一般支配関係の効率的な検査法

滝本宗宏[†] 武田正之[†]

コード最適化やプログラム解析では，プログラム点間の支配関係の情報を利用するものが多い．プログラムの開始点からあるプログラム点 p へ至るすべての実行パスがプログラム点 q を含むとき，q は p を支配するといい，その関係を支配関係と呼ぶ．この支配関係は，複数のプログラム点の集合を仮想的に1つのプログラム点と考えることによって，プログラム点の集合が，あるプログラム点を支配するという関係に拡張することができる．この支配関係を一般支配関係と呼ぶ．支配関係を利用してコード最適化やプログラム解析を行う場合，通常の支配関係の代わりに一般支配関係を用いることによって，それらの効果をさらに高めることが可能になる．本稿では，コード最適化やプログラム解析において，与えられた複数のプログラム点が，あるプログラム点を一般支配するかどうかを，必要に応じて効率良く検査する手法を提案する．通常の支配関係の検査は，支配木をたどるだけで実現できるのに対し，一般支配関係の検査は，支配木以外に制御フローグラフの辺をたどる必要があり，計算コストが大きいことが知られている．本稿では，制御フローグラフから上昇辺を除いたグラフを考え，そのグラフの深さを利用した手法を導入する．この値を調べることによって，支配木の辺だけでなく制御フローグラフの辺もたどる必要があるかどうかを予測することができ，一般支配関係を効率良く検査することが可能となる．本手法の効率は，SSA 形式上での利用可能式の発見という問題をとりあげて，データフロー解析による手法との比較によって示す．

Efficient Checking of Generalized Dominance Relation

Munehiro Takimoto[†] and Masayuki Takeda[†]

To realize practical code optimizations and sophisticated static program analysis, the dominance relation plays an important role because each dominator lies on every execution paths from the start point to its dominated points. The concept of dominator can be extended to a set of program points which virtually dominate a program point. Such dominance relation is called generalized dominance relation. We propose here an approach to efficiently check the generalized dominance relation for each target problem. Traditional dominance relation can be checked by only traversing a dominator tree, while generalized dominance relation may require traversing edges of control flow graph, so it is very costly. Our technique is intended to suppresses unnecessary traversal of the dominator trees based on the depth of control flow graph without up-edges. Since reachability to relevant points of dominance relation is expected by the depth of the points, relevant points unreachable along control flow graph edges can be ignored, and it is enough to traverse dominator tree edges. We show the efficiency of our technique by comparing experimental results of applying it to the available expression problem on SSA form with ones of dataflow analysis approach.

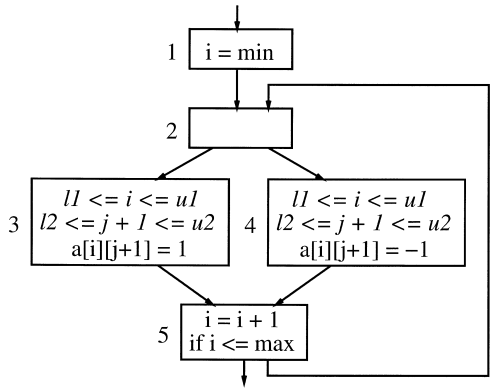
1. はじめに

コンパイラにおけるコード最適化やプログラムの静的解析の中には，大域的情報として，支配の概念を必要とするものがある．プログラムに対応する**制御フローグラフ**（control flow graph，以降 CFG と呼ぶ）において，CFG の開始節からある節 w へ至るすべての実行パスが節 v を含むとき，v は w を**支配する**（dominate）といい，v を w の**支配節**（dominator）[1),3),14)]という．また，このとき v と w は，**支配関係**（dominance relation）[1),3),14)]にあるという．

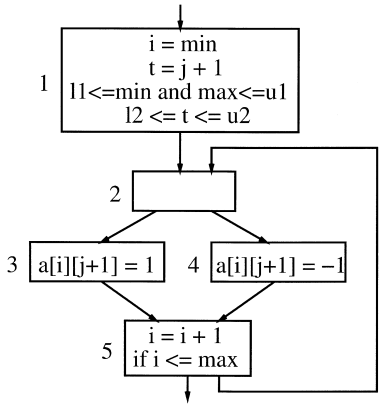
プログラム変形の条件として支配関係を用いるとき，従来，支配節は単一の CFG 節に限られることが多く，実際は可能な変形であっても，そのために変形が制限される場合があった．

例：ループ不変式（loop-invariant expression）や**配列境界検査**（array bound check）[1),3),10)]は，これらを含むプログラム点が，ループのすべての出口を支配する場合に，ループのプリヘッダ（preheader）へ移動することができる．**図 1** (a) に示したプログラムにおいて，ループの出口に到達するどの実行パスにおい

† 東京理科大学理工学部情報科学科
Department of Information Sciences, Faculty of Science and Technology, Tokyo University of Science



(a) 元のプログラム（original program）



(b) 変形後のプログラム（transformed program）

図 1 ループ不変式の移動と配列境界検査
Fig. 1 Propagation of loop invariants and array bound checks.

ても，添字の範囲検査を行うための同じパターンの条件式（斜体で示した式）が存在するので，それらの式をループのプリヘッダへ移動し，図 1 (b) に示すように変形することが可能である．しかし，それらの条件式を含む節 3，4 は，ループの出口である節 5 を支配してはいないので，支配関係の制約から，通常は移動が不可能であるとして扱われる場合が多い．∎

一方，単一の CFG 節に限定していた支配節の概念は，開始節から節 w への実行パス上に存在する節の集合 V を仮想的な 1 つの節と見なして，V が w を支配するという概念に拡張することができる．この複数の節によって構成される支配節は，**多重支配節**（multiple-vertex dominator）と呼ばれる．支配節が単一の節である場合を**単一支配節**（single-vertex dominator）と呼び，単一支配節と多重支配節を支配節として扱う場合には，**一般支配節**（generalized dominator）と呼ぶ．一般支配節の概念を用いると，単一支配節を基にした解析を容易に拡張することができる．

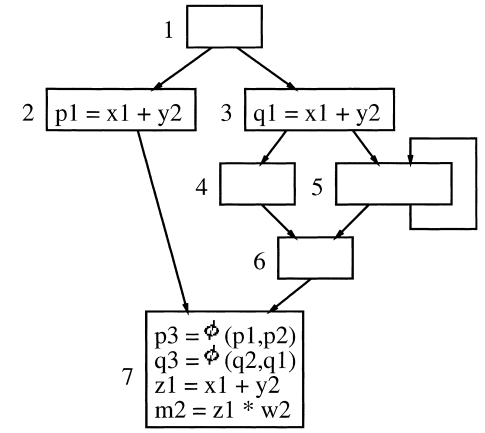
例：図 1 (a) において，節 3，4 は，ループの出口である節 5 に対して多重支配節を構成し，それらはともに同じパターンの式を含んでいるので，斜体の式をループのプリヘッダへ移動することができる．∎

