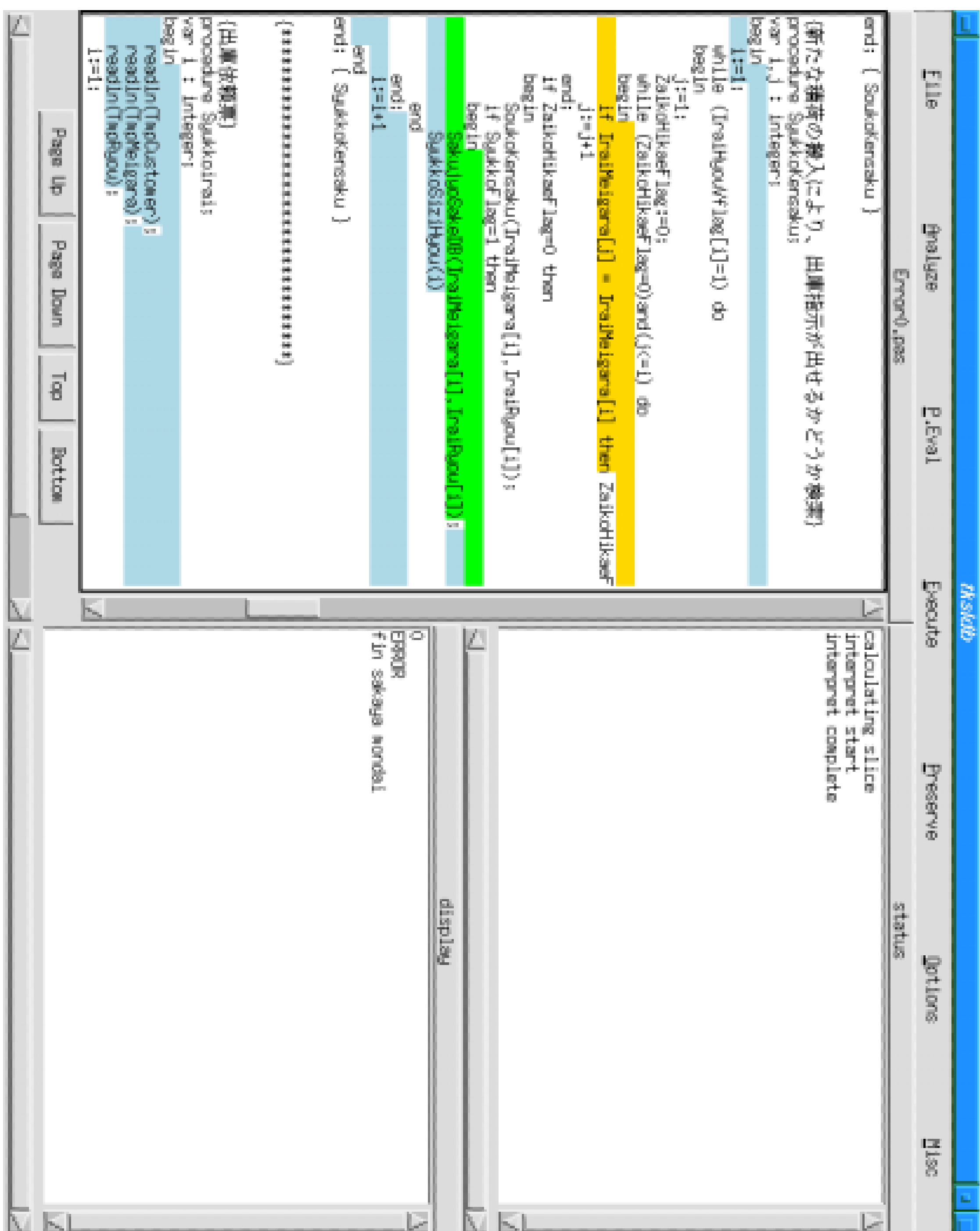


図 2.15: ツール実行画面

計算することに比べ，解析時間が大幅に削減される，特に規模の大きいプログラムに対して有効であることがわかる．

2.8.2　計算量

PDG の再計算・更新にかかわる要素を表 2.4 に示す．集合演算にその要素数に比例する計算量が必要だとしたときの，PDG 再計算・本アルゴリズムの計算量は表 2.5 のようになる．

表 2.4: PDG 再計算・更新にかかわる要素

P	手続きの総数
G	大域変数の総数
L	手続きの局所変数の最大値
S_i	手続き呼び出しの総数
S_t	文の総数
V	PDG の節点の総数
E	PDG の有向辺の総数

表 2.5: PDG 更新アルゴリズムの計算量

PDG 再計算	$O(P \cdot S_t \cdot (G+L))$
単一関数内削除	$O((V+E) \cdot (G+L))$
単一関数内挿入	$O((V+E) \cdot (G+L))$
単一関数内修正	$O((V+E) \cdot (G+L))$
関数間削除・挿入・修正	$O(P \cdot S_t \cdot (G+L)+$ $S_i \cdot G \cdot (V+E)(G+L))$

2.8.3　考察

PDG の再計算及び更新アルゴリズムの時間計算量を示した．

本アルゴリズムの時間計算量が最悪となるのは PDG が完全グラフであり，かつ特殊節点に関する操作をすべての関数で行う時である．しかし，通常のプログラムでこのようになることは無い．

また，一つの関数の大きさはプログラム全体の大きさとは独立で，ある定数以下であると仮定すると前定義節点の発見手続きは $O(1)$ で計算できる．

以上の仮定のもとでは，本アルゴリズムでは関数間の削除・挿入・修正は $O(V+E+S_t\cdot(G+L))$ で計算できる．一方，PDG 全体を再計算すると $O(S_t\cdot(G+L))$ になる．

PDG 再計算と本アルゴリズムでは解析の対象がそれぞれソースプログラム，PDG と異なるため，計算量の表現に用いられる変数が異なっている．本アルゴリズムでは関数境界を越える依存関係を PDG 再計算と同様に解析しており，表記上の計算量が大きくなる．

本アルゴリズムでは関数間の依存関係の変化を解析するために，関数で定義・参照する変数を収束するまで調べる．この部分の計算量は PDG 再計算と同じだけ必要である．しかし，PDG 再計算ではこの繰り返しにおいてすべての節点でデータ依存辺の挿入を行っているが，本アルゴリズムでは節点の内容を調べるだけで，辺の挿入は行わない．

PDG 再計算のためにはプログラム全体の構文解析などが必要であるが，本アルゴリズムでは部分的な構文解析（必要ない場合もある）だけで済み，大部分の節点や辺の作成も必要ないため，現実的には PDG 再計算に比べて短い時間で解析できる．

また，本アルゴリズムのために PDG 中に保持しておく必要のある情報は関数ごとの定義・参照変数情報だけである．しかし，変更前に PDG を調べることによりこの情報も復元できるため，PDG のみあれば変更を行うことができる．これは，すべての節点で到達定義集合を保存しておくという文献 [37] のアルゴリズムに比べ使用する作業領域面で効率が良い．

2.9 既存手法との比較

2.9.1 PDG 計算アルゴリズム

PDG の計算アルゴリズム [35] の一部を以下に示す．このアルゴリズムではプログラムの変更が行われた際に，プログラム全体の解析を行う．計算量としては提案した PDG 再計算アルゴリズムと同等であるが，解析対象のサイズ，及び構文解析の有無など，実際のプログラムの解析に必要となる時間及びメモリ空間のオーバヘッドは非常に大きくなる．

プログラム全体の解析

プログラムは手続きを一つの単位として解析される．手続きの解析を開始する前に，すべての手続きに対して，以下のような初期化を行う (f を手続きとする).

$$SuDEF(f) \ \leftarrow \ \phi$$

$$PoDEF(f) \leftarrow \phi$$

$$ImUSE(f) \leftarrow \phi$$

また，P を手続きの集合として，以下のようにプログラム全体の解析を行う.

1. $P \leftarrow \{p|p$ はプログラム内の手続き $\}$
2. すべての $q \in P$ について q の解析を行う. (2.9.1 節参照)
3. すべての $q \in P$ について q のすべての g–in 節点に入ってくる，またはそこから出ていく DD 関係辺をすべて消去する.
4. $P \neq \phi$ の間，次の操作を繰り返す.
   - $P \leftarrow P - \{q\}$
   - q の解析をする. (2.9.1 節参照)
   - もし $SuDEF(q)$, $PoDEF(q)$, $ImUSE(q)$ のうちのどれかの値が変化したら，$P \leftarrow P \cup \{r|r$ は q を直接呼び出す手続き $\}$
5. 主プログラムを解析する.

関数 及び 手続きの宣言部に対する処理

手続き f は，図 2.16 のような構造で表される．ここでは，その本体の領域を B と表す. この f は以下のような手順で解析される.

function f (...) : ⋯;
var ⋯
begin
⋮
end; } B

図 2.16: 手続き定義の概略

1. RD_{in} の初期値として，

   $RD_{in} \leftarrow \{\langle w, f_w\text{–}Pin\rangle | w$ は手続き f の仮引数変数 $\} \cup \{\langle v, f_v\text{–}Gin\rangle | v \in ImUSE(f)\}$

   とする.

2. 手続き本体 B を解析する．

3. $SuDEF(B)$, $PoDEF(B)$ から大域変数以外の要素を除く．

4. f において,

$$\begin{aligned} SuDEF(f) &== SuDEF(B) \wedge \\ PoDEF(f) &== PoDEF(B) \wedge \\ ImUSE(f) &== ImUSE(B)(2.9.1\text{ 参照}) \end{aligned}$$

が成り立っていないなら,

$$\begin{aligned} SuDEF(f) &\leftarrow \{\langle v, f_v\text{–}Gout\rangle | \langle v, f_v\text{–}Gout\rangle \in SuDEF(B)\} \\ PoDEF(f) &\leftarrow \{\langle w, f_w\text{–}Gout\rangle | \langle w, f_w\text{–}Gout\rangle \in PoDEF(B)\} \\ ImUSE(f) &\leftarrow \{t | t \in ImUSE(B)\} \end{aligned}$$

とする．

5. $RD_{out}(f)$ の中から，関数の戻り値に関する定義を探し，その定義節点 n から関数の $exit$ 節点への DD 関係辺 ($n \xrightarrow{n} f\text{–}exit$) を作る．また，すべての大域変数 g に関する定義を探しだし，その定義節点 m から対応する g_{empty}–out 節点への DD 関係辺 ($m \xrightarrow{g} f_g\text{–}Gout$) を作る．同様に，すべての変数渡し仮パラメータ p に関する定義を探しだし，その定義節点 n から対応する手続きの $para_{empty}$–out 節点への DD 関係辺 ($n \xrightarrow{p} f_p\text{–}Pout$) を作る．

文の解析

条件文，繰り返し文，代入文，入出力文，手続き呼出文の解析は文献 [35] のものに以下の修正を加えるだけである．

S : 文の集合もしくは式

新たに $ImUSE(S)$ なる変数の集合を定義し,

$ImUSE(S) \leftarrow \{v_{var}|$ v は S で参照される変数のうち外部からの影響を受ける大域変数で, $v \in RD_{in}(S)$ $\}$

