論　文

Go to なしプログラムのデータ・フロー解析*

原　田　賢　一**

Abstract

Data flow analysis which determines data relationships at each point in a program is needed for production type compilers to generate efficient object code. And it also provides a programmer with useful information on usage of variables in a program. The data flow problems are devided into two classes: what definitions can affect computations at a point, and what uses can be affeced by computations at that point.

In this paper, recursive descendant algorithms for solving the latter problem on a goto-less program are described. First, the basis of the data flow problem is given and some terminologies are defined. Second, the intraprocedural data flow analysis algorithm is shown which uses only available information within a procedure. Finally, the interprocedural data flow analysis algorithm is presented taking account of calling relations between procedures.

1. はじめに

電子計算機の性能が向上し，大規模なものが広く用いられるようになるにつれて，その性能を十分に生かすために，効率のよいオブジェクト・コードの生成を主目的としたコインパイラはますます必要となってくる．とくにそのような機能が要求されるのは，言語プロセッサなどのシステム・プログラムの作成を容易にするために開発された高レベル言語に対するコンパイラである．効率のよいコードを生成するには，与えられたソース・プログラム全体にわたって制御およびデータの流れに関する情報を収集し，解析を行わなければならない．全域的な最適化 (global optimization) はそれらの解析結果に基づいて行われるものであり，一般にはつぎに示すような技法が広く用いられている[2),10),11)]．

(i)　共通部分式の除去
(ii)　不変コードの移動 (code motion)
(iii)　定数の伝播 (constant propagation)
(iv)　未使用コードの除去 (dead code elimination)
(v)　高速レジスタの有効割付け

このような技法を適用するには，与えられたプログラムを実行するときに，各文またはそれらの構成要素について，つぎのことが明らかになっていなければならない．

(1)　変数の参照があった場合，その変数の値はプログラムのどこで定義されたものが影響しているか，
(2)　どの変数があとのプログラムの実行に影響を与えるか．

上述の(i)～(iii)による最適化を行うには(1)の情報が必要であり，(iv)と(v)では(2)の情報が必要とされる．

データ・フロー解析はこれらの情報を収集することを目的としている．そのためには与えられたプログラムの制御の流れを解析し，そのすべての流れにそって変数の使われ方を解析していかなければならない．一般にはつぎの3つの処理からなっている[3)]．

(1)　与えられたプログラムを基本ブロック (basic block) と呼ぶ単位に分割する．基本ブロックは一連の文またはその構成要素，あるいは中間言語で表現されたテキストの列 (sequence) からなる．基本ブロックがいくつかの文からなって

* Data Flow Analysis for a Goto-less Program by Ken'ichi HARADA (Keio Institute of Information Science, Keio University)

** 慶応義塾大学情報科学研究所

いる場合を想定すると，ある基本ブロックに制御が与えられたときには，まずその先頭の文が実行される．以後，その文が逐次実行され，最後の文が実行されたときは，無条件に，あるいは単純な判定条件によって制御は他のブロックに移る．ブロックの途中から他のブロックへ制御が移ることはない．

(2) 各基本ブロックを節 N (node), それらの間の制御の流れを辺 E (edge) として，もとのプログラムの制御の流れをフロー・グラフ $G=(N, E, s)$ で表わす．s は実行開始点となる節である．

(3) Gを走査して各節に対応した基本ブロックについて，上述の情報を集める．その方法には2通りのものがある．1つはGの各辺を巡回しながら，各節に対応した基本ブロックのもつ情報が収束するまで反復を行う方法である[4),6),8)]．もう1つは，Gをインターバルと呼ぶ単位で分割して新しい表現のグラフを作る方法である．それと同時に各インターバルに含まれる節に対応した基本ブロックの情報を集約する．Gが単一の節だけになるまでこの操作を繰り返す．つぎに逆の操作を行いながら，前半の処理で集めた情報を各基本ブロックに分配していく[3),7)]．

ここでは，BLISS[12)]や SIMPL[5)]などのように go to 文のないプログラミング言語に対するコンパイラでのデータ・フロー解析，とくに前述の(2)の問題について考える．**2.**ではデータ・フロー解析に関する基本的な考え方を述べ，用語を定義する．**3.**では go to なしプログラミング言語のモデルを示し，基本ブロックを拡張したブロックについて述べる．**4.**ではこのブロックに基づく1つの手続き内での解析 (intraprocedural analysis)，**5.**では手続きの呼出し関係を考慮した手続き間の解析 (interprocedural analysis) について述べる．

