ソフトウェア工学 94－5
(1993. 9. 21)

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

植田 良一[†] 練 林[†] 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—1
大阪大学 基礎工学部 情報工学科 鳥居研究室
Tel: 06-844-1151(内線 4816)
E-mail : r-ueda@ics.es.osaka-u.ac.jp

あらまし

プログラムスライシング技術は,プログラム内のある地点のある変数の値に影響を与える文（slice）を抽出する技術で,様々な分野で応用されている.本論文では,プログラム依存グラフおよび到達定義集合をもとにして,再帰を含むプログラムにも適応可能なプログラムスライシングアルゴリズムを紹介する.この手法は従来のアルゴリズムに比べて,対話的にプログラムのslice情報を提供するシステムに容易に組み込むことができる.

和文キーワード 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

Dependence Analysis and Program Slicing of Recursive Programs

Ryoichi UEDA,[†] Lin LIAN,[†] Katsuro INOUE,[†] and Koji TORII[‡]

[†] Department of Information and Computer Sciences
Faculty of Engineering Science, Osaka University

[‡] Advanced Institute of Science and Technology, Nara

1-1 Machikaneyama, Toyonaka, Osaka 560, Japan
Tel: 06-844-1151(4816)
E-mail: r-ueda@ics.es.osaka-u.ac.jp

Abstract

The program slicing technique, originally introduced by Mark Weiser, is so useful that it is used in various fields. The slices of a program are collection of all statements that might affect the value of selected variables at a selected point in the program. In this paper we propose a slicing algorithm, where we introduce the extended program dependence graphs and the reaching definitions sets. This algorithm can be naturally applicable to self / mutual recursive programs as well as non recursive ones. Also, it would be easily implemented for a program development system which interactively gives the developers various slice information.

英文 key words Program Dependence Graph, Reaching Definition, Recursion, Control Dependence, Data Dependence

1 まえがき

プログラム P のスライス (slice) とは, 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 slice, 特定の入力データに対して解析したものを動的 slice という (本論文では静的 slice のみを扱うので, 以下単に slice と言えばそれは, 静的 slice を意味するものとする). slice の技法は Mark Weiser [2] によってはじめて考案され, 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 [5, 8]. slice 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, ポインタ型変数や再帰が存在するような現実的な言語上で正確な解析をするのは容易ではない.

Weiser は, slice を計算するために, データフロー方程式を使ったが, これはひとつの手続き内を対象としたものだった. Ottenstein [7] が, この slice 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 [3] が, 手続きの境界を越えて slice 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 slice 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なうため, 実際のシステム作成には応用しにくい. また, Hwang [6] は, 再帰を含むプログラムに対して最小浮動点を求める手法を用いて, slice 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 slice 情報を提供するシステムには組み込みにくい.

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 slice 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.

関数および手続きがあることで問題になるのは, ある文に関数呼び出しがある時, その文には現れない変数の定義や参照が発生することである. よって, データフロー方程式における *kill* と *gen* の集合の値が計算しにくい事が問題となる. 例えば, 図 1 のようなプログラムを解析する時, 文 1

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

図 1: プログラムの一部

の定義が文 3 へ到達するかどうかは関数 f の内容に依存する. slice を計算することを目的とする場合のひかえめな解は, 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 通りに分けることで, より正確な解を得ることができる.

- f が必ず g を定義する時, 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 を定義する可能性がある時, f 内の定義と 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時, 1 が 3 に到達する.

また, 再帰を許すことで問題になるのは, 再帰的に定義された関数が, 1. 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, 2. if 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, 繰り返し文と似た性質を持っていることや, 3. ある関数の解析を終える前に, その関数の呼び出しを含む文を処理しなければならない可能性があることである.

そこで, 我々はこれらに対処するために, *kill* と *gen* を直接計算するのではなく, その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.