手続き呼出文，式の解析については，引数として変数渡しも許したため少し異なる．それらの解析方法を以下に示す．

手続き呼出文の解析 $S : p(exp_1, ..., exp_n);$ とする．

1. $RD_{in}(exp_1) \leftarrow RD_{in}(S)$ として式 exp_1 の処理を行う．
2. $RD_{in}(exp_k) \leftarrow RD_{out}(exp_{k-1})(1 < k \leq n)$ としてすべての引数についての処理をする．
3. $RD_{in}(S) \leftarrow RD_{out}(exp_n)$
4. $RD_{in}(S)$ 内のすべての大域変数 v の定義 d それぞれに対して d からそれに対応する p の $g - in$ 節点への DD 関係辺（ $d \xrightarrow{v} p_v\text{–}Gin$ を作る．
5. $RD_{out}(S) \leftarrow \{d | d \in RD_{in}(S) \wedge d_{var}$ と同じ変数名の要素が $SuDEF(p)$ にない $\}$ $\cup \{ \langle v, p_v\text{–}Gout\rangle \, | v$ と同じ名前の変数が $SuDEF(p)$ の中にある $\} \cup PoDEF(p)$ $\}$
6. $SuDEF(S) \leftarrow SuDEF(p) \cup \bigcup_{k=1}^{n} SuDEF(exp_k)$
7. $PoDEF(S) \leftarrow PoDEF(p) \cup \bigcup_{k=1}^{n} PoDEF(exp_k)$
8. $ImUSE(S) \leftarrow ImUSE(p) \cup \bigcup_{k=1}^{n} ImUSE(exp_k)$

式の解析 節点 n に含まれている式 exp は以下のようにして解析される．

1. $RD_{out}(exp) \leftarrow RD_{in}(exp)$
2.
$$
\begin{aligned}
SuDEF(exp) &\leftarrow \phi \\
PoDEF(exp) &\leftarrow \phi \\
ImUSE(exp) &\leftarrow \phi
\end{aligned}
$$
と初期化する．
3. 式を左から読み込み，
   - ある変数 v の参照があれば，その時点での $RD_{out}(exp)$ の中から v に関する定義 d をすべて探して，その定義節点から節点 n への DD 関係辺（ $d \xrightarrow{v} n$ ）を作る．
   - ある関数 g の呼び出しがあれば，その引数を先に式の列として処理した後，g の $exit$ 節点から n への DD 関係辺（ $g\text{–}exit \xrightarrow{g} n$ ）を作り，$RD_{out}(S)$ 内

のすべての大域変数 v の定義 d それぞれに対して d からそれに対応する g の g–in 節点への DD 関係辺（ $d \xrightarrow{v} g_v\text{–}Gin$ ）を作る．さらに，以下のように各集合の値を更新する．

$RD_{out}(exp) \leftarrow \{\langle v, n\rangle \,|v \in SuDEF(g)\} \cup \{PoDEF(f)\,\} \cup \{d|d \in RD_{out}(exp)$

$\wedge\, d_{var}$ と同じ名前の変数が $SuDEF(f)$ に存在しない $\}$

$SuDEF(exp) \leftarrow SuDEF(exp) \cup SuDEF(g)$

$PoDEF(exp) \leftarrow PoDEF(exp) \cup PoDEF(g)$

$ImUSE(exp) \leftarrow ImUSE(exp) \cup ImUSE(g)$

という処理を式の終りまで続ける．

4. exp が手続きの引数 p の時，現節点から対応する関数 g の $para$–in 節点への DD 関係辺（ $n \xrightarrow{p} g_p\text{–}Pin$ ）を作る．また，その引数が変数渡しをされている時には，対応する関数 f の $para$–out 節点から現節点への DD 関係辺（ $f_p\text{–}Pout \xrightarrow{p} n$ ）を作る．

2.9.2　既存の更新アルゴリズム

既存の更新アルゴリズム [37] の一部を以下に示す．このアルゴリズムは各節点において到達定義集合を保存し，変更が行われた時にその変化を後ろの節点に伝えることによりデータ依存辺を引き直す．この手法では，各節点において到達定義集合を保存する必要があるため，解析に必要となるメモリ空間が非常に大きくなる．また，関数境界を越えて伝わる依存関係について考慮されていないため，複数関数を持つプログラムに適用することができないという問題がある．

削除アルゴリズム

ある文を削除することを考え，その文に対応する PDG 上の節点を便宜上 “削除節点” と呼ぶ．

概要

削除される文において変数が定義されていれば，それ以降の文，つまり削除節点からフロー辺を順方向に辿った先の節点における到達定義に影響を与える．そのため，フロー辺の先の節点での到達定義を順次再計算する．到達定義の変更が無くなった時点で終了する．

アルゴリズム

削除節点が分岐文あるいは繰り返し文の場合は，その節点から制御依存辺が出ている節点すべてについて以下を繰り返す．

1. 削除節点において定義される変数がない場合，5. に進む．
2. 削除節点において定義される変数がある場合，削除節点の RD_{out} を削除節点の RD_{in} に置き換える．
3. 削除節点からフロー辺を順方向に辿る．
   (a) この時，現在辿っている節点を現節点とする．
   (b) 前節点の RD_{out} を求める．この時，前節点が複数あれば，すべての前節点の RD_{out} の和集合を求める．
   (c) 前節点の RD_{out}（の和集合）と現節点の RD_{in} を比較する．もし，その結果が等しければフロー辺を辿ることを終了し，異なれば前節点の RD_{out}（の和集合）を現節点の RD_{in} として，現節点の RD_{out} を再計算する．
   (d) 現節点からフロー辺を順方向に辿り（辿るフロー辺がない場合 5. へ），3a. からまた繰り返す．現節点からフロー辺が複数本出ていれば，そのそれぞれについて，3a. からの処理を繰り返す．
4. 削除節点 s で変数 w を定義し $s \xrightarrow{w} t$ という関係の節点 t がある時，前定義節点 u を求め，データ依存辺 $u \xrightarrow{w} t$ を引く．
5. 削除節点の前節点と後節点をフロー辺で結ぶ．
6. 削除節点に関係する辺（データ依存辺，制御依存辺，フロー辺）をすべて削除し，最後に削除節点自体を削除する．

但し，関数（手続き）の境界を越えた依存関係は特殊節点を中継したデータ依存関係がある．そのため，関数（手続き）呼び出し文を含む文を削除する場合，削除する文に対応する PDG 上の節点に関係する辺（節点に入ってくるあるいは出ていく，データ依存，制御依存辺，フロー辺）だけでなく，関数（手続き）の特殊節点から直接出ていく辺あるいは特殊節点に直接入ってくる辺も削除する．

挿入アルゴリズム

ある文を挿入する場合を考え，その文に対応する PDG 上の節点を便宜上 " 挿入節点" と呼ぶ.

概要

挿入する文において変数が使用されていれば，その変数を定義した節点から挿入節点へのデータ依存関係が生じる.

また，挿入する文において変数が定義されていれば，それ以降の文，つまり挿入節点からフロー辺を順方向に辿った先の節点における到達定義に影響を与える．そのため，フロー辺の先の節点での到達定義を順次再計算し，それにデータ依存関係辺を更新する．到達定義への影響が無くなった時点で終了する.

アルゴリズム

1. 挿入節点自体を作成する．この節点の RD_{in} は前節点の RD_{out} と等しくする.
2. 前節点から挿入節点へ，また，挿入節点から後節点へフロー辺を引き，前節点と後節点の間のフロー辺を削除する.
3. 挿入節点において参照する変数があれば，挿入節点の到達定義集合を調べる．もし参照する変数に関する前定義節点があれば，その前定義節点から挿入節点に対してデータ依存辺を引く.
4.
   - 挿入節点において定義する変数がなければ，終了.
   - 挿入節点において定義する変数があれば，この節点の RD_{out} を計算する.
5. 挿入節点からフロー辺を順方向に辿る.
   (a) この時，現在辿っている節点を現節点とする.
   (b) 前節点の RD_{out}（の和集合）を求める.
   (c) 前節点の RD_{out}（の和集合）と現節点の RD_{in} を比較する．もし，等しければフロー辺を辿ることを終了する．異なれば現節点の RD_{in} を前節点の RD_{out}（の和集合）に置き換えて 5d. へ進む.
   (d) 現節点へのデータ依存辺をすべて削除し，RD_{in} 合をもとにして，現節点へのデータ依存辺を引き直す.

(e) 現在の RD_{in} をもとにして，現節点の RD_{out} を再計算する．

(f) 現節点からフロー辺を順方向に辿り（辿る辺がない場合終了する），5a. からまた繰り返す．現節点からフロー辺が複数本出ていれば，そのそれぞれについて，5a. からの処理を繰り返す．

但し，分岐文・繰り返し文を挿入する場合は，まず，条件式部分を挿入し，その後実行部分を挿入する．条件式部分を挿入した時は，合流節点を条件式の後節点として挿入する．また，実行部分を挿入した時は条件式部分からの制御依存辺を引く．

2.10 まとめ

プログラムの部分的変更が行われた際に，PDG を部分的に変更する手法を提案した．また，既存のシステムに組み込みその高速実行性を確認した．我々のデバッグ支援システムはこの機能を組み込んだため，頻繁にソースプログラムの変更をしてもスライスを効率良く抽出できるようになり，よりインタラクティブにデバッグ作業を行えるようになった．

本手法は，今回実際に組み入れたシステムに限らず，PDG を利用しその一部が頻繁に変更される可能性のあるシステム使用時の作業効率を向上させることが期待される．

本研究では Pascal 風言語を入力言語として扱ったが，2.2.4 節で示した PDG を用いて解析を行っていれば，他の手続き型言語にも適用可能である．

今後の課題としては以下が挙げられる．

一つめは，変更対象の問題である．本アルゴリズムでは変数や関数の宣言部などに対する変更を考えていない．そのため，本アルゴリズムのみを用いたデバッグの際に不都合を生じる可能性がある．

二つめは，計算量の問題である．本研究で示したアルゴリズムは最悪時に PDG 再計算よりも計算量が大きくなる場合がある．但し，現実のプログラムではこのような場合はほぼあり得ず，また，構文解析や節点・辺の作成の大部分が不要であることから，実際に使用する上での問題は無い．

三つめは，実装の問題である．現在の実装は変更を行う文を指定してから変更の種類を指定する，といった方法を取っているが，通常のエディタなどでデバッグを行うのと変わらない環境を実現する方が望ましい．

第3章　準動的解析を用いたプログラムスライシング手法

3.1　導入

テスト・保守工程でのデバッグ作業において，プログラムスライシング技術はフォールト位置特定に有効である [48]．このとき，テストケースは一般に有限個であり，入力データも特定のものに限定される．本来，デバッグ作業では対象プログラムの実行は必要不可欠であり，そこから得られる情報を利用することで，静的スライスよりも小さい（正確な）スライスを抽出することができる．

動的スライシングはそのようなアプローチの一つであるが,（実行経路 および 各実行時点での変数の状態など）プログラム実行に関する全履歴を必要とするため，プログラム実行時のオーバヘッド（解析に必要な時間・メモリ空間など）が非常に大きくなる．

実行時オーバヘッドの削減を目的として，静的解析と動的解析を組み合わせてプログラムの解析を行う**準動的解析**手法が提案されている．本研究では，準動的解析手法を用いたスライシング手法である**準動的スライシング手法**について，解析の正確性と実行時オーバヘッドのトレードオフを実現した，**依存キャッシュスライシング**（*Dependence-Cache Slicing*）手法を提案する．

以降，3.2 において準動的スライシング手法の説明およびその分類を行う．3.3 では提案する依存キャッシュスライシング手法について述べる．3.4 では依存キャッシュスライシングの実行例を示し，3.5 では依存キャッシュスライシングの計算時間に関する実験について述べる．また，3.6 において考察を行う．

3.2　準動的スライシング手法の分類

1.2.4 で述べたように，動的解析のオーバヘッド削減を目的として，静的情報と動的情報を組み合わせた準動的解析手法が必要とされている．このような準動的スライシング手法は，動的に収集する情報の違いによって，

