# 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

佐藤 慎一† 小林 孝則† 飯田 元‡ 井上 克郎† 鳥居 宏次‡

†大阪大学 基礎工学部 情報工学科
‡奈良先端科学技術大学院大学 情報科学研究科
〒５６０ 大阪府豊中市待兼山町1-3
大阪大学 基礎工学部 情報工学科 知能情報処理学講座
Tel:06-850-6571(内線6571)
E-mail:s-sato@ics.es.osaka-u.ac.jp

ソフトウェアが大規模化，複雑化するにつれ，プログラム中のバグ位置を特定するのが容易ではなくなっている．本研究では，プログラムの依存関係解析に基づいて，スライス抽出や部分解釈を行ない，プログラム全体からバグに関係のある部分のみを抽出し，それを対象にデバッグを行なえるようなシステムの試作を行なった．本システムは，もとのプログラムからスライスを抽出したり，部分解釈に基づいたプログラムの簡単化を行なう．また，得られた部分プログラムを実行させたり，特定の変数に直接影響を及ぼす文や変数を参照したりしながらデバッグを行なうことができる．本システムを用いれば，バグ位置の特定を比較的効率良く行なうことができるため，結果としてソフトウェアの開発効率が上がると考えられる．

依存関係解析, プログラム依存グラフ, スライス, 部分解釈, デバッグ

# Debug support system based on program dependence analysis

Shinichi Sato[†] Takanori Kobayashi[†] Hajimu Iida[‡] Katsuro Inoue[†] Koji Torii[‡]

[†]Department of Information and Computer Sciences,
Faculty of Engineering Science,Osaka University
[‡]Nara Institute of Science and Technology

1-3 Machikaneyama, Toyonaka, Osaka 560, Japan
Tel:06-850-6571(6571)
E-mail:s-sato@ics.es.osaka-u.ac.jp

Debugging large and complex software is not easy task since localizing and identifing faults are very diffcult. We have depeloped a debug system which extracts program slices and partially evaluates program texts based on program dependence analysis. Using this system, we can extract parts of the program which would relates to bugs, and also we can execute and trace the extracted parts. Therefore we consider efforts of debugging would be reduced, and its efficiency will be improved.

program dependence analysis, program dependence graph(PDG), program slice, partial interpretation, debugging

# 1　まえがき

通常プログラムをデバッグする際には，対象となるプログラム全体を扱う．従って，プログラムが大きくなるとデバッグ時にはエラーの原因の特定だけで時間がかかってしまい，このことがプログラム開発の効率を悪化させている．

このような場合に，エラーに関係があると思われる部分だけをプログラムから抽出するような機能があれば，開発者はプログラムの一部分だけに注目してエラーの位置の特定ができ，開発効率の向上が期待される．

本研究では，スライスや部分解釈を用いて，プログラム全体から注目する文や変数に関係のある部分だけを抽出し，その部分のみに対して処理を行なうようなデバッグ支援システムの試作を行なった．スライスとは，プログラム中のある文sでのある変数の値に影響を与える文の集合，もしくはsでの変数の定義が影響を与える文の集合である．スライスを用いることにより，ある出力変数に関係のある文のみをプログラムから抽出することができる．また，部分解釈では，プログラムに対する入力変数の値をある値に固定することにより，プログラムサイズの縮小を目的として，プログラムの変換を行なう．

これまで，いくつかのスライスを利用したデバッグ支援ツールが発表されている[2, 5]．しかし，これらはいずれもプログラムの実行系列を解析して得られるスライス(動的スライス)に基づくもので，実行させるたびにスライスを計算しなければならない上，実行系列を保存しておく必要があるために解析結果が膨大な量になる．

本研究では，動的スライスではなく，プログラムの依存関係を解析し，それに基づいて求められるスライス(静的スライス)を利用することにより，抽出したスライスを一つのプログラムとして扱ってデバッグを行なうことができる．また，静的スライスを利用することにより，デバッグだけでなくソフトウェアの部品化や保守などにも役立つと考えられる．