本論文ではこれ以降, 2 節で, プログラム依存グラフに関する定義を, 3 節で, 入力プログラムからプログラム依存グラフを作る方法を, 4 節で, プログラム依存グラフ上で, slice を計算する方法を, 5 節で, 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プログラムスライシング

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持つ Pascal 風言語を想定している. その言語には文として条件文 (if), 代入文, 繰り返し文 (while), 入力文 (readln), 出力文 (writeln), 手続き呼びだし文, 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 (図 5 参照). 手続きおよび関数内では, 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, 他の関数内の局所変数は参照できない. 関数および手続きは, 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 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時, s_1 と s_2 の間に CD 関係が発生する. これを CD 関係辺 ($s_1 \dashrightarrow s_2$) 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**[1]時, 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.

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

[1] 「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

定部分, 代入文, 入出力文, 手続き呼びだし文を表し, その有向辺は2つの文の間の制御依存および, データ依存関係を表す. ただし, 関数や手続きの境界を越えて, slice を計算できるようにするために, 以下に述べるような, いくつかの特殊な節点も存在する.

entry 節点 関数および手続きの入口を表す節点で, 各関数および手続きにひとつずつある (関数 f に関するものを f–*entry* と表す).

exit 節点 関数の出口を表す節点で, 各関数にひとつずつある (関数 f に関するものを f–*exit* と表す).

in 節点 関数および手続き外からの大域変数の影響を伝えるための節点で, 各関数および手続きに対して, その中で参照される大域変数それぞれに対して, ひとつずつある (大域変数 g に関するものを g–*in* と表す).

out 節点 関数および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, 中で定義される可能性のある大域変数それぞれに対して, ひとつずつある (大域変数 g に関するものを g–*out* と表す).

引数節点 関数および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 (引数 p に関するものを p–*par* と表す).

文 s における slice とは, CD 関係辺, または, DD 関係辺をたどって s に到達できるすべての文の集合である. 例えば, 図2 のプログラムの文 `writeln(c)` に関する slice は, 図3 の様になり, この部分だけを実行しても指定した文での結果は同じものになる.

```
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 2: プログラム atoi

3 解析方法

変数 v と節点 n との組 d を $\langle v, n\rangle$ と表し, 変数を d_{var} で, 節点を d_{vertex} で参照することにする. プログラムのある領域 S (連続する0個以上の文または式の集合) の到達定義集合 (Reaching Definition) を $RD_{in}(S)$ と書き, これを上で述べた組の集合とする ($RD_{in}(S) \stackrel{\text{def}}{=} \{\langle v, n\rangle \mid$ 節点 n で変数 v が定義される$\wedge n$ が S に到達する $\}$) .

```
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 3: atoi の文 `writeln(c)` に関する slice

プログラムのある領域 B について, B を実行した時に (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, $SuDEF(B)$ 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, $PoDEF(B)$ と表す. さらに, ある定義集合 S に対して,

$$S_{var}(B) \stackrel{\text{def}}{=} \{d_{var} \mid d \in S(B)\}$$

と定義する.

3.1 プログラム全体の解析

プログラムは関数および手続きをひとつの単位として解析される. 関数および手続きの解析を開始する前に, すべての関数および手続きに対して, その確実定義集合と潜在定義集合を以下のように定義する (ただし, $SuDEF(f)$ は関数 f 全体に対する確実定義集合を意味する).

$$SuDEF(f) \stackrel{\text{def}}{=} \{\langle v, v\text{–}out\rangle \mid v\text{は大域変数}\}$$

$$PoDEF(f) \stackrel{\text{def}}{=} \{\langle v, v\text{–}out\rangle \mid v\text{は大域変数}\}$$

また, P を関数および手続きの集合として, 以下の操作でプログラム全体の解析を行なう.