- 実行経路の収集に着目した手法
- データ依存関係の収集に着目した手法

の 2 つに分類可能である．

これらについて以下に述べる．

3.2.1 実行経路の収集に着目した手法

プログラムの実行経路に関する情報を収集・利用することでスライスサイズの削減を行う．軽量な実行時オーバヘッドで経路情報を収集することに重点が置かれる．

- Profiling Method[1]

  動的スライシングを単純化したものとして提案された．各文の実行の有無を記録し，静的に生成された PDG から実行されなかった文を削除する．この手法では，実行系列の保存は不要である．

- コールマークスライシング (Call-Mark Slicing)[21]

  実行時，関数・手続き呼び出し文の実行の有無のみ記録する．しかし，静的解析により導出される実行経路情報と組み合わせることで，呼び出し文以外で かつ 確実に実行されることのない文もスライス計算対象から排除できる．この手法は，Profiling Method と比べ，実行時に記録する文が少なくなる．

- ハイブリッドスライシング (Hybrid Slicing)[9]

  ユーザの設定したブレークポイントの実行履歴または各関数・手続きの呼び出し履歴を記録することで，実行経路の予測を行う．有効な実行経路情報を得るため，ユーザはブレークポイントの設定位置に注意を払う必要がある．

3.2.2 データ依存関係の収集に着目した手法

プログラムの実行経路を静的解析により予測できれば，整数やブールなどの単純型の変数に関するデータ依存関係は容易に抽出できる．しかし，配列やポインタ変数に関するデータ依存関係の抽出には限界がある．

```
s1:  a[0] := 0;
s2:  a[1] := 1;
s3:  readln(i);
s4:  c := a[i];
s5:  writeln(c);
```

図 3.1: 配列変数に関するデータ依存

例として，図 3.1 に示すプログラム断片を考える．この場合，実行経路は唯一に定まるが，それだけでは文 s4 が文 s1 及び文 s2 にデータ依存するかを決定することはできない．

通常，この問題の解決には，**完全実行履歴**（*Full Execution History*）を用いて DDG 構築を行う．DDG ではすべての配列要素が展開されており，完全なデータ依存関係を保持することができるが，完全実行履歴の蓄積に要するオーバヘッドは極めて大きい．そこで，データ依存関係の正確性と実行時オーバヘッドとのトレードオフを考慮した手法が提案されている．

- Reduced DDG Method[1]

  DDG 中に存在する同一構造を持つ部分グラフを一つに集約し，DDG 全体の大きさを削減する．これにより DDG のサイズは小さくなるが，実行時オーバヘッドは類似性チェックのため増加する．

データ依存関係を動的に収集する手法では，ポインタや配列などの変数の依存関係についても正確に解析することができるため，フォールト位置特定に非常に有用である．しかし，Reduced DDG Method では DDG のサイズを小さくすることで DDG 蓄積に必要なメモリ空間の削減と，DDG 作成後のスライス計算時間を短縮することができる反面，実行時のオーバヘッドが増大するという問題がある．

そこで，本研究では，実行時オーバヘッドを削減するとともにデータ依存関係の正確性をも保つことを目的として，依存キャッシュスライシング手法を提案する．

3.3　依存キャッシュスライシング

3.3.1　概要

通常，ポインタ変数や配列要素のデータ依存関係を静的解析により得ることは非常に難しい [11, 12, 18]．一方，ある入力データを与えプログラムを実行し，その際にポインタ変

数の参照先や配列の添字値を把握する機構を実現することは容易である．しかし，プログラム実行のために非常に大きなオーバヘッドを必要とする．

そこで，依存関係抽出の正確性と実行時オーバヘッドとのトレードオフを考慮したスライシング手法である，**依存キャッシュスライシング**（*Dependence Cache Slicing*）手法を提案する．以下は依存キャッシュスライスの計算手順の概略である．

Step 1 静的制御依存関係解析

PDG の部分グラフ PDG_{DS} を静的に生成する．

まず，静的スライシングで利用する PDG と同様，文または制御文に対応する節点を用意する．そして，文間に制御依存関係が存在すれば，対応する節点間に制御依存辺を引く．ただし，PDG$_{DS}$ にデータ依存辺は加えない．

Step 2 動的データ依存関係解析

対象プログラムをある入力データで実行する．

実行の際，次節で示すデータ依存関係抽出アルゴリズムに基き，動的なデータ依存関係を計算し，PDG$_{DS}$ にデータ依存辺を追加する．プログラム実行が終了した時点で，PDG$_{DS}$ の完成となる．

Step 3 PDG$_{DS}$ によるスライス計算

PDG$_{DS}$ を用いて，静的スライシングと同様の方法でスライス計算を行う．

例えば，スライシング基準 (s_c, v) に関する依存キャッシュスライスを抽出する場合，まず，s_c に対応する節点から制御依存辺及び v に関するデータ依存辺を逆向きに辿ることで到達可能な節点集合を導出する．そして，この節点集合に対応する文が求めるスライスとなる．

3.3.2　データ依存関係収集アルゴリズム

図 3.2 に **3.3.1** 節の Step 2 で使われるデータ依存関係抽出アルゴリズムを示す．まず，プログラム中で用いられるすべての変数 v に対して，キャッシュ（$C(v)$ と記す）を用意する．大域変数など静的な変数はプログラム実行開始時に，スタック上の Automatic 変数やヒープ中の変数など動的な変数はその変数の割付け時にキャッシュを用意する．プログラムの各実行時点において，$C(v)$ は最も新しく v を定義した文に対応する節点を保持し，文 s に

入力

PDG_{DS}: 部分的に生成された PDG
P: 対象プログラム
I: P への入力

作業変数

P 中の各変数 v に対する依存キャッシュ$C(v)$

出力

OUT: 入力 I に対するプログラム P の実行の出力
PDG_{DS}: 完成した PDG

アルゴリズム本体

1. P 中の各静的変数 v に対し, $C(v) := \perp$

   { 各キャッシュの未代入マークによる初期化.
   (注) 動的に割当てられる変数は割当てられた時点でキャッシュを用意し，$\perp$ を代入する． }

2. P が停止するまで以下を繰り返し実行する
   { P を入力 I で最初から停止するまで文ごとに実行 }

   (a) I に関して，P の次の一文 s を実行
   (b) s で使用 (参照) される各変数 u について, $C(u) \neq \perp$ かつ，データ依存辺 $C(u) \xrightarrow{u} s$ が存在しなければ PDG$_{DS}$ に $C(u) \xrightarrow{u} s$ を追加する．
   (c) sで定義される各変数 w について，$C(w) := s$

図 3.2: データ依存関係収集アルゴリズム

おいて v が使用（参照）された時，$C(v)$ が保持する節点から s に対応する節点に対してデータ依存辺を (すでに存在しなければ) 追加する．一方，s で v が定義された時，$C(v)$ は s に対応する節点に更新する．これらをすべての変数に対して行う．

配列変数や構造体に対しては，すべての要素に対してキャッシュを用意する．例えば，$A[1], A[2], \ldots, A[10]$ という 10 個の要素を持つ配列変数 A は，キャッシュ$C(A[1]), C(A[2]), \ldots, C(A[10])$ を持つ．ポインタ変数 p が文 s において使用された場合，使用された p 自身だけでなく，p を介して間接参照された変数 $p\uparrow$ をも考慮しなければならない．つまり，直接と間接の参照（$C(p) \xrightarrow{p} s$ と $C(p\uparrow) \xrightarrow{p\uparrow} s$ の両者）をデータ依存辺に含めなければならない．また，文 t における $q\uparrow := \ldots$ のようなポインタ変数 q を介した間接的な代入については，キャッシュ$C(q\uparrow)$ が t において更新され，また t において q が使用されたと考える．

動的変数に対しては，個々のインスタンスごとにキャッシュを用意する．また，配列や構造体では，参照されうる要素ごとにキャッシュが必要である．例えば，要素 v_1 と v_2 から構成される構造体 v に対しては，キャッシュ$C(v_1), C(v_2)$ を用意する．各 v_1, v_2 への定義及び参照時の操作は図 3.2 のアルゴリズムと同じである．文 s で構造体 v 全体への定義が行われる時，$C(v_1) := s, C(v_2) := s$ を行う．また，s で v 全体の参照が行われる時は，データ依存辺 $C(v_1) \xrightarrow{v_1} s$ 及び $C(v_2) \xrightarrow{v_2} s$ を PDG$_{DS}$ に追加する．(注: 辺上のラベル v_1,v_2 は静的に決まりうる要素名)

このアルゴリズムでは，プログラム実行中の変数の使用状況に比例したキャッシュ空間と，変数へのアクセス回数に応じた実行時間のオーバヘッドが必要である．

3.4 依存キャッシュスライシングの実行例

3.4.1 実行例 1

図 3.1 で示したプログラムの一部に対し，動的スライシング基準 (文入力 $i = 0, s5, c$) に対して依存キャッシュスライスを計算する例を示す．

最初に，静的解析によって節点と制御依存辺のみから構成される PDG_{DS} を生成する．図 3.1 のプログラムでは制御依存関係が存在しないため，PDG_{DS} は節点のみから構成される．

次に，データ依存関係収集アルゴリズムによって，文間のデータ依存関係を求める．

1. このプログラムで使用される変数は，配列 a，変数 i, c である．そのため，これらの

変数の初期化を行う．

初期状態: $C(a[0]) = \bot, C(a[1]) = \bot, C(i) = \bot, C(c) = \bot$

2. 文 s1 では，参照される変数は無く，$a[0]$ が定義されている．そのため，$a[0]$ のキャッシュに文 s1 を代入する．

   文 s1 実行後: $C(a[0]) = s1, C(a[1]) = \bot, C(i) = \bot, C(c) = \bot$

3. 文 s2,s3 では，文 s1 と同様に参照される変数は無く，変数 $a[0], i$ がそれぞれ定義されている (i への入力は 0 とする).

   文 s2 実行後: $C(a[0]) = s1, C(a[1]) = s2, C(i) = \bot, C(c) = \bot$
   文 s3 実行時: $C(a[0]) = s1, C(a[1]) = s2, C(i) = s3, C(c) = \bot$

4. 文 s4 では，変数 $a[i]$ が参照され，c が定義されている．ここで，$i = 0$ であり，$C(a[0]) = s1$ であるので，データ依存辺 $C(a[0]) \xrightarrow{a[0]} s4$，つまり $s1 \xrightarrow{a[0]} s4$ を追加する．また，変数 c に関するキャッシュの更新を行う．

   文 s4 実行時: $C(a[0]) = s1, C(a[1]) = s2, C(i) = s3, C(c) = s4$

5. 文 s5 では，変数 c が参照され，定義される変数は無い．そのため，データ依存辺 $C(c) \xrightarrow{c} s5$，つまり $s4 \xrightarrow{c} s5$ を追加する．s5 では定義される変数が無いため，キャッシュの更新は行わない．

これらの結果，生成される PDG_{DS} は図 3.4.1 のようになる．この PDG の節点 s5 に対しスライスを計算する．この結果，求められる依存キャッシュスライスは図 3.4 のようになる．

一方，同じプログラムに対し静的スライスを計算する場合，構成される PDG は図 3.4.1 のようになる．静的スライスではプログラムの実行を行わないため，文 s3 の入力値を確定することができないず，文 s1,s2 の両方から文 s4 に対しデータ依存辺が存在する．この PDG から静的スライスを求めた結果は，ソースプログラムと同一である．

また，動的スライシングの場合は，依存キャッシュスライシングと同様に文 s3 の入力値を確定させることができるため，結果のスライスは依存キャッシュスライスと同じものとなる．