さらに，部分解釈を用いることにより，スライスを求めにくいようなプログラムに対しても対象プログラムのサイズを縮小し，デバッグや保守を効率良く行なえるようになる．

本稿では，これ以降，2章でプログラムの依存関係解析の概要を，3章で作成したシステムの概要を，4章でシステムの実行例を，5章で考察を述べる．

# 2　プログラムの依存関係解析

本論文で述べるデバッグ支援システムは，プログラムの依存関係解析の結果得られるプログラム依存グラフ(Program Dependence Graph，略してPDG)から，スライシングや部分解釈を行なうことによってプログラムの参照範囲を縮小することによりデバッグの支援を行なう．

## 2.1　プログラム依存グラフ（PDG）

PDGはプログラム内の文の依存関係を表すグラフである．PDGの節点はプログラム中の各文及びif文やwhile文の条件判定部分を表し，辺は変数の影響を伝えるデータ依存(Data Dependence，略してDD)関係及び条件文や繰り返し文の制御の影響を伝える制御依存(Control Dependence，略してCD)関係を表す．

DDは，各頂点の到達定義集合(Reaching Definitions，略してRD)を求めることによって得られる．PDG上でのある頂点$t$のRDとは，変数$v$と頂点$s$との組$\langle v, \mathtt{s} \rangle$の集合である．これは，

- プログラム中の文$s$で変数$v$を定義している．
- プログラム中の二つの文$s$と$t$の間に$v$を**必ず**定義するような文がない．

ことを示している．$t$のRDに$\langle v, \mathtt{s} \rangle$が含まれ，かつ$t$が$v$を参照する時，$s$から$t$へのDD関係があるという．

また，ある条件判定部分$s$の結果により文$t$の実行の有無が決定される時，$s$と$t$との間にCDが存在するものとする．すなわちCDはif文やwhile文の条件判定部分からそれらの内部ブロックに属する文への影響であり，これはプログラムを解析すれば容易に求められる．

一般にプログラムには複数の手続きが定義されている．各手続き間には引数や大域変数を通じてDD関係が生じる．これらのDD関係を表すため

に，PDG にプログラム中の文とは直接対応しない節点 (中継節点と呼ぶ) を用意する．

PDG の作成は，プログラムを解析し，プログラムの各文を PDG の節点に切りわけ，プログラム中の各文における RD を求め，それをもとにして PDG の各辺を生成することによって行なわれる．詳細は，[7, 8] を参照されたい．

例として 図1 のプログラムに対応する PDG を図2 に示す．図の PDG の中で角の丸い四角がプログラム中の各文に対応する節点で，楕円が中継節点である．また，有向辺のうち，実線で名前がついているものが DD 関係の辺で，破線のものが CD 関係の辺である．実線についている名前は，その DD 関係の辺が影響を伝える変数の名前である．また，破線で囲まれている部分はプログラム上の一つの手続きを表す．

```
 1  program euclid(input,output);
 2    var x,y,g,l:integer;
 3    function gcd(m,n:integer):integer;forward;
 4    procedure swap(var a,b:integer);
 5      var     temp:integer;
 6      begin
 7        temp:=a;
 8        a:=b;
 9        b:=temp;
10      end;
11    function lcm(a,b:integer):integer;
12      var     c:integer;
13      begin
14        c:=gcd(a,b);
15        lcm:=(a div c)*(b div c)*c
16      end;
17    function gcd;
18      var     w:integer;
19      begin
20        if m<n then begin
21          swap(m,n);
22        end;
23        while n<>0 do begin
24          w:=m mod n;
25          m:=n;
26          n:=w;
27        end;
28        gcd:=m;
29      end;
30    begin
31      writeln('Input x and y');
32      readln(x,y);
33      writeln('x=',x,' y=',y);
34      g:=gcd(x,y);
35      l:=lcm(x,y);
36      writeln('gcd=',g);
37      writeln('lcm=',l);
38    end.
```