1. $P \leftarrow \{p \mid p$はプログラム内の関数および手続き$\}$
2. $P \neq \phi$ の間, 次の操作を繰り返す
   - $q \in P$ をひとつ選ぶ
   - $P \leftarrow P - \{q\}$
   - q の解析をする (3.2 節参照)
   - もし, $SuDEF(q)$ または $PoDEF(q)$ の値が変化したら, $P \leftarrow P \cup \{r \mid r$は$q$を直接呼び出す関数または手続き$\}$
3. メインプログラムを解析する.

3.2 関数および手続き定義の処理

図 4 のような関数定義は, 以下の手順で解析される. 手続きの場合, exit 節点を作らないので, 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.

1. $RD_{in}(B) \leftarrow \{\langle w, w\text{-}par\rangle \mid w$は$f$の仮引数変数$\} \cup \{\langle v, v\text{-}in\rangle \mid v$は大域変数$\}$
2. $f\text{-}entry$ から B 内の各文 s への CD 関係辺 ($f\text{-}entry \dashrightarrow s$) を作り, B 内の文を 3.3 節から 3.9 節で定義される手順で解析する.
3. 大域変数だけについて見た時, $(SuDEF_{var}(f) = SuDEF_{var}(B)) \wedge (PoDEF_{var}(f) = PoDEF_{var}(B))$ が満たされていないなら, $SuDEF(f) \leftarrow \{\langle v, v\text{-}out\rangle \mid v \in SuDEF_{var}(B)\}$, $PoDEF(f) \leftarrow \{\langle w, w\text{-}out\rangle \mid w \in PoDEF_{var}(B)\}$ として, 満たされるようになるまで B の解析を繰り返す.
4. $RD_{out}(f)$ 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 exit 節点への DD 関係辺 ($n \xrightarrow{f} f\text{-}exit$) を作る. また, すべての大域変数 g に関する定義を探し, その定義節点 m から, out 節点への DD 関係辺 ($m \xrightarrow{g} g\text{-}out$) を作る.

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

図 4: 関数定義の例

このひとつの関数および手続きの定義の解析が終るかどうかは, 3 で条件が満たされるかどうかにのみ依存する. 3.6 節の定義からわかるように, $SuDEF_{var}(f)$ は f の各文の $SuDEF$ の和集合からなるので, 文の $SuDEF$ の変化に依存する. 関数呼び出しを含まない文の $SuDEF$ は, 最初の解析以降変化することはない. 関数呼び出しを含む文に関して, $SuDEF(f)$ は大域変数に関して最大 (すべての大域変数がその関数で定義される) から始まるので, 2 回目の解析以降でその要素が増えることはあり得ない. よって, 各文の $SuDEF$ は必ず以前の $SuDEF$ の部分集合になる. これはすなわち, $SuDEF(f)$ が必ず収束することを示しており, $PoDEF(f)$ についても同じことがいえる. ゆえに, このアルゴリズムは停止する.

3.3 代入文での処理

S: $v := exp$ の時

1. $RD_{in}(exp) \leftarrow RD_{in}(S)$ として exp の処理をする.
2. $RD_{in}(S) \leftarrow RD_{out}(exp)$
3. $SuDEF(S) \leftarrow \{\langle v, S\rangle\} \cup SuDEF(exp)$
4. $PoDEF(S) \leftarrow \{\langle v, S\rangle\} \cup PoDEF(exp)$
5. $RD_{in}(S)$ 内の変数 v に関する定義を取り除いて, $RD_{out}(S)$ とする.
   $RD_{out}(S) \leftarrow \{d \mid d \in RD_{in}(S) \wedge d_{var} \neq v\} \cup \{\langle v, S\rangle\}$