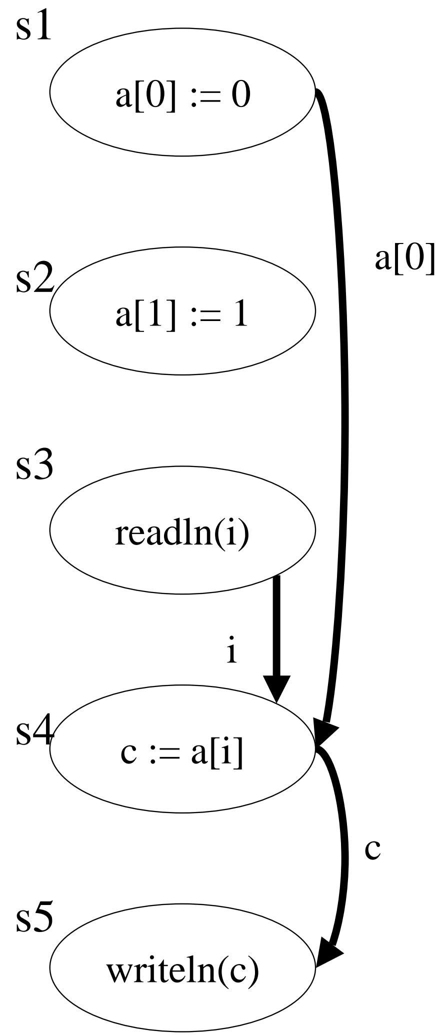


図 3.3: データ依存関係収集アルゴリズムによって生成される PDG

s1: a[0] := 0;
s2:
s3: **readln**(i);
s4: c := a[i];
s5: writeln(c);

図 3.4: 依存キャッシュスライス計算結果

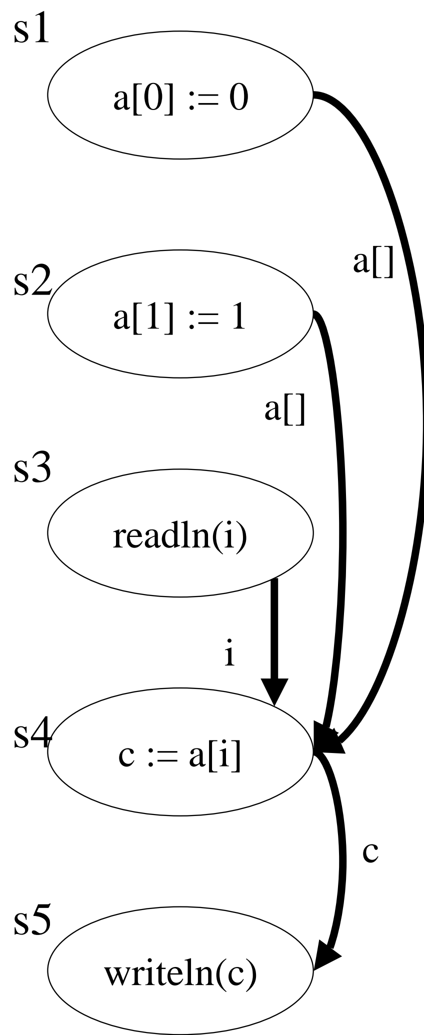


図 3.5: 静的スライス計算時に生成される PDG

3.4.2 実行例 2

```
s1:  i := 0 ;
s2:  a[0] := 0;
s3:  a[1] := 1;
s4:  while (i < 2) do begin
s5:          b := a[i];
s6:          i := i + 1
s7:  end;
s8:  writeln(b);
```

図 3.6: サンプルプログラム 2

次に，図 3.6 に示すサンプルプログラムについて考える．

依存キャッシュスライシングでは，節点と制御依存辺のみからなる最初に PDG_{DS} を生成する．この段階の PDG_{DS} を図 3.4.2 に示す．

次に，PDG_{DS} に対し，データ依存関係収集アルゴリズムによってデータ依存辺を追加する．

1. 最初に，各変数のキャッシュを初期化する．

   初期状態: $C(i) = \perp, C(a[0]) = \perp, C(a[1]) = \perp, C(b) = \perp$

2. 文 s1,s2,s3 では，それぞれ定義される変数のキャッシュを更新する．

   文 s3 実行後: $C(i) = s1, C(a[0]) = s2, C(a[1]) = s3, C(b) = \perp$

3. 文 s4 では，変数 i が参照されており，$C(i) = s1$ であるため，データ依存辺 $s1 \xrightarrow{i} s4$ を追加する．

4. 文 s5 では，変数 $i, a[0]$ が参照，b が定義されている．そのため，データ依存辺 $s1 \xrightarrow{i} s5, s2 \xrightarrow{a[0]} s5$ を追加し，キャッシュを更新する．

   文 s5 実行後: $C(i) = s1, C(a[0]) = s2, C(a[1]) = s3, C(b) = s5$

5. 文 s6 では，変数 i が参照及び定義されている．データ依存辺 $s1 \xrightarrow{i} s6$ を追加し，キャッシュを更新する．

   文 s6 実行後: $C(i) = s6, C(a[0]) = s2, C(a[1]) = s3, C(b) = s5$

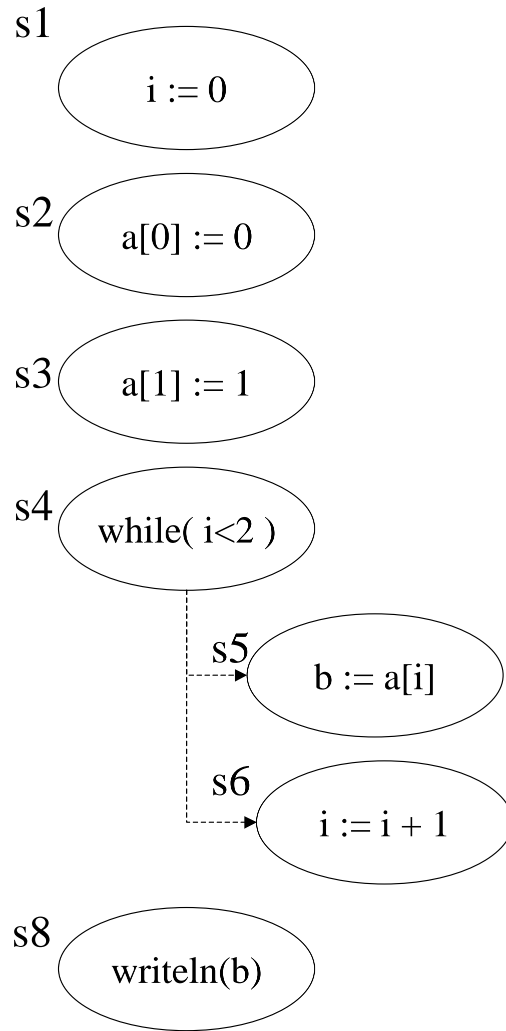


図 3.7: PDG 初期状態

6. 文 s4(2 回目) では，データ依存辺 $s6 \xrightarrow{i} s4$ を追加する．

7. 文 s5(2 回目) では，変数 $i, a[1]$ が参照されているため，データ依存辺 $s2 \xrightarrow{i} s5, s6 \xrightarrow{a[0]} s5$ を追加し，キャッシュを更新する ($C(b) = s5$ であったので実際にはキャッシュの内容は変化しない)．

   文 s5 実行後: $C(i) = s6, C(a[0]) = s2, C(a[1]) = s3, C(b) = s5$

8. 文 s6(2 回目) では，データ依存辺 $s6 \xrightarrow{i} s6$ を追加し，キャッシュを更新する (キャッシュの内容は変化しない)．

   文 s6 実行後: $C(i) = s6, C(a[0]) = s2, C(a[1]) = s3, C(b) = s5$

9. 文 s8 では，変数 b が参照されている．データ依存辺 $s5 \xrightarrow{b} s8$ を追加する．

これによって生成される PDG を図 3.4.2 に示す．この PDG に対する，スライシング基準 ($s8$, b) に関する依存キャッシュスライスの計算結果はソースプログラムと同じである．

同じプログラムに対する静的スライシングを求める際に生成される PDG，及び静的スライスの計算結果は依存キャッシュスライスと同じとなる．

一方，動的スライシングでは，図 3.9 のような実行系列が保存される．この結果，計算される動的スライスは図 3.10 のようになる．

3.4.3 実行例に関する考察

以上，2 つのプログラムに対する依存キャッシュスライシングの実行例を示した．3.4.1 では，依存キャッシュスライスは動的スライシングと同一の結果となり，静的スライシングよりも良い結果となった．一方，3.4.2 では，依存キャッシュスライスは静的スライシングと同一の結果となり，動的スライシングよりも悪い結果となった．

これは，依存キャッシュスライシングが文ごとにデータ依存関係を収集しているため同一の文が複数回実行された際にその繰り返しを区別できないのに対し，動的スライシングでは実際に実行した文を保存し，その履歴に対してスライスの計算を行うという違いによる．

また，静的スライシングは実行を伴わないため，配列の要素変数の値を判断することができず，すべての要素を仮定した安全な近似を行うため，依存キャッシュスライシングに比べ悪い結果となる．

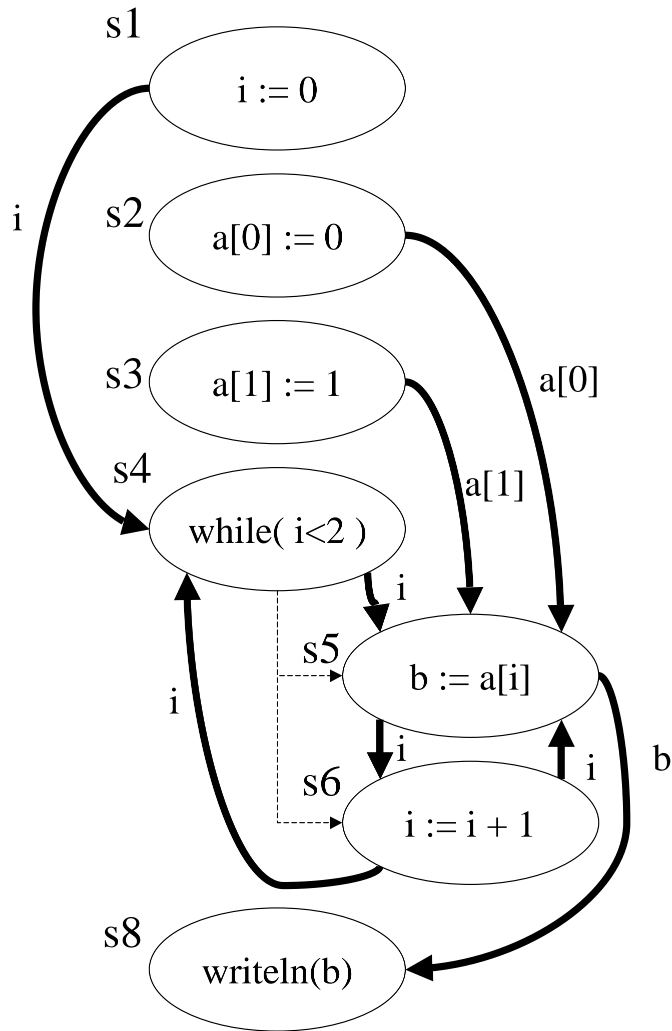


図 3.8: 依存キャッシュスライシングによって生成される PDG

s1	i := 0
s2	a[0] := 0
s3	a[1] := 1
s4	while (i < 2)
s5	b := a[0]
s6	i := i + 1
s4	while (i < 2)
s5	b := a[1]
s6	i := i + 1
s8	writeln(b)

図 3.9: サンプルプログラム 2 の実行履歴

```
s1:   i := 0 ;
s2:
s3:   a[1] := 1;
s4:   while (i < 2) do begin
s5:           b := a[i];
s6:           i := i + 1
s7:   end;
s8:   writeln(b);
```