これらの解析方法はつぎのような特徴をもっている．

- 言語の性質上，実行時における制御の流れはプログラムの文の種類によって明確にとらえることができる．したがって，プログラムを基本ブロックにまとめなおしたり，フロー・グラフを作ったりする必要はなく，コンパイラの過程で作り出される中間言語のテキストを直接利用して解析ができる．
- recursive descendant な方法で各文は1回走査するだけで解析ができる．したがって，その処理に要するコストは解析しようとするプログラムの文の数nに正比例するので，time complexity は $O(n)$ である[1)]．
- 手続き間の解析において，各手続きは1回走査するだけでよい．

2. データ・フロー解析

まず，**Fig. 1** に示すような基本ブロック B とその直接先行ブロック (immediate predecessor) と直接後続ブロック (immediate successor) の間の関係を考える．基本ブロック $P_1, P_2, \cdots$, または P_n を実行したあと，制御はBに移り，Bの実行が終了したあとは，その結果によって後続の基本ブロック $S_1, S_2, \cdots$, あるいは S_m に制御が移るものとする．ブロック B に制御が移される点をBの入口，Bから制御がはなれる点をBの出口という．

プログラムで使用している変数の集合をVとする．プログラムのある点以降の実行あるいは1つの基本ブロックの実行において，変数 $v \in V$ の使われ方をつぎのように分類する．

(i) まず参照が行われる．そのあと値の再定義あるいは参照が行われることがある．

(ii) まず値の再定義が行われる．そのあと参照あるいは再定義が行われることがある．

(iii) 再定義も参照も行われない．

変数vの実際の使われ方は，これらの場合の1つ，あるいは制御の流れ方によってそれらを組み合わせたものとなる．プログラム中のある点p以降の実行において，変数vの使われ方として(i)に該当することがある場合，vはpにおいて **busy** であるという．そうでない場合には，vはpにおいて **free** であるという．

基本ブロックBの入口で busy な変数の集合を $IN(B)$, 出口で busy な変数の集合を $OUT(B)$ とする，

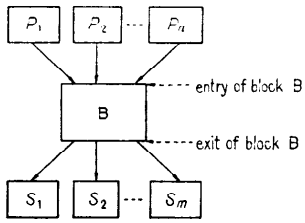


Fig. 1 A basic block

Bの出口で busy な変数は，その各直接後続ブロック S_i, $i=1,2,\cdots,m$ の入口で busy な変数の和集合であるから，つぎの関係が成立つ．

$$\mathrm{OUT}(B)=\mathrm{IN}(S_1)\cup\mathrm{IN}(S_2)\cup\cdots\cup\mathrm{IN}(S_m)$$
$$=\bigcup_{i=1}^{m}\mathrm{IN}(S_i) \qquad (1)$$

つぎに IN(B) と OUT(B) との関係を考える．Bを実行したときに，そこで使用されている変数の中で上述(i)の使われ方をすることがあるものの集合をREF(B) とする．変数 $v\in V$ がBの実行において，上述の(i)あるいは(iii)の使われ方をすることがある場合，vはBにおいて **definition free** (以下，D-free と書く) であるという．すなわち，つぎの場合である．

(1) $v\in$REF(B) または

(2) vはBの中では使用されていない．

Bで D-free な変数の集合を DFR(B) で表わす．Bの入口で busy な変数は，REF(B) の要素，あるいはその出口で busy でかつBでは D-free なものである．したがって，つぎの関係が成立つ．

$$\mathrm{IN}(B)=\mathrm{REF}(B)\cup[\mathrm{OUT}(B)\cap\mathrm{DFR}(B)]$$
$$=\mathrm{REF}(B)\cup[[\bigcup_{i=1}^{m}\mathrm{IN}(S_i)]\cap\mathrm{DFR}(B)] \qquad (2)$$

本論文で述べるデータ・フロー解析は，各ブロックの入口と出口における busy な変数の集合を求めることが目的である．プログラムはいくつかの手続きからなっているものとすると，1つの手続きの実行が終了したときには，すべての変数は free となる．したがって，手続きの実行の終りを1つの基本ブロックEとすると，

$$\mathrm{IN}(E)=\phi \quad (\phi \text{ は空集合を表わす})$$

である．これが busy な変数の集合を求めるときの初期値となる．

データ・フロー解析の例を **Fig. 2** に示す．たとえば，Fig. 2(b)のブロック $B6$ では，変数 a, b および t が参照されるので，その入口で busy な変数の集合は，

