ソフトウェア工学 102-3
(1995. 1.26)

COBOLを対象としたプログラムスライス計算方式

友納 正裕* 大竹 和雄**
小泉 昌紀*** 川崎 洋治* 中島 震*

*NEC C&C 研究所 **NEC クライアントサーバソフト技術研究所 ***NEC CASE 事業部

プログラムスライス技術は、ソースプログラムから注目する部分に関連するコードを切り出す技術であり、デバッグ、最適化、並列化、プログラム理解などに有用である。本稿では、プログラム理解支援を目的として、COBOLにおけるスライスの計算方式を検討する。COBOLプログラムには、手続き(サブルーチン)内にプログラム終了命令や他の手続きに飛び出すGOTO命令を記述することができ、この場合、従来の手法では正しいスライスを求めることができない。本稿では、手続き呼出命令および手続き本体に関して制御フローグラフを拡張し、これをもとに作成したプログラム依存グラフからスライスを求めることでこの問題を解決する方法を提案する。また、計算方式の正当性について簡単に考察する。

Static Program Slicing for COBOL

Masahiro Tomono* Kazuo Otake**
Masaki Koizumi*** Yoji Kawasaki* Shin Nakajima*

*C&C Research Laboratories, NEC Corporation
1-1, Miyazaki 4-Chome, Miyamae-ku, Kawasaki, Kanagawa 216, Japan

**Client-server Software Laboratories, NEC Corporation
11-5, Shibaura 2-Chome, Minato-ku, Tokyo, 108, Japan

***CASE Division, C&C Application Software Operation Unit, NEC Corporation
14-22, Shibaura 4-Chome, Minato-ku, Tokyo, 108, Japan

A static program slice is an extract of a program with respect to a program point and a variable. It is useful in program debugging, optimization, parallelization, and program understanding. This paper concerns the problems of program slicing for COBOL. The existing methods cannot handle the COBOL programs having STOP-RUN statements in procedures (sub-routines) and GOTO statements which jump to the labels in other procedures. It proposes the methods to cope with these problems by augmenting the Control Flow Graph with respect to procedure calls and procedure bodies. The correctness of the methods is briefly discussed.

1　はじめに

プログラムスライス技術は、ソースプログラムから注目する部分に関連するコードを切り出す技術である。抽出された部分をプログラムスライスあるいは単にスライスと呼ぶ。スライスは Weiser によって提案され、デバッグ、最適化、並列化、プログラム理解などに対する有用性が指摘されている [Weiser 84]。最近になって、C などの実用言語に対する研究が行なわれるようになった [JZR 91]。

本稿では、 COBOL プログラムのスライス計算方式を検討する[1]。 COBOL は膨大な既存資産をもつため、その保守におけるプログラム理解支援の基盤技術としてスライスが有効であるといわれている。

スライス計算の問題点として、

(1) プログラムが手続き呼出 (サブルーチンコール) を含む場合

(2) GOTO などの非構造化命令を含む場合

がある。 (1) に関しては、 [HRB 90] で構造化命令だけをもつ言語に対する方式が提案されている。 (2) に関しては [CF 94] で手続き呼出を考慮しない場合の方式が提案されている。しかし、 COBOL では両者が混在することがあり、その場合の検討はなされていない。

本稿では、上記の問題に対処するスライス計算方式を提案する。以下、第 2 章でスライスの定義を述べ、COBOL プログラムのスライスを求める際の問題点を整理する。第 3 章で PERFORM 命令を含むプログラムのスライス計算方式を述べる。第 4 章で STOP-RUN 命令を含む手続きを呼出すプログラム、第 5 章で GO TO 命令を含む手続きを呼出すプログラムのスライス計算方式について説明する。第 6 章で、本稿で提案するスライス計算方式の正当性について簡単に考察する。

2　プログラムスライス

スライスは次のように定義される [HRB 90]。

定義 1 プログラム P の文 p と変数 x に関するスライスとは、 p における x の値に影響を与える可能性のある P の文および述語の集合である。

これは静的スライス (static slice) と呼ばれ、プログラムへの入力とは独立に求められる。これに対して、ある入力を仮定した場合の動的スライス (dynamic slice) があるが、本稿では静的スライスを対象とする。

