論　文

再帰を含むプログラムのスライス計算法

非会員　植田　良一[†]　　非会員　練　　林[††]
正　員　井上　克郎[††]　　正　員　鳥居　宏次[†††]

An Algorithm of Computing Slices for Recursive Program

Ryoichi UEDA[†], Lin LIAN[††], *Nonmembers*, Katsuro INOUE[††] *and* Koji TORII[†††], *Members*

あらまし　プログラムスライスは，デバッグ，テスト，プログラム合成などに利用される有用な技術である．プログラムPのスライスとは，直感的にはP中のある地点nおよびある変数vに対して，nにおけるvの値に影響を与えるP中の各文や式の集合を言う．スライスの計算には，プログラム内の文の間の依存関係の正確な解析が必要であるが，再帰が存在するプログラムを解析するのは容易ではない．本研究では，再帰を含むプログラムの依存関係の静的解析と，その結果に基づいてスライスを求めるためのアルゴリズムを提案する．本アルゴリズムでは，再帰的な手続き定義に対応するために，計算に必要な情報をあらかじめ仮定し，その情報をより正しいものへと変化させ，それが収束するまで解析を繰り返すという方法をとった．更に，手続きの境界を超えてスライスを計算できるように，独自の改良を施したプログラム依存グラフを定義した．このアルゴリズムは，従来のものに比べ，再帰を含むプログラムを効率的に解析することができ，対話的にスライス情報を提供するシステムに組み込むことができる．

キーワード　プログラム依存グラフ，到達定義集合，再帰，制御依存関係，データ依存関係

1. まえがき

プログラムPのスライスとは，直感的にはP中のある地点nおよびある変数vに対して，nにおけるvの値に影響を与えるP中の各文や式の集合を言う．すべての可能な入力データに対して解析したものを静的スライス，特定の入力データに対して解析したものを動的スライスと言う(本論文では静的スライスのみを扱うので，以下単にスライスと言えばそれは，静的スライスを意味するものとする)．スライスの技法はMark Weiser[(1)]によって提案され，当初はプログラムのデバッグを支援するために使われていたが，現在では，デバッグだけでなくテストや保守，プログラム合成などにも利用されている[(4),(7)]．スライスの計算には，プログラム内の文の間の依存関係の正確な解析が必要であるが，再帰が存在するプログラムを正確に解析するのは容易ではない．

Weiserは，スライスを計算するために，データフロー方程式（詳しくは**3.**参照）を使ったが，正確な計算ができるのは一つの手続き内だけだった．Ottenstein[(6)]が，このスライスの計算をグラフ上の到達可能性問題に置き換える手法を考案した．この手法を使って，Horwitz[(2),(3)]が，手続きの境界を超えてスライスが計算できるアルゴリズムを紹介したが，このアルゴリズムでは各手続き内のデータフロー解析とスライス計算のためのグラフ作成等をいくつかのフェーズに分けて行うため，効率が悪いものとなっている．また，Hwang[(5)]は，再帰を含むプログラムに対して最小不動点を求める手法を用いて，スライスを直接計算する方法を提案したが，一般に，プログラム中の地点n，変数vごとに再帰方程式を解く必要があり，我々が目指す，対話的にスライス情報を提供するシステムには組み込みにくい．

† (株)日立製作所・システム開発研究所，川崎市
System Development Laboratory, Hitachi, Ltd., Kawasaki-shi, 215 Japan
†† 大阪大学基礎工学部情報工学科，豊中市
Faculty of Engineering Science, Osaka University, Toyonaka-shi, 560 Japan
††† 奈良先端科学技術大学院大学情報科学研究科，生駒市
Graduate School of Information Science, Nara Institute of Science and Technology, Ikoma-shi, 630-01 Japan

```
        ⋮
1       g := 0;
2       l := f(5);
3       h := g + 10;
        ⋮
```

図1 プログラムの一部: g は大域変数で，関数 f 内で定義可能
Fig. 1 A part of a program: g is a global variable, so can be redefined in the function f .

そこで，本論文では，再帰を含むプログラムの依存関係の静的解析と，その結果に基づいてスライスを求めるための，対話的システムに組み込みやすいアルゴリズムを提案する．我々は，ユーザがさまざまな地点でのスライスの計算結果を参照したり、部分実行するような対話的なデバッグ環境でも使えるようにすることを目標にしているので，本アルゴリズムでは，再帰の存在を前提としてソースコードを解析しプログラム依存グラフを作り，その上でスライスを計算するという方法を採用した．

我々のアルゴリズムが対象とする入力言語では，再帰的に手続きを定義することができる．通常，依存関係の解析には，データフロー方程式が使われるが，ある文に手続き呼出しがある場合，陽に現れない変数の定義や参照が発生し，その影響を知ることが容易ではない．

例えば，図1のようなプログラムを解析するとき，文1で定義された変数 g の内容がそのまま文3で使われる（文1の定義が文3に到達すると言う．正確には **2.** で定義する）かどうかは関数 f の内容に依存する．スライスを計算することを目的とする場合の控え目な解（無駄なプログラム断片を含むかもしれないが実行に影響を与えるものすべては必ずスライスとして残る）は，到達すると考えることであるが，f を注意深く解析すれば，次のような3通りに場合分けすることで，より正確な解を得ることができる．

- f 内で必ず g を定義するとき，1は3には到達せず，代わりに，f 内の g の定義が3へ到達する
- f が g を定義する場合と定義しない場合の両方の可能性があるとき，f 内の g の定義と1の両方が3に到達する
- f が g を定義する可能性が全くないとき，1が3に到達する