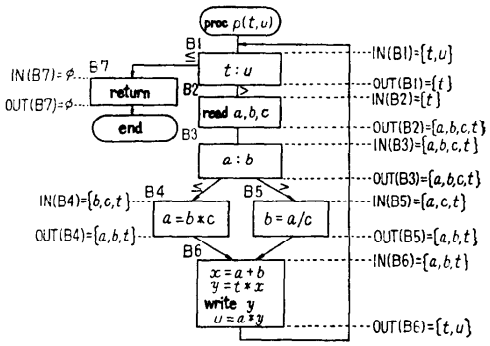
$$\mathrm{IN}(B6)=\{a,b,t\}$$

となる．$B6$ が実行されたとき，そこで使用されている変数 a, b, t, u, x, y のうちで，xとyとは制御が $B6$ をはなれたあとはどこでも参照されることはないので，それらは出口で free となる．tとuとは以後の実行で参照される ($B1$ で) ので

$$\mathrm{OUT}(B6)=\{t,u\}$$

```
1  proc p(t,u);
2    dcl a,b,c,x,y;
3    while (t>u)
4      do;
5        read a,b,c;
6        if a≤b
7          then a=b*c;
8          else b=a/c;
9        do; x=a+b; y=t*u;
10           write y; u=a*y; end;
11     end;
12   return;
13   end;
```

(a) source program



(b) result of the analysis

Fig. 2 An example of data flow analysis for busy variables

である．

3. Go to なしプログラミング言語のモデルとブロック

ここで解析の対象とするプログラムは，**Fig. 3** (次頁参照) に示すようなプログラミング言語Lで書かれているものとする．データ・フロー解析では，言語Lの制御構造と各文における変数の使われ方だけに着目するので，以下に言語Lの概要を簡単に説明する．

(1) プログラムは全域的変数の宣言 (global decralation) で始まり，そのあとに手続きの定義を並べたものが続く．

(2) 手続きはその中で有効な局所的変数の宣言 (local decralation) で始まり，そのあとに文の並びを書いたものである．

(3) 各種の文については，Pascal や PL/I にあるものなので説明を省略する．case 文を実行すると，式eの値をiとしたとき $c_j=i$ となるようなjがあれば，S_j が実行される．そのような c_j がなかったとき，**else** の指定があれば S_{k+1} が実行される．また，**else** の指定がなけ

```
global decralation
proc p1(a1, a2, ···, an)
    local decralation
    ⟨statement list⟩
        ⋮
proc c pk(a1, a2, ···, am)
    local decralation
    ⟨statement list⟩
```

where $p_1, \cdots, p_k$: procedure names, $a_1, a_2, \cdots$: formal parameters

⟨statement list⟩::=[⟨statement list⟩] ⟨statement⟩;

⟨statement⟩::=⟨basic statement⟩|⟨compound statement⟩|⟨if statement⟩|⟨repeat statement⟩|⟨while statement⟩|⟨case statement⟩|⟨return statement⟩

⟨basic statement⟩::=v=e|**read** $v_1, v_2, \cdots, v_n$|**write** $v_1, v_2, \cdots, v_m$|**call** $p(e_1, e_2, \cdots, e_l)$

⟨compound statement⟩::=**do**; S **end**

⟨if statement⟩::=**if** e **then** S_1 [; **else** S_2]

⟨repeat statement⟩::=**repeat** ⟨statement⟩ **until** (e)

⟨while statement⟩::=**while** (e) ⟨statement⟩

⟨case statement⟩::=**case** e **of** $\backslash c_1 \backslash S_1 \backslash c_2 \backslash S_2 \cdots \backslash c_k \backslash S_k$ [; **else** S_{k+1}]

⟨return statement⟩::=**return** [(e)]

where, $e, e_1, e_2, \cdots, e_l$: expressions, $v_1, v_2, \cdots$: variables, $S, S_1, S_2, \cdots$: ⟨statement list⟩, $c_1, c_2, \cdots, c_k$: constants, p: procedure name, [ ] means optional syntax.

Fig. 3 *A* **go to** less programming language *L*

れば，ただちにつぎの文の実行に移る．

(4) 変数は単純変数だけとする．式は変数と関数の引用とからなるものとする．説明を簡単にするために関数のパラメタのすべて値とり (call by value) とする．

Fig. 2(a)はこの言語でのプログラム例である．

ここに示した文 (⟨statement⟩) およびその中で用いられる一部の式 (⟨expr⟩) を構成するコードの列をブロックと呼び，その対応づけをつぎのようにする．

(1) 各文は1つのブロックを構成する．

(2) 文の並び (⟨statement list⟩) も1つのブロックを構成する．

(3) if 文，repeat 文，while 文，および case 文においては，条件式(e)も1つのブロックを構成しているものとする．

(4) ブロックどおしは，与えられたプログラムにおける制御の流れに従って互いに関連づけられるものとする．

1つのブロックは必ずしも分岐命令で終るとはかぎらないことと，ブロック自身がいくつかのブロックを含むことがあるという点において，ブロックと基本ブロックは異なっている．