本稿では，ある CFG 節が一般支配されているかどうかを効率的に検査するための手法を提案する．節集合 V と節 wが与えられるとき，V に含まれる節によって w が一般支配されるかどうか判定することを，w の V に対する**一般支配検査**と呼ぶ．また，V を w の**支配節候補**と呼ぶ．従来研究されてきた一般支配節の計算法は，**支配木**（dominator tree）を拡張した**支配 DAG**（dominator DAG，以降 DDAG と呼ぶ）[2),9),11)]に基づくものであり，DDAG の構成にコストがかかるという問題があった．また，DDAG に基づく一般支配検査は，効率的な方法が存在しないという問題があった．

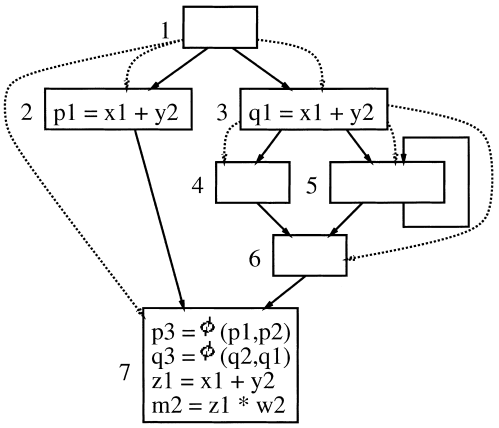
本稿で提案する手法は，CFG 辺よりも支配木の辺を優先してたどる疎な解析（sparse analysis）によって，一般支配検査を実現する．すなわち，ある節 w の V に対する一般支配検査を行う際に，まず，w の単一支配節をたどりながら，それらが V に含まれているかどうか判定する．その結果，単一支配節が V に含まれなかった場合に限って，多重支配節が V に含まれるかどうか判定する．多重支配節の判定は，w の先行節をたどることによって行うが，w の単一支配節から w までのパス上に，もし V に含まれる節がないことが分かれば，先行節をたどらずに V が w の多重支配節を含まないことが分かる．本手法では，各 CFG 節に対して開始節からの**深さ**（depth）を割り当て，支配節候補に含まれる節の深さから，多重支配節が支配節候補に含まれている可能性があるかどうかを予測する．

例：静的単一代入（static single assignment，以降 SSA と呼ぶ）形式[7),15)]のプログラムにおいて共通部分式の除去を行うためには，同じパターンの式を含むプログラム点によって一般支配されているプログラム点の式を除去すればよい．**図 2** (a) で示した SSA 形式の CFG に支配木の辺を付加したものを，図 2 (b) に示す．CFG 節の辺は実線で表現し，支配木の辺は点線で表現する（以降の図でも同様の表現を用いる）．また，CFG 節の開始節からの深さを，開始節にあたる節 1 が 0，節 2，3 が 1，節 4，5 が 2，節 6 が 3，節 7 が 4 であるとする．

節 7 の式 `x1 + y2` が共通部分式であるかどうかを



(a) 元の SSA 形式（original SSA form）



(b) 変形後の SSA 形式（transformed SSA form）

図 2 SSA 形式における利用可能式

Fig. 2 Propagation of loop invariants and array bound checks.

調べることを考えると，`x1 + y2` を含む節 $\{2,3\}$ を支配節候補として，節 7 の一般支配検査を行えばよい．

まず，節 7 から，支配木の辺をたどって節 7 の単一支配節 1 が支配節候補に含まれないことが分かるので，次に節 7 の多重支配節が支配節候補に含まれるかどうか調べる．そのためには，節 7 の先行節をたどる必要がある．節 7 の単一支配節 1 の深さは 0 であり，それと節 7 の深さ 4 との間には，深さ 1 において，支配節候補に含まれる節が存在している．このことから，節 7 はそれらの節によって多重支配されている可能性があり，そのことを調べるためにすべての先行節，すなわち節 2，6 をたどる．まず，先行節 2 が支配節候補に含まれるので，次に先行節 6 をたどる．節 6 は支配節候補に含まれないので，さらにその単一支配節 3 をたどる．節 3 は支配節候補に含まれるので，結果として，節 4，5 をたどらずに，節集合 $\{2,3\}$ が節 7 の多重支配節を構成することが分かる．節 7 は，一般支配されていることから，節 7 の式は共通部分式であることが分かる．∎

本手法で用いる一般支配節の概念は，Gupta[11]が定義したものとは異なる場合があるので，特に**セミ一般支配節**（semi-generalized dominator）と呼ぶ．本稿では，セミ一般支配節が，一般支配節を部分集合として含むことを示し，セミ一般支配節の存在が一般支配節の存在を保証することを示す．セミ一般支配節の性質から，節 w の支配節候補 V に対する一般支配検査は，w のセミ一般支配節が V に含まれるかどうか判定することと等しいことを示すことができる．

本手法では，次の手順によって，w のセミ一般支配節が V に含まれているかどうかを判定する．

(1) CFG における，各節の深さを計算する．
(2) CFG から一般支配検査に不要な辺を取り除く．
(3) V に含まれる節の深さを記録した表（以降，**深さ表**と呼ぶ）を作成する．
(4) w から支配木の辺を優先してたどり，深さ表によって支配節候補が存在する可能性があるかどうかを調べ，その可能性がある場合にだけ，先行節をたどるという方法で，V に含まれる節に到達できるかどうかを判定する．

(1) と (2) は，1 つのプログラムに対して，1 回計算するだけで済むので，問題ごとに必要となる処理は，(3) と (4) だけである．

本稿における以降の構成を次に示す．2 章で，本稿で用いる記法や用語をまとめる．3 章で，一般支配検査に重要なセミ一般支配節の定義を述べ，一般支配節との関係を述べる．また，セミ一般支配節が支配節候補に含まれるかどうか判定する基本的なアルゴリズムを示す．4 章で，効率的な一般支配検査のために重要な，深さの概念を示し，取り除くことのできる不要な CFG 辺を分類する．5 章で，深さ表の実現方法と，深さ表を用いた効率的な一般支配検査アルゴリズムを示す．6 章で，SSA 形式のプログラムにおける利用可能式の計算[4]を例に，本手法と従来法であるデータフロー解析による方法とについて，実験結果に基づく計算コストの比較を行う．そして，7 章で関連研究を述べたのち，最後に結論を述べる．

2. 用語と記法

本手法の適用に際しては，原始プログラムに対応する CFG がすでに作成されているものとする．CFG は，途中に分岐を持たない連続した文からなる**基本ブロック**（basic block）を節とする節集合 N，基本ブ

ロック間の制御の流れを辺とする辺集合 $E \subset N \times N$，特別な節である開始節 **s** と終了節 **e** からなる 4 つ組 $(N, E, \mathbf{s}, \mathbf{e})$ である．CFG 節 n について，その先行節集合 $\{n' | (n', n) \in E\}$ を $PRED(n)$ で表し，後続節集合 $\{n' | (n, n') \in E\}$ を $SUCC(n)$ で表す．それぞれの先行節と後続節は，$pred(n)_i$ と $succ(n)_i$ のように添字を付けて表記する．また，CFG 節は，必ず開始節から終了節までの実行パス上に存在するものとする．