図 1: PDG の元のプログラム

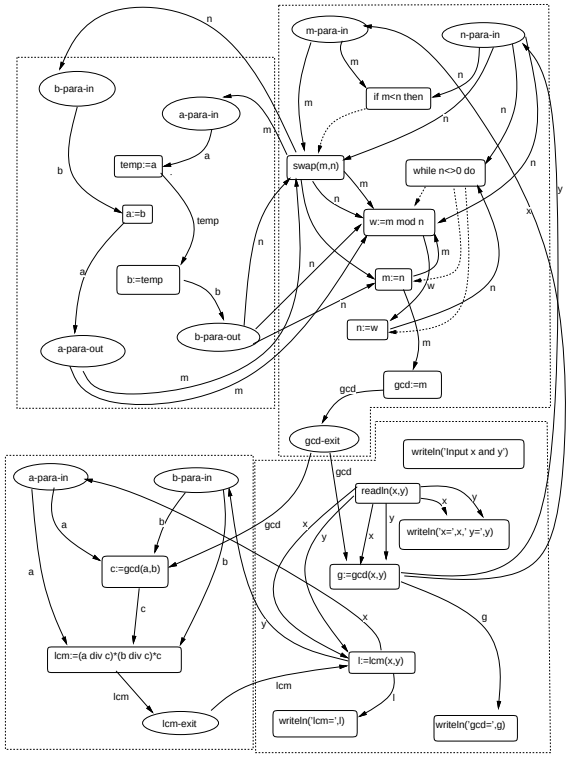


図 2: PDG の例

## 2.2 スライス

スライスは，前節で述べた PDG を探索することにより得られる．PDG を与えられた頂点から辺の順方向に探索していくものを forward slice と呼び，逆方向に探索していくものを backward slice を呼ぶ．

スライスの例として，図1 のプログラムのうち，36 行目の変数 g に関する backward slice が 図1 の下線部分である．g に影響を及ぼさない部分，すなわち関数 lcm に関する部分はスライスに含まれていない．

## 2.3 部分解釈

本稿で述べる部分解釈とは，プログラムに対する入力変数の取る値を固定したときにプログラム中の一部の文で参照される式や変数の値を静的に決定できるときには，その値をそれらに代入し，その結果不要な文や手続きが現れた場合にはそれらを削除することである．

入力変数の固定によるプログラム $P$ の部分解釈の手順を次に示す.

1. $P$ の大域変数の中から固定の対象とする入力変数 $x$ (複数個でも良い) を選ぶ.
2. 目的とする機能を選び出すのに応じて $x$ の値 (特定の固定値 $a$) を定め, 変数とその値の組 $\langle x,\ a\rangle$ を作る．その他の大域変数と局所変数 $y$ については不定値 $\perp$ との組 $\langle y,\ \perp\rangle$ を作る．(これらの組の集合をプログラム状態と呼ぶ.)
3. 主プログラムの本体（複合文）$M$ および2. のプログラム状態 $MState$ を引数として関数 $simple\ (M,\ MState)$ を呼び, その返り値を出力する.
4. 呼び出しが削除されなかった各手続きについて, その手続きが呼び出された時のプログラム状態を集めたプログラム状態集合 $F_iAllState$ 中のすべてのプログラム状態を演算 $\circ$ を使って併合して, その手続きの初期プログラム状態 $F_iState$ を作成する.
5. 呼び出しが削除されなかった各手続きについて, 手続きの本体（複合文）$F_i$ およびその手続きの初期プログラム状態 $F_iState$ を引数として関数手続き $simple\ (F_i,\ F_iState)$ を呼び, その返り値を出力する.

ここで，$simple\ (S,\ State)$ は与えられたプログラム状態 $State$ を利用して文 $S$ を部分解釈した文 $S'$ を返す．$simple$ は関数 $exec$ とともに定義される．$exec\ (S,\ State)$ は与えられたプログラム状態 $State$ で文 $S$ を部分計算した結果の新しいプログラム状態を表す．$simple, exec$ の詳細は本稿の最後に添付した付録で述べる.

