# 実行履歴　利用した　　スプログラムの部分解析に基づく スライス抽出技法の提案

芦田 佳行　大畑 文明　井上 克郎

大阪大学大学院基礎工学研究科情報数理系専攻
〒 560-8531 大阪府豊中市待兼山町 1-3
Phone: 06-6850-6571　Fax: 06-6850-6574
E-mail: {asida, oohata, inoue}@ics.es.osaka-u.ac.jp

**らまし**

本研究では，静的解析と動的解析を組み合わせたプログラムスライス計算法を 4 つ提案する．(1)Statement-Mark スライス：文の実行履歴を用いて，実行されなかった文 (スライスに不要な文) を取り除く．(2) プログラムの部分解析：関数の実行履歴を用いて，実行された文だけを静的に解析することで依存解析のコストを削減する．(3) 動的なデータ依存解析：制御依存解析を静的に，データ依存解析を動的に行うことで，より正確なデータ依存関係を抽出する．(4) 配列，ポインタ解析：ポインタ，配列のデータ依存解析のみを動的に，その他の依存解析を静的に行うことで，(3) と比較して実行時間の削減をめざす．

**ワ　ド**　プログラムスライス，静的解析，動的解析，実行履歴

# Proposal of Slicing Algorithms Using Static and Dynamic Analysis Information

Yoshiyuki Ashida, Fumiaki Ohata and Katsuro Inoue

Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
Phone: +81-6-6850-6571　Fax: +81-6-6850-6574
E-mail: {asida, oohata, inoue}@ics.es.osaka-u.ac.jp

**Abstract**

In this paper, we propose four slicing algorithms using static and dynamic analysis information. (1) Statement-Mark Slice : removes the unnecessary statements using a execution history of the statements. (2) Partial Program Analysis : reduces the static analysis cost using invocation history of procedures. (3) Dynamic Data Dependence Analysis : extracts precise data dependence relations using dynamic data dependence analysis. (4) Array and Pointer Analysis : improves the efficiency of analysis(3) by dynamically analyzing pointer and array variables only.

**Key words**　program slice，static analysis，dynamic analysis，execution history

# 1 まえがき

プログラムスライス [12] とは，直観的には，ある文 s のある変数 v の値に影響を与え得る文の集合である．プログラムスライスを計算するには，プログラム文間の依存関係を把握する必要がある．

プログラムスライスは，デバッグ，テスト，プログラム解析，プログラム理解，プログラム統合等に有効であると考えられている [3, 4, 12]．また我々もデバッグ，保守にプログラムスライスが有効であることを確認した [2, 6, 7].

プログラムスライスは，静的スライスと動的スライスの 2 種類に大別される．前者は，ソースプログラムの解析し，全ての入力データを考慮した依存関係を抽出する．後者は，実際に入力データを与えて実行することによって，特定の入力において生じる依存関係をのみを抽出する．静的スライスは，全ての入力を考慮するため，不必要な情報を含むことがあり，その結果，スライスサイズが大きくなってしまう．動的スライスは，実行系列 (実際に実行された文の並び) を扱うため，スライスサイズを減らすことができるが，全ての実行系列を保存しそこから依存関係を抽出するために，多大な時間，空間コストを必要とする．

スライスの計算に必要なコストとスライスサイズはトレードオフの関係にあり，さまざまな研究がなされている．

本研究では，動的情報と静的解析を組み合わせることで，スライス計算に必要なコストを削減する手法を提案する．また，ポインタや配列といった，静的解析が困難であるものについても，動的情報を利用した解析方法を提案する．以降，2 では，静的スライス，動的スライスについて述べる．3 では，Statement-Mark スライスについて述べる．4 では，実行履歴を使ったプログラムの部分解析法について述べる．5 では，動的にデータ依存関係解析を行う方法について述べる．6 では，動的にポインタ，配列の解析を行う方法について述べる．最後に 7 で，まとめと今後の課題について述べる．