支配関係は，推移律（transitive）が成り立つので，木構造によって表現することができる．この木構造を支配木と呼ぶ．支配木において，節 x が節 y の直接の親であるとき，x は y を**即支配**（immediately dominate）するといい，x は y の**即支配節**（immediate dominator）であるという．本稿では，解析対象の CFG について支配木がすでに作成されているものとする．

3. セミ一般支配節

ある節 w の支配節候補 V に対する一般支配検査は，w の一般支配節が V に含まれるかどうかを判定することと同じである．本手法では，w の一般支配節を含むセミ一般支配節 V' が V に含まれるかどうかを判定することによって，一般支配検査を実現する．

本章では，一般支配節の定義を示し，セミ一般支配節の定義を与える．次に，一般支配節とセミ一般支配節との関係を述べ，セミ一般支配節が支配節候補に含まれるかどうかを判定することによって，一般支配検査が実現できることを示す．最後に一般支配検査の基本的なアルゴリズムを示す．

Gupta [11] による一般支配節の定義は次のとおりである．

定義 1（一般支配節）：次の 2 つの条件が満たされる場合に，節集合 V は節 w を一般支配する．

(1) CFG の開始節から w へ到達するどの実行パスも，V に含まれる節を少なくとも 1 つは通る．
(2) 各節 $v \in V$ について，v を含み V 内の他の節を含まないで w へ到達する実行パスが，少なくとも 1 つ存在する．

定義 1 を満たす節集合 V は，要素数を $|V|$ として，$|V| = 1$ の場合，単一支配節を表し，$|V| > 1$ の場合，多重支配節を表す．

本手法では，定義 1 の 2 つの条件のうち，(1) の条件だけを満たす節集合 V' を扱う．この節集合 V' を特にセミ一般支配節と呼び，V' は w をセミ一般支配するという．

定義 2（セミ一般支配節）：節集合 V' について，CFG の開始節から節 w へ到達するどの実行パスも，V' に含まれる節を少なくとも 1 つ通るとき，V' は w をセミ一般支配する．

定義 2 の条件を満たす節集合 V' は，$|V'| = 1$ の場合，一般支配節と同様に，単一支配節を表す．$|V'| > 1$ の場合は，定義から，多重支配節を含む節集合を表す．この V' を特に，**セミ多重支配節**（semi-multiple vertex dominator）と呼ぶ．

ある節 w に対して V' がセミ一般支配節であれば，定義 2 から V' は，節 w の一般支配節を少なくとも 1 つ含んでいることが分かる．すなわち，節 w の一般支配節が支配節候補 V に含まれるかどうかは，w のセミ一般支配節集合を $SGD(w)$ として，次の $sgd \in SGD(w)$ の条件を検査すればよい．

$$sgd \subseteq V \tag{1}$$

したがって，節 w の一般支配検査は，w のセミ一般支配節が支配節候補に含まれるかどうかを判定すれば十分である．一般支配節は，セミ一般支配節から定義 1 の (2) を満たす節だけを残して，そのほかの節は取り除くという方法で計算されるので，セミ一般支配節の方が容易に計算することができる．

3.1 セミ一般支配節の計算

定義 1 から，節集合 V が節 w を一般支配するとき，同時に V は，w のすべての先行節を一般支配する．これは，一般支配節が各先行節の一般支配節を含んでいることを意味し，w の各先行節に対する一般支配節の和が V を含むことを意味する [11]．したがって，w に対する一般支配節集合を $GD(w)$ とすると，$gd \in GD(w)$ と $gd_i \in GD(pred(w)_i)$ は，次の関係式を満たす．

$$gd \subseteq \bigcup_{i=1}^{|PRED(w)|} gd_i \tag{2}$$

節 w のセミ一般支配節集合を $SGD(w)$ で表現することにすると，$sgd \in SGD(w)$，$gd \in GD(w)$ に対して，$gd \subseteq sgd$ となる gd が少なくとも 1 つ存在する．したがって，$sgd_i \in SGD(pred(w)_i)$ として，次の関係式を満たす $gd \in GD(w)$ が存在する．

$$gd \subseteq \bigcup_{i=1}^{|PRED(w)|} sgd_i \tag{3}$$

関係式 (3) の右辺もセミ一般支配節を表していることから，関係式 (3) で得られる集合の要素は，再帰的に関係式 (3) の右辺で置き換えることができる．すな

わち，節 wの一般支配節が節集合 V に含まれるかどうかは，次の単純なアルゴリズムで決定することができる．

アルゴリズム 1（基本検査）：w から V に含まれる節または開始節に到達するまで，すべての先行節への訪問を繰り返す．その間，開始節の入口へ到達することがあれば，w の一般支配節が V に含まれることはない．いいかえれば，どの先行節への訪問においても，V に含まれる節に到達するのであれば，w の一般支配節は V に含まれる．∎

支配節候補 V が，w の一般支配節と一致する場合に，アルゴリズム 1によって，どの先行節への訪問でも V 中の節に到達することは，次のように証明できる．

定理 1（一般支配節への到達）：アルゴリズム 1 は，支配節候補 $V = gd \in GD(w)$ に到達する．

証明：
アルゴリズム 1 における先行節への訪問を繰り返すことによって，開始節の入口へ到達したと仮定すると，開始節から節 wへの実行パスに，V 内の節を含まないものが存在することになる．これは，$V = gd \in GD(w)$ に反する．∎

次の章では，さらに効率的に一般支配検査を行うための工夫を示す．

4. 効率的な検査法

アルゴリズム 1 は，即支配節あるいは即支配節の一般支配節が支配節候補に含まれない場合にだけ，先行節をたどるように改良することができる．単一支配節が支配節候補に含まれる場合には，従来の単一支配関係に基づく方法と同じコストで一般支配検査を行うことができる．この検査法を**アルゴリズム 1'** と呼ぶことにする．

アルゴリズム 1'では，先行節をたどる際に CFG 辺を利用し，即支配節をたどる際に支配木を利用することができる．本稿では，説明を簡単にするために，CFG の各節に相当する節を，CFG 辺と支配木の辺に相当する辺で結んだグラフ表現を用いる．以降，このグラフを**探索グラフ**（traversal graph）と呼ぶ．探索グラフにおいても，CFG 辺と支配木の辺は区別して扱う．

例：図 2 (b) は，図 2 (a) の探索グラフを示している．∎

本章では，先行節への不要な訪問を避けることによって，アルゴリズム 1' の効率を改良する方法を述べる．その方法は，次の 2 つである．

(1) 各節に付加した深さによって，先行節をたどる必要があるかどうかを判定する．
(2) 探索グラフから一般支配検査に不要な CFG 辺を取り除く．

まず，深さの定義と，深さに基づいて先行節をたどる必要性を判定する方法について述べ，次に，一般支配検査に不要な CFG 辺の定義と除去について述べる．

4.1 CFG の深さ

一般に，CFG は閉路を持つグラフ構造を形成するので，開始節から深さ優先順序の逆順に付けた番号を基に，辺を分類することができる．特に，ある節の子孫から祖先に向かう辺は，**上昇辺**（up-edge）と呼ばれる．上昇辺を取り除くと，グラフは DAG 構造を持つことが知られている．