また，再帰的な手続き定義を許すことで問題になるのは，再帰的に定義された手続きが，ある実行パス中のある文がより前の文へ影響を与える可能性がある，条件文の条件成立時に実行される文が，不成立時に実行される文へ影響を与える可能性があるなど，繰返し文と似た性質をもっていることや，ある手続きの解析を終える前に，その手続きを呼び出す別の手続きを処理しなければならない場合があることである．

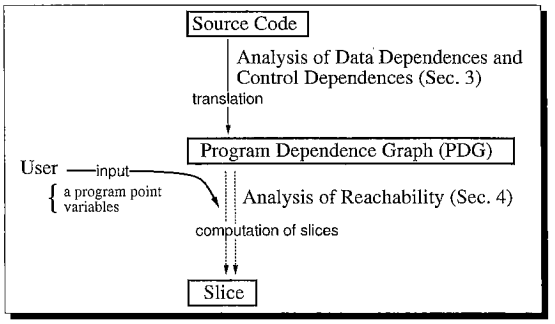


図2 試作システムの全体像
Fig. 2 The overview of our prototype system.

そこで，我々はこれらに対処するために，計算に必要な情報をあらかじめ仮定し，その情報をより正しいものへと変化させ，それが収束するまで解析を繰り返すという方法をとった．更に，手続きの境界を超えてスライスを計算できるように，独自の改良を施したプログラム依存グラフを定義した．

図2にスライスの計算を行うアルゴリズムの概要を示す．またこのアルゴリズムに従ってプログラムを解析し，ユーザの入力に応じてスライスを計算し表示するシステムを試作し，その動作を確認した．

本論文ではこれ以降，**2.** で，入力言語とプログラム依存グラフに関する定義を，**3.** で，入力プログラムからプログラム依存グラフを作る方法（図2の細矢印部分）を，**4.** で，プログラム依存グラフ上でスライスを計算する方法（図2の破線矢印部分）を，**5.** で，アルゴリズムの複雑さについて，それぞれ述べる．

2. プログラム依存グラフとスライス

我々は，Ottenstein, Horwitzと同様に，ソースコードを解析しプログラム依存グラフを作り，その上でスライスを計算するという方法を採用した．この方法は，プログラム依存グラフを作ることにより，はじめの1回の解析を終えた後は，より少ない時間でスライスを計算できる（**5.** 参照）もので，対話的にスライス情報を提供するシステムには適していると考えられる．こ

の節では，入力言語，プログラム依存グラフおよびその上でのスライスを定義する．

2.1 入力言語

ここで紹介する解析アルゴリズムの入力言語として，以下のような特徴をもつPascal風言語を想定している．この言語は現実の言語に比べ，簡単化されてはいるが，本質的な構造をすべて含んでおり，一般の言語に拡張できる．

その言語には文として条件文(if)，代入文，繰返し文(while)，入力文(readln)，出力文(writeln)，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でポインタ型はない．プログラムは，大域変数宣言，手続き（および関数）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およ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，自己再帰的およ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 制御依存とデータ依存

二つの文の間の関係として制御依存(Control Dependence，以下CD)関係とデータ依存(Data Dependence，以下DD)関係がある．文sがif文のような条件判定を含む文であり，かつ，sの条件判定部分s_1の実行結果が文s_2の実行の有無に直接影響を与えるとき，s_1とs_2の間にCD関係が発生する．これをCD関係辺($s_1 - - \rightarrow s_2$)と表す．また，文s_1がある変数vを定義し，vを参照している文s_2にこの定義が到達する†とき，s_1とs_2の間にDD関係が発生する．これをDD関係辺($s_1 \xrightarrow{v} s_2$)と表す．

2.3 プログラム依存グラフ(PDG)

プログラム依存グラフ(Program Dependence Graph，以下PDG)とは，プログラム内の文の間の依存関係を表す有向グラフであり，その節点は，プログラムに含まれる条件判定部分，代入文，入出力文，手続き呼出し文を表し，その有向辺は二つの節点の間の制御依存および，データ依存関係を表す．但し，手続きの境界を超えて，スライスを計算できるようにするために，表1に挙げるような，いくつかの特殊節点も存在する．例えば，大域変数gと，仮引数pをとる関数fをもつプログラムには図3にあるように，exit節点f–$exit$, in節点f_g–in, out節点f_g–out，引数節点f_p–parが存在する．

このように、我々のアルゴリズムではHorwitzの方法と同様に、プログラムには現れない特殊節点を使って手続き間のデータフローを検出する。

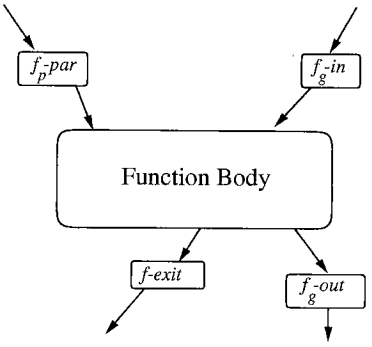


図3 関数fに対するPDGの概略
Fig. 3 The outline of PDG for a function f.