# 2 プログラムスライス

## 2.1 静的スライス

今，ソースプログラム $p$ 中の文 $s_1$，$s_2$ について考える．次の条件をすべて満たすとき，文 $s_1$ から文 $s_2$ への制御依存 $(ControlDependence, CD)$ があるという．

- 文 $s_1$ が条件文である．
- 文 $s_2$ が実行されるかどうかは，文 $s_1$ の結果に依存する．

この関係を $s_1 \dashrightarrow s_2$，もしくは $CD(s_1, s_2)$ と表す．

また，以下の 3 つの条件をすべて満たすとき，文 $s_1$ から文 $s_2$ へ変数 $v$ に関してデータ依存 $(DataDependence, DD)$ があるという．

- 文 $s_1$ で変数 $v$ を定義している．
- 文 $s_2$ で変数 $v$ を参照している．
- 文 $s_1$ から文 $s_2$ への実行可能なパスが少なくとも一つは存在し，さらに，そのパスにおいて文 $s_1$ から文 $s_2$ 間に変数 $v$ を定義している文が存在しない．

この関係を $s_1 \overset{v}{\dashrightarrow} s_2$，もしくは $DD(s_1, v, s_2)$ と表す．

ソースプログラムに含まれるこれら 2 つの依存関係を表すため，プログラム依存グラフ $(ProgramDependenceGraph, PDG)$ が用いられる．PDG の各節点は，プログラムに含まれる代入文，入出力文，条件判定文，手続き呼出し文などの各文を表し，辺は 2 つの文間の依存関係を表す．図 1 の Pascal のソースプログラムに対応する PDG は図 2 のようになる．

プログラム中の文 $s$ の変数 $v$ に関する静的スライス (組 $(s, v)$ をスライシング基準と呼び，どの文に関してスライスを計算するのかを表す．) は，スライシング基準に対応する節点から 制御依存辺，データ依存辺を逆向きに辿って到達できる節点に

```
 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: ソースプログラム

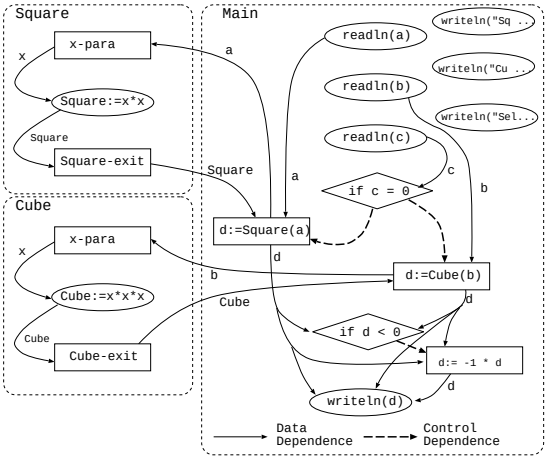


図 2: プログラム依存グラフ

対応する文の集合となる¶1.

例えば，図 1 のプログラムの文 24 の変数 $d$ をスライシング基準として静的スライスを計算すると，図 2.1 のように write 文 (12，14，16 行) を除いたすべての文が含まれる.

### 2.2 動的スライス

静的スライスの計算は，ソースコードを対象として依存関係を計算し，スライスを抽出していたが，動的スライスの計算では，依存関係を計算する対象は実行系列である．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 $p$ 番目の文の実行のことを実行時点 $p$ 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．次の条件をすべて満たすとき，実行時点 $r_1$ から実行時点 $r_2$ への動的制御依存 ($DynamicControlDependence$, $DCD$) があるという.

- 実行時点 $r_1$ が条件文である.
- 実行時点 $r_2$ が実行されるかどうかは，実行時点 $r_1$ の結果に依存する.

次の条件をすべて満たすとき，実行時点 $r_1$ から実行時点 $r_2$ へ変数 $v$ に関して動的データ依存 ($DynamicDataDependence$, $DDD$) があるという.

- 実行時点 $r_1$ で変数 $v$ を定義している.
- 実行時点 $r_2$ で変数 $v$ を参照している.
- 実行時点 $r_1$ から $r_2$ の間に変数 $v$ を定義している実行時点が存在しない.

¶1データ依存辺を辿るには，まず $v$ に関する辺を辿り，以降，影響を与える変数の辺のみを推移的に辿る.