CFG から上昇辺を取り除いた DAG を基に，その中の節 vの深さ $level(v)$ を次のように定義する．

定義 3（深さ）：
(1) v が開始節なら，$level(v) = 0$ である．
(2) そうでなければ，$PRED(v)$ の最大の深さを $level(v_{max})$ として，
$level(v) = level(v_{max}) + 1$ である．∎

定義から，DAG の辺 $(v1, v2)$ について，$level(v1) < level(v2)$ が成り立つので，次の補助定理が得られる．

補助定理 1（DAG 上の深さ関係）：DAG 上の節 vから v' へのパスを P とするとき，P 上の節 w（$\neq v$ かつ $\neq v'$）に関して次の関係が成り立つ，

$$level(v) < level(w) < level(v')$$

証明：辺間の深さの大小関係は，推移律が成り立つので明らか．∎

補助定理 1 を利用すると，ある節 w とその即支配節 id の間の実行パス上に存在する節 v に関して，次のことを示すことができる．

定理 2（深さと即支配節）：$level(id) < level(up) \leq level(w)$ を満たす，上昇辺 (x, up) が指す節 upが存在しないとき，即支配節 idから w への実行パス上にある任意の節 v は，$level(id) < level(v) < level(w)$ を満たす．

証明：$level(id) < level(up) < level(w)$ を満たす up が存在しなければ，補助定理 1 から id と w の間の DAG 上に up は存在しない．さらに $w \neq up$ であれば，id から w までの実行パスは，すべて DAG 上のパスである．したがって，補助定理 1 か

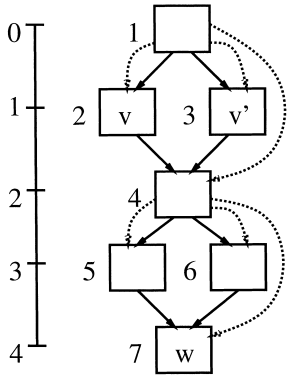


図 3 深さを利用した一般支配検査
Fig. 3 Checking based on depth.

ら，id から w までの実行パス上にある任意の節 v は，$level(id) < level(v) < level(w)$ を満たす．∎

定理 2 から，節 wとその即支配節 id の間の DAG 上に，上昇辺が指す節が存在せず，支配節候補 V に含まれるどの節 vも，$level(v) \leq level(id)$ あるいは $level(v) \geq level(w)$ であれば，w の一般支配節が V に含まれるかどうかは，id の一般支配節が V に含まれているかどうかで決まることが分かる．この性質を利用すると，セミ一般支配節が V に含まれているかどうかを判定する際に，たどる必要のない節への訪問を避けることができる．

例：図 3 は，上昇辺を持たない探索グラフを表している．探索グラフの左には，各節の深さを示してある．節 wに対する支配節候補として $\{v, v'\}$ が与えられたとき，その一般支配検査を考える．w の深さは 4 であり，w の即支配節 4 の深さは 2 である．一方，支配節候補の各節の深さは 1 であり，両者の深さの範囲に入っていないことから，一般支配検査は即支配節 4 の先行節をたどればよいことが分かる．結果として，節 5，6 をたどらずに，w が $\{v, v'\}$ によって一般支配されていることが分かる．∎

4.2 不要な辺の除去

本手法では，支配節候補の中にセミ一般支配節が含まれるかどうかを判定するために探索グラフを基にして検査を行うが，その際，不要な訪問を避けるために次の性質を持つ辺を取り除いておくことができる．

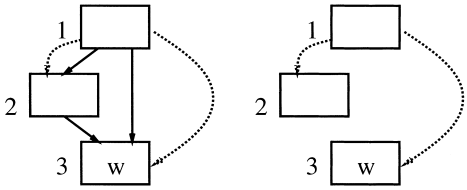
不要な辺の除去は，探索グラフ中の節 w についての単一支配関係から，次の 2 つの場合に分類することができる．

(1) w のある先行節 p が，w の即支配節である場合，w の各先行節 x からの CFG 辺 (x, w) をすべて取り除く．

(2) w が，上昇辺 (x, w) の行先であり，出元 x を支配する場合，その上昇辺を取り除く．

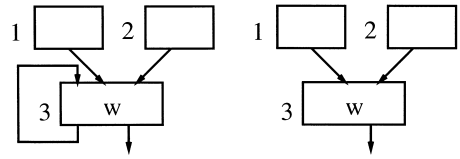
(1) の場合，p 以外のどの先行節によっても，w は一般支配されることはないので，w の支配節候補に対する一般支配検査には，支配木の辺 (p, w) を残しておくだけでよい．

例：図 4 (a) に示す探索グラフは，(1) の不要な辺の除去によって，図 4 (b) のように変形できる．∎



(a) 元の探索グラフ（original traversed graph）
(b) 変形後（transformed graph）

図 4 不要な辺の除去（1）
Fig. 4 Removing unnecessary edges (1).



(a) 元の探索グラフ（original traversed graph）
(b) 変形後（transformed graph）

図 5 不要な辺の除去（2）
Fig. 5 Removing unnecessary edges (2).

(2) の場合，w が上昇辺 (x, w) の出元の節 x を支配するならば，w を一般支配する節として x が含まれることがないので，上昇辺を取り除くことができる．この上昇辺は**戻辺**（back-edge）[1),3),14)]と呼ばれる．

例：図 5 (a) の上昇辺 $(3, 3)$ は，支配関係の反射律から節 3 が節 3 を支配するので，取り除くことができる．結果として，図 5 (b) のように変形できる．∎

図 2 (b) の探索グラフから不要な辺を取り除いた結果を**図 6** に示す．

不要な辺を除去した探索グラフは，一般支配節の解析に共通に利用できるので，不要な辺は 1 度除去すれば済む．

5. 一般支配検査アルゴリズム

本章では，一般支配検査を行う効率的なアルゴリズムの詳細を示す．まず，ある節とその即支配節の間に

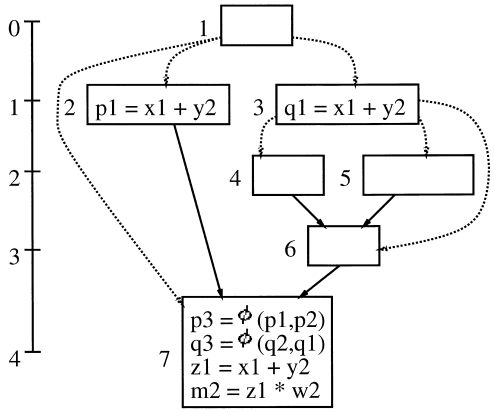


図 6 図 2 (b) における不要な辺の除去
Fig. 6 Removing unnecessary edges for Fig. 2 (b).