3.4 条件文での処理

S: **if** exp **then** B_t **else** B_f の時

1. $RD_{in}(exp) \leftarrow RD_{in}(S)$ として exp の処理をする.
2. B_t, B_f 内の各文へ S の条件判定部分からの CD 関係辺を作る.
3. $RD_{in}(B_t) \leftarrow RD_{out}(exp)$, $RD_{in}(B_f) \leftarrow RD_{out}(exp)$ として B_t, B_f の処理をする.
4. B_t, B_f が共に定義する変数に関する定義を S の確実定義とする ($SuDEF(S) \leftarrow \{d \mid d_{var} \in SuDEF_{var}(B_t) \wedge d_{var} \in SuDEF_{var}(B_f)\}$).
5. $PoDEF(S) \leftarrow PoDEF(B_t) \cup PoDEF(B_f)$
6. $RD_{out}(S) \leftarrow PoDEF(S) \cup \{d \mid d \in RD_{in}(S) \wedge d_{var} \notin SuDEF_{var}(S)\}$
7. $SuDEF(S) \leftarrow SuDEF(exp) \cup SuDEF(S)$
8. $PoDEF(S) \leftarrow PoDEF(exp) \cup PoDEF(S)$

3.5 繰り返し文での処理

S: **while** exp **do** B の時

1. $RD_{in}(exp) \leftarrow RD_{in}(S)$ として exp の処理をする.
2. $RD_{in}(S) \leftarrow RD_{out}(exp), RD_{in}(B) \leftarrow RD_{out}(exp)$ として B の処理をする.
3. $RD_{out}(S) \leftarrow RD_{in}(S) \cup PoDEF(B)$ とする.
4. $RD_{in}(S) \leftarrow RD_{out}(S)$ として, 1 から 3 までをもう一度繰り返す.
5. B 内の各文へ S の条件判定部分からの CD 関係辺を作る.
6. $SuDEF(S) \leftarrow SuDEF(exp)$
7. $PoDEF(S) \leftarrow PoDEF(exp) \cup PoDEF(B)$

3.6 複合文での処理

S: **begin** $S_1; \cdots; S_n$ **end** の時

1. $RD_{in}(S_1) \leftarrow RD_{in}(S)$ として, S_1 の処理をする.
2. $RD_{in}(S_k) \leftarrow RD_{out}(S_{k-1})$ $(1 < k \leq n)$ として, すべての文についての処理をする.
3. $RD_{out}(S) \leftarrow RD_{out}(S_n)$
4. $SuDEF(S) \leftarrow \bigcup_{k=1}^{n} SuDEF(S_k)$
5. $PoDEF(S) \leftarrow \bigcup_{k=1}^{n} PoDEF(S_k)$

3.7 手続き呼び出し文での処理

S: $p(exp_1, \ldots, exp_n)$ の時

1. $RD_{in}(exp_1) \leftarrow RD_{in}(S)$ として, 式 exp_1 の処理をする.
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. $RD_{out}(S) \leftarrow \{d \mid d \in RD_{in}(S) \wedge d_{var} \notin SuDEF_{var}(p)\} \cup \{\langle v, S\rangle \mid v \in SuDEF_{var}(p)\} \cup PoDEF(p)$
5. $SuDEF(S) \leftarrow SuDEF(p) \cup \bigcup_{k=1}^{n} SuDEF(exp_k)$
6. $PoDEF(S) \leftarrow PoDEF(p) \cup \bigcup_{k=1}^{n} PoDEF(exp_k)$

3.8 入出力文での処理

S: **readln**(v) の時, v への代入文と同じとして処理する.

S: **writeln**(v) の時, v を参照する式と同じ様に処理する.

3.9 式の処理

節点 n に含まれている式 exp は以下の手順で処理される.