```
 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
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 3: 24 行目変数 $d$ に関する静的スライス

動的スライスのスライシング基準 ($x$，$r$，$v$) は，入力 $x$，実行時点 $r$，変数 $v$ の 3 つからなる．スライシング基準 ($x$，$r$，$v$) に対応する節点から動的制御依存辺，動的データ依存辺を逆向きに辿って到達できる節点に対応する文の集合となる.

図 1 のプログラムにおいて，スライシング基準が入力 ($a = 2, b = 3, c = 0$)，実行時点 13(24 行目の文を実行した時点)，変数 $d$ である動的スライスの計算結果を図 4 に示す.

### 2.3 静的スライスと動的スライスの特徴

静的スライスの計算は，プログラムを実行せずソースプログラムの依存解析を行い，得られた PDG 上で行う．PDG を構築する際，制御依存関係はプログラム構造を調べることで容易に求まる．また，データ依存関係はデータフロー方程式 [1] を解くことで求まる．静的スライス計算の複雑さは，その評価の基準によって変わるが，現実には比較的短い時間で計算できる [11, 9]．しかし，すべての実行可能なパスについて考えているため，静的スライスに元のプログラムの大部分が含まれてしまい，デバッグやプログラム理解等の効率向上は期待できない場合がある.

一方，動的スライスは特定の入力でプログラムを実行して得られる実行系列から計算するため，その実行に関係のない部分はスライスに含まれない．つまり，一般的に抽出されるサイズは静的スライ

```
 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 4: 入力データ $(a = 2, b = 3, c = 0)$,24 行目変数 $d$ に関する動的スライス

スよりも小さくなる．

特定のテストデータに対して不具合が発生し，その欠陥の原因をプログラム中で捜すことを考える．静的スライスでは，そのテストデータでは実行されていないパスに関する依存関係まで計算してしまうため，不具合の原因とは全く関係のない部分までスライスに含まれてしまうが，動的スライスはテストデータの実行に関連する部分の中から計算するため，不具合に関係のない部分がスライスに含まれるようなことはない．

このように特定のテストデータで存在するフォールト位置特定を行う場合には，動的スライスは非常に有効となる．

動的スライスの計算には，プログラム実行前の解析は必要ではないが，実行中に動的データ依存関係と動的制御依存関係の情報を主記憶等に記憶しなければならない．このため，動的スライスの計算には多くの記憶領域と計算時間を要する．また，動的スライスを抽出する対象となる実行系列はプログラムが実行した文の数に比例することから，入力データによっては非常に大きくなるために，抽出する時間も非常に要することがある．

## 3 Statement-Mark スライス

本節では，文の実行履歴を用いた Statement-Mark スライスを提案する．

### 3.1 Call-Mark スライス

我々は，静的スライスと動的スライスの中間に位置付けされるものとして，Call-Mark スライス [5, 8] を提案している．Call-Mark スライスの計算手順は，次のようになる．

(1) 静的スライスと同じ方法で PDG を作成する．
(2) プログラムに入力データを与えて実行させる．その際，関数の実行履歴 (各関数が呼び出されたか否か) を取得する．
(3) 取得した関数の実行履歴を用いて実行されなかった文の一部を特定し，それらの文をスライスから取り除く．

関数の実行履歴という簡単な動的情報を用いることで，動的スライスの欠点であった実行時のオーバーヘッドを大幅に減らしている．

### 3.2 Statement-Mark スライス

Call-Mark スライスでは，動的情報として関数の実行履歴を用いているが，これを文の実行履歴にすれば，取得すべき情報は多くなるが，より正確なスライスを抽出することができると考えられる．そこで，文の実行履歴を用いた Statement-Mark スライスを提案する．

#### 3.2.1 アルゴリズム

基本的には Call-Mark スライスと同じ手順で計算を行うことになる．

(1) 静的な依存関係解析を行い，PDG を構築する．
(2) 入力データを与えて実行する．ある文が実行されるごとに，対応する PDG の節点に対してフラグを立てる．
(3) PDG において，スライシング基準に対応する節点から依存辺を逆向きに探索する．実行されていない節点に着いた時は，(その節点も含めて) その節点以降の探索を止め，他の分岐を探す．

#### 3.2.2 例

図 1 のプログラムについてのスライシング基準が入力 $(a = 2, b = 3, c = 0)$，文 24 の変数 $d$ である Statement-Mark スライスの計算結果を図 5 に示す．

動的スライスと全く同じ結果が得られているが，一般に，動的スライスは Statement-Mark スライスの部分集合である．

#### 3.2.3 考察

Statement-Mark スライスを，実際にスライシングツール [9] に実装し，スライスサイズと実行時間