基本ブロックBに関して述べた，IN(B), OUT(B), REF(B) および DFR(B) の定義を以下ブロックに対してもそのまま適用することにする．REF(B) は，ブロックBに制御が与えられて，Bから他へ制御が移る

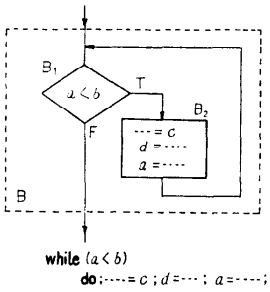


Fig. 4 An example of a **while** block

間のすべての制御の流れを考えたとき，ある流れにおいて **2.** の(i)に該当する使われ方がなされることのある変数の集合を表わすものとする．DFR(B) についても同様でつぎのように定義する．

DFR(B)=REF(B)∪{B内のある制御の流れにおいては使用されることのない変数}

While 文のブロックの例を **Fig. 4** に示す．この図において B_2 内の各代入文に対するブロックの表示は省略してある．Bに制御が与えられて，B_1 と B_2 が何回か実行されるものとすると，変数 a, b, c はそれぞれまず参照が行われるので，

$$\mathrm{REF}(B)=\{a, b, c\}$$

となる．Bの実行において，B_1 を1回実行しただけで制御がBから他へ移ったとすると，変数 a, b が参照されるだけであるから，Vをその手続きで使用している変数の集合とすると，

$$\mathrm{DFR}(B)=\mathrm{REF}(B)\cup[V-\{a, b\}]$$
$$=V.$$

4. 手続き内データ・フロー解析

はじめに，1つの手続きの中だけで得られる情報を用いた解析方法について述べる．

データ・フロー解析は，プログラムを構成している各ブロック B_i, $i=1,2,\cdots,n$ に対する IN(B_i) および OUT(B_i) を求めるのが最終の目的である．最適コードの生成に必要な情報としては OUT(B_i) だけで実際には役立つ．仮定として，コンパイルの意味解析の段階で代入文や入出力文などの基本文 (⟨basic statement⟩) に対するブロック B_i に関しては call 文を除いて，REF(B_i) とのブロック内で値が定義される変数の集合 DEF(B_i) とが正確に得られているものとする．また，各手続きの制御構造はすでに明らかにされているものとする．このとき，IN(B_i) および OUT(B_i) を求めるのに必要となる D-free な変数の

集合 DFR(B_i) はつぎのようにして決定することができる.

[アルゴリズム 1] D-free な変数の決定

入力: 手続きPに含まれるすべての基本文のブロック B_i, $i=1,2,\cdots,m$ に対する REF(B_i) と DEF(B_i)

出力: 各ブロックに対する DFR(B_j) と REF(B_j) $j=1,2,\cdots,m$

手続きPを文の列 $S_1; S_2; \cdots; S_n$ とする. 各文およびそこに含まれる構成要素のブロックに対する DFR と REF は, P で与えられた文の順序に従って, 以下に示す式を再帰的に適用することによって求められる.

いま処理しようとしているブロックをBとする. このアルゴリズムを最初に適用するとき, Bは手続きPに対するブロックとする.

(1.1) Bが基本文または式のとき

$$\mathrm{DFR}(B)=\mathrm{REF}(B)\cup[V_P-\mathrm{DEF}(B)]$$

ここで, V_P は手続きPにおける局所的変数 (local variable) および非局所的変数 (global variable) の集合である. call 文の場合には, 手続き呼出しによって変数がどのような影響を受けるのかわからないので, 最適化によって最悪の状態を想定しなければならない. すなわち, 実パラメタおよび非局所的変数は手続き呼出しによって, 呼び出された手続きの中でまず参照が行われ, それから値が再定義されることを仮定する.

(1.2) Bが文の並びまたは複合文 (〈compound statement〉) のとき

Fig. 5 に示すような構造を考える.

$$\mathrm{REF}_i(B)=\mathrm{REF}_{i-1}(B)\cup\mathrm{REF}(B_i)\quad i=1,2,\cdots,n$$

ただし, $\mathrm{REF}_0(B)=\phi$ とする.

$$\mathrm{DFR}(B)=\bigcap_{i=1}^{n}[\mathrm{REF}_{i-1}(B)\cup\mathrm{DFR}(B_i)],$$

$$\mathrm{REF}(B)=\mathrm{REF}_i(B)\cap\mathrm{DFR}(B).$$

Bに含まれる各ブロック B_i, $i=1,2,\cdots,n$ について再帰的にこのアルゴリズムを適用し, REF(B_i) と DFR(B_i) を求めてから, B に対する計算を行う.