また, プログラム状態の併合を行なう演算 $\circ$ は, 以下のように定義される．いま，2つのプログラム状態 $State_1$ と $State_2$ が与えられており,

$$
\begin{aligned}
State_1 &= \{\langle x_1,\ v^1_{x_1}\rangle, \ldots, \langle x_n,\ v^1_{x_n}\rangle\} \\
State_2 &= \{\langle x_1,\ v^2_{x_1}\rangle, \ldots, \langle x_n,\ v^2_{x_n}\rangle\}
\end{aligned}
$$

であるとする．これらに演算 $\circ$ を適用した時の結果は次のようになる.

$$
State_1 \circ State_2 = \{\langle x_1,\ v^3_{x_1}\rangle, \ldots, \langle x_n,\ v^3_{x_n}\rangle\}
$$

$$
v^3_{x_i} = \begin{cases} v^1_{x_i}, & v^1_{x_i} = v^2_{x_i} \text{ のとき} \\ \perp, & v^1_{x_i} \neq v^2_{x_i} \text{ のとき} \end{cases}
$$

この定義の意味は変数 $x_i$ が両方のプログラム状態 $State_1$ と $State_2$ で同じ値を持つならば新しいプログラム状態 $State_3$ においてその値が保存され, そうでなければその値は $\perp$ となるということである.

上記の手順でプログラムが部分解釈されると, 元のプログラムにあった手続きの呼び出しや, 変数の定義, 参照がなくなることが起こり得る．その結果, それらの手続きや変数自体が不要となり，さらに文が削除できる場合がある.

手続きが不要となる場合は，その手続きが一度も呼び出されない場合である．そのような場合には, その手続き自体を削除できる.

次に, 変数が不要となるのは次の2つの場合である.

- 参照する文がない.
- 自分自身の再定義にしか参照されない.

したがって, この場合は以下のような手順で不要な代入文と入力文を削除することができる.

1. 前述の簡素化結果のプログラムをPDGに変換する.
2. そのPDGの頂点のうち代入文もしくは, 入力文の頂点を調べてそこから到達できる頂点が存在しない場合, その代入文もしくは, 入力文は削除する.

このような手続きおよび変数が不要となる場合に不要な手続き宣言と不要な文の削除を行なった結果, 新たに手続きの呼び出し，変数の定義や参照がなくなることが起こる場合があるので, その場合はその結果のプログラムに対して, 上に述べたような文の削除ができないか確認する必要がある．よって, 上に述べたような文の削除は新たに削除する文がなくなるまで繰り返しておこなう.

部分解釈の例として， 図3 のプログラムの入力変数のうち，変数 `b` を `'n'` に，変数 `c` を `'y'` に固定して部分解釈を行なった結果のプログラムを 図4 に示す.

入力変数 `b` を `'n'` に固定した結果，入力文が削除され，また，その値が `'y'` のときに実行されるは

ずであった27行目から38行目の部分が削除されている．また，入力変数cを'y'に固定した結果，24行目のif節が削除されている．

```
 1  program wordcount(in,out);
 2    var a,b,c:char;
 3        in:array[0..1000] of char;
 4        i:integer;
 5        letter,word,line:integer;
 6        isinword:boolean;
 7
 8
 9    begin
10      writeln('Count the letter?(y/n)');
11      readln(a);
12      writeln('Count the word?(y/n)');
13      readln(b);
14      writeln('Count the line?(y/n)');
15      readln(c);
16      i := 0;
17      letter := 0;
18      word := 0;
19      line := 0;
20      isinword := false;
21      while in[i] <> EOF do begin
22        if a = 'y' then
23          letter := letter + 1;
24        if c = 'y' then
25          if in[i] = EOL then
26            line := line + 1;
27        if b = 'y' then
28          if (in[i]<>' ')and(in[i]<>EOL) then
29            begin
30              if isinword = false then
31                begin
32                  isinword := true;
33                  word := word +1
34                end
35            end
36          else
37            isinword := false;
38        i := i +1
39      end;
40      if a = 'y' then
41        writeln('letter =',letter);
42      if b = 'y' then
43        writeln('word   =',word);
44      if c = 'y' then
45        writeln('line   =',line)
46    end.
```