```
 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 5: 入力データ $(a = 2, b = 3, c = 0)$,24 行目変数 $d$ に関する Statement-Mark スライス

表 1: スライスサイズ (行)

| program | static | Call-Mark | Statement -Mark | dynamic |
|---|---|---|---|---|
| P1 | 27 | 22 | 15 | 14 |
| P2 | 176 | 155 | 141 | 139 |
| P3 | 324 | 166 | 148 | 50 |

(Pentium-II 300MHz with 256MB Memory)

を計測した．(表 1，表 2)

Call-Mark スライスと比較して，スライスサイズが 10∼30%の減少が見られ，実行時間は 15∼30%増加している．
静的スライスと比較すると，スライスサイズが 20∼55%の減少が見られ，実行時間は 30∼60%増加している．

## 4 ソースプログラムの部分解析

静的スライスにおいては，入力データを考慮せずに全ての可能な実行パスを想定して依存関係解析を行っている．しかし 1 度入力データを与えて実行すれば，その実行において使用されていない文が特定できる．使用されなかった部分を解析しないことによって，

(1) スライスのサイズの減少
(2) PDG 構築のコストの削減

が期待できる．

表 2: 実行時間 (ms)

| program | static | Call-Mark | Statement -Mark | dynamic |
|---|---|---|---|---|
| P1 | 38 | 47 | 62 | 87 |
| P2 | 48 | 53 | 62 | 903 |
| P3 | 4,064 | 4,104 | 5,318 | 31,635 |

(Pentium-II 300MHz with 256MB Memory)

### 4.1 部分解析法

Call-Mark スライス計算法を改良した，部分解析法を提案する．

ある実行 $E$ において使用された関数の集合を $UsedF(E)$ とする．[11] のアルゴリズムは，

(1) 関数内の依存関係解析
(2) プログラム全体の依存関係解析

となっている．そこで，$f \notin UsedF(E)$ である関数 $f$ の依存解析は行わなければよい．また，プログラム全体の解析の際には，関数呼び出し文の出現による PDG の接続等の作業を行わない．

この部分解析法は，関数の実行履歴を利用している．このため，結果として Call-Mark スライスと等しいスライスが得られる．また，文の実行履歴を利用することで，さらに効率良い依存関係解析が期待できる．その際に得られるスライスは，Statement-Mark スライスと等しい．

### 4.2 適用例

部分解析法は，何らかの形で実行されていない文を調べ，その部分を解析しないことによって，そのコストを削減するというものである．

そこで，関数の実行履歴を用いることで，ソースプログラムの何% が不要になるかを調べた．

対象となるプログラムとして，ftp を選んだ．このプログラムは，ソースプログラムが 6294 行で，定義された関数が 130 個である．ftp を実際に実行し，使われなかった関数をソースプログラムから削除し，解析が必要な部分がどれだけかを調べた．