表1 特殊節点

	特殊節点	表記例
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，各関数に一つずつある	f–$exit$
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，一つずつある	f_g–in
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，一つずつある	f_g–out
引数節点	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一つずつある	f_p–par

2.4 スライス

プログラムをPDGで表現し，「文sにおける変数vに関するスライスとは，PDG上でCD関係辺またはDD関係辺をたどって文sの変数vに到達できる節点集合に対応する文の集合である」と定義する．

例えば，図4のプログラムの文 `writeln(c)` に関するスライスは，図5のようになり，この部分だけを実行しても指定した文でのcの値は同じものになる．

3. PDGへの変換

ソースコードをPDGに変換するには，プログラム中の各文の間のCD関係およびDD関係を知る必要がある．CD関係は入力言語仕様より容易に求めることができるので，ここでは，DD関係の求め方を詳しく

† 「s_1からs_2への，vの再定義を含まないような実行パスが存在する」と同義．

```
program atoi(input,output);
    var c:integer;
        n:integer;
        ch:char;
begin
    n := 0;
    c := 0;
    readln(ch);
    while (ch >= '0') and (ch <= '9') do begin
        n := n * 10 + (ch - '0');
        c := c + 1;
        readln(ch);
    end;
    writeln(n);
    writeln(c);
end.
```

図4 プログラムatoi:入力文字列を整数に変換してその数とけた数を表示する

Fig. 4 Program atoi: converts an input string to the corresponding integer and displays its integer and its number of places.

```
program atoi(input,output);
    var c:integer;
        ch:char;
begin
    c := 0;
    readln(ch);
    while (ch >= '0') and (ch <= '9') do begin
        c := c + 1;
        readln(ch);
    end;
    writeln(c);
end.
```

図5 atoiの文`writeln(c)`に関するスライス:けた数を表示する部分が抽出されている

Fig. 5 The slice at the statement `writeln(c)`: the only part which calculates the number of places is extracted.

説明する.

3.1 諸 定 義

変数vと節点nとの組dを$\langle v,n\rangle$と表し，変数部をd_{var}で，節点部をd_{vertex}で表す．プログラムのある領域S(連続する0個以上の文の集合)に対する到達定義集合(Reaching Definition Set) $RD_{in}(S)$を，次のように定義する.

$$RD_{in}(S) \equiv \{\langle v,n\rangle \mid \text{節点}n\text{で変数}v\text{が定義される} \wedge n\text{が}S\text{の先頭へ到達する}\}$$

また，Sを出て次の領域へ到達する定義集合を$RD_{out}(S)$と書く．一般に式(1)のような関係が成り立つ．但し，$kill(S)$はSで消滅する†定義の集合を，$gen(S)$はSで新たに発生する定義の集合を意味する.

$$RD_{out}(S) = (RD_{in}(S) - kill(S)) \cup gen(S) \quad (1)$$

プログラムのある領域Sについて，Sを実行したときに(どのパスが実行されても)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．定義より，二つの定義集合間には次のような関係がある.

$$SuDEF(S) \subseteqq PoDEF(S)$$

更に，次のように定義する.

$$SuDEF_{var}(S) \equiv \{d_{var} \mid d \in SuDEF(S)\}$$
$$PoDEF_{var}(S) \equiv \{d_{var} \mid d \in PoDEF(S)\}$$

この確実定義集合と潜在定義集合を使うと，式(1)は次のように書き換えることができる.

$$RD_{out}(S) = (RD_{in}(S) - \{\langle v,n\rangle \mid v \in SuDEF_{var}(S)\}) \cup PoDEF(S) \quad (2)$$

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

$ImUSE(p)$に含まれる変数は手続きpの呼出し地点sには直接現れないが，pの呼出しによって参照されると解釈して，$RD_{in}(s)$に含まれる定義節点からsへのDD関係を作るために使われる.

3.2 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pの解析にはp自身の構造に関する情報以外にpが直接呼び出す手続きqの確実定義集合($SuDEF(q)$)，潜在定義集合($PoDEF(q)$)，暗使用される変数集合($ImUSE(q)$)に関する情報が必要となる．しかし，再帰を含むプログラムを解析する際には，qを解析する前にqを呼び出す別の手続きpを解析しなければならない状況が起こり得る．よって，手続きの解析を開始する前に，すべての手続きに対して，そのSuDEF,

† S以降の文に到達しない.

PoDEF, ImUSE の初期値を与える必要がある．

手続き r が別の手続き s を呼び出すが，s は r を呼び出さないとき，r の定義集合は s の定義集合に依存するので，s の解析の後に，r の解析をする方が，r の定義集合が早く収束することは明らかである．このように，手続き間の呼出し関係を解析し，手続きの解析順序をうまく選ぶことにより，手続きの総解析回数を少なくできる．

解析順序を決定するためには，手続き間の「呼ぶ—呼ばれる」関係を知る必要があるので，入力プログラムに対して，手続き名を節点で，また，「呼ぶ—呼ばれる」関係を，呼ぶ側から呼ばれる側への有向辺で表した呼出しグラフ (Call Graph，以下CG) を作る．あるプログラムのCGが有向無閉路グラフ (Directed Acyclic Graph，以下DAG) になるとき，そのプログラムには，相互/自己再帰呼出しがない．

手続きが再帰的に定義されているとき，その解析に使われる情報が不正確な場合があり，通常，その手続きを一度解析しただけでは，その定義集合は収束していない．そこで，関連する再帰呼出しを含む手続き集合を，その他の手続き集合と分離して，それぞれの手続きの定義集合が収束するまで，その集合内の手続きのみを解析し続ける．

関連する再帰呼出しを含む手続き集合は，直接または間接的に相互に呼び出す可能性のある手続きの集合であるから，CG上では強連結成分 (Strongly Connected Component) として現れる．再帰呼出しを含まない手続きは，その手続き一つだけからなる強連結成分として抽出される．CG上での強連結成分の抽出方法については文献(8) の10章を参照されたい．