支配節候補に含まれる節が存在する可能性があるかどうかの判定のために深さ表を利用できることを述べ，深さ表の詳細を述べる．次に，アルゴリズム 1' を深さに基づいて効率化したアルゴリズムの詳細を仮想コードを用いて示す．

5.1 深 さ 表

定理 2 から，ある節とその即支配節の間の節が支配節候補に含まれるかどうかは，深さの関係によって予測することができる．特に，深さがそれぞれ $upper$ と $lower$（$upper < lower$）である節を考えたとき，$lower$ の深さを持つ節が上昇辺の先でなく，$upper < l < lower$ の深さ l を持つ節が，上昇辺が指す先でも，支配節候補の要素でもなければ，深さ $upper$ と $lower$ を持つ節の間には，支配節候補に含まれる節が存在しないことが分かる．すなわち，その場合には $upper$ と $lower$ の間にある節については，一般支配検査をする必要がないことが保証される．

深さ l を持つ節が支配節候補に含まれないことを調べる方法は，次の 2 種類が考えられる．

(1) 支配節候補に含まれる各節について，その深さが i であれば，ビット位置 i には 1，そうでなければ 0 を立てたビットベクトルを用意する．上述の判定には，ビットの位置が $upper$ 以下と $lower$ 以上の部分をマスクして，ビットベクタが 0 になるかどうかを調べる．

(2) 深さ $i = 0, 1, 2, \cdots$ について，i 以下の深さを持ち，支配節候補に含まれる節のうちで最大の深さを得ることができる表を用いる．添字を深さ i に対応付けた配列 $depthTable$ を用意し，配列の要素 $depthTable[i]$ には，支配節候補に含まれる節の深さの中で，i 以下で最大の深さを格納しておく．上述の判定は，$depthTable[lower - 1] \leq upper$ であるかどうかを調べることによって決まる．

(1) の方法は，実現が容易であるが，ビットベクタのサイズがワードサイズを超えた場合に，繰返し計算が必要になる可能性がある．一方，(2) の表を用いる方法は，定数時間で検査が可能である．この表を深さ表と呼ぶ．

例：節 2，3，7 を支配節候補とするとき，図 6 の深さ表は，$depthTable[0] = -1$，$depthTable[1] = 1$，$depthTable[2] = 1$，$depthTable[3] = 1$，$depthTable[4] = 4$ となる．$depthTable[0]$ の値 -1 は，添字以下の深さを持つどの節も支配節候補に含まれないことを表す． ∎

$depthTable$ の初期化は，次の手順で行うことができる．

(1) 開始節が支配節候補の要素であれば，$depthTable[0] = 0$，そうでなければ，$depthTable[0] = -1$ とする．

(2) 深さ $= 1, 2, \cdots$ について順に CFG 中の節を訪問する．このとき，深さ i に，支配節候補に含まれる節が 1 つでも存在すれば，$depthTable[i] = i$ とする．そうでなければ，$depthTable[i] = depthTable[i - 1]$ とする．

探索グラフに上昇辺が残っている場合は，その上昇辺が指す節も深さ表に記録する必要がある．上昇辺をたどる必要性は，深さ表では判断できないので，上昇辺をたどることを回避しないようにしなければならない．

5.2 アルゴリズム

深さに基づいて，アルゴリズム 1' を効率化した一般支配検査アルゴリズムを次に示す．

アルゴリズム 2（効率的な検査）：

Input: 開始節を s とする探索グラフ，
検査対象となる節 $checkedNode$ と
支配節候補節 $domCand$，
各節 n の深さ $level[n]$ と即支配節 $idom[n]$，
深さ表 $depthTable$

Output: 一般支配されるていれば $True$，
そうでなければ $False$

```
Function SGDom(checkedNode)
{
1:  search = False
2:  return Visit(checkedNode, 0)
}

Function Visit(node, upper)
```

```
{
3:  if (search && node ∈ domCand)
4:    return True
5:  if (node == s)
6:    return False
7:  if (visited[node])
8:    return True
9:  if (node != destination of up-edge
10:   && depthTable[level[node] − 1] <= upper)
11:     return False
12: search = True
13: visited[node] = True
14: if (Visit(idom[node], 0))
15:   visited[node] = False
16:   return True
17: if (node != destination of up-edge &&
18: depthTable[level[node] − 1] <= level[idom[node]])
19:   visited[node] = False
20:   return False
21: foreach p ∈ PRED(node)
22:   if (! Visit(p, level[idom[node]]))
23:     visited[node] = False
24:     return False
25: visited[node] = False
26: return True
}
```

アルゴリズム 2 では，上昇辺以外の CFG 辺を用いることによって各節の深さを計算してあり，不要な CFG 辺は探索グラフから取り除かれていることを前提とする．

アルゴリズム 2 は，一般支配検査を行いたい節 *checkedNode* を引数として関数 *SGDom* を呼び出すことから始まる．実際に，*CheckedNode* の一般支配検査を行うのは，関数 *Visit* であるが，*Visit* は，訪問節 *node* が支配節候補に含まれる場合に検査を終了してしまうので，*checkedNode* が支配節候補に含まれる場合には，支配節候補への到達を無視して検査を継続する必要がある．そこで，*SGDom* は，支配節候補へ到達したらただちに終了することを指示する変数 *search* を *False* に初期化することによって，*checkedNode* における検査の継続を保証する（3 行目）．*search* は，*Visit* の呼び出し後，*True* となる（12 行目）．

実際の一般支配検査を行う *Visit* は，次の 4 つのいずれかの場合に検査を終了して結果を返す．

(1) 支配節候補に含まれる節に到達したとき（3 行目）
(2) 開始節に到達したとき（5 行目）
(3) 検査の間に訪問した節に再び到達したとき（7 行目）
(4) 検査の上限 *upper* に達したとき（9，10 行目）

(1) の場合は，支配節候補に到達したことから，*True* を返す．逆に，(2) は，開始節に到達するまでに支配節候補に出会わないパスが存在することを意味するので，*False* を返す．

(3) の場合は，閉路によって訪問済みの節に達したことを意味する．各節は，開始節から終了節までの実行パス上にあることから，開始節から閉路上の各節への実行パス P が存在し，一般支配検査の結果は P に依存する．したがって，ここでは単に *True* を返す．

(4) は，これ以上訪問しても結果が *False* であると分かっている深さに達したことを意味する．先行節をたどるのは，即支配節を検査した結果が *False* であったときだけなので，即支配節の手前の深さ *upper* までに，支配節候補の要素が存在しなければ，*False* を返す．*upper* は，*Visit* の第 2 引数として与える．

さらに節をたどる部分は，次の 2 つである．

(1) 即支配節をたどる（14 行目）
(2) すべての先行節をたどる（21，22 行目）

(1) は，結果が *True* であった場合だけ，*True* を返す．結果が *False* であった場合は，現在の節 *node* が上昇辺が指す節でなく（17 行目），*node* と即支配節の間の節が支配節候補に含まれなければ（18 行目），*False* を返す．

(2) は，すべての先行節の検査結果が *True* だった場合にだけ *True* を返し，それ以外は *False* を返す．

例：図 2 (a) において節 7 の `x1 + y2` の共通部分式を発見する問題は，支配節候補を $\{2, 3, 7\}$ として，節 7 の一般支配検査を行えばよい．ここで，各式に対する共通部分式の発見に対して，1 つの支配節候補で済ませるために，支配節候補に検査対象の節を含めている．図 6 の探索グラフと深さ表 $depthTable[0] = -1$，$depthTable[1] = 1$，$depthTable[2] = 1$，$depthTable[3] = 1$，$depthTable[4] = 4$ に基づいて，一般支配検査は，次の手順となる．