2 回の実行を行い，データを取得した．

1 回目は簡単な実行で，ls でファイルを確かめ，get でファイルを獲得し，bye で終了するというものであった．この実行では，30 個の関数が使用され，必要な部分は 3227 行 (51.3%) であることが分かった．

2 回目は 73 個のコマンドを実行するものであった．この実行では，87 個の関数が使用され，必要な部分は 4838 行 (76.9%) であると分かった．

```
1  a[0]:=0;
2  a[1]:=1;
3  a[2]:=2;
4  readln(c);
5  b:=a[c]+5;
6  writeln(b);
```

図 6: 配列を含むプログラム

```
1  a=2;
2  b=1;
3  c=&a;
4  d=&c;
5  *c=5;
6  **d=b;
7  printf("%d",a);
8  printf("%d",**d);
```

図 7: ポインタを利用するプログラム

### 4.3 Call-Mark スライス，Statement-Mark スライス，部分解析法の特徴

関数の実行履歴を用いた部分解析法と Call-Mark スライス，文の実行履歴を用いた部分解析法と Statement-Mark スライスは，共にスライスのサイズとしては同じになる．

部分解析法を用いれば，他の 2 つのスライス技法と比べて，依存関係解析のコストを減らすことができる．しかし，入力データに依存した PDG を構築するため，別の実行を行った際には，新たに PDG を再構築する必要がある．

## 5 動的なデータ依存関係解析

### 5.1 配列，ポインタの解析

配列変数を静的に解析する際に，配列の添字の値を把握するのは困難であり，それゆえ不要に多くの依存関係を抽出してしまうことがある．

図 6 は，配列を含んだ簡単なプログラムである．文 6 の変数 $c$ の値が分からないため，文 6 の配列 $a$ が文 1~3 全てにデータ依存していると解析される．

ポインタ解析では，ポインタによる alias によって，陽には表れないデータ依存関係が発生する．その依存関係を知るために，各ポインタが何を指しているかを把握しておく必要がある．point-to グラフを使った手法 [10] などが提案されているが，多大なコストを必要とするものが多く，静的に解析する方法には限界がある．

図 7 は，ポインタを利用する簡単なプログラムである．文 7 の変数 $a$ は文 6 によって定義されているが，こういった依存関係は簡単には解析できない．

### 5.2 方針

配列，ポインタの静的な解析は，コストがかかる上に完全な解析は期待できない．そこで，実行中にデータ依存関係を抽出する方法が考えられる．実行中は，配列の添字，ポインタの値を把握するのは簡単である．動的スライスとの差異として，次の 2 つが挙げられる．

- 制御依存関係は静的に解析する．
- 実行系列を保存することはしない．

これによって，動的スライス抽出に比べて実行時間の短縮を図る．

### 5.3 概要

実行中に，ある文 $s$ である変数 $v$ が参照された時，$v$ がどの文 ($t$) で定義されたかが分かれば，$DD(t, v, s)$ どいうデータ依存関係があることが分かる．逆に言えば，$v$ を定義してる文さえ分かればデータ依存関係を知ることができる．

そこで，全ての変数に対して，その値を定義したのはどの文か，という情報を持たせておき，その変数の参照があった場合には，その情報からデータ依存関係を把握する．

また，得られたデータ依存関係は各文に持たせる．各文 $s$ は集合 $DDS(s)$ を持つとする．この $DDS(s)$ の要素は，2 つの要素からなる組であり，(“依存関係の原因となる変数 $v$”，“$s$ がデータ依存している文”) となっている．

### 5.4 アルゴリズム

ある実行時点において，変数 $v$ を最後に定義した文を $DefS(v)$ とする．ここで $v$ は，動的に生成される変数も含めた全ての変数を考える．配列では，各要素にも $DefS$ を考える．