図 3.10: サンプルプログラム 2 に対する動的スライス

3.5　依存キャッシュスライシングの評価実験

3.5.1　Osaka Slicing System の概要

プログラム解析を行うプラットフォームとして，様々なシステムが考案されている [24, 25, 26, 28, 50]．我々の研究グループでは様々なスライシングアルゴリズムを評価するため，*Osaka Slicing System*(OSS)[39] と呼ぶソフトウェア開発・デバッグ環境を構築している．対象言語は Pascal である．

このシステムはプログラムの実行とデバッグを行うことができる．また，静的スライシング，動的スライシングの機能を持つ．今回，コールマークスライシング，そして **3.3.2** 節で提案した依存キャッシュスライシングの機能を追加した．

3.5.2　サンプルプログラムの実行

このシステムを用い，様々なプログラムを実行し幾つかの評価尺度を得た．プログラム $P1$ はカレンダープログラム，$P2$ は在庫管理プログラム，$P3$ はプログラム $P2$ の在庫管理プログラムの拡張版である．

これらのプログラムに対し，静的スライシング，コールマークスライシング，依存キャッシュスライシング，動的スライシングのそれぞれを実行した．なお，静的スライシングは [35]，コールマークスライシングは [21]，動的スライシングは [49] のアルゴリズムを用いた．

実行にあたり，無作為に選んだ文と，その文中で定義されている変数をスライシング基準として複数選んだ．それぞれのスライシング基準に対し，実行前解析時間，実行時解析時間（実行時間と，スライス計算を行わない場合の実行時間との差），スライス計算時間，

及びスライスサイズを測定し，これらの結果の平均値を求めた．

図 3.11 は 3 つのサンプルプログラムのスライスサイズである（表中，CM はコールマークスライス，DC は依存キャッシュスライスを示す）．

図 3.12 は実行前解析に要した時間を示している．静的スライシングの場合，この値は PDG 構築に要した時間である．コールマークスライシングでは PDG の構築と実行経路情報の静的解析の両方に要した時間である．依存キャッシュスライシングでは初期の PDG_{DS} 構築に要した時間である．また，動的スライシングでは実行前解析は不要である．

図 3.13 に実行時解析時間を示す．静的スライシングの場合，実行時に解析を必要としない．動的スライシングの実行は DDG の生成時間である．依存キャッシュスライシングの実行時解析時間は依存関係をキャッシュする時間と PDG_{DS} を生成する時間である．また，コールマークスライシングでは呼び出し文を記録する処理に必要な時間となる．

図 3.14 は依存グラフを用いてスライスを計算する時間を示している．静的スライシング，依存キャッシュスライシング，動的スライシングの場合，それぞれ PDG，PDG_{DS}，DDG を探索する時間である．また，コールマークスライシングでは，実行されなかったと判る文の頂点を PDG から削除し、探索するのに要した時間である．

次節でこれらの結果について議論を行う．

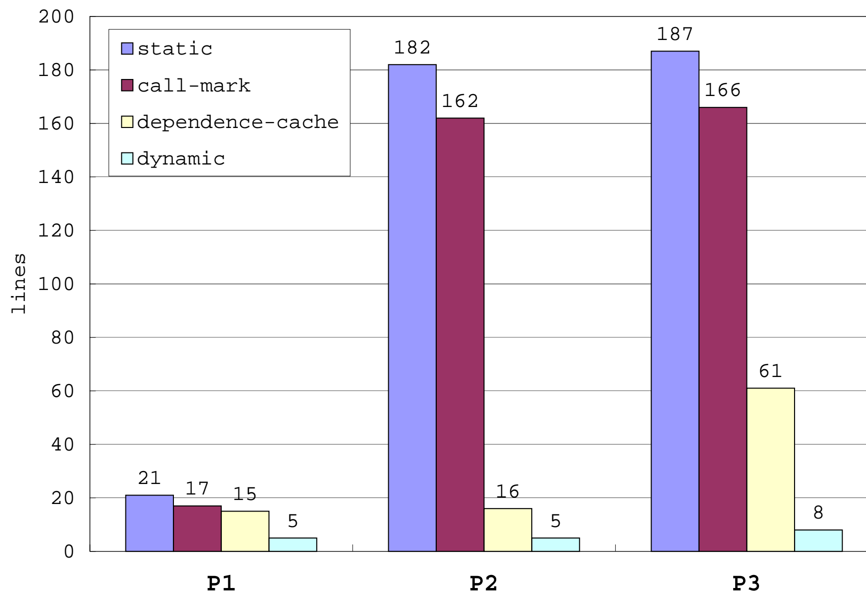


図 3.11: スライスサイズ (行)

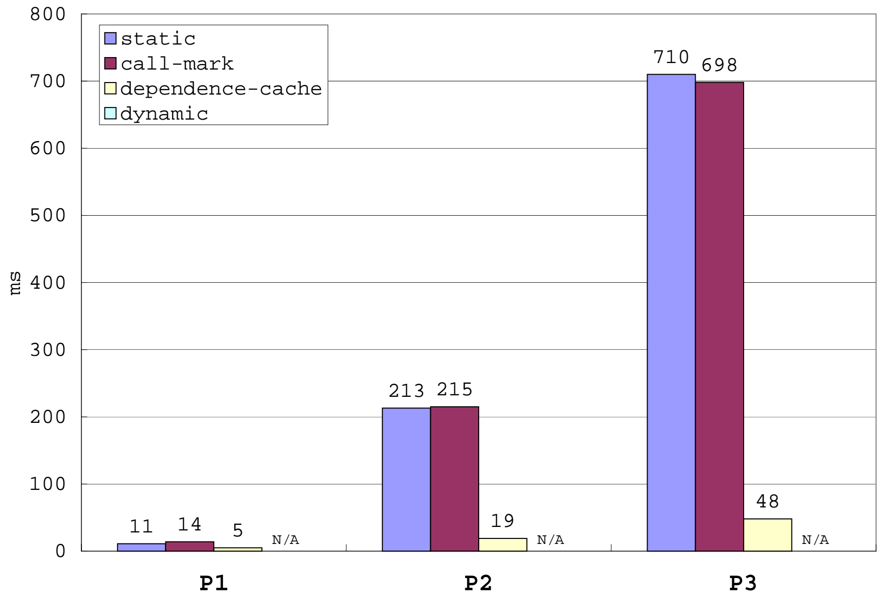


図 3.12: 実行前解析時間 (ms)

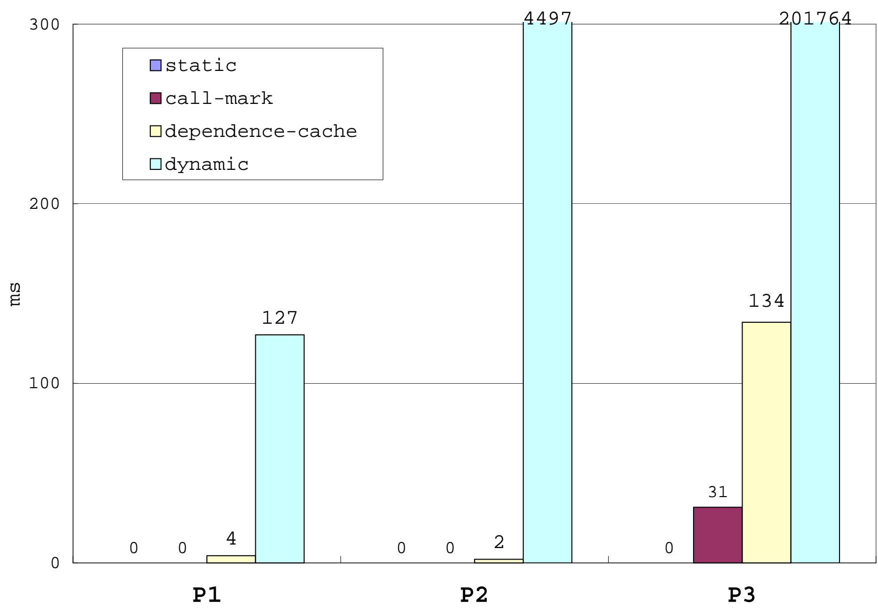


図 3.13: 実行時解析時間 (ms)

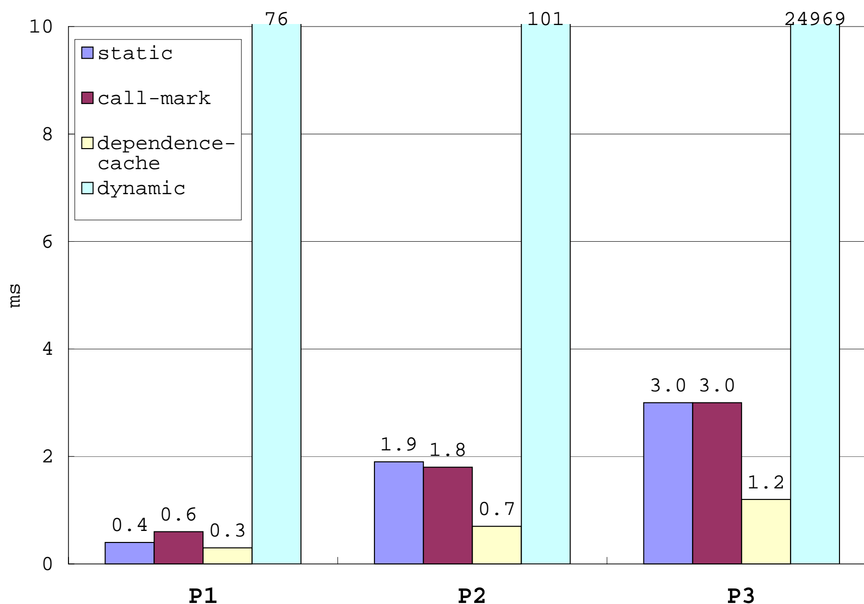


図 3.14: スライス計算時間 (ms)

3.6 考察

3.6.1 実験データの解釈

- スライスサイズ

  図 3.11 はスライスのサイズを示している．依存キャッシュスライスのサイズは静的スライスの 9–71% となっている．

  依存キャッシュスライスのサイズは静的スライスと動的スライスの中間となっている．これは常に静的スライスより小さく (良く)，動的スライスより大きい (悪い)．また，依存キャッシュスライスはコールマークスライスよりも良い (小さい) といえる．これは，コールマークスライシングが静的に解析したデータ依存関係から実行されない部分を削除するだけなのに対し，依存キャッシュスライシングが特定の実行パスに依存するデータ依存関係を反映するためである．依存キャッシュスライスのサイズの削減度合はプログラム $P2$ と $P3$ では，$P1$ に比べ大きい．これは $P1$ が整数やブールなどの単純型の変数しか使用していないのに対し，$P2$ と $P3$ が依存キャッシュスライシングもしくは動的スライシングのみが詳細に解析できる配列変数を使用しているためである．

- 実行前解析

  図 3.12 で示したように，依存キャッシュスライシングでは，静的スライシングに比べ短時間で実行前解析が可能である．これは，静的スライシングが制御依存関係とデータ依存関係の解析を行っているのに対し，依存キャッシュスライシングでは制御依存関係のみの解析を行えばよいためである．

- 実行時解析時間

  図 3.13 で示した実行時解析時間は動的スライシングのオーバヘッドが極端に大きいことを示している．ループを繰り返し実行するなどしてプログラムの実行が長くなると，深刻な性能低下を引き起こす．依存キャッシュスライシングは静的スライシングよりも実行時解析時間が長く，動的スライシングよりも短くなっている．

  ここで示した実行時解析時間はインタープリタ型のシステムに基づいているため，これらはインタープリタのオーバヘッドによって隠蔽されている可能性がある．この問題については 3.6.2 節で議論する．