スライス計算の方法として、データフロー方程式を解く方法 [Weiser 84] と、プログラム依存グラフ (Program Dependece Graph、以下 PDG) を用いる方法 [FOW 87] がある。前者がスライス計算のたびにデータフロー方程式を解くのに対して、後者では、あらかじめ PDG を求めておき、 PDG 上での注目ノードからの到達可能範囲としてスライスを求める。後者は、データフロー方程式の計算を PDG を作成する際にまとめて行なうので、個々のスライスの計算量は少なくてすむ。本稿では後者を採用する。

PDG は制御フローグラフ (Control Flow Graph、以下 CFG) から制御依存グラフとデータ依存グラフを求めることにより得られる。単一手続き内の PDG は、制御依存グラフは [FOW 87] により、データ依存グラフは Kildall の反復解法 [ASU 86] により計算することができる。しかし、前述のように、手続き呼出や GOTO などの非構造化命令が混在する場合は、うまく対処できない。

COBOL で上記問題の原因となる命令を以下に示す。

(1) 手続き呼出命令 (PERFORM 命令)
COBOL では、 PERFORM、 SORT、 MERGE が手続き呼出機能をもつが、一般性を失わないので PERFORM に限って考える。たとえば、「PERFORM L1 TO L2」は段落 L1 から段落 L2 の終りまでをサブルーチンとして呼び出す。引数は記述できない。呼び出された手続きと呼び出した手続きがソースコードを共有できるという特徴がある (後述)。

(2) STOP-RUN 命令
プログラムを終了させる。呼び出された手続きに STOP-RUN 命令が含まれる場合に問題が生じる。

(3) GOTO 命令
COBOL では、 GOTO 命令の行き先として他の手続き内にある段落 (ラベル) を記述できる。行き先が同一手続き内であれば [CF 94] の方法で対処できるが、他の手続き内である場合は問題が生じる。

これらの命令に対して以下の節でそれぞれ検討する。ただし、簡単のため、制御依存関係を中心に議論を進め、データ依存関係の説明は必要最小限にとどめる。また、スライスの対象は手続き部 (PROCEDURE DIVISION) だけとし、環境部やデータ部は考えない。

3　PERFORM を含む場合

複数の手続きからなる COBOL プログラムのスライス計算方式を述べる。

3.1　プログラム依存グラフの作成

プログラム依存グラフは、 CFG から制御依存関係とデータ依存関係を抽出することにより作成する。まず、複数の手続きからなる CFG を次のように定義する。

定義 2 制御フローグラフは有向グラフ

$$CFG = < N_{CFG}, E_{CFG}, N_0 >$$

である。 N_{CFG} はノードの集合で COBOL の文や句に対応するノード、手続きの開始を表す Entry ノード、手

[1] 言語仕様は COBOL85(以下単に COBOL と呼ぶ) に従う。

続きの終端を表す Exit ノード、プログラムの終端を表す Stop ノードからなる。 E_{CFG} はノード間の制御の遷移を表すアークの集合、 N_0 は各手続きの Entry ノードの集合である。

COBOL の文はいくつかの句から構成されるが、基本的にはこれらの句をそれぞれ CFG のノードに対応させる。句は制御の分岐や合流、データ項目の設定を表すものが多く、独立のノードとした方が CFG の作成やその後の解析がしやすいためである。 Stop、 Exit、 Entry の各ノードは COBOL の文には対応しない付加的なノードである。 Stop ノードはプログラムに 1 つだけ付加し、 STOP-RUN 命令から Stop ノードにアークを張る。また、プログラムが最後の命令で終了する場合は、その命令から Stop ノードにアークを張る。 Exit ノードは各手続きごとに付加し、各手続きの末尾ノードから Exit ノードにアークを張る。 Entry ノードも各手続きごとに付加し、各手続きの Entry ノードから先頭ノードにアークを張る。図 1に例を示す。ただし、簡単のため、図 1では主手続きの Exit ノードは省略した[2]。

制御依存関係は、逆支配という概念を用いて次のように定義される [FOW 87]。

定義 3 制御フローグラフ G において、ノード V から Stop ノードに至るすべての経路にノード W が含まれるならば、 G 中で W は V を逆支配する。

定義 4 制御フローグラフ G において、以下の条件を満たすときノード Y はノード X に制御依存するという。

(1) X から Y へ至る経路上のどのノード Z(X、 Y を除く) も Y に逆支配される。

(2) X は Y に逆支配されない。

直観的には、分岐命令 X によって、命令 Y がある場合には実行され、他の場合には実行されないとき、 Y は X に制御依存するという。

データ依存関係は CFG に Kildall の反復解法を適用して求める [ASU 86]。ただし、 Entry ノードから Exit ノードへの CFG アークは計算対象から除く。また、呼出側の PERFORM ノードおよび呼ばれる手続きの Entry ノード、 Exit ノードに、その手続き内で使用されるデータ項目を集めて自己代入文を作る (図 1の ($A \rightarrow A$), ($B \rightarrow B$))。これにより、 PERFORM ノードと呼び出される手続きとの関係づけを行なう (3.2節参照)。