**(1)**ある文 $s$ を実行する前に，$DDS(s) = \phi$ とする．

**(2)**入力データを与えてプログラムの実行を行う．

今，文 $s$ が実行されたとする．

- 文 $s$ で変数 $v$ が参照[¶2] された場合，$DDS(s) \leftarrow DDS(s) \cup (v, DefS(v))$．とする．
- 文 $s$ で変数 $v$ が定義された場合，$DefS(v) = s$ とする．

[¶2]ポインタ変数が出現した場合，何が参照されたかを判断する際に注意が必要となる．例えば，代入文の右辺や出力文などに $**v$ という 2 階のポインタが現れた場合，$v$，$*v$，$**v$ が参照されたと考える．また，代入文の左辺に現れた変数は普通参照されたとは考えないが，例えば $*v$ という一階のポインタが現れた場合は，$v$ を参照したと考える．

表 3: 図 6 のプログラムにおける，各実行時点での $DefS$ の推移

| 実行文 | $a[0]$ | $a[1]$ | $a[2]$ | $b$ | $c$ |
|---|---|---|---|---|---|
| 1 | 1 | | | | |
| 2 | | 2 | | | |
| 3 | | | 3 | | |
| 4 | | | | | 4 |
| 5 | | | | 5 | |
| 6 | | | | | |

この作業によって得られた $DDS(s)$ は，全てのデータ依存関係を表しており，$DDS(s) = \{(v,t)|DD(t,v,s)$ が成り立つ $\}$ となっている．

**(3)**集合 $DDS$ を使って PDG を構築する．文 $s$ の解析において，

- $DDS(s) \neq \phi$ の間，次の操作を繰り返す．
  - $DDS(s) \leftarrow DDS(s) - \{(v,t)\}$.
  - データ依存辺 $t \overset{v}{\dashrightarrow} s$ を引く．
- $s$ が制御文であれば，その制御文内の全ての文 $t$ に対して，制御依存辺 $s \dashrightarrow t$ を引く．

### 5.5 適用例

#### 5.5.1 図 6 のプログラム

各実行時点における $DefS$ の推移を表 3 に表す．

文 1，2，3 では，それぞれ変数 $a[0], a[1], a[2]$ が定義されているので，文 3 を実行した時点で $DefS(a[0]) = 1, DefS(a[1]) = 2, DefS([2]) = 3$ となる．

文 4 では，$c$ を定義しているので，$DefS(c) = 4$ となる．

文 5 では，$a[0], c$ を参照しているので，$DDS(5) \leftarrow DDS(5) \cup (a[0], DefS(a[0])) \cup (c, DefS(c))$．つまり，$DDS(5) = \{(a[0],1), (c,4)\}$ となる．

#### 5.5.2 図 7 のプログラム

各実行時点における $DefS$ の推移を表 4 に表す．

文 1, 2, 3, 4 では，それぞれ変数 $a, b, c, d$ が定義されているので，文 4 を実行した時点で $DefS(a) = 1, DefS(b) = 2, DefS(c) = 3, DefS(d) = 4$ となる．

文 5 では，$c$ を参照しているので，$DDS(5) \rightarrow \phi \cup (c, DefS(c))$．つまり，$DDS(5) = (c,3)$ となる．また，$*c$ を定義しているので，$DefS(*c) = 5 (DefS(a) = 5)$ となる．

文 6 では，$b$，$d$，$*d$ を参照しているので，$DDS(6) = \{(b,2), (d,4), (*d,3)\}$ となる．
さらに，文 6 では $**d$ を定義しているので，$Def(**d) = 6 (DefS(a) = 6)$ となる．

文 7 では，$a$ を参照しているので，$DDS(7) = \{(a,6)\}$ となる．

表 4: 図 7 のプログラムにおける，各実行時点での $DefS$ の推移

| 実行文 | $a$ | $b$ | $c$ | $d$ |
|---|---|---|---|---|
| 1 | 1 | | | |
| 2 | | 2 | | |
| 3 | | | 3 | |
| 4 | | | | 4 |
| 5 | 5 | | | |
| 6 | 6 | | | |