(1.3) Bが if 文または case 文のとき

一般に **Fig. 6** に示すような制御構造を考える.

(a) **else** の指定がある場合

$$\mathrm{DFR}(B)=\mathrm{REF}(e)\cup\left[\bigcup_{i=1}^{n+1}\mathrm{DFR}(B_i)\right],$$

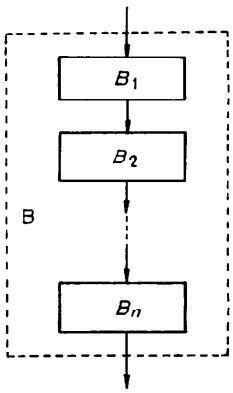


Fig. 5 Control structure of a sequential operation

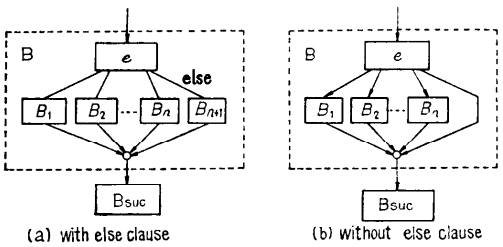


Fig. 6 Control structure of a selection

$$\mathrm{REF}(B)=\mathrm{REF}(e)\cup\left[\bigcup_{i=1}^{n+1}\mathrm{REF}(B_i)\right].$$

(b) **else** の指定がない場合

$$\mathrm{DFR}(B)=V_P,$$

$$\mathrm{REF}(B)=\mathrm{REF}(e)\cup\left[\bigcup_{i=1}^{n}\mathrm{REF}(B_i)\right].$$

(1.4) Bが while文のとき

while (e) B_W

という形の文に対するブロックを考えると, つぎのようになる.

$$\mathrm{DFR}(B)=V_P,$$

$$\mathrm{REF}(B)=\mathrm{REF}(e)\cup\mathrm{REF}(B_W).$$

(1.5) Bが repeat 文のとき

repeat B_R **until** (e)

という形の文に対するブロックを考えると, つぎのようになる.

$$\mathrm{DFR}(B)=\mathrm{DFR}(B_R),$$

$$\mathrm{REF}(B)=\mathrm{REF}(B_R)\cup[\mathrm{REF}(e)\cap\mathrm{DFR}(B)].$$

(1.6) Bが **return** (e) のとき

$$\mathrm{DFR}(B)=V_P,$$

$$\mathrm{REF}(B)=\mathrm{REF}(e).$$

ただし, eがないときには REF(e)=ϕ とする. ☒

以上の関係からも明らかなように，これらの計算は recursive descendant な方法で意味解析と同時に行うことができる．Fig. 2 の例ではつぎのような結果が得られる．Fig. 2(a)に示した各文に対するブロックを S_i で表わす．ただし，i は図における行番号とする．

$$V_P=\{a,b,c,t,u,x,y\}$$

$\mathrm{DFR}(B1)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(B1)=\{t,u\}$.
$\mathrm{DFR}(S5)=\{t,u,x,y\}$, $\mathrm{REF}(S5)=\phi$.
$\mathrm{DFR}(B3)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(B3)=\{a,b\}$.
$\mathrm{DFR}(S7)=\{b,c,t,u,x,y\}$, $\mathrm{REF}(S7)=\{b,c\}$
$\mathrm{DFR}(S8)=\{a,c,t,u,x,y\}$, $\mathrm{REF}(S8)=\{a,c\}$
$\mathrm{DFR}(S6)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(S6)=\{a,b,c\}$.

9, 10 行目の各代入文に対する結果は省略する．

$\mathrm{DFR}(S9)=\{a,b,c,t\}$, $\mathrm{REF}(S8)=\{a,b,t\}$.
$\mathrm{DFR}(S4)=\{t\}$, $\mathrm{REF}(S4)=\{t\}$.
$\mathrm{DFR}(S3)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(S3)=\{t,u\}$.
$\mathrm{DFR}(S12)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(S12)=\phi$.
$\mathrm{DFR}(S1)=\{a,b,c,t,u,x,y\}$, $\mathrm{REF}(S1)=\{t,u\}$.