制御依存関係、データ依存関係は、 CFG ノード間にラベルつきアークを張ることで表す。これらのアークをそれぞれ制御依存アーク、データ依存アークと呼ぶ。この他に、手続きの呼出関係を表すために、手続きの Entry ノードから PERFORM ノードに CALL アークを設定する。 PDG は、 CFG のノードとこれらのアークから構成される。

定義 5 プログラム依存グラフは有向グラフ

$$PDG = < N_{PDG}, E_{PDG}, N_0 >$$

である。ただし、 $N_{PDG} = N_{CFG}$、 E_{PDG} は制御依存アーク、データ依存アーク、 CALL アークの集合である。

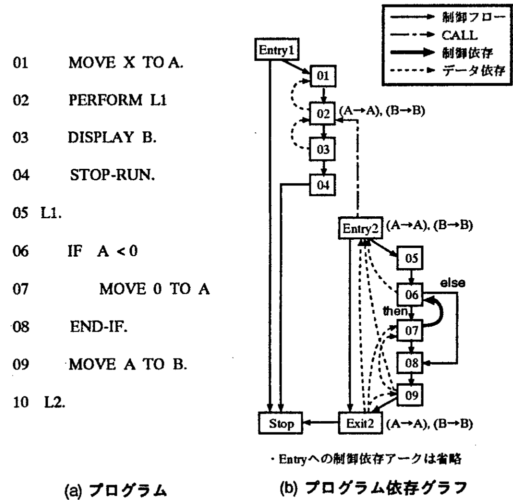


図 1: プログラム依存グラフ

3.2 スライス計算

スライスを求めるには、注目ノード (スライス起点) から到達可能なノードを集めればよい (Entry と Exit ノードは除く)。図 2にそのアルゴリズムを示す。手続き内のデータ項目に関するデータ依存アークが PERFORM ノードに集約されており、ステップ 6 で手続き側の Exit ノードに連結されるので、手続きをまたがってデータ依存関係をたどることができるようになる。

ただし、このアルゴリズムでは、同一手続きの呼出命令が複数個ある場合に、余分なスライスが得られる。また、 PERFORM ノードにデータ依存アークが集約されるため、注目したデータ項目と関係ないデータ項目に関するスライスが得られることもある。これに対処するには、 [HRB 90] のように手続き要約情報を用いるなど種々の方法が考えられるが、制御依存関係を中心としている本稿では扱わない。

4 STOP-RUN を含む場合

4.1 問題点

3.1節で述べた方式では、呼び出される手続き内に STOP-RUN 命令があると、制御依存関係が正しく求められな

[2] 図 1では、ノード 04 が STOP-RUN 命令なので、これを主手続きの終端としたが、一般には COBOL の主手続きの終端を自動的に求めるのは困難であり、プログラム全体の終端を主手続きの終端とするのが妥当である。

1. s= スライス起点
2. L={s}
3. **while** L $\neq$ {}
4. p = L の 1 つの要素
L = L − {p}
5. **for** (p, q) $\in E_{PDG}$
if q がマークづけされていない
q にマークづけし、 q を L に加える
6. **if** p は呼出命令か?
p が呼ぶ手続きの Exit ノードを L に加える

図 2: スライス計算

い。たとえば、図3は、図1の行 07 を「STOP-RUN」に、行 03 を「DISPLAY X」に変更したものである。図3では、行 06 の条件が真となって STOP-RUN が実行された場合、行 03 〜 04 は実行されない。したがって、行 03 〜 04 は行 06 に制御依存するが、 3.1節の方式ではこの関係を求めることはできない。