次に，強連結成分間の解析順序を決めなければならない．強連結成分を一つの節点として，強連結成分間をつなぐCG上の辺のみをその辺としてもつ縮約グラフ (Reduced Graph) CG′ を作ると，これは強連結成分の定義からDAGになる．CG′ 内の辺は，強連結成分間の「呼ぶ—呼ばれる」関係を表しているので，CG′ 上をその根から深さ優先探索をして，ポストオーダで順序付けすれば，これが強連結成分間の最適な解析順序になる．

以上をまとめると，次のようになる．

アルゴリズム AnalyzeProgram

Input: プログラム P

Output: P のPDG

Step 1. プログラム P のCG上で強連結成分を抽出して，その縮約グラフを作り，その根から深さ優先探索して，ポストオーダで順序付けする（但し，1回の探索ですべての節点に到達できないとき (グラフが連結でないとき) は，未到達の節点がなくなるまで繰り返し，すべての節点を順序付けする）．

Step 2. 各強連結成分内の要素を任意の要素から深さ優先探索して，ポストオーダで順序付けする

Step 3. 各手続きの SuDEF, PoDEF, ImUSE を以下のように初期化する (但し，変数 $SuDEF(f)$ は手続き f 全体に対する確実定義集合を意味する)．

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

Step 4. Step 1で決定した順序で強連結成分 S を一つずつ選び以下の手順で解析する

(a) S 内の各手続きを Step 2 で決定した順序で1回ずつアルゴリズム AnalyzeProcedure を使って解析する（アルゴリズム AnalyzeProcedure は **3.3.** で説明する）．

(b) S 内のすべての手続きの SuDEF, PoDEF, ImUSE が変化しなくなるまで (a) を繰り返す†．

Step 5. メインプログラムを解析する

3.3 一つの手続きの処理

一つの関数 f は，一般に 図6 のような構造で表される．ここではその本体の領域を B とする．この f は以下の手順で解析される（手続きの場合，関数と違って戻り値がなく，exit節点が不要なので，この節点に関する辺を作らないところを除けば関数の場合と同じである）．

アルゴリズム AnalyzeProcedure

Input: 手続き f (その本体の領域を B とする)

Output: $SuDEF(f)$，$PoDEF(f)$，$ImUSE(f)$ の変化の有無

Step 1. $RD_{in}(B) \leftarrow \{\langle u, f_u\text{–}par\rangle \mid u$ は f の仮引数変数$\} \cup \{\langle v, f_v\text{–}in\rangle \mid v$ は大域変数$\}$

† 但し，S が再帰呼出しを含まない手続き一つだけからなるときは，繰り返し解析してもその手続きの SuDEF, PoDEF, ImUSE は変化しないので，その手続きを1回解析するだけで S の解析を終える．

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

図6　関数定義の概略
Fig. 6　The outline of a function definition.

表2　各文での確実定義集合の計算方法

	各文の確実定義集合
代入文	左辺の変数が確実定義集合に入る
条件文	then節とelse節それぞれの確実定義集合の積集合が文全体の確実定義集合となる
繰返し文	確実定義はない(繰返し本体が一度も実行されない可能性のある前判定型であるため)
複合文	各文の確実定義集合の和集合が全体の確実定義集合となる
手続き呼出し文	呼び出される手続きの確実定義集合が文全体の確実定義集合となる
入力文	入力値を代入される変数が確実定義集合に入る
出力文	確実定義はない

表3　各文での潜在定義集合の計算方法

	各文の潜在定義集合
代入文	左辺の変数が潜在定義集合に入る
条件文	then節とelse節それぞれの潜在定義集合の和集合が文全体の潜在定義集合となる
繰返し文	条件成立時に実行される文の潜在定義集合が全体の潜在定義集合となる
複合文	各文の潜在定義集合の和集合が全体の潜在定義集合となる
手続き呼出し文	呼び出される手続きの潜在定義集合が文全体の潜在定義集合となる
入力文	入力値を代入される変数が潜在定義集合に入る
出力文	潜在定義はない

但し，複合文において，同じ変数に対する定義がある場合，より後の定義が全体の定義集合に含まれる(式(2)参照)．また，文が含む式に関数呼び出しがあるとき，呼び出される関数の定義集合をその文の定義集合に加える．また，呼び出される関数で暗使用される変数がその文で参照されたようにDD関係辺をつなぐ．

Step 2.　B 内の文を表2，表3に基づいて解析する†．これによって，f 内のCD関係辺，DD関係辺が作られ，$RD_{out}(B)$，$SuDEF(B)$，$PoDEF(B)$，$ImUSE(B)$ の値が計算される（但し，$ImUSE(B)$ は f が呼び出す手続きの現在の `ImUSE` の値を使って計算し直した結果を意味する）．

Step 3.　次のようにして，各変数の変化を調べる（但し，*Global* は変数集合の中の大域変数だけを選ぶことを意味する）．

$$SuDEF_{var}(f)==Global(SuDEF_{var}(B))$$
$$PoDEF_{var}(f)==Global(PoDEF_{var}(B))$$
$$ImUSE(f)==ImUSE(B)$$

左辺の各変数は，初期値として与えたもの，または，この解析が2度目かそれ以降のときは，前回の解析時に代入されたものである．これらのうち，一つでも変化したものがあるときは，次の操作をして定義集合が変化したことを記録しておく．

$$SuDEF(f)\leftarrow\{\langle v, f_v\text{-}out\rangle \mid v \in Global(SuDEF_{var}(B))\}$$
$$PoDEF(f)\leftarrow\{\langle u, f_u\text{-}out\rangle \mid u \in Global(PoDEF_{var}(B))\}$$
$$ImUSE(f)\leftarrow ImUSE(B)$$