- スライス計算時間

  図 3.14 で示すように，動的スライシングはスライス結果を収集するのに長時間を必要とする．依存キャッシュスライシングでは，静的スライシングよりも短くなっている．これは，依存キャッシュスライシングが PDG よりも小さい PDG_{DS} を用いているためである．動的スライシングについては，巨大な DDG を探索するのに非常に時間がかかる．

3.6.2 依存キャッシュスライスの適用領域と限界

3.5 節で示した実験はインタープリタ型の実行システムの下で行われた．ここでは，スライシングに対する解析時間の特徴がコンパイラ環境でも同様かどうかを考察する．

C でマージソートのプログラムを書き，実行中に依存キャッシュの更新とデータ依存関係を収集するようにソースプログラムを変更した．このプログラムの実行時間は変更前の 8.6 倍であった．これは依存キャッシュスライシングのオーバヘッドがコンパイラ環境では大きくなることを示している[1]．しかし，単純型変数のデータ依存情報は静的解析で容易

[1] これは，実行中に DDG の探索が必要な動的スライシングではオーバヘッドが許容できないほど大きくなることも示唆している．

に決定できるため，データ依存情報を配列とポインタ変数についてのみ動的に収集することで高速化が可能である．この考えに基づき，プログラムを再び変更した．この部分的依存キャッシュスライシングプログラムの実行時間は変更前の 3.4 倍となり，これは実用的に許容できると考える．

図 3.11 からわかるように，依存キャッシュスライスは常に動的スライスよりも大きくなっている．これは依存キャッシュスライシングは同じ文の複数回の出現を区別せず，キャッシュの中に直近の Def-Use 関係を保持しているからである．

3.6.3　他手法との関連

データフロー情報の収集に注目した動的手法についてはほとんど研究されていない．**3.2.2** 節で紹介した Reduced DDG Method はこれらのうちの一つであり，動的スライシングの正確さと，解析に必要なメモリ空間の削減を実現している．しかし，DDG の重複チェックのために実行オーバヘッドは削減されず，増加する可能性もある．

文献 [11, 12] では，ポインタや配列変数の静的解析について述べられている．これらは不確実性を残している [23] のに対し，依存キャッシュスライシングは動的情報を利用しており，軽量なオーバヘッドでどんな種類の変数に対しても，実用的なスライスの正確さを実現している．

文献 [5] では，静的と動的スライスを一般化させた，**制約スライス**（*Constrained Slice*）が提案されている．これはプログラムの入力の部分集合をプログラムの実行と捉え，この入力制限を用い，依存関係の再計算を行う．しかし，このアプローチの実行効率やその実用性については不明である．

3.7　まとめ

プログラマの注意をソフトウェアの小さな一部分に特化させることは，プログラムのデバッグや保守の効率を改善するために非常に重要である．伝統的なプログラムスライシングは精度と効率の十分なトレードオフを提供していない．

本研究では依存キャッシュスライシングという軽量な準動的スライシング手法を提案した．

スライス結果は同一スライシング基準での動的スライスより大きくなるが，静的スライスより小さくなる．また，これらのスライシングアルゴリズムを実験インタープリタシス

テムに実装し，サンプルプログラムを実行することで提案手法の有効性を確認した．

我々は現在のインタープリタシステムではなくコンパイラシステムに基づくデバッグ環境の構築を予定している．次期システムの機能として，

- 依存キャッシュ機能を持つオブジェクトコードの生成
- 実行時に収集した動的データ依存関係を用いた依存キャッシュスライスの計算
- 動的データ依存関係の表示
- 動的データ依存関係の他アルゴリズムへの提供

を計画している．

第4章　準動的解析を用いたブロック単位スライシング手法

4.1　導入

プログラムのフォールト位置特定を主目的として，さまざまなプログラム解析手法が提案されている．これらの手法は大きく静的解析・動的解析の2つに分類可能である．一般に，静的解析は解析オーバヘッドは小さいが正確なデータ依存関係を求めることが難しく，逆に，動的解析は実際にプログラムを実行した際の依存関係を保持することで正確なデータ依存関係を求めることができるが解析オーバーヘッドが大きくなる．

ポインタ変数や配列変数のデータ依存関係を効率良く解析する手法として，依存キャッシュスライシングがある．依存キャッシュスライシングは静的解析と動的解析の中間的な準動的手法であり，簡単なキャッシュを用いることによって，動的スライシングと比べ小さいオーバヘッドで動的データ依存関係を求めることができる．

依存キャッシュスライシングはインタープリタ型言語に適用した場合，静的スライシングとほぼ同等の実行時間で抽出が可能である．一方，コンパイラ型言語に適用した場合，静的スライシングと比べ約8〜10倍の実行時間を必要とする．これは現実のデバッグ作業に適用することは難しいと考えられる．

そこで本研究では，依存キャッシュスライシングの基本的な考え方を基に，実行時オーバヘッドをさらに削減した手法であるブロック単位スライシングの提案を行う．

以下，4.2で**ブロック単位スライシング**を提案する．また，4.3において効率化及び利便性の向上を目的としてブロック単位スライシングの拡張を行う．次に，4.4においてブロック単位スライシングをC言語プログラムに適用した実験を行う．

4.2　ブロック単位スライシング

4.2.1　概要

依存キャッシュスライシングはスライスの正確さと実行効率のトレードオフを実現した

スライシング手法である．しかし，コンパイラ型言語に適用した場合，その実行オーバヘッドは無視できないものとなる．そこで，実際のデバッグ環境への適用を目的として，依存キャッシュスライシングをさらに効率化させた手法である**ブロック単位スライシング**を提案する．

ブロック単位スライシングは，依存キャッシュスライシングの基本的な考え方である，静的な制御依存解析とキャッシュを用いた動的なデータ依存関係解析の組合せに加え，複数節点を 1 つのブロックとして考え，ブロック単位での依存関係を求める．

以下に，ブロック単位スライシングの計算手順を示す.

STEP1 実行前解析 (ブロック化・静的制御依存解析)

ソースプログラムから，以下の手順により，PDG の部分グラフ PDG_{BL} を静的に生成する．

まず，4.2.2 節に述べるブロック化アルゴリズムに従い，文の集合とブロックとの対応関係を得る．

次に，ブロックに対応する節点を用意し，ブロック間に制御依存関係が存在すれば，対応する節点間に制御依存辺を引く．ただし，データ依存辺は加えない．

STEP2 実行時解析 (動的データ依存関係解析)

対象プログラムをある入力データで実行する．

実行の際，4.2.3 節で示すデータ依存関係抽出アルゴリズムに基づき，動的なデータ依存関係を計算し，PDG$_{BL}$ にデータ依存辺を追加する．プログラム実行が終了した時点で，PDG$_{BL}$ の完成となる．

STEP3 スライス計算

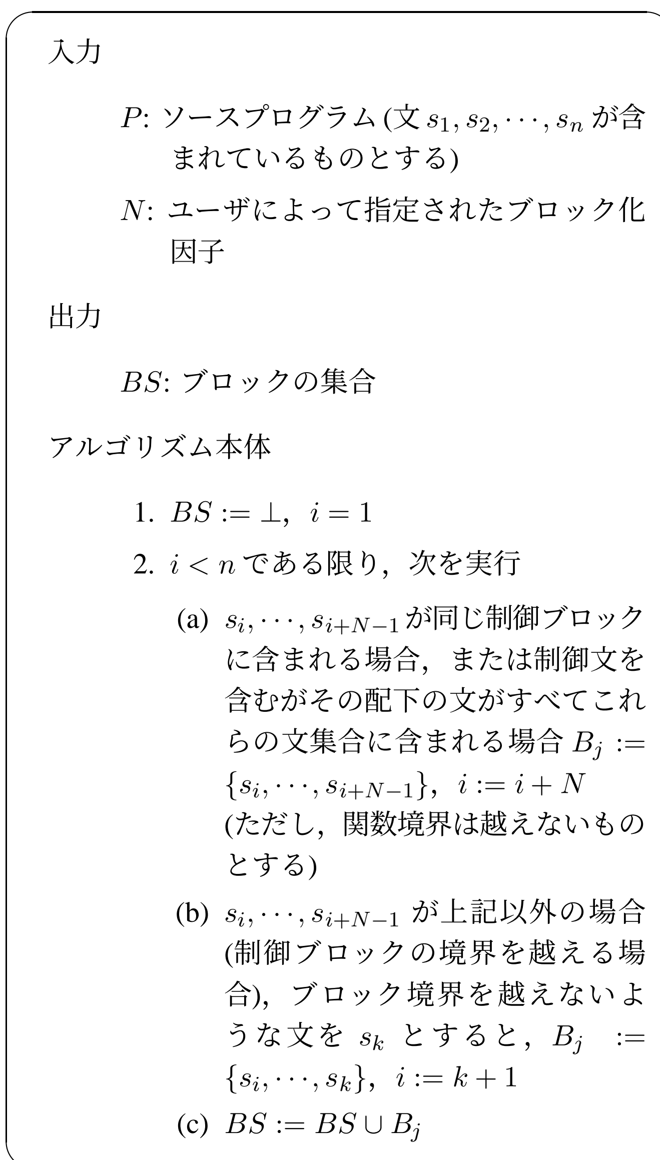
PDG$_{BL}$ を用いて，静的スライシングと同様の方法でスライス計算を行う．

例えば，スライシング基準 (s_c, v) に関するスライスを抽出する場合，まず，s_c に対応する節点から制御依存辺及び v に関するデータ依存辺を逆向きに辿ることで到達可能な節点集合を導出する．そして，この節点集合に対応する文が求めるスライスとなる．

4.2.2 ブロック化アルゴリズム

4.2.1 節の Step 1 で使われるブロック化アルゴリズムを図 4.1 に示す．

このアルゴリズムでは，ユーザの設定したブロック化因子 N に基づき，任意の粒度でのブロック化が可能である．



入力

- P: ソースプログラム (文 $s_1, s_2, \cdots, s_n$ が含まれているものとする)
- N: ユーザによって指定されたブロック化因子

出力

- BS: ブロックの集合

アルゴリズム本体

1. $BS := \perp$，$i = 1$
2. $i < n$ である限り，次を実行
   (a) $s_i, \cdots, s_{i+N-1}$ が同じ制御ブロックに含まれる場合，または制御文を含むがその配下の文がすべてこれらの文集合に含まれる場合 $B_j := \{s_i, \cdots, s_{i+N-1}\}$，$i := i + N$ (ただし，関数境界は越えないものとする)
   (b) $s_i, \cdots, s_{i+N-1}$ が上記以外の場合 (制御ブロックの境界を越える場合)，ブロック境界を越えないような文を s_k とすると，$B_j := \{s_i, \cdots, s_k\}$，$i := k + 1$
   (c) $BS := BS \cup B_j$

図 4.1: ブロック化アルゴリズム

ブロック化の実例

図 4.2 に示すプログラムに対して，ブロック化因子を変化させた際の実例を以下に示す．

```
s1: ...
s2: ...
s3: if (..)  then begin
s4:   ...
s5:   ...
s6:   ...
    end
    else begin
s7:   ...
    end
s8: ...
s9: ...
```

図 4.2: ブロック化サンプルプログラム

ブロック化因子 $N = 2$ の場合

$s1, s2$ は制御構造を含まないため，$B1 = \{s1, s2\}$ となる．次に，$s3$ は制御文であり，かつ $s3, s4$ は $s3$ 配下の文すべてでは無いため制御ブロックの境界を越える場合と見なされ，ブロック $B2 = \{s3\}$ となる．$s3$ 配下の文 $s4, s5$ には制御構造を含まないため，$B3 = \{s4, s5\}$ となる．$s6, s7$ は制御ブロックの境界を越えるため，$B4 = \{s6\}$ となる．$s7, s8$ は制御ブロック境界を越えるため，$B5 = \{s7\}$ となり，$B6 = \{s8, s9\}$ となる．