(1) 支配木の辺 $(1, 7)$ をたどると，開始節に到達するので *False* を返す．
(2) $depthTable[level[7] - 1] \leq level[idom[7]]$ は成り立たないので，次に先行節をたどる．
(3) CFG 辺 $(2, 7)$ をたどると，支配節候補に含まれる節 2 に到達するので *True* を返す．
(4) CFG 辺 $(6, 7)$ をたどり，次に支配木の辺 $(3, 6)$ をたどると，支配節候補に含まれる節 3 に到達するので，それぞれ *True* を返す．

最終的に，節 7 のすべての先行節が *True* を返すので，節 7 は，支配節候補の部分集合によって一般支配

されることが分かる．

6. 評　　価

本手法の効果を示すために，SSA 形式上における共通部分式の発見を本手法と従来法でそれぞれ実現し，実行効率の比較を行った．実現には，**並列コンパイラ向け共通インフラストラクチャ**（a compiler infrastructure project，以降 COINS と呼ぶ）[6]から提供されている C コンパイラを用いた．COINS コンパイラは，C のソースコードを高水準中間表現から低水準中間表現（以降，LIR と呼ぶ）に変換し，LIR から目的コードを生成する．

評価結果を比較するために，次の 3 つの手法を実現した．いずれの手法も，LIR 上の最適化パッケージの 1 つである `coins.ssa` パッケージに組み込むことによって実現した．

(1) スロットワイズ法：**利用可能式**（available expression）のデータフロー解析を各式のパターンごとに計算するワークリスト法[8]

(2) ワードワイズ法：利用可能式のデータフロー解析において，各式のパターンをビットに対応付け，1 ワードで表現できる 32 パターンの式を 1 度に計算するワークリスト法[3),13)]

(3) 本手法：式のパターンの個数分，深さ表を用意して，式のすべての出現に対して一般支配検査を行う手法

これらの手法は，CFG 節（基本ブロック）内の共通部分式はすでに発見されているものとし，CFG 節間の共通部分式を発見することを目的とする．

評価に用いたプログラムは，SPEC CINT2000 ベンチマークの 6 つのプログラム（mcf，bzip2，gzip，parser，var，crafty）と SPEC CFP2000 ベンチマークの 2 つのプログラム（art，equake）である．実行効率の比較には，共通部分式発見のために各手法ごとにかかった CFG 節への訪問回数を用いた．

図 7 に，各手法の訪問節数をグラフにして，プログラムサイズの小さい順に示す．CFG の節を単位とする各プログラムのサイズは**図 9** に示すとおりである．スロットワイズ法（Slot）と比較して，ワードワイズ法（Word）と本手法（GD）の訪問節数が，顕著に少ないことが分かる．**図 8** に，ワードワイズ法と本手法の結果を拡大して示すと，本手法は，最も効率の良い解法として知られるワードワイズ法と比べても，gzip を除いて半分程度の訪問節数であることが分かる．gzip では，ワードワイズ法の方が本手法よりも訪

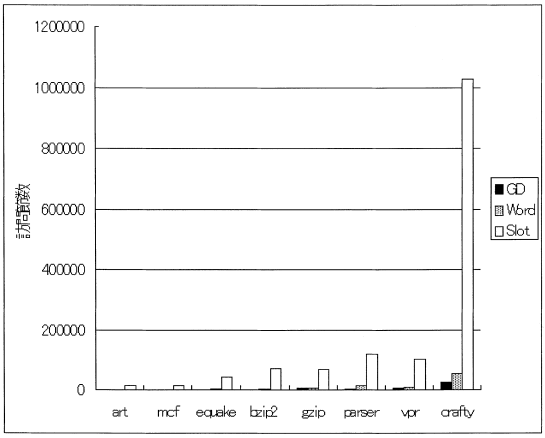


図 7 訪問節数の比較（1）
Fig. 7 Number of visited nodes (1).

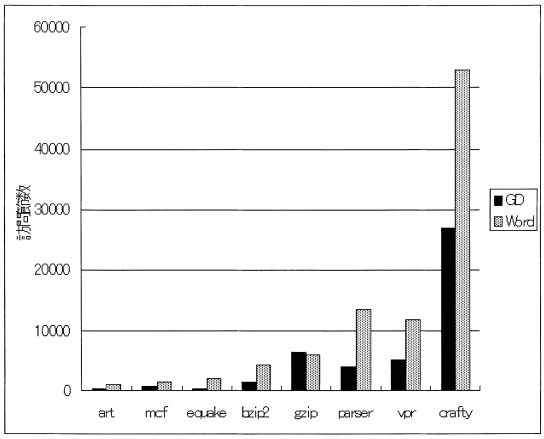


図 8 訪問節数の比較（2）
Fig. 8 Number of visited nodes (2).

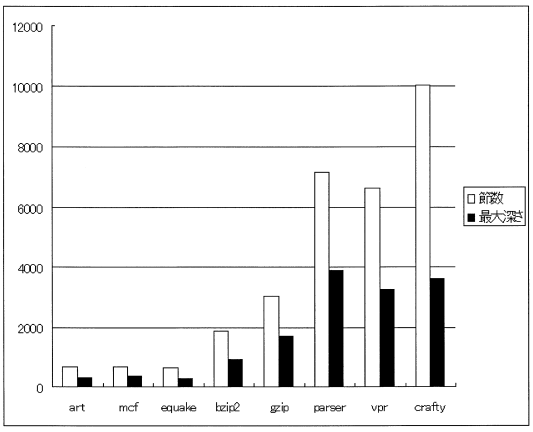


図 9 CFG 節数と最大深さ
Fig. 9 Size of CFG and depth.

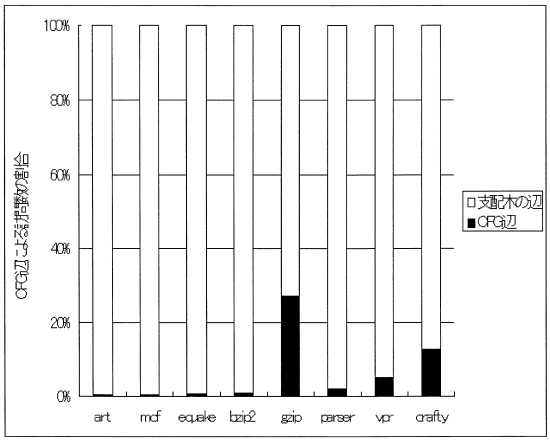


図 10 CFG 辺をたどった割合
Fig. 10 Ratio of visited CFG edges.