1. $RD_{out}(exp) \leftarrow RD_{in}(exp)$
2. $SuDEF(exp) \leftarrow \phi$
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 $(d_{vertex} \xrightarrow{v} n)$ を作る.
   - ある関数 g (g は exp を含む関数と同じであっても良い) の呼び出しがあれば, その引数を先に式の列として処理をした後, g–$exit$ から n への DD 関係辺 $(g\text{–}exit \xrightarrow{g} n)$ を作り, その時点での, $RD_{out}(exp)$ 内のすべての大域変数 v の定義 d それぞれに対して, v の定義から g の v–in 節点への DD 関係辺 $(d_{vertex} \xrightarrow{v} v\text{–}in)$ を作る. さらに, 以下のように各集合の値を更新する

     $RD_{out}(exp) \leftarrow \{\langle v, n\rangle \mid v \in SuDEF_{var}(g)\} \cup PoDEF(f) \cup \{d \mid d \in RD_{out}(exp) \wedge d_{var} \notin SuDEF_{var}(g)\}$

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

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

   という処理をその式の終りまで続ける.
4. exp が関数または手続きの引数である時, 現節点から対応する関数の引数節点への DD 関係辺 $(n \xrightarrow{v} p\text{–}par)$ を作る (v は exp 内で参照されている変数を, p は対応する仮引数変数を, それぞれ表す).

解析例

図 5 のプログラムを上で示した方法で解析すると, 図 7 の様になる. このプログラムには故意に無駄な代入文 21, 27 を入れてあるが, これらの影響は関数 f には伝わっていない. しかも, 代入文 13 の影響は f の g–out 節点を通して 9 へ伝わっている. 出力文 24 での変数 a に対する slice を計算した結果を図 6 に示す.

4 slice の求め方

3 節の方法でプログラムから PDG G ができたら, プログラム内の文 S での変数 v に関する slice を表す節点の集合 V は, E を G に含まれるのすべての辺の集合として, 以下に示すような手順で計算できる.

1. $V \leftarrow \{n \mid n$ は Sに対応する節点 $\}$
2. $N \leftarrow \{n \mid \langle v, n\rangle \in RD_{in}(S)\}$
3. $N \neq \phi$ の間, 次の操作を繰り返す.
   - $l \in N$ をひとつ選ぶ
   - $N \leftarrow N - \{l\}$
   - $V \leftarrow V \cup \{l\}$
   - $N \leftarrow N \cup \{k \mid (k \xrightarrow{*} l \in E \vee k \dashrightarrow l \in E) \wedge k \notin V\}$ (ただし $k \xrightarrow{*} l$ は k から l への DD 関係辺ならどのような変数のものでも構わないことを意味する)

節点とプログラムの断片は一対一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.

5 アルゴリズムの複雑さ

3 節で述べた解析方法では, プログラム内のすべての関数および手続きが, 相互再帰に関わっている場合, 最もコストがかかる. 関数 f の 1 回の解析の結果, ひとつの大域変数が $SuDEF(f)$ から削除されるという変化が f を解析するたびに起こると, 各関数および手続きの定義を, 大域変数の総数回解析することになる. また, ひとつの関数および手続きを解析する際, その中に繰り返し文が含まれているとその内側の文をもう一度解析する必要が生じる (3.5 節参照) ので, n 重のループがある時, 最も内側にある文は 2^n 回解析されることになる (しかし, n は現実のプログラムにおいてはある定数で抑えられると考えられる).

よって, プログラム内の大域変数の総数を G, 関数および手続きの総数を P, ループの深さの最大値を N, ひとつの関数または手続きに含まれる文の最大数を S とすると, 解析にかかるコストは, $O(G \cdot P + S \cdot 2^N)$ となる.

```
 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.
```