D-free な変数が求められたときには，**2.** の(1)および(2)式の関係にもとづいて，1つの手続き内の各ブロックについて，その入口あるいは出口で busy な変数の集合を求めることができる．

まず，つぎのような手続きを考える．

$$\mathrm{LVIO}(B, B_{suc})$$

ここで，B は文または文の並びに対するブロック，B_{suc} は B の直接後続ブロックとする．LVIO は B_{suc} の入口で busy な変数が求められているとき，B およびそこに含まれるブロックの入口および出口で busy な変数を決定するための手続きで，つぎのように定義される．

[アルゴリズム 2] ブロックの入口および出口で busy な変数の決定: $\mathrm{LVIO}(B, B_{suc})$

入力: B および B に含まれるブロックを b としたとき，$\mathrm{DFR}(b)$ と $\mathrm{REF}(b)$，ならびに B_{suc} の入口で busy な変数の集合 $\mathrm{IN}(B_{suc})$

出力: 各 b の入口および出口で busy な変数の集合 $\mathrm{IN}(b)$ および $\mathrm{OUT}(b)$

B のブロックの型によって，それぞれ以下の計算を行う．

(2.1) B が **return** (e) のとき

$$\mathrm{OUT}(B)=\phi,$$
$$\mathrm{IN}(B)=\mathrm{REF}(e).$$

(2.2) その他の場合は，**2.** の(2)式によって，まず $\mathrm{OUT}(B)$ と $\mathrm{IN}(B)$ を計算する．

$$\mathrm{OUT}(B)=\mathrm{IN}(B_{suc}),$$
$$\mathrm{IN}(B)=\mathrm{REF}(B)\cup[\mathrm{OUT}(B)\cap\mathrm{DFR}(B)].$$

B が基本文以外の文に対するブロックであるときには，さらに以下の計算をする．

(2.3) B が文の並びまたは複合文のとき

一般に Fig. 5 に示した構造を考える．B を構成している各ブロック $B_1, B_2, \cdots, B_n$ についてそれぞれ，IN と OUT を計算するために，この手続きを再帰的に呼ぶ．すなわち，

call $\mathrm{LVIO}(B_i, B_{i+1})$, $i=n, n-1, \cdots, 1$

ただし $B_{n+1}=B_{suc}$ とする．

(2.4) B が if 文または case 文のとき

Fig. 6 に示した構造を考える．

$$\mathrm{IN}(e)=\mathrm{IN}(B).$$

(a) **else** の指定がある場合

$B_1, B_2, \cdots, B_{n+1}$ における IN および OUT を計算する．すなわち，

call $\mathrm{LVIO}(B_i, B_{suc})$, $i=1,2,\cdots,n+1$

e の出口で busy な変数の集合は **2.** の(1)式の関係からつぎのようになる．

$$\mathrm{OUT}(e)=\bigcup_{i=1}^{n+1}\mathrm{IN}(B_i).$$

(b) **else** の指定がない場合

$B_1, B_2, \cdots, B_n$ における IN および OUT を計算する．すなわち，

call $\mathrm{LVIO}(B_i, B_{suc})$, $i=1,2,\cdots,n$

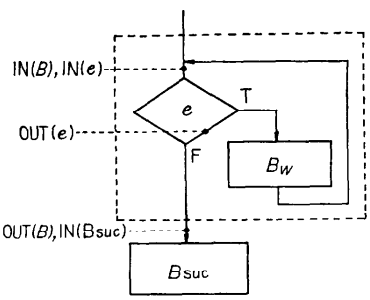
$$\mathrm{OUT}(e)=\left[\bigcup_{i=1}^{n}\mathrm{IN}(B_i)\right]\cup\mathrm{IN}(B_{suc}).$$

(2.5) B が while 文のとき

Fig. 7 に示す構造を考える．

$$\mathrm{IN}(e)=\mathrm{IN}(B).$$

e は B_W の直接後続ブロックでもあるので，IN



while (e) B_W ; Bsuc

Fig. 7 A structure of a **while** block

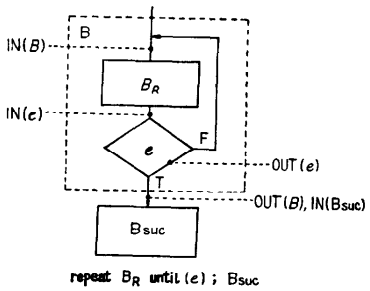


Fig. 8 A structure of a **repeat** block

(e)を用いて，IN(B_W)と OUT(B_W) を求める．すなわち，

call LVIO(B_W, e).