問節数が少なかったが，両者は僅差であることが示されている．

本手法とワードワイズ法とで，評価プログラムによって訪問節数の比率が異なる理由を調べるために，プログラムサイズと最大深さとの比較と，一般支配検査の際にたどった CFG 辺の割合を，それぞれ図 9 と**図 10** に示す．図 9 から，訪問節数の比率の違いにかかわらず，最大深さは，プログラムサイズの半分程度になることが分かる．一方，図 10 から，CFG 辺をたどる割合が最も大きいものが gzip であることから，本手法は，支配節候補に含まれるセミ多重支配節の存在を判定するために CFG 辺をたどることが多いと効率が下がることが分かる．この結果は，深さ表によって不要な CFG 辺をたどらないようにすることが，本手法の効率向上に重要な役割を果たしていることを意味している．

6.1 深さ表の効率的な初期化

配列を用いた深さ表の初期化には，深さサイズを d，式のパターン数を e として，$e*d$ のコストがかかる．一方，ビットベクタ表現を用いたデータフロー解析の初期化は，ワードごとの初期化が可能なことから，CFG のサイズを n，式の出現を c，ワードサイズを wrd として，$c+e*n/wrd$ のコストで済む．

ワードごとの初期化が可能な深さ表は，支配節候補に含まれる節が深さ i を持つとして，ビット位置 i には 1，そうでなければ 0 を立てたワードを要素とする配列 $depthBit$ と，値 1 のビットを含む $depthBit$ の添字を格納した配列 $depthWord$ を用意することによって実現できる．$depthWord[k]$ には，$j<k$ かつ $depthBit[j]\neq 0$ である最大の j の値を格納しておく．

$depthBit$ と $depthWord$ に基づく深さ表から，深さ $upper$ と $lower$ の間に支配節候補に含まれる節 v が存在するかどうかを，次のように判定することができる．

(1) $lower/wrd = upper/wrd$ の場合，
$depthBit[lower/wrd]$ のビット位置 $lower$ 以上，$upper$ 以下の部分をマスクして 0 かどうかを検査する．結果が 0 でなければ v が存在し，0 であれば存在しない．

(2) $lower/wrd > upper/wrd$ の場合，
$depthBit[lower/wrd]$ のビット位置 $lower$ 以上の部分をマスクして 0 かどうかを検査する．結果が 0 でなければ v が存在する．0 であれば次の条件を検査する．
  (a) $depthWord[lower/wrd] < upper/wrd$ の場合，v は存在しない．
  (b) $depthWord[lower/wrd] = upper/wrd$ の場合，$depthBit[depthWord[lower/wrd]]$ をビット位置 $upper$ 以下の部分をマスクして 0 かどうか検査する．結果が 0 でなければ v が存在し，0 であれば存在しない．
  (c) $depthWord[lower/wrd] > upper/wrd$ の場合，v が存在する．

例：ワードサイズを 4 として，$depthBit$ が $depthBit[0]=0000$，$depthBit[1]=1010$，$depthBit[2]=0000$，$depthBit[3]=0100$，$depthBit[4]=0000$ とすると，$depthWord$ は，$depthWord[0]=-1$，$depthWord[1]=-1$，$depthWord[2]=1$，$depthWord[3]=1$，$depthWord[4]=3$ となる． ∎

$depthBit$ と $depthWord$ の初期化の手順は，次のとおりである．

(1) 支配節候補の深さに対応する $depthBit$ のビットを 1 にする．
(2) $depthWord[0]=-1$ にする．
(3) $depthWord$ の添字 $i>0$ に対して，順に次の処理を行う．
  (a) もし，$depthBit[i-1]\neq 0$ であれば，$depthWord[i]=i-1$
  (b) そうでなければ，$depthWord[i]=depthWord[i-1]$

$depthBit$ と $depthWord$ に基づく深さ表の初期化は，ワードごとの初期化によって，$c+e*d/wrd$ の計算量に抑えることができる．

6.2 共通部分式の除去

一般支配検査の結果を利用して，共通部分式の除

去を実現する 1 つの方法は，各文 $z = x\ op\ y$ に対して，式のパターン $x\ op\ y$ ごとに異なる**一時変数**（temporary）t を用意し，$z = x\ op\ y$ という形式の文を $t = x\ op\ y;\ z = t$ と変形しておくことである．いったんこの変形を行っておけば，各 $t = x\ op\ y$ について他の $t = x\ op\ y$ の出現に対する一般支配検査を行い，結果が $True$ であったものを除去すればよい．

例：図 11 の `x1 + y1` について，共通部分式の除去を行った結果を**図 12** (a) に示す．∎

この方法は，図 12 (a) の結果のように，SSA 形式を壊してしまう可能性があるので，以降の SSA 形式に基づく解析を難しくする．

アルゴリズム 2 は，到達する変数を管理するためのスタック操作を加えることによって，SSA 形式を保持したまま，共通部分式の除去を扱えるように変更することができる．式 e の解析で行うスタック操作は，次のとおりである．

(1) 支配節候補に含まれる節 n に達した場合，n における e の代入先変数をスタックに積む．

(2) 訪問済みの節 n に再び到達した場合，新しい一時変数 t を生成し，t をスタックに積む．同時に，以降使用される変数として，t を配列 use の要素 $use[n]$ に記録しておく．

(3) 節 n から CFG 辺をたどった結果が，すべて $True$ であった場合，探索グラフの先行節の数だけスタックから変数を取り出し，その変数を引数として ϕ 関数を作成する．ϕ 関数の代入先は，新しい一時変数 t とする．もし，探索グラフに含まれない n への戻り辺が存在するなら，対応する引数を t にする．
作成した ϕ 関数を n の入口に挿入し，t をスタックに積む．$use[n] \neq \emptyset$ なら，各 $t' \in use[n]$ を代入先としてコピー代入 $t' = t$ を生成し，n の出口に挿入する．

最終的に，関数呼出し $SDom(n)$ の返値が $True$ であった場合，スタックの一番上にある変数によって，n に含まれる e を置き換える．返値が $False$ であった場合は，挿入した文をすべて削除する．

各代入文の挿入は，挿入する文を管理するスタックに積んでおくことで，$SDom$ の返値が得られるまで遅らせることができるので，$SDom(n) = False$ の際の削除処理は省くことができる．

例：図 11 における節 4 の `x1 + y1` について，SSA 形式を保存しながら，共通部分式の除去を行った結果を図 12 (b) に示す．∎

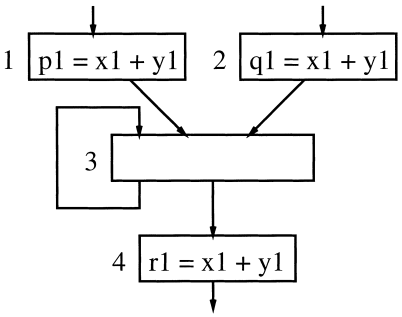
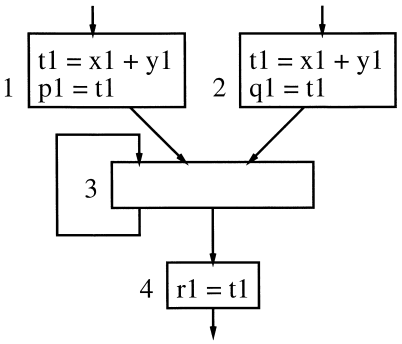
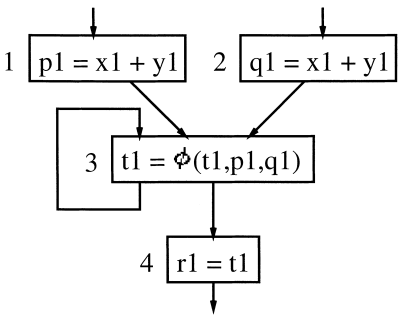


図 11 共通部分式の除去（除去前）
Fig. 11 Original program.