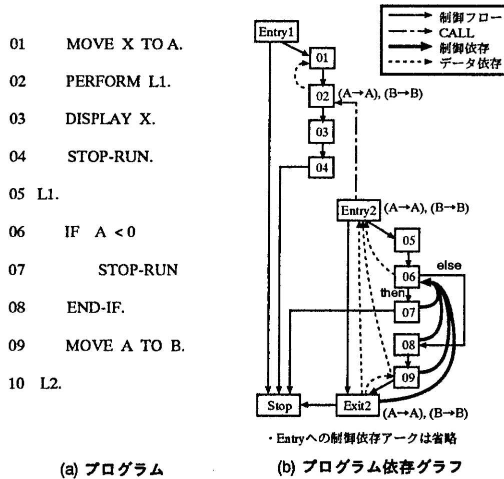


図 3: STOP-RUN を含む例

4.2 プログラム依存グラフの作成

CFG を拡張することにより、上記の問題に対処する。STOP-RUN を含む手続きを呼ぶ PERFORM ノードから Stop ノードへアークを張る。

定義 6 手続き呼出命令 C_i が呼び出す手続きを P_i、また P_i に含まれる手続き呼出命令の集合を $C(P_i)$ と表す。C_i の**呼出手続き集合** $F(C_i)$ を次のように定める。

$$F(C_i) = \{P_i\} \cup \bigcup_{C_j \in C(P_i)} F(C_j)$$

COBOL には再帰呼び出しは許されないため、呼出関係は木になる。定義 6の呼出手続き集合は、呼出木における P_i のすべての子孫手続きに相当する。

定義 7 手続き呼出命令 C_i の手続き呼出集合 $F(C_i)$ に属する 1 つ以上の手続きが STOP-RUN 命令を含むとき、C_i を**終了しうる呼出命令**と呼ぶ。

終了しうる呼出命令を求めるには、呼出木の末端手続きからボトムアップに STOP-RUN 命令の存在を検査し、どれかの下位手続きで STOP-RUN 命令が存在した場合、それより上位のすべての手続きを終了しうる呼出命令にする。

定義 8 $CFG =< N_{CFG}, E_{CFG}, N_0 >$ の**拡張制御フローグラフ (拡張 CFG)** とは、

$$ACFG_1 =< N_{CFG}, E_{CFG} \cup E_{Stop}, N_0 >$$

である。ただし、 E_{Stop} はすべての終了しうる呼出命令ノードから Stop ノードへのアークの集合である。

$ACFG_1$ から 3.1節と同様にして PDG を求め、スライスを計算する。求めた PDG を図 4に示す。ノード 03 から Stop ノードに CFG アークが張られると、ノード 02 で分岐が生じるため、ノード 03 〜 04 からノード 02 に制御依存アークが設定される。この PDG に図 2のアルゴリズムを適用すると、たとえばノード 03 を起点とした場合、ノード 02、 06、 01 が得られる。

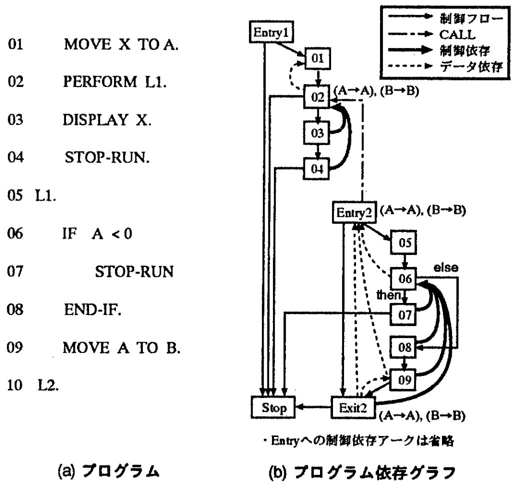


図 4: 拡張 PDG

5 手続きの重複と GOTO 命令

5.1 問題点

COBOL では、手続きの範囲が重複したり GOTO 命令による飛び出しがあるため問題が生じる。まず用語を整理する。

定義 9 呼出命令「PERFORM L1 THRU L2」において、ソースコード上で段落 L1 の先頭から段落 L2 の終りまでの範囲にある命令の集合をこの呼出命令で呼び出される手続きの**構文範囲**と呼ぶ。

定義 10 手続き m 中の命令 P_i から出ている制御フローアークが、m の構文範囲外のノードを終点としてもつとき、P_i を m からの**飛び出し命令**と呼ぶ。

COBOL では、呼出命令が存在する親手続きと呼ばれる子手続きの構文範囲が重複してもよい。さらに、飛び出し命令がある場合は、飛び出した部分が他の手続きと重複する。このような場合、PDG が正しく作成されないことがある。