つぎに IN(B_W) を用いて OUT(e)を計算する．

$$OUT(e)=IN(B_{suc})\cup IN(B_W).$$

(2.6) Bが repeat 文のとき

Fig. 8 に示す構造を考える．

$$OUT(e)=IN(B)\cup OUT(B),$$
$$IN(e)=REF(e)\cup OUT(e).$$

つぎに，IN(e) を用いて，IN(B_R) および OUT(B_R) を計算する．

call LVIO(B_R, e). ⊠

上述の手続き LVIO を用いた手続き内のデータ・フロー解析のアルゴリズムはつぎのようになる．

[アルゴリズム 3] 手続き内データ・フロー解析

入力: 解析しようとする手続きP内の各ブロックBにおける DFR(B) および REF(B)

出力: Pの各ブロックBにおける IN(B) および OUT(B)

手続きPを定義している文の並びに対するブロックQをとすると，

call LVIO(Q, S)

によって，上述の結果が得られる．ここで，Sは仮りのブロックで，IN(S)=ϕ とする． ⊠

4. 手続き間データ・フロー解析

1つの手続きの中だけでどのように情報を集めても，call 文の出現によって最悪の場合を想定したのでは，正確な解析結果を得ることはできない．そこで，手続き間の呼出しの関係を考慮した解析方法を考える．実行に必要な手続きはすべて与えられており，再帰的な手続きはないことを仮定する．

手続きPの中で他の手続きQを呼んでいるとき，その呼出しによって受ける変数の変化を求めるには，手

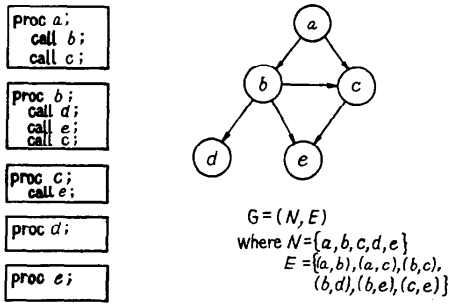


Fig. 9 An example of a call graph G

続きPを解析する前に，手続きQおよびQから呼び出されるすべての手続きがすでに解析されていなければならない．手続きの呼出しの関係をグラフで表現する．これを呼出しグラフ (call graph) と呼ぶ[4]．呼出しグラフ $G=(N, E)$ は各手続きを1つの節としたものの集合Nと呼出しの関係を辺としたものの集合Eで表わされる．プログラムの実行において，最初に制御が与えられる手続き以外のものは，必ず他の手続きから1回は呼び出されるものとする．**Fig. 9** に例を示す．

再帰的な呼出しを含まないという仮定から，Gはループを含まない．手続き P_i から手続き P_j への直接あるいは間接的な呼出しを $P_i \prec P_j$ と表わすと，Gの要素間には $\prec$ に関して半順序 (partial order) の関係が成り立つ．したがって，G を topological sort[9] することによって，つぎの性質を満たす集合 $K=(P_1, P_2, \cdots, P_n)$ を作ることができる．

$P_i \prec P_j$ ならば $i<j$ である． ⊠

手続きの解析順序は K を逆転したもの (inverse invocation order) $\bar{K}=(P_n, P_{n-1}, \cdots, P_1)$ で与えられる．Fig. 9 の例では

$K=(a, b, c, d, e)$, $\bar{K}=(e, c, d, b, a)$ となる．

手続きPが手続きQを直接呼び出している場合に，文 **call** Qのブロックに対する DFR, REF, および DEF の求め方について考える．このとき，パラメタの型には値とりと番地とり (call by address) の2種類があるものとする．

[アルゴリズム 4] call 文に対する DFR, REF, および DEF の決定．

手続きPにおける手続きQの呼出し **call** Qに対するブロックをBとする．

入力: Bにおける call 文のパラメタとして使用されている変数および DFR(Q), REF(Q) お

および DEF(Q)

出力: DFR(B), REF(B), および DEF(B)

(4.1) DFR(Q), REF(Q), および DEF(Q) のうちで，非局所的変数と仮パラメタだけを残し，仮パラメタをそれぞれに対応した実パラメタで置換した集合をつくる．それらをそれぞれ DFR(B), REF(B), および DEF(B) に代入する．

(4.2) Qで使用しているすべての非局所的変数とBで実パラメタとして使用している変数の集合 V_B を求める．すなわち，

$$V_B=\mathrm{REF}(B)\cup\mathrm{DEF}(B).$$

(4.3) 値とりのパラメタに対する実パラメタの値は手続きの呼出しによる影響を受けない．したがって，Qの値とりの仮パラメタに対する **call** Qの実パラメタ $u_1, u_2, \cdots, u_m$ は，Bにおいて D-free である．すなわち，

$$\mathrm{DFR}(B)=\mathrm{DFR}(B)\cup\{u_1, u_2, \cdots, u_m\}.$$

$u_1, u_2, \cdots, u_m$ の中でPの局所的変数，および DEF(Q) に含まれない非局所的変数を $v_1, v_2, \cdots, v_l$ とすれば，それらは，手続きQの呼出しなよって実際に値が再定義されることはないので，DEF(B) からとり除く．