図 5: 数列の計算をするプログラム

```
 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 6: 出力文 24 での変数 a についての slice

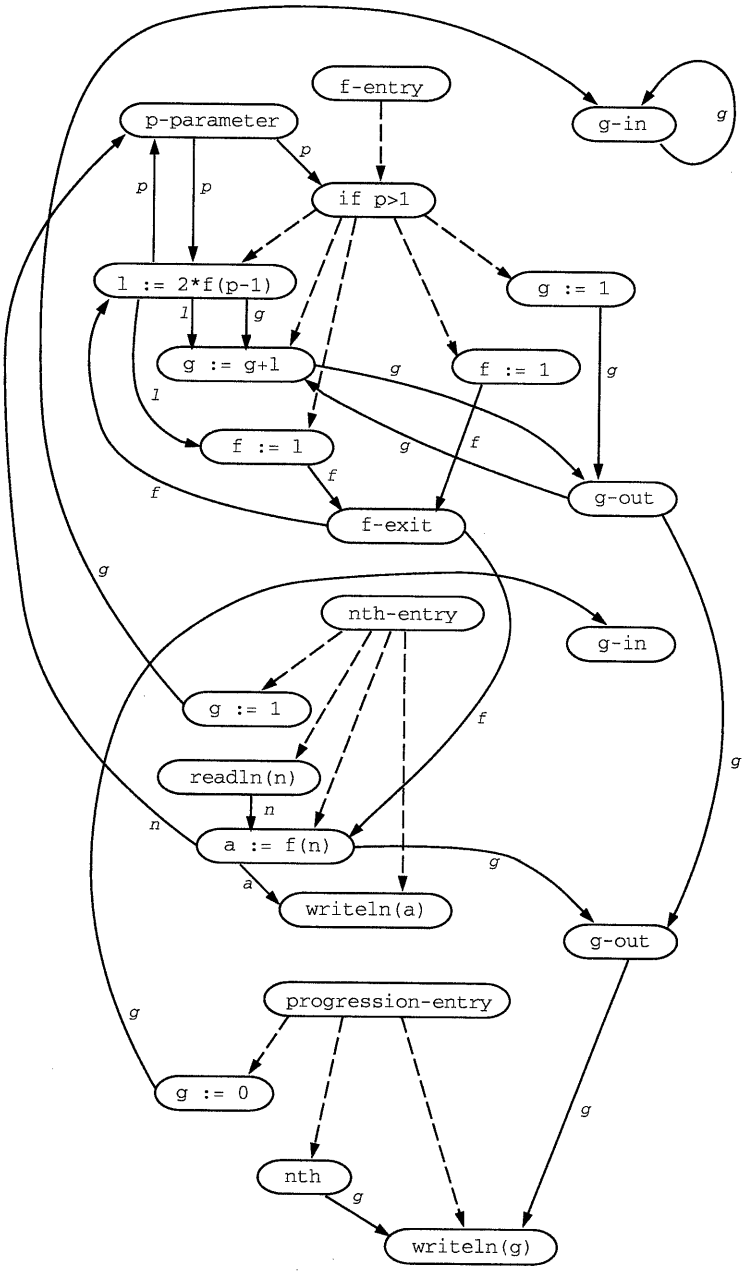

図 7: 数列の計算をするプログラムに対する PDG

6 まとめ

本論文では, 再帰を含むプログラムにおける, 文の間の依存関係の解析方法と, その結果できる PDG 上での slice の計算方法を紹介した. この中で我々が使った方法は, 実際の解析が行なわれる前に, ひかえめ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.

再帰を含むプログラムの slice を計算するアルゴリズムは, 他に, Hwang によるものがある [6]. このアルゴリズムは, 再帰呼び出しの深さ毎に, slice 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 slice を計算する我々のアルゴリズムに比べると, slice を何度も計算する時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.

Horwitz [4] による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 slice が計算できる点で, 有利であると考えられる.

参考文献

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

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

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] 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).

[8] 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, Vol. 33, No. 9, pp. 1078–1086(1992).

[9] Horwitz, S., Prins, J. and Reps, T.:"Integrating Non-interfering Versions of Programs", ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, pp. 345–387(1989).

[10] Horwitz, S., Pfeiffer, P. and Reps, T.:"Dependence Analysis for Pointer Variables", Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, pp. 28–40(1989).

[11] Venkatesh, G.A.:"The Semantic Approach to Program Slicing", Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pp. 107–119(1991).

[12] Korel, B. and Laski, J.:"Dynamic Program Slicing", Information Processing Letters 29, pp. 155–163(1988).

[13] 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).