たとえば、図 5(a) のプログラムでは、行 01 の PERFORM 命令が行 03 ～ 07 の手続きを呼んでいる。PERFORM 命令を含む手続きをメイン、呼ばれる手続きを L1 と呼ぶと、メインの構文範囲は行 00 ～ 12、L1 の構文範囲は行 03 ～ 07 であり、構文範囲が重複している。さらに、行 05 の GOTO 命令によって L1 の構文範囲を飛び出して行 08 に制御が移り、行 11 から行 06 に戻り、行 07 で L1 を終了してメインに制御が戻る。その後、メインでの命令として行 02 ～ 12 が実行される。

ここで、L1 の制御依存関係を考える。図 5(b) のようにメインと L1 が重複したままだと、ノード 08 からノード 04 への制御依存アークは存在しない。これは、メインでは行 07 の次に行 08 が実行されるので、行 08 は行 04 によらず実行されるためである。ところが、L1 だけで見ると、行 07 の次に行 08 は実行されないので、行 08 の実行は行 04 に制御依存しており、正しい制御依存関係は図 5(c) のようになる。

同様の理由で、データ依存関係も正しく設定されないことがある。

5.2 プログラム依存グラフの作成

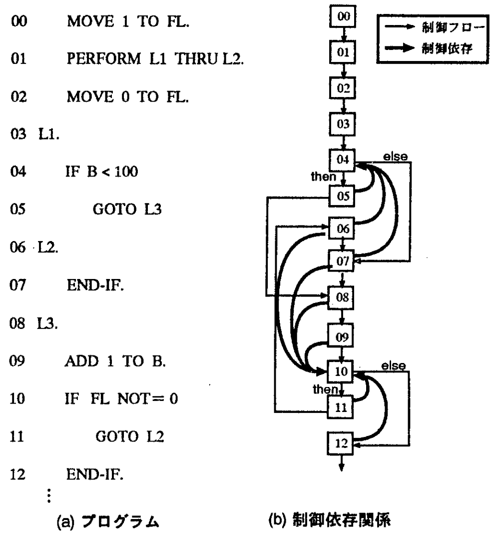
各手続きの実行範囲の CFG を別々に作成して重複部分を解消することにより、上記の問題に対処する。

定義 11 手続き m の構文範囲内のノードの集合を N、アークの集合を E、m 中の飛び出し命令の集合を G とするとき、m の**実行範囲グラフ** $< N(m), E(m), n(m) >$ を次のように定める。

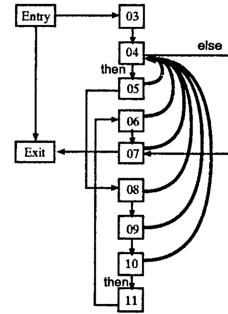
$$N(m) = N \cup \bigcup_{P \in G} R(P)$$

$$E(m) = E \cup \bigcup_{P \in G} Q(P)$$

ただし、$R(x), Q(x)$ は、それぞれもとの制御フローグラフ上でノード x から到達可能な m の構文範囲外のノード、アークの集合である。$n(m)$ は開始ノード (Entry ノード) である。



(a) プログラム　(b) 制御依存関係



(c) 手続きL1の実行範囲グラフ

図 5: 飛び出し命令を含む例

$n(m)$ から m の先頭命令のノードにアークが張られる。m は終端ノード Exit をもち、末尾ノードから Exit ノードにアークが張られる。また、$N(m)$、$E(m)$ の各要素には m 固有の識別子をつけるものとする。

この実行範囲グラフを用いて制御フローグラフを作成する。

定義 12 プログラム P に含まれる手続きの集合を M とする。P の**拡張制御フローグラフ** $ACFG_2$ は、

$$ACFG_2 = < \bigcup_{m \in M} N(m), \bigcup_{m \in M} E(m), \bigcup_{m \in M} n(m) >$$

である。

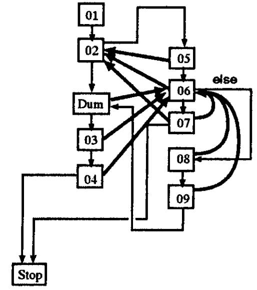
$ACFG_2$ では、手続きの実行範囲の重複はなくなっているので、上述の問題な生じない。たとえば、図 5(c) は手続き L1 の $ACFG_2$ である。ノード 12 はプログラム終了命令に至らなかったノードとして実行範囲には含めていない (後述)。上位手続きとの重複により生じてい

たノード07からノード08への制御フローアークがなくなっているため、ノード08からノード04への制御依存アークが正しく設定されている。スライスは図2と同じアルゴリズムで求める。