図 3: 部分解釈前のプログラム

# 3　システムの概要

本章では，本研究で試作したデバッグ支援システムの概要について述べる．

## 3.1　言語仕様

本システムが対象とする言語は，以下のような仕様を持つPascalのサブセットである．

- 文として，代入文，条件文，繰り返し文，手続き呼出文，begin–endで囲まれる複合文を扱う．
- 手続きは再帰呼び出しも扱う．ただし，部分解釈ではこれは扱えない．また，手続きの引数の渡し方については，値渡しと変数渡しの二種類を許す．
- 変数のデータ型はスカラー型のみでポインタ型は扱わない．具体的には整数型，文字型，論理型およびそれらを要素に持つ配列型とした．

```
program wordcount(in,out);
  var a,b,c:char;
      in:array[0..1000] of char;
      i:integer;
      letter,word,line:integer;
      isinword:boolean;

  begin
    writeln('Count the letter?(y/n)');
    readln(a);
    writeln('Count the word?(y/n)');
    writeln('Count the line?(y/n)');
    i:=0;
    letter:=0;
    line:=0;
    while in[i] <> EOF do begin
      if a='y' then

        letter:=letter+1;
      if in[i]=EOLN then

        line:=line+1;
      i:=i+1
    end;
    if a='y' then

      writeln('letter =',letter);
    writeln('line =',line)
  end.
```

図 4: 部分解釈後のプログラム

このように，簡単化されてはいるものの，プログラミング言語の基本的な文の構造はすべて含んでいるので，拡張は容易に行なうことができる．

## 3.2　システム構成

本システムの構成図を 図5 に示す．本システムは，おおまかには，ソースコードの構文解析を行ないつつプログラム中の各文をPDGの各節点と対応づけるパーザ部，その結果に基づいてPDGを作成するアナライザ部，PDGを探索してスライスを得るスライサ部，PDGを利用してプログラムの部分解釈を行なう部分解釈部，プログラムの実行を行なうインタープリタ部に分けられる．

ユーザがコード化が終了したプログラムを入力すると，システムは，アナライザ部でそれを解析

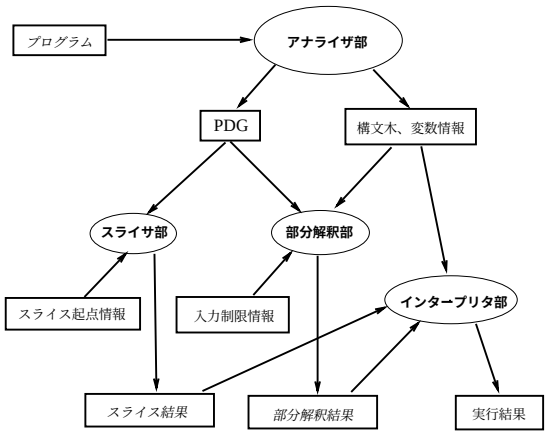


図 5: システムの構成図

する．解析が終了すると，PDG および実行に必要な構文木や変数の情報などが生成される．

ユーザはこの段階でシステムにどの動作をさせるかを指定する．ユーザがプログラムを実行させた場合には，システムは, インタープリタ部にアナライザ部が生成した構文木や変数の情報などを渡して，プログラムを実行させる．その際，通常の実行だけでなく，ステップ実行やブレークポイントの設定，変数値の参照などのデバッグに必要な作業も行なうことができる．

また，実行させた結果誤った出力が得られた場合，ユーザはプログラム全体を対象とするのではなく，その出力に関係のある部分のみをプログラムから抽出し，その抽出部分をデバッグの対象とすることができる．

部分解釈で入力変数を固定する場合には，システムは, PDG および構文木などの情報を部分解釈部に渡す．また，ユーザはどの入力変数をどの値に固定するのかといった情報を入力する．それらをもとにして部分解釈部で部分解釈が行なわれ，その結果, プログラムの一部の機能のみを実現するプログラムが生成される．