Step 4.　$RD_{out}(B)$ 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exit節点へのDD関係辺 $(n \xrightarrow{f} f\text{-}exit)$ を作る．また，同様に，各大域変数 g に関する定義を探し，その定義節点 m から，out節点へのDD関係辺 $(m \xrightarrow{g} f_g\text{-}out)$ を作る．

3.4　解　析　例

ここでは，**3.2.** で示したアルゴリズムANALYZEPROGRAMを使って，図7の左のプログラムを解析する様子を示す．

はじめに，Step 1でプログラムprogressionのCGを作り，強連結成分を抽出すると（図8），手続きの解析順として，$\{f\} \Rightarrow \{nth\}$ が得られる．次に，Step 2を行い，Step 3で各変数に初期値を代入する．その後，Step 1で得られた解析順に従って，Step 4(a)で $\{f\}$ 内の手続き f をANALYZEPROCEDUREを使って解析するとその本体 B での定義集合が次のようになる．

$$SuDEF_{var}(B)=\{f, g\}$$
$$PoDEF_{var}(B)=\{f, g, l\}$$
$$ImUSE(B)=\phi$$

† より詳しくは，文献(9)を参照されたい．

```
 1  program progression(input,output);
 2      var     g:integer;
 3
 4      function f(p:integer):integer;
 5          var     l:integer;
 6      begin
 7          if p>1 then begin
 8              l := 2 * f(p-1);
 9              g := g + l;
10              f := l;
11          end
12          else begin
13              g := 1;
14              f := 1;
15          end;
16      end;
17
18      procedure nth;
19          var     n,a:integer;
20      begin
21          g := 1;
22          readln(n);
23          a := f(n);
24          writeln(a);
25      end;
26  begin
27      g := 0;
28      nth;
29      writeln(g);
30  end.
```

```
 1  program progression(input,output);
 2      var     g:integer;
 3
 4      function f(p:integer):integer;
 5          var     l:integer;
 6      begin
 7          if p>1 then begin
 8              l := 2 * f(p-1);
 9
10              f := l;
11          end
12          else begin
13
14              f := 1;
15          end;
16      end;
17
18      procedure nth;
19          var     n,a:integer;
20      begin
21
22          readln(n);
23          a := f(n);
24          writeln(a);
25      end;
26  begin
27
28
29
30  end.
```

図7　スライスの計算例
Fig. 7　An example of calculating a slice.

f は大域変数gを文9で参照するが，この参照の前に文8の手続き呼出しで，gを必ず定義するので，$ImUSE(B)$ にgは含まれない．このとき，$SuDEF_{var}(B)$ と $PoDEF_{var}(B)$ の値は初期値から変化しているので，各変数の値は以下のように更新され，Step 4(b)によって再度$\{f\}$に対して Step 4(a)が実行される．

$$SuDEF(f)\leftarrow\{\langle g, f_g\text{–}out\rangle\}$$
$$PoDEF(f)\leftarrow\{\langle g, f_g\text{–}out\rangle\}$$
$$ImUSE(f)\leftarrow\phi$$

($\{f\}$ は一つの要素しかもたないが，その手続き f が再帰呼出しを含むので再度解析しなければならない.)

2回目の$\{f\}$の解析では定義集合が変化しないので$\{f\}$の解析が終る．次に Step 4(a)により，$\{nth\}$の解析をする．$\{nth\}$の1回目の解析により，その定義集合は次のように更新される．

$$SuDEF(nth)\leftarrow\{\langle g, nth_g\text{–}out\rangle\}$$
$$PoDEF(nth)\leftarrow\{\langle g, nth_g\text{–}out\rangle\}$$
$$ImUSE(f)\leftarrow\phi$$

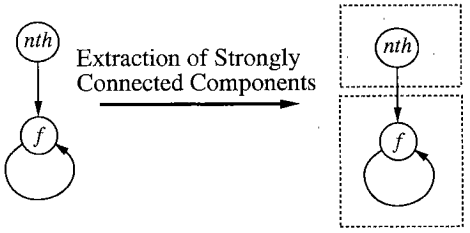


図8　プログラムprogressionに対するCG
Fig. 8　The call graph for the program progression.

$\{nth\}$ は再帰呼出しを含まない手続き一つからなる強連結成分なので解析を繰り返してもその定義集合は変化しない．よって，$\{nth\}$ の解析は1回で終わる．

最後に，Step 5でメインプログラムを解析してプログラム全体の解析を終える．

4.　スライスの計算

この章ではPDG上でスライスを計算する方法を示す（図2の破線矢印部分）．本アルゴリズムでは，「文 s における変数 v に関するスライスとは，PDG上でCD関係辺またはDD関係辺をたどって文 s の変数 v に到達できる節点集合に対応する文の集合である」と

定義したので，PDG上でのスライスの計算は節点間の到達可能性の判定と同じである．但し，我々が定義したPDGには手続きの境界を超えてスライスを計算できるようにいくつかの拡張がなされているので，たどることのできる辺に制限が付く．

プログラム内の文sでの変数vに関するスライスを表す節点の集合Vは，以下に示すような手順で計算できる．但し，n_sはsに対応する節点である．

アルゴリズム GETSLICEONPDG

Input: PDG，文s，変数v

Output: スライスを表す節点の集合V

Step 1. $V \leftarrow \{n_s\}$

Step 2. $N \leftarrow \{n \mid n \xrightarrow{v} n_s\}^{\dagger} \cup \{m \mid m - - \rightarrow n_s\}$