しかし、静的解析により実行範囲を求めると現実的でない結果が得られることがある。たとえば、図5のように手続きL1内に戻るGOTO命令がIF文で条件づけされている場合、手続きL1内に戻らず行12以降に進む経路も存在する。図5ではフラグFLで制御しているためそのように実行されることはないが、静的解析では判断できない。このため、実行範囲が本来の範囲を越えて巨大になりすぎる可能性がある。これに対処するには、実行範囲をユーザが指定するか、近似的な実行範囲を抽出するヒューリスティックを与えることが考えられる。後者の例として、他の手続きの末尾ノードを越える経路を切捨てる方法があるが、安全ではない。

```
01      MOVE X TO A.
02      PERFORM L1.
03      DISPLAY X.
04      STOP-RUN.
05 L1.
06      IF  A < 0
07          STOP-RUN
08      END-IF.
09      MOVE A TO B.
10 L2.
```

(a) プログラム



・Entry, Exitノードは省略

(b) 制御フローグラフ

図 6: 展開制御フローグラフ

6 考察

計算量は、STOP-RUN、飛び出し命令どちらの場合も、グラフのノード数の線形オーダですむ。ただし、前述のように手続きの実行範囲が巨大になると現実的でないので、回避策が必要である。

次にスライスの正当性について検討する。スライスの正当性は、元のプログラムとスライスの実行軌跡(各命令ステップにおける変数とその値の対の集合)を比較することにより示せるが[CF 94]、ここではやや直観的に考察する。まず、複数の手続き呼出がある場合、呼び出された手続きの実行軌跡を呼出ごとに区別する必要があるため、呼出ごとに手続きのCFGを複製して呼出箇所で展開し、意味的に等価なCFGを作る。図3のプログラムに対する展開CFGを図6に示す。呼出命令02は常に真となる条件文に置換する。Dumノードは合流用で、02からDumへのアークは02が制御依存関係によってスライスに含まれるようにするためのアークであり、データ依存関係の計算の際には除外する。

この展開CFGから得られるスライス[3]と本方式によるスライスを比較することにより、本方式のスライスの正当性を議論する。詳細を述べる余裕はないが、STOP-RUNを含む場合の制御依存関係については、展開CFGによるPDGでは、親手続きのノードから展開された子手続きのノードに制御依存アークが直接張られるのに対して、本方式では、呼出命令ノード、子手続きのEntryノード、Exitノードを中継点として、制御依存関係がつながるので、結果として同一のスライスが得られることが示せる。データ依存関係については、3.2節で述べたように安全ではあるが余分なスライスが得られる。

飛び出し命令の場合は手続きごとにCFGを複製していることになるので、展開CFGによるスライスの正当性に帰着する。

[3] このスライス自体の正当性の証明が必要であるが、ここでは考えない。

7 まとめ

COBOLプログラムのスライスを求める際の問題点を整理し、手続き内にSTOP-RUN命令を含む場合、飛び出し命令を含む場合のスライスの計算方法を述べた。

今後の課題として、GOTOの行き先を変更するALTER命令、別プログラムを呼び出すCALL命令、階層的なデータ構造への対処がある。また、精度の高いスライスを求めるのは計算コストが高く、プログラム理解支援という観点からは、完全なスライスを求めるのがよいとは限らない。他の解析手法や効果的なGUIと併せて、有用な解析情報を提示するアプローチが重要である。

参考文献

[ASU 86] Aho,A.V.,Sethi,R, and Ullman,J.D., *Compilers - Principles, Techniques, and Tools*, Addison-Wesley Publishers Limited (1986). 原田賢一訳、コンパイラ (I,II), サイエンス社 (1990).

[CF 94] Choi,J.D.,Ferrante,J., Static Slicing in the Presence of Goto Statements, *ACM Trans. Program. Lang. Syst. 16, 4*, (July 1994), 1097-1113.

[FOW 87] Ferrante,J., Ottenstein,K.,and Warren,J. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9, 3*, (July 1987), 319-349.

[HRB 90] Horwitz,S., Reps,T.,and Binkley,D., Interprocedural slicing using dependence graphs, *ACM Trans. Program. Lang. Syst. 12, 1*, (Jan.1990), 26-60.

[JZR 91] Jiang,J., Zhou,X.,and Robson,D.J., Program slicing for C - the problems in implementation. *Proceedings of Conference on Software Maintenance 1991*, (Oct. 1991), 182-190, IEEE Computer Society Press.

[Weiser 84] Weiser,M., Program slicing. *IEEE Trans. Softw. Eng. SE-10, 4*, (July 1984), 352-357.