スライスを抽出する場合には，スライサ部に，PDG やどの変数に関するスライスを抽出するのかといったユーザからの入力が渡され，スライスが計算される．そのスライス結果は，元のプログラムの部分プログラムとして，スライサ部から出力される．

また，部分解釈部やスライサ部から出力された簡素化プログラムをインタープリタ部で実行することができる．この場合にも上述のようなデバッグ作業を行なうことができる．このようにして，デバッグの対象を限定することにより，効率良くデバッグ作業を進めることができる．

# 4 実行例

本システムの実行例として， 図6 に示されるバグを含むプログラムをデバッグする例を考える．このプログラムは，入力された５つの数字の合計, 最大，最小, 平均を計算し，出力するプログラムである．

1. プログラムを解析し，インタープリタ部で実行させる．この実行結果が 図7 である．これを見ると，明らかに, 最大値を示す `MAX` の値が誤っている．

2. この値を出力した変数 ( 図6 における 32 行目の変数 `Max`) に関する backward slice を抽出する．このとき， 図8 のようなウィンドウが現れる．この画面で，スライスを抽出したい変数名や CD および DD のどちらの依存関係について PDG を探索するか，PDG をどのレベルまで探索するかなどの情報を入力する．ここでは，変数名 `Max` とし, CD，DD の両方の依存関係について PDG をできる限り探索することにする．こうして得られるスライス結果は 図9 中の網掛け部分として参照される．

3. 32 行目の `Max` に直接定義を行なっている文は，14 行目の文である．これはスライスを計算する際に, レベルの設定を 1 などの適当な値に設定することにより得られる．

4. この文にブレークポイントを設定し，スライス部分のみを実行させる．すると，この文自体は何度も実行されているにも関わらず，`Max` の値が変化していないことがわかる．これは，この文で参照される変数が `A[1]` となっているためで，それを `A[i]` に修正することにより正しい結果が得られる．

```
 1  program coverage(input,output);
 2    var   Sum, Max, Min, Mean: integer;
 3          A: array[1..5] of integer;
 4          n: integer;
 5
 6  procedure calc(var sum, max, min, mean: integer);
 7    var   i: integer;
 8    begin
 9      i := 1 + 1;
10      while i <= n do
11        begin
12          sum := sum + A[i];
13          if A[i] > max then
14            max := A[1]
15          else if A[i] < min then
16            min := A[i];
17          i := i + 1
18        end;
19      mean := sum div n;
20    end;
21
22  begin
23    n := 5;
24    writeln('Please Input ',n,' values');
25    readln(A[1],A[2],A[3],A[4],A[5]);
26    Sum := A[1];
27    Max := A[1];
28    Min := A[1];
29    Mean := A[1];
30    calc(Sum, Max, Min, Mean);
31    writeln('SUM      : ', Sum);
32    writeln('MAX      : ', Max);
33    writeln('MIN      : ', Min);
34    writeln('MEAN     : ', Mean)
35  end.
```

図 6: バグを含んだプログラム

```
display
Please Input 5 values:
1 2 3 4 5
SUM      : 15
MAX      : 1
MIN      : 1
MEAN     : 3
```

図 7: 実行結果

## 5 まとめ

プログラムから依存関係にもとづいてバグに関係のある部分のみを抽出するデバッグ支援システムを作成した．本システムを用いることにより，プログラム全体を対象としていた従来のデバッガを用いるよりも効率の良いデバッグを行なえることが期待される．また，プログラムの一部の機能を実現する部分のみを抽出できることから，ソフトウェアの部品化などの保守作業にも役立つと考えられる．