Step 3. $N \neq \phi$の間，以下の操作を繰り返す

(a) $n \in N$を一つ選ぶ

(b) $N \leftarrow N - \{n\}$

(c) $V \leftarrow V \cup \{n\}$

(d) nがin節点でも引数節点でもないなら次の操作をする

$N \leftarrow N \cup \{m | m \notin V \wedge (m \longrightarrow n \vee m - - \rightarrow n)\}$

(但し$m \longrightarrow n$はmからnへの任意の変数のDD関係辺を表す)

スライスを計算する対象を文s全体にするにはStep 2を次のように変更すればよい．

Step 2'. $N \leftarrow V$

また，文の集合Sに対するスライスを計算するときは，更に，Step 1を次のように変更すればよい．

Step 1'. $V \leftarrow \bigcup_{s_i \in S} (s_i$を表す節点$)$

節点とプログラムの断片は1対1に対応しているので，Vが求まると，それに対応したプログラムを作り上げることは簡単である．例えば，図9のPDG上で文`writeln(a)`での変数`a`についてのスライスを計算した結果が図9中の破線四角形で示された節点であり，この節点集合に対応するソースコードの部分を集めたものが図7の右のプログラムとなる．

5. アルゴリズムの複雑さ

この章では**3.**，**4.**で紹介したアルゴリズムの複雑さについて述べる．我々のアルゴリズムは，ソースコードをPDGに変換する部分(ANALYZEPROGRAM)と，PDG上でスライスを計算する部分(GETSLICEON-PDG)とに分けられるので，複雑さについても両者を分けて考える．

5.1 PDGを作るコスト

ソースコードからPDGを作るための解析にかかわる要素を以下の表のように定義する．

PDG作成にかかわる要素	
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数の最大値
S_i	手続き呼出しの総数
S_t	文の総数

[補題 1] プログラム中のすべての手続きが再帰呼出しを含むときが解析に最も時間がかかる．

(証明) すべての手続きを少なくとも1回は解析しなければならないことは明らかである．再帰呼出しを含まない手続きは，その手続き一つだけからなる強連結成分に含まれる．この強連結成分の解析はそれが含む手続きを1回解析するだけで終わるので，このような手続きが存在しないときに，すなわち，プログラム中のすべての手続きが再帰呼出しを含むときに，プログラム全体の解析に最も時間がかかる． □

[補題 2] 手続きのSuDEF, PoDEF, ImUSEに含まれる要素は，解析の繰返しによって単調に増加する．

(証明) 手続きp内での，手続き呼出しによらない直接的な，確実定義集合をS_dとすると，$SuDEF(p)$はS_dとpが直接呼び出す別の手続きのSuDEFだけを変数としてもつ，否定を含まない積と和だけからなる論理式で表現される．この論理式とS_dの値はプログラムが変化しない限り変化せず，各手続きのSuDEFは最小（空集合）に初期化されるので，$SuDEF(p)$に含まれる要素は単調に増加する．

同様のことが$PoDEF(p)$と$ImUSE(p)$についても言えるので，これらに含まれる要素も単調に増加する． □

[補題 3] アルゴリズムANALYZEPROGRAMでは一つの強連結成分Sに含まれる手続きの解析（Step 4(a)）の繰返しはたかだか$|S|$回で終わる（但し，$|S|$

† より正確には$\{n \mid \langle v, n \rangle \in RD_{in}(s)\}$であるが，一般に，ユーザは変数$v$の参照を含む文を$s$に指定する場合が多いのでここでは，このわかりやすい表現を用いた．

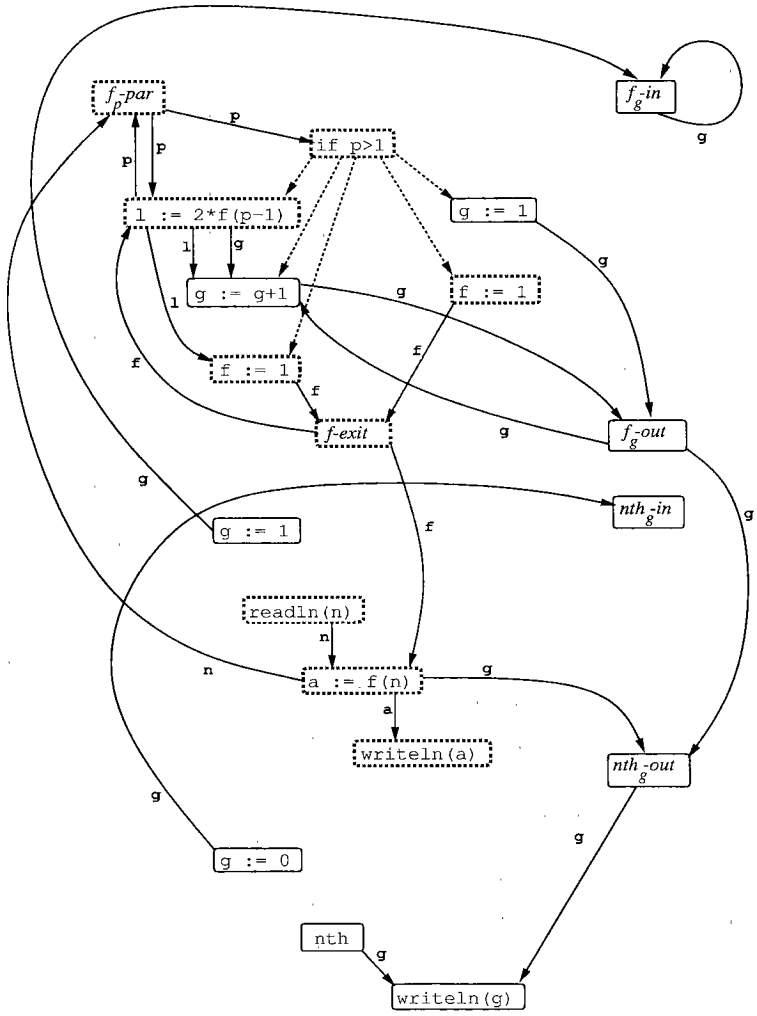


図9 プログラムprogressionに対するPDG: 破線四角形は文 `writeln(a)` での変数 `a` に関するスライスを計算したときの節点集合に含まれる節点である

Fig. 9 The PDG for the program progression; vertices enclosed with a dashed rectangle means a slice with respect to the statement `writeln(a)` and the variable `a`.