以上から，

B1: s1, s2

B2: s3

B3: s4, s5

B4: s6

B5: s7

B6: s8, s9

という B1〜B6 の 6 つのブロックが得られる．

ブロック化因子 $N = 5$ の場合

$N = 2$ の場合と同様に，5 つの文を対象にブロック化を進めていく．この結果は

B1: s1, s2

B2: s3

B3: s4, s5, s6

B4: s7

B5: s8, s9

となる．

ブロック化因子 $N = 7$ の場合

$\{s4, \cdots, s7\}$ はすべて $s3$ の if 文の配下であり，$N = 7$ の場合は，これらすべての文が同じブロックに含まれる．そのため，この結果は

B1: s1, s2, s3, s4, s5, s6, s7

B2: s8, s9

となる．

4.2.3 データ依存関係収集アルゴリズム

図 4.3 に **4.2.1** 節の Step 2 で使われるデータ依存関係抽出アルゴリズムを示す．まず，プログラム中で用いられるすべての変数 v に対して，キャッシュ（$C(v)$ と記す）を用意する．大域変数など静的な変数はプログラム実行開始時に，スタック上の Automatic 変数やヒープ中の変数など動的な変数はその変数の割付け時にキャッシュを用意する．プログラムの各実行時点において，$C(v)$ は最も新しく v を定義した文に対応する節点の属するブロックを保持し，文 s において v が使用（参照）された際，$C(v)$ が保持する節点から s の属するブロックに対応する節点 ($B(s)$) に対してデータ依存辺を (すでに存在しなければ) 追加する．一方，s で v が定義された際，$C(v)$ は $B(s)$ に対応する節点に更新する．これらをすべての変数に対して行う．

配列変数や構造体に対しては，すべての要素に対してキャッシュを用意する．例えば，$A[1], A[2], \ldots, A[10]$ という 10 個の要素を持つ配列変数 A は，キャッシュ$C(A[1]), C(A[2]), \ldots, C(A[10])$ を持つ．ポインタ変数 p が文 s において使用された場合，使用された p 自身だけでなく，p を介して間接参照された変数 $p\uparrow$ をも考慮しなければならない．つまり，直接と間接の参照（$C(p) \xrightarrow{p} B(s)$ と $C(p\uparrow) \xrightarrow{p\uparrow} B(s)$ の両者）をデータ依存辺に含めなければ

入力

PDG_{BL}: 部分的に生成されブロック化された PDG

P: 対象プログラム

I: P への入力

作業変数

P 中の各変数 v に対する依存キャッシュ$C(v)$

出力

OUT: 入力 I に対するプログラム P の実行の出力

PDG_{BL}: 完成した PDG

アルゴリズム本体

1. P 中の各静的変数 v に対し, $C(v) := \bot$

   { 各キャッシュの未代入マークによる初期化.
   (注) 動的に割当てられる変数は割当てられた時点でキャッシュを用意し，$\bot$ を代入する． }

2. P が停止するまで以下を繰り返し実行する
   { P を入力 I で最初から停止するまで文ごとに実行 }
   (a) I に関して，P の次の一文 s を実行
   (b) s で使用 (参照) される各変数 u について，$C(u) \neq \bot$ かつ，データ依存辺 $C(u) \xrightarrow{u} B(s)$ が存在しなければ PDG$_{BL}$ に $C(u) \xrightarrow{u} B(s)$ を追加する.
   ここで，$B(s)$ は s を含むブロックを表す.
   (c) s で定義される各変数 w について，$C(w) := B(s)$

図 4.3: データ依存関係収集アルゴリズム

ばならない．また，文 t における $q\uparrow := \ldots$ のようなポインタ変数 q を介した間接的な代入については，キャッシュ$C(q\uparrow)$ が t において更新され，また t において q が使用されたと考える．

動的変数に対しては，個々のインスタンスごとにキャッシュを用意する．また，配列や構造体では，参照されうる要素ごとにキャッシュが必要である．例えば，要素 v_1 と v_2 から構成される構造体 v に対しては，キャッシュ$C(v_1), C(v_2)$ を用意する．各 v_1, v_2 への定義及び参照時の操作は図 4.3 のアルゴリズムと同じである．文 s で構造体 v 全体への定義が行われる際，$C(v_1) := B(s), C(v_2) := B(s)$ を行う．また，s で v 全体の参照が行われる場合は，データ依存辺 $C(v_1) \xrightarrow{v_1} B(s)$ 及び $C(v_2) \xrightarrow{v_2} B(s)$ を PDG_{BL} に追加する．(注: 辺上のラベル v_1, v_2 は静的に決まりうる要素名)

このアルゴリズムでは，プログラム実行中の変数の使用状況に比例したキャッシュ空間と，変数へのアクセス回数，及びブロック数に応じた実行時間のオーバヘッドが必要である．

4.2.4 ブロック単位スライシングの例

図 4.4 に示したサンプルプログラムに対して，入力 $a = 2, b = 3, c = 0$，ブロック化因子 $N = 2$ としたときのブロック単位スライスを図 4.5 に示す．このときのブロックは，

B1: 5

B2: 9

B3: 12, 13

B4: 14, 15

B5: 16, 17

B6: 18

B7: 19

B8: 21

B9: 22

B10: 23

B11: 24

となっている．

```
1   program Square_Cube(input, output);
2   var a, b, c, d : integer;
3   function Square(x : integer) : integer;
4   begin
5     Square := x * x
6   end;
7   function Cube(x : integer) : integer;
8   begin
9     Cube := x * x * x
10  end;
11  begin
12    writeln("Squared Value ?");
13    readln(a);
14    writeln("Cubed Value ?");
15    readln(b);
16    writeln("Select Feature! Square: 0 Cube: 1");
17    readln(c);
18    if c = 0 then
19      d := Square(a)
20    else
21      d := Cube(b);
22    if d < 0 then
23      d := −1 * d;
24    writeln(d)
25  end.
```

図 4.4: サンプルプログラム

```
1   program Square_Cube(input, output);
2   var a, b, c, d : integer;
3   function Square(x : integer) : integer;
4   begin
5     Square := x * x
6   end;
7
8
9
10
11  begin
12    writeln("Squared Value ?");
13    readln(a);
14
15
16    writeln("Select Feature! Square: 0 Cube: 1");
17    readln(c);
18    if c = 0 then
19      d := Square(a)
20
21
22
23
24    writeln(d)
25  end.
```

図 4.5: スライシング基準 ($\{a = 2, b = 3, c = 0\}$, 24, d) に対するブロック単位スライス ($N = 2$)

4.3　ブロック単位スライシングの拡張

ブロック単位スライシングの実行時効率及び利便性を向上させるため，以下を考える．

- スカラ型変数の静的解析

  配列やポインタ型の変数のデータ依存関係を静的に解析することは非常に困難であるが，スカラ型変数については比較的容易に解析することが可能である．

  そこで，PDG_{BL} 作成時に，スカラ型変数のデータ依存関係についても解析し，実行時にはポインタ・配列・構造体などの変数についてのみ依存関係を解析することで，実行時オーバヘッドを削減できる．

- ブロック内局所変数の解析省略

  ブロック内でのみ使用される変数 (関数の局所変数など) は，ブロック外へ依存関係が伝播することは無い．そこで，このような変数についてはキャッシュの作成・データ依存辺の追加を共に省略することにより，実行時間・消費メモリを削減することができる．

- 基本ブロック単位のブロック化

  ブロック化因子の特別な場合として，基本ブロックを一つのブロックとして計算すれば，ユーザが特にブロック化因子を指定する必要の無い場合に有用である．また，制御構造の境界を越えることが無くなり，無駄なブロックが少なくなると考えられる．図 4.4 に示したサンプルプログラムに対し，基本ブロック単位でブロック化を行い，入力 $a = 2, b = 3, c = 0$ としたときのブロック単位スライスを図 4.6 に示す．

4.4　ブロック単位スライシングの評価実験

4.4.1　概要

ブロック単位スライシングの有効性を確認するため実行時間に関する実験を行った．依存キャッシュスライシングのオーバヘッドはコンパイラ型言語の場合に特に大きくなるため，C 言語で記述された 2 つのプログラム $P1$，$P2$ を対象とし，動的データ依存収集の動作を追加した．

```
1   program Square_Cube(input, output);
2   var a, b, c, d : integer;
3   function Square(x : integer) : integer;
4   begin
5     Square := x * x
6   end;
7
8
9
10
11  begin
12    writeln("Squared Value ?");
13    readln(a);
14    writeln("Cubed Value ?");
15    readln(b);
16    writeln("Select Feature! Square: 0 Cube: 1");
17    readln(c);
18    if c = 0 then
19      d := Square(a)
20
21
22
23
24    writeln(d)
25  end.
```

図 4.6: スライシング基準 ($\{a = 2, b = 3, c = 0\}$, 24, d) に対するブロック単位スライス (基本ブロック単位でブロック化)

プログラム $P1$ はマージソートを行うプログラムであり，配列に格納されたデータの整列を行う．プログラム $P2$ はクイックソートを行うプログラムであり，キーとデータの 2 つの構成要素を持つ構造体の配列に対し，キーを基に整列を行う．

これらに対して，静的スライシング，依存キャッシュスライシング，ブロック単位スライシング の実行時解析時間を複数測定し，その平均時間を求めた．なお，ブロック単位スライシングについては，4.3 節で示した拡張を取り入れたアルゴリズムを使用した．

この実験の結果を表 4.1 に示す．

表 4.1: 平均実行時間 (秒)

	$P1$	$P2$
静的スライシング	0.017	0.266
依存キャッシュスライシング	0.141	2.692
ブロック単位スライシング	0.058	1.413

(M-PentiumIII 600MHz CPU with 256MB Memory)

4.4.2　考察

表 4.1 より，依存キャッシュスライシングでは静的スライシングの約 9〜10 倍程度の実行時間となっているのに対し，ブロック単位スライシングでは静的スライシングの約 3〜6 倍程度となっている．また，ブロック単位スライシングは依存キャッシュスライシングの実行時間は約 1/2 に短縮できていることがわかる．

この理由について以下に簡単な考察を行う．

ブロック単位スライシングでは，ブロック化を行うことにより PDG の節点数を減らすことができる．そのため，実行時に追加するデータ依存辺は依存キャッシュスライシングに比べ少なくなり，また，依存辺の追加に関するオーバヘッドも少なくなる．

また，関数内局所変数については，依存キャッシュスライシングではキャッシュを作成し，そのデータ依存辺をも追加しているが，ブロック単位スライシングではその局所変数が単一ブロックの中でしか使用されない場合，解析を行わない．これは，頻繁に呼びだされ制御構造が単純な小さな関数などについて非常に有効であると考えられる．

今回の実験では，実行時間のみに注目し，ブロック単位スライシングの有効性を示したが，本来は実行時間とスライスの正確性が共に達成される必要がある．現状のブロック単位スライシングでは，スライスの粒度がブロック単位であり，スライスサイズは他手法と

比べ大きくなる．

この問題は，スライス計算時に抽出されたブロックの内部を，静的解析によって文単位で抽出することで解決可能である．本アルゴリズムでは，関数境界を越えないようにブロック化を行う．そのため，ブロック内部のデータ依存関係について複雑な解析を行う必要はなく，変数の定義・参照関係及び制御依存関係がわかれば解析が可能である．これは静的スライシングの関数内解析と同じ処理であり，ブロック内部の正確性については静的スライシングと同じとなる．したがって，ブロック単位スライスのサイズは，最大の場合（すべてのブロックがスライスに含まれている場合）においても静的スライシングと同程度となる．