今後の課題としては，実際にシステムを使用してその評価を行うことや，対象言語を拡張し，実

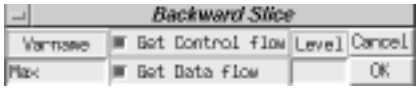


図 8: スライス選択画面

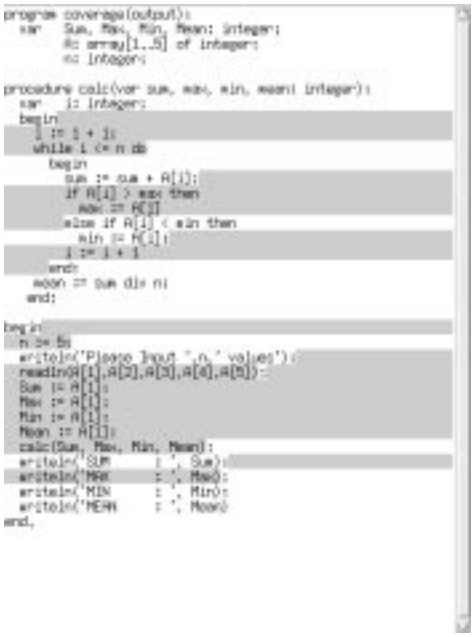

図 9: スライス結果

際に使用されている手続き型言語 (C,Pascal など) にも適用させること，また，このような対話的なプログラムの依存関係解析システムに適した，インクリメンタルに解析を行なえるような解析アルゴリズムの適用などが挙げられる．

## 参考文献


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

[2] Agrawal, H., Demillo, R.A. and Spafford, E.H.:“Debugging with Dynamic Slicing and Backtracking”, Software–Practice and Experience, Vol.23(6), pp. 589–616(1993).

[3] 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).

[4] 小林 孝則:“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(1995–03).

[5] 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).

[6] 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”, 大阪大学大学院基礎工学研究科修士学位論文(1994–02).

[7] 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依存関係解析とそれに基づくプログラムスライシング”, 信学技報, SS93–24, pp. 33–40(1993–09).

[8] 植田 良一:“再帰を含むプログラムのスライスを求めるアルゴリズムの提案”, 大阪大学大学院基礎工学研究科修士学位論文(1994–02).

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

# 付録：simple および exec 関数

(1) 空文: $\phi$

$$\begin{aligned} exec(\phi,\ State) &= State \\ simple(\phi,\ State) &= \phi \end{aligned}$$

(2) 複合文: $(S_1; S_2; \cdots; S_n)$

$$\begin{aligned} &exec((S_1; S_2; \cdots; S_n), State) \\ &\quad = exec((S_2; \cdots; S_n), exec(S_1, State)) \\ &simple((S_1; S_2; \cdots; S_n), State) \\ &\quad = simple\ (S_1,\ State)\ ; \\ &\qquad simple\ ((S_2; \cdots; S_n),\ exec\ (S_1,\ State)) \end{aligned}$$

(3) 入出力文: $\mathbf{input}(x_i)$, $\mathbf{output}(x_i)$

$$\begin{aligned} &exec(\mathbf{input}(x_i), State) \\ &\quad = \{\langle x_1, v_{x_1}\rangle, \ldots, \langle x_i, v'_{x_i}\rangle, \ldots, \langle x_n, v_{x_n}\rangle\} \end{aligned}$$

$$\begin{aligned} exec(\mathbf{output}(x_i), State) &= State \\ simple(\mathbf{input}(x_i), State) &= \mathbf{input}(x_i) \\ simple(\mathbf{output}(x_i), State) &= \mathbf{output}(x_i) \end{aligned}$$

ただし, $State = \{\langle x_1, v_{x_1}\rangle, \ldots, \langle x_n,\ v_{x_n}\rangle\}$, $v'_{x_i}$ は $Input\text{–}Var$ 中の $x_i$ の値である.