はSに含まれる手続きの数を表す)．

(証明) 強連結成分Sに含まれる手続きpの$SuDEF_{var}(p)$に大域変数gが含まれていることを$g_p = true$で，含まれていないことを$g_p = false$で表現する．

$SuDEF(p)$は，手続きp内での，手続き呼出しによらない直接的な，確実定義集合とpが直接呼び出す別の手続きの`SuDEF`だけを変数としてもつ，積と和だけからなる論理式で表現される．

我々の入力言語仕様では別名(Alias)が発生しないので，どの大域変数も互いに影響を及ぼすことはない．よって，$SuDEF(p)$を変数ごとに分解することができ，g_pはpが直接呼び出す手続き$q_1, q_2, \ldots$に関する$g_{q_1}, g_{q_2}, \ldots$を変数としてもつ，積と和だけからなる論理式で表現される．この$g_{q_1}, g_{q_2}, \ldots$のうち，Sに含まれない手続きに関するものは既に値が決定しており，変化することはない．

また，補題2 より，ある g_p が初期値 $false$ からいったん $true$ に変化すると再び $false$ に変化することはない．

S 全体を1回解析して，S に含まれる手続き p のどの g_p も変化しなければ，その後の解析で変化することはないので，S 全体を1回解析するごとに S に含まれる一つの手続き p の g_p にだけ変化が起こるとき，S の解析の回数が最大となる．これは，SuDEF だけでなく，PoDEF, ImUSE に対しても同様である．これは AnalyzeProgram が停止することと同時に，S に含まれる手続きの解析はたかだか $|S|$ 回で終わることを示している． □

[定理 1] アルゴリズム AnalyzeProgram の時間計算量は $O(P + S_i + P \cdot S_t \cdot (G + L))$ である．

(証明) アルゴリズム AnalyzeProgram の Step 1, Step 2 に要する時間はプログラムの CG の節点と有向辺の数の和に比例する．節点の数は P で，有向辺の数は S_i で，それぞれ抑えられる．Step 3 は手続きの数に比例する時間で終わる．

AnalyzeProgram の繰返し部分 Step 4(a) は，補題1，3 より，プログラム中のすべての手続きが一つの強連結成分に含まれているとき，最も時間がかかる．このとき，プログラム中のすべての手続きが最大 P 回ずつ解析される．

プログラム中のすべての手続きを1回ずつ解析する，すなわち，プログラム中のすべての文を1回解析するのに要する時間は，文の数に比例する．一つの文の解析には定数回の集合演算が必要である．1回の集合演算にかかる時間が集合に含まれる要素の数に比例すると仮定すると，一つの文の解析の時間は大域変数と局所変数の数の和に比例する．

故に，アルゴリズム AnalyzeProgram の時間計算量は $O(P + S_i + P \cdot S_t \cdot (G + L))$ である． □

5.2 PDG上でスライスを計算するコスト

PDG にかかわる要素を以下のように定義する．

PDG上でのスライスの計算に関わる要素
V PDGの節点の総数
E PDGの有向辺の総数

アルゴリズム GetSliceOnPDG では，PDG上でスライスを計算する際，すべての節点とすべての有向辺を多くとも1回しかたどらないので，そのコストは $O(V + E)$ である．

5.3 解析全体にかかるコスト

ソースコードから始めてスライスを計算するまでに $O(P + S_i + P \cdot S_t \cdot (G + L) + V + E)$ の時間がかかるが，その後ソースコードに変更がなければ，$O(V + E)$ の時間でスライスの再計算ができる．

但し，これはプログラム中のすべての手続きが互いに呼び合う可能性がある場合で，このようなプログラムは，非常にまれにしか存在しないと考えられる．よって，一般的なプログラムに対しては，$O(P + S_i + S_t \cdot (G + L) + V + E)$ の時間でスライスの計算ができる．

6. むすび

本論文では，再帰を含むプログラムにおいて，文の間の依存関係を解析し，プログラムをPDGに変換し，その上でのスライスを計算するアルゴリズムを紹介した．本アルゴリズムは，実際の解析が行われる前に，控え目な解を初期値として定義しておき，必要に応じて解析を繰り返して真の解に収束させるというもので，再帰を含むプログラムの解析には適したものであると思われる．

我々のアルゴリズムはWeiserのものと比べ，手続き間のスライス計算をより正確に行うことができる点，および，再帰を取り扱える点で有利である．例えば，図7のプログラムの文21や27は文29での変数gの値には影響を及ぼしていないにもかかわらず，Weiserのアルゴリズムでは及ぼしていると解釈されスライスに含められる．