文 8 では，$d, *d, **d$ を参照しているので，$DDS(8) = \{(d,4), (*d,3), (**d,6)\}$ となる．

### 5.6 考察

本アルゴリズムでは，全ての変数に対して $DefS$ を考える必要がある．このため，大きなメモリ空間を必要とする可能性がある．また，動的スライスほどではないとしても，実行時間が大きくなってしまう．

そこで，配列やポインタの解析のみを動的に行い，その他の解析は静的に行う方法が考えられる．

## 6 配列，ポインタ解析

### 6.1 概要

ここで提案する手法は，配列，ポインタのデータ依存解析のみを動的に行うものである．5 節の手法よりも実行時間の短縮，空間コストの削減が期待できる．

本アルゴリズムで仮定している言語は，ポインタで指すことができる変数は $new$ などによって動的に生成された変数と限る (例：Pascal)．

上記の仮定により，動的に生成された変数，配列の各要素についてのみ $DefS$ を考えればよい．

### 6.2 アルゴリズム

本アルゴリズムと 5 節のアルゴリズムの差異のみを述べる．

- 配列の個々の要素と動的に生成された変数に対してのみ，$DefS(a[i])$ を考える．
- PDG 構築の際は，ポインタ，配列に関する依存関係解析が必要な場合のみ $DDS$ を使った解析 (5 節) を行い，その他は静的な依存関係解析を行う．

## 7 まとめ

本研究では，動的情報と静的情報を組み合わせた手法を提案した．

(1) Statement-Mark スライス (3 節)
(2) プログラムの部分解析法 (4 節)

(3) 動的なデータ依存関係解析法 (5 節)
(4) 動的なポインタ，配列解析法 (6 節)

(1),(2) に関しては，コンパイラ (インタプリタ) のわずかな変更で動的情報を取得することが可能である．しかし，(3),(4) に関しては，実行中に依存関係解析を行うため，特別なコンパイラ (インタプリタ) が必要となる．

今後の課題として，(2)〜(4) を実装し，評価することが挙げられる．

## 参考文献


[1] Aho, A.V., Sethi, R., and Ullman, J.D. : "Compilers: Principles, Techniques, and Tools", *Addison Wesley*, Massachusetts, 1986.

[2] 芦田, 西松, 楠本, 井上 : "プログラムスライスに基づいたデバッグ支援システムの評価のための追証実験", **第 *58* 回 *(*平成 *11* 年前期*)* 全国大会**, **講演論文集** *(1)*, pp. 259–260, 1999.

[3] Gallagher, K.B. and Lyle, J.R. : "Using program slicing in software maintenance", *IEEE Transactions on Software Engineering, 17(8)*, pp. 751–761, 1991.

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

[5] 地平, 西松, 楠本, 井上 : "関数呼び出し履歴を利用したプログラムスライスの提案と実現", **信学技報**, *SS98-7*, pp. 9–15, 1998.

[6] 西江, 神谷, 楠本, 井上 : "プログラムスライスに基づくデバッグ支援ツールの実験的評価", **ソフトウェアシンポジウム** *97* **予稿集**, pp. 142–147, 1997.

[7] 西松, 楠本, 井上 : "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", **電子情報通信学会技術研究報告**, *SS98-3*, pp. 17–24, 1998.

[8] 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*, Los Angeles, CA, USA, 1999.

[9] 佐藤, 飯田, 井上 : "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", **情処学論**, *Vol. 37, No. 4*, pp. 536–545, 1996.

[10] Steensgaard, B. : "Points-to analysis in almost linear time", *Technical Report MSR-TR-95-08*, Microsoft Research, 1995

[11] 植田, 練, 井上, 鳥居 : "再帰を含むプログラムのスライス計算法", **信学論**, *Vol. J78-D-I, No. 1*, pp. 11–22, 1995.

[12] Weiser, M. : "Program Slicing", *Proceedings of the Fifth International Conference on Software Engineering*, pp. 439–449, 1981.