(4) 代入文: $x_i := expr\ (x_1, \ldots, x_n)$

$$\begin{aligned} &exec(x_i := expr(x_1, \ldots, x_n), State) \\ &\quad = \{\langle x_1, v_{x_1}\rangle, \ldots, \langle x_i, v'_{x_i}\rangle, \ldots, \langle x_n, v_{x_n}\rangle\} \\ &simple(x_i := expr(x_1, \ldots, x_n), State) \\ &\quad = x_i := eval(expr(x_1, \ldots, x_n), State) \end{aligned}$$

ただし,

$$\begin{aligned} State &= \{\langle x_1,\ v_{x_1}\rangle, \ldots, \langle x_n, v_{x_n}\rangle\} \\ v'_{x_i} &= \begin{cases} n & \cdots eval\ (expr\ (x_1, \ldots, x_n),\ State) \text{が} \\ & \qquad \text{定数値} n \text{を返すとき} \\ \perp & \cdots \text{それ以外のとき} \end{cases} \end{aligned}$$

ここで関数 $eval\ (expr,\ State)$ は与えられたプログラム状態 $State$ で与えられた式 $expr$ を評価した式を示す. 結果は $State$ で値が決まる変数をすべてその値で置き換えた式である. また, 置き換えによって式が計算できるようになった場合はその値を返す.

(5) 分岐文: $\mathbf{if}\ expr\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$

$$exec\ (\mathbf{if}\ expr\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2,\ State)$$
$$= \begin{cases} exec\ (S_1,\ State) & \\ \qquad \cdots eval\ (expr,\ State) = \mathbf{true} \text{のとき} & \\ exec\ (S_2,\ State) & \\ \qquad \cdots eval\ (expr,\ State) = \mathbf{false} \text{のとき} & \\ exec\ (S_1,\ State)\ \circ\ exec\ (S_2,\ State) & \\ \qquad \cdots \text{それ以外のとき} & \end{cases}$$

$$simple\ (\mathbf{if}\ expr\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2,\ State)$$
$$= \begin{cases} simple\ (S_1,\ State) & \\ \qquad \cdots eval\ (expr,\ State) = \mathbf{true} \text{のとき} & \\ simple\ (S_2,\ State) & \\ \qquad \cdots eval\ (expr,\ State) = \mathbf{false} \text{のとき} & \\ \mathbf{if}\ eval\ (expr,\ State)\ \mathbf{then} & \\ \quad simple\ (S_1,\ State) & \\ \mathbf{else} & \\ \quad simple\ (S_2,\ State) & \\ \qquad \cdots \text{それ以外のとき} & \end{cases}$$

(6) 繰り返し文: $\mathbf{while}\ expr\ \mathbf{do}\ S\ \mathbf{end}$

$$exec\ (\mathbf{while}\ expr\ \mathbf{do}\ S\ \mathbf{end},\ State)$$
$$= \begin{cases} exec\ (\mathbf{while}\ expr\ \mathbf{do}\ S\ \mathbf{end},\ exec\ (S,\ State)) & \\ \qquad \cdots eval\ (expr,\ State) = \mathbf{true} \text{のとき} & \\ State \quad \cdots eval\ (expr,\ State) = \mathbf{false} \text{のとき} & \\ State' \quad \cdots \text{それ以外のとき} & \end{cases}$$

$$simple\ (\mathbf{while}\ expr\ \mathbf{do}\ S\ \mathbf{end},\ State)$$
$$= \begin{cases} \phi \qquad \cdots eval\ (expr,\ State) = \mathbf{false} \text{のとき} & \\ \mathbf{while}\ eval\ (expr,\ State)\ \mathbf{do} & \\ \quad simple\ (S,\ State') & \\ \mathbf{end} & \\ \qquad \cdots \text{それ以外のとき} & \end{cases}$$

ただし, $State = \{\langle x_1,\ v_{x_1}\rangle, \ldots, \langle x_n,\ v_{x_n}\rangle\}$ とすると,

$$State' = \{\langle x_1,\ v'_{x_1}\rangle, \ldots, \langle x_n,\ v'_{x_n}\rangle\}$$

$$v'_{x_i} = \begin{cases} \perp & x_i \text{が} S \text{中で定義される可能性が} \\ & \text{あるとき} \\ v_{x_i} & \text{それ以外のとき} \end{cases}$$