また，本アルゴリズムでは，ブロック化を行った後に制御依存解析を行っている．ブロック単位スライスを一度のみ求める際にはこの方法が適していると考えられるが，ブロック単位スライスのブロック化因子を何度も変更するような場合には，あらかじめ制御依存解析を行った PDG の部分グラフを作成しておき，この PDG に対して節点集約 (ブロック化) を行うことで，実行前解析の時間を短縮できると考えられる．

4.5 まとめ

一般に，配列やポインタなどを含んだプログラムのデータ依存関係を静的に解析するのは非常に困難であり，また，動的に解析するのは非常に実行時オーバヘッドが必要となる．本稿では，ブロック単位スライシングというアルゴリズムを提案した．この方式では，複数の文をブロックとして扱い，ブロック単位で動的にデータ依存関係を求めることで，実行時オーバヘッドを大幅に削減できる．

今後は，ブロック単位スライシングの正確性について実験を行うとともに，4.4.2 節で示したブロック内静的依存解析についても検討，及び，ブロック化因子を様々な数値に変動させた場合の，実行時間と正確性についての考察をも行う．また，静的・動的・依存キャッシュスライシング等の機能を実装したデバッグシステム [39] への実装を行い，実ユーザによるデバッグ実験などの検証も行う予定である．

第5章　むすび

5.1　まとめ

本論文では，プログラム中の文間の依存関係解析に関して，解析を効率化するための3つの手法を提案した．まず，プログラムに変更が加えられた際の再解析を効率化し，インタラクティブなデバッグ作業を実現するプログラム依存グラフの効率的な更新手法を提案した．次に，静的解析情報と動的解析情報を組み合わせた準動的解析手法を用いることにより，現実的な実行時オーバヘッドで高精度のプログラムスライスを抽出できる依存キャッシュスライシング手法を提案した．最後に，依存キャッシュスライシング手法の実行時オーバヘッドをさらに削減したブロック単位スライシング手法を提案した．

また，これらの提案手法についてそれぞれの有効性を確認した．

5.2　今後の研究方針

本論文では，プログラム依存グラフを用いたプログラムスライシングについて効率的な解析手法を提案し，有効性を確認した．

これらの手法はデバッグ作業における作業者の負担を軽減し，デバッグ作業の効率化，ひいてはソフトウェア開発の効率化を行うためのものである．本論文では，それぞれの手法について，テストプログラムを用いて解析を行い，解析時間を中心に有効性を評価した．現実のソフトウェア開発への適用を行うためには，実際の作業者によるデバッグ作業における実験を通した評価を行う必要がある．

提案手法は対象言語を簡単な手続き型言語と仮定している．しかし，現実のデバッグ環境では，ポインタ変数などの**エイリアス**(Alias) の解析が必要となる．提案手法のうち，PDG更新手法は，PDGが既に完成されていることが前提であり，PDG中にエイリアス情報を持たせる，またはエイリアスグラフの概念を導入することで対応可能である．また，依存キャッシュスライシング及びブロック単位スライシングはデータ依存関係解析に動的情報を用いているため，容易に対応可能である．

一方，近年オブジェクト指向言語が広く普及してきている．オブジェクト指向言語では継承 (Inheritance) や動的束縛 (Dynamic Binding) などの概念が導入されているため，従来の手法では解析を行うことができない．そのため，オブジェクト指向言語の解析に関する研究 [4, 32, 36] が行われている．今後は本論文で提案した手法のオブジェクト指向言語への適用を検討する．

参考文献

[1] Agrawal, H. and Horgan, J. : “Dynamic Program Slicing”, *SIGPLAN Notices*, Vol.25, No.6, pp. 246–256 (1990).

[2] Aho, A. V., Sethi, R., and Ullman, J. D.: “Compilers: Principles, Techniques, and Tools”, *Addison Wesley*, Massachusetts (1986).

[3] Canfora, G., Cimitile, A., and De Lucia, A.: “Conditioned Program Slicing”, *Information and Software Technology*, Vol. 40, no. 11/12, November 1998, pp. 595-607 (1998).

[4] 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).

[5] Field, J., and Ramalingam, G.: “Parametric Program Slicing”, *Proc. of 22nd ACM Symposium on Principles of Programming Languages*, pp. 379–392, San Francisco, USA, January (1995).

[6] Fritzson, P., Gyimothy, T., Kamkar, M. and Shahmehri, N.: “Generalized aigirithmic debugging and testing”, *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, Vol. 26, No. 6, pp. 317–326 (1991).

[7] Fritzson, P., Shahmehri, N. and Kamkar, M.: “Generalized Algorithmic Debugging and Testing”, *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 4, pp. 303–322 (1992).

[8] Gallagher, K. B. and Lyle, J. R.: “Using Program Slicing in Software Maintenance”, *IEEE Transactions on software engineering*, Vol. 17, No. 8, pp. 751–761 (1991).

[9] Gupta, R., Soffa, M.L., and Howard, J. : “Hybrid Slicing: Integrating Dynamic Information with Static Analysis”, *ACM Transaction on Software Engineering and Methodology*, Vol. 6, No. 4, pp. 370–397 (1997).

[10] Harman, M. and Danicic, S.: “Amorphous program slicing”, *IEEE International Workshop on Program Comprehesion (IWPC’97)*, pp. 70-79, May 1997, Dearborn, Michigan, USA, (1997).

[11] Hind, M., Burke, M., Carini, P., and Choi, J.:“Interprocedural Pointer Alias Analysis”, *ACM Trans. on Programming Languages and Systems*, Vol.21, No. 4, pp. 848–894 (1999).

[12] Horwitz, S. Pfeiffer, P., and Reps, T.:“Dependence Analysis for Pointer variables”, *Proceedings of SIGPLAN ’89 Conference on Programming Language Design and Implementation*, pp.28–40, SIGPLAN Notices Vol. 24, No. 6 (1989).

[13] Horwitz, S. and Reps, T.: “The Use of Program Dependence Graphs in Software Engineering”, *Proceedings of the 14th International Conference on Software Engineering*, Dallas, Texas, Association for Computing Machinery, pp. 392–411 (1992).

[14] Horwitz, S., Reps, T. and Binkley, D.: “Interprocedural Slicing Using Dependence Graphs”, *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pp. 26–60 (1990).

[15] Korel, B., and Laski, J. : “Dynamic Program Slicing”, *Information Processing Letters*, Vol.29, No,10, pp. 155–163 (1988).

[16] Kusumoto, S., Nishimatsu, A., Nishie, K. and Inoue, K. : “Experimental Evaluation of Program Slicing for Fault Localization”, *Empirical Software Engineering*, Vol. 7, No. 1, pp. 49-76 (2002).

[17] Lian, L., Aizawa, M., Inoue, K. and Torii, K. : “Development of Program Difference Tool Based on Tree Mapping”, *IEICE Transactions on Information and Systems*, Vol. E78-D, No. 10, pp. 1261-1268 (1995).

[18] Liang, D., and Harrold, M. J., : "Efficient Points-To Analysis for Whole-Program Analysis", *Proc. of 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp.199–215, Toulouse, France (1999).

[19] Lyle, J. R. and Weiser, M.: "Automatic program bug location by program slicing", *Proceedings 2nd International Conference on Computers and Applications*, pp. 877–883 (1987).

[20] Ning, J. Q., Engberts, A., Kozaczynski, W. V. : "Automated Support for Legacy Code Understanding", *Communications of the ACM*, Vol. 37, No. 5, pp.50–57, May (1994).

[21] Nishimatsu, A., Jihira, M., Kusumoto, S. and Inoue, K. : "Call-Mark Slicing: An Efficient and Economical Way of Reducing Slice", *Proceedings of The 21st International Conference on Software Engineering*, pp.422-431, Los Angeles, CA, USA (1999).

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

[23] Ramalingam, G.: The Undecidability of Aliasing, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1467–1471 (1994).

[24] Reasoning Systems, Palo Alto, CA, *REFINE User's Guide* (1990).

[25] Reasoning Systems, Palo Alto, CA, *DIALECT User's Guide* (1990).

[26] Reasoning Systems, Palo Alto, CA, *INTERVISTA User's Guide* (1991).

[27] Ryder, B. G. and Pall, M. C.: "Incremental Data-Flow Analysis Algorithm", *ACM Transactions on Programming Languages and Systems*, Vol. 10, No. 1, pp. 1–50 (1988).

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

[29] Shimomura, T.: "Bug Localization Based on Error–Cause–Chasing Methods", *Transactions of Information Processing Society of Japan*, Vol. 34, No. 3, pp. 489–500 (1993).

[30] Sommerville, I. : "Software Engineering", Addison Wesley (2000).

[31] Takada, T., Ohata, F. and Inoue, K.: "Dependence-Cache Slicing: A Program Slicing Method Using Lightweight Dynamic Information", *10th IEEE International Workshop on Program Comprehension*, pp. 169–177, June 2002, Paris, France (2002).

[32] Tonella, P., Antoniol, G. Flutem, 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*, pp. 433–443 (1997).

[33] Weiser, M.: “Program Slicing”, *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, pp. 439–449 (1981).

[34] 會澤 実, 練　林, 荻原 剛志, 井上 克郎, 鳥居 宏次: “構文木の比較によるプログラム開発履歴分析ツールの試作”, 情報処理学会ソフトウェア工学研究会研究報告 SE94–100, pp. 55–62 (1994).

[35] 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, pp. 11–22 (1995).

[36] 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, pp. 1–11 (2001).

[37] 鍛冶 武志: “プログラムの部分的変更にともなう依存解析グラフの更新手法”, 大阪大学基礎工学部情報工学科 特別研究報告 (1996).

[38] 小林 孝規: “入出力変数の制限情報を利用したプログラム簡素化ツールの試作”, 大阪大学基礎工学部情報工学科 特別研究報告 (1995).

[39] 佐藤 慎一, 飯田 元, 井上 克郎: “プログラムの依存関係解析に基づくデバッグ支援システムの試作”, 情報処理学会論文誌, Vol. 37, No. 4, pp. 536–545 (1996).

[40] 下村 隆夫: “プログラムスライシング技術と応用”, 共立出版 (1995).

[41] 高田 智規, 佐藤 慎一, 飯田　元, 井上 克郎: “ソースコード解析ツール開発支援システムの試用”, 電子情報通信学会論文誌 D-I, Vol. J80-D-I, No. 3, pp. 317–318 (1997).

[42] 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 D-I, Vol. J81-D-I, No. 3, pp. 253–260 (1998).

[43] 高田 智規, 大畑 文明, 芦田 佳行, 井上 克郎: “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 D-I, Vol. J85-D-I, No. 2, pp. 228–235 (2002).

[44] 高田 智規, 井上 克郎: “制限された動的情報を用いたブロック単位スライシング手法の提案”, 電子情報通信学会論文誌, (条件付採録).

[45] 高谷 暢之: “入出力の制限情報を利用したプログラム簡素化手法の提案”, 修士論文, 大阪大学基礎工学部情報工学科 (1994).

[46] 二村 良彦ほか: “プログラム変換”, 共立出版, chapter 4, pp. 63–79 (1987).

[47] 西松 顕, 楠本 真二, 井上 克郎: “保守プロセス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. 8, pp. 1121–1123 (1999).

[48] 西松 顕, 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, pp. 1336–1344 (1999).

[49] 西松 顕: “プログラム文の依存関係を利用した開発支援システムの機能拡張〜ダイナミックスライサの実現〜”, 大阪大学基礎工学部情報工学科 特別研究報告 (1997).

[50] 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, pp. 1990–1998 (1998).