再帰を含むプログラムのスライスを計算するアルゴリズムは，ほかに，Hwangによるもの[5]とHorwitzによるもの[3]がある．前者のアルゴリズムは，再帰呼出しの深さごとに，スライスを計算し，それらの和集合が収束するまでその計算をするというもので，PDGを作らない．よって，プログラムを解析してPDGで表し，その上でスライスを計算する我々のアルゴリズムに比べると，スライスを何度も計算するときに非常に不利になり，実際のデバッグ環境を考えると満足できるものであるとは言いがたい．また，ソースコードの変更の影響が及ぶ箇所を知るためなどに使われる順方向スライス (Forward Slice) を計算するために，PDGをそのまま使うことができる点でも我々のアルゴリズムは有利である．

後者のアルゴリズムに比べると，我々のアルゴリズムでは，プログラム内の各文の到達定義集合を計算して

いるので，プログラムの任意の地点の任意の変数に対するスライスが計算できる点，および，ソースコードが変更されたときに，再解析が必要になる部分をより少なくでき，インクリメンタルにPDGを更新する方法をとることができる点で有利であると考えられる．更に，時間計算量を比較すると，HorwitzによるものがP_rを仮引数の数の最大値，$D \leq (S_t+1)\cdot(G+P_r)$としたとき，$O\left((P+S_i)\cdot D^2\cdot(G+P_r)^2\right)$であるのに対して，我々のアルゴリズムでは$O(P+S_i+P\cdot S_t\cdot(G+L))$である．

また，関連するアルゴリズムとしては，Harroldらの「定義–参照」関係の計算法がある[12]．この方法では，手続きにまたがるDD関係を正確に解析することができるが，**3.2.**で述べるような関数間の解析順を考慮していないため，通常，手続き間の解析情報の伝搬に多くの時間を要する．また，Pandeらはある種のポインタを含むCプログラムの関数間にまたがる「定義–参照」関係を求めるアルゴリズムを提案している[13]．

我々は，本アルゴリズムに従ってスライスを計算し，表示する試作システムを作成し，本アルゴリズムが正しく動作することを確認した．このシステムはSUN SPARCstation ELC上で動作し，13個の手続き，7個の大域変数を含む再帰呼出しのない249行のプログラムを0.80秒で，また，8個の手続き，9個の大域変数を含み，すべての手続きがお互いに呼び合う可能性のある275行のプログラムを1.82秒で，それぞれ解析することができる．今後，このシステムをより使いやすいユーザインタフェースを備えた本格的なシステムにする予定である．

文　献

(1) Weiser M.: "Program Slicing", Proceedings of the Fifth International Conference on Software Engineering, pp. 439–449(1981).

(2) Horwitz S., Reps T. and Binkley D.: "Interprocedural Slicing Using Dependence Graphs", Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, pp. 35–46(1988).

(3) Horwitz S., Reps T. and Binkley D.: "Interprocedural Slicing Using Dependence Graphs", ACM Transactions on Programming Languages and Systems, **12**, 1, pp. 26–60(1990).

(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) Hwang J.C., Du, M.W. and Chou C.R.:"Finding Program Slices for Recursive Procedures", Proceedings of the IEEE COMPSAC '88, pp. 220–227(1988).

(6) Ottenstein K.J. and Ottenstein L.M.:"The Program Dependence Graph in a Software Development Environment", Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5) pp. 177–184(1984).

(7) 下村隆夫:"Program Slicing 技術とテスト，デバッグ，保守への応用", 情報処理, **33**, 9, pp. 1078–1086(1992).

(8) Kingston J.H.:"Algorithms and Data Structures : Design, Correctness, Analysis ", Addison-Wesley, 1990.

(9) 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(1993–09).

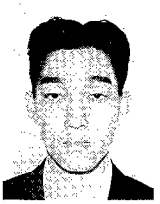
(10) Aho A.V., Sethi S. and Ullman J.D.:"Compilers : Principles, Techniques, and Tools", Addison-Wesley(1986).

(11) Shimomura T.:"Bug Localization Based on Error-Cause–Chasing Methods", Transactions of information Processing Society of Japan, **34**, 3, pp. 489–500(1993).

(12) Harrold M. J. and Soffa M. L. :"Efficient Computation of Interprocedural Definition–Use Chains", ACM Transactions on Programming Languages and Systems, **16**, 2, pp. 175–204(1994).

(13) Pande H. D., Landi W. A. and Ryder B. G. :"Interprocedural Def–Use Associations for C systems with Single Level Pointers", IEEE Transactions on Software Engineering, **20**, 5, pp. 385–403(1994).

(平成6年3月7日受付，8月15日再受付)



植田　良一

平4阪大・基礎工・情報中退．平6同大大学院修士課程了．同年日立製作所システム開発研究所入所．プログラムスライスの研究に従事．

練　　林

昭59中国四川大・計算機科学系卒．昭62同大修士課程了．現在阪大大学院博士後期課程在学中．プログラムスライス，プログラム比較ツールおよびテストプロセスの研究に従事．

井上　克郎

昭54阪大・基礎工・情報卒．昭59同大大学院博士課程了．同年同大・基礎工・情報・助手．昭59～昭61ハワイ大マノア校・情報工学科・助教授．平1阪大・基礎工・情報・講師．平3年11月同学科・助教授．工博．ソフトウェア工学の研究に従事．ACM，IEEE，情報処理学会各会員．

鳥居　宏次

昭37阪大・工・通信卒．昭42同大大学院博士課程了．同年電気試験所（現電子技術総合研究所）入所．昭50ソフトウェア部言語処理研究室室長．昭59阪大・基礎工・情報・教授．平4奈良先端科学技術大学院大学・教授，阪大・基礎工・情報・教授併任．工博．ソフトウェア工学の研究に従事．情報処理学会，日本ソフトウェア科学会，人工知能学会，ACM，IEEE各会員．