$$\mathrm{DEF}(B)=\mathrm{DEF}(B)\cap[V_B-\{v_1, v_2, \cdots, v_l\}].$$

(4.4) 手続き P における局所的変数および非局所的変数の集合を V_P とすれば，

$$\mathrm{DFR}(B)=V_P-[V_B-\mathrm{DFR}(B)].$$ ☒

この方法においては，実パラメタとして1つの変数が何回か使用されているときに，それらに対応する仮パラメタが1つでも D-free であれば，その変数はBにおいて D-free となる．また，実パラメタとして非局所的変数 g が使用され，Qにおいても g が非局所的変数として使用されているときにも，同様のことがいえる．これらの場合について正確な結果を得るには，Qにおける制御の流れをたどって，定義と参照のいずれが先行するかを決定しなければならない．

いくつかの手続きからなるプログラムが与えられたとき，それらの手続き間の呼出しの関係を考慮に入れたデータ・フロー解析はつぎのようになる．

[アルゴリズム 5] 手続き間データ・フロー解析

(5.1) 手続きの呼出しグラフGを作成する．

(5.2) Gを topological sort して手続きの解析順序 $\overline{K}=(P_n, P_{n-1}, \cdots, P_1)$ を決定する．

(5.3) 手続 P_i, $i=n, n-1, \cdots, 1$ について，アルゴリズム1を適用して，各ブロックにおける DFR および REF を計算する．このとき，call 文のブロックの処理に対してアルゴリズム4を適用する．

(5.4) 各手続きについてアルゴリズム3を適用して各ブロックの入口と出口で busy な変数の集合を求める．このときの手続きの解析順序は任意でよい． ☒

5. むすび

Go to なしプログラミング言語を用いて書かれるプログラムは，一般に多数の小さな手続きの集まりからなるという特徴がみうけられる．したがって，手続き間の呼出しの関係を考慮しないで，1つの手続きの中だけで得られる情報を使って最適なコードを生成しようとすると，どんなに最適化の技法を用いても良いコードが作り出せないということもある[13]．このようなとき，ここで述べた手続き間データ・フロー解析は有効な情報を効率よく収集できるので大いに役立つ．

最後に，本研究をまとめるきっかけを与えてくださった，Dr. Mathew S. Hecht, Department of Computer Science, University of Maryland, ならびに日頃御指導をいただいている浦昭二教授に深く感謝いたします．

参考文献

1) Aho, A. V., Hopcroft, J. E. and Ullman, J. D.: The Design and Analysis of Computer Algorithms, p. 470, Addison-Wesley, Reading, Mass (1974).
2) Allen, F. E.: Program Optimization, Anual Review Automatic Programming, Vol. 5, pp. 239-307, Pergamon Press, New York (1969).
3) Allen, F. E.: Control Flow Analysis, SIGPLAN Notices, Vol. 5, No. 7, pp. 1-19 (1970).
4) Allen, F. E.: Interprocedural Data Flow Analysis, Proc. IFIP Congress 74 (1974).
5) Basili, V. R.: SIMPL-X: A Language for Writing Structured Programs, Technical Report TR-223, p. 43, Dept. of Computer Science, Univ. of Maryland (1973).
6) Hecht, M. S. and Ullman, J. D.: Analysis of a Simple Algorithm for Global Data Flow Problems, Conf. Record of ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, pp. 207-17, Boston, Mass., (1973).
7) Kennedy, K.: A Global Flow Analysis Algorithm, International J. Computer Math., Vol. 3, pp. 5-15 (1971).
8) Kildall, G. A.: A Unified Approach to Global

Program Optimization, Conf. Record of ACM Symposium on Principles of Programming Languages, pp. 194-206, Boston, Mass (1973).
9) Knuth, D. E.: Fundamental Algorithms, The Art of Computer Programming p. 634, 2 nd editioin, Addison-Wesley, Reading, Mass (1973).
10) Lowry, E. S. and Medlock, C. W.: Object Code Optimization, CACM, Vol. 12, No. 1, pp. 13-22 (1969).
11) Rustin, R., ed.: Disign and Optimization for Compiler, Courant Computer Science Symposium 5, p. 141, Prentice-Hall, Englewood Cliffs, N. J (1972).
12) Wulf, W. A., Rusell, D. B. and Habermann, A. W.: BLISS, A Language for System Programming, CACM, Vol. 14, No. 12, pp. 780-790 (1971).
13) Zelkowitz, M. V. and Bail, W. G.: Optimization of Structured Programs, Software Practice and Experience, Vol. 4, No. 5, pp. 51-57 (1974)

(昭和50年4月18日受付)
(昭和51年6月9日再受付)