(a) SSA 形式の破壊（not keeping SSA form）



(b) SSA 形式の保持（keeping SSA form）

図 12 共通部分式の除去（除去後）
Fig. 12 Transformed program.

本手法は，セミ一般支配節に存在する共通部分式を発見するので，一般支配節の場合に比べて，共通部分式の数が多くなり，ϕ 関数の挿入が多くなる可能性がある．

6.3 議　　論

本手法の計算量は，式の出現数を c，CFG のサイズを n とすると，$O(c * n)$ である．一方，ワークリスト法を用いたデータフロー解析の計算量は，式のパターン数を e として，$O(e * n)$ である．データフロー解析は，各式のパターンについて，解析を 1 度行えばよいのに対して，本手法は，各式の出現についてそれ

ぞれ解析を行う必要があるので，本手法の計算量は，データフロー解析の計算量より大くなる場合がある．

評価の対象として用いた SSA 形式上での共通部分式の発見は，1 つの共通部分式を除去するたびに，コピー伝播を実行すると，多くの共通部分式を除去できることが知られているので，各式の出現ごとの効率的な解析が重要である．本手法が，この共通部分式発見において，最も効率の良いデータフロー解析の解法と同等かそれ以上の効率を示したことは，インクリメンタルなコード最適化やプログラム解析に対して有効な手段となることを意味している．

7. 関連研究

支配節を拡張した一般支配節の概念は，Gupta[11]によって最初に提案された．Gupta は，DDAG に基づく一般支配節の計算法を提案したが，DDAG の作成に要する計算量は，即一般支配節（immediate multiple-vertex dominator）の要素数を C，CFG 節の数を N とすると，$O(C*2^C*N+N^C)$ であり，高いコストを要する．DDAG を効率的に計算する方法には，CFG 辺の数 E として $O(E^2)$ の計算量の Gao ら[9]の手法と，構造化されたプログラムを対象とした線形の計算量の Alstrup ら[2]の手法がある．本稿で提案した手法は，DDAG のような特別な構造を必要とせず，多くのコード最適化やプログラム解析において一般に使用される CFG と支配木だけを利用している．

Gao らが DDAG の計算に使用している **DJ グラフ**[9),15)]は，本手法が一般支配検査に用いる探索グラフと同じ構造を持つ．しかし，DJ グラフは，支配木に対する深さを利用するのに対して，探索グラフは，CFG から上昇辺を取り除いた DAG の深さを利用している点で，DJ グラフとは異なる．

CFG 上の解析において，不要な節を訪問しない疎な解析法には，Choi らの**評価グラフ**（evaluation graph）[5]，Weise らの**値依存グラフ**（value dependence graph）[16]，Johnson らの**プログラム構造木**（program structure tree）に基づく**高速伝播グラフ**（quick propagation graph）[12]の研究があるが，いずれもそれぞれ変数や式のパターンごとに異なるグラフを作成して行うデータフロー解析である．本手法は，必要とするグラフ構造が共通であり，異なるのは表だけなので，実現が容易である．

8. 結論

本稿では，必要に応じて与えられた節が一般支配されているかどうかを効率良く検査する手法を提案した．

本手法は，CFG から上昇辺を取り除いた DAG から各節の深さを計算し，その深さに基づいて支配節候補の中に多重支配節が含まれることがあるかどうかを判定する．その可能性がある場合にだけ，CFG 辺をたどり，そうでない場合は，支配木の辺をたどればよいので，効率的な解析が可能である．また，一般支配検査に不要な CFG 辺を前もって取り除いておくことによって，さらに不要な検査を回避することができる．

本手法の効率を示すために，C コンパイラに本手法を実現し，訪問した節数をデータフロー解析による解法と比較した．その結果，本手法は，データフロー解析を最も効率的に解く方法と比較しても，同等かそれ以下の訪問節数で結果を得ることができることを示した．

本手法を用いることによって，支配関係に基づいたインクリメンタルなプログラム解析やコード最適化は，コストを犠牲にせずに容易に拡張することができる．

謝辞 本稿を執筆するにあたり，慶應義塾大学理工学部原田賢一教授と東京工業大学大学院情報理工学研究科佐々政孝教授に，的確なご指摘と有益なコメントをいただいた．

なお，本研究の成果は，科学技術振興調整費「並列化コンパイラ向け共通インフラストラクチャの研究」に基づくプロジェクトとの共同研究によるものである．

参考文献

1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley (1986).
2) Alstrup, S., Lauridsen, P.W. and Thorup, M.: Generalized Dominators for Structured Programs, *Proc. Int. Static Analysis Symposium* (*SAS'96*), Vol.1145 of LNCS, Aachen, pp.24–26, Springer-Verlag (1996).
3) Appel, A.W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
4) Briggs, P., Cooper, K.D. and Simpson, L.T.: Value Numbering, *Softwre-Practice and Experience*, Vol.27, No.6, pp.701–724 (1997).
5) Choi, J.D., Cytron, R. and Ferrante, J.: Automatic Construction of Sparse Data Flow Evaluation Graphs, *Proc. Principles of Programming Languages* (*POPL'91*), pp.55–66, ACM (1991).
6) COINS: A Compiler Infrastructure Project. http://www.coins-project.org/
7) Cytron, R., Ferrante, J., Rosen, B.K. and Wegman, M.N.: Efficiently Computing Static Single Assignment Form and Control Depen-

dence Graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (1991).
8) Dhamdhere, D.M., Rosen, B.K. and Zadeck, F.K.: How to Analyze Large Programs Efficiently and Informatively, *Proc. Programming Language Design and Implementation* (*PLDI'92*), pp.212–223, ACM (1992).
9) Gao, G.R., Lee, Y.F. and Sreedhar, V.C.: DJ-graphs and Their Application to Flow Graph Analysis, Technical Report 70, McGill University, School of Computer Science, ACAPS (1994).
10) Gupta, R.: A Fresh Look at Optimizing Array Bound Checking, *Proc. Programming Language Design and Implementation* (*PLDI*), pp.272–282, ACM (1990).
11) Gupta, R.: Generalized Dominators and Post-dominators, *Proc. Principles of Programming Languages* (*POPL'92*), pp.246–257, ACM (1992).
12) Johnson, R., Pearson, D. and Pingali, K.: The Program Structure Tree, *Proc. Programming Language Design and Implementation* (*PLDI*), pp.171–185, ACM (1994).
13) Khedker, U.P. and Dhamdhere, D.M.: A Generalized Theory of Bit Vector Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.5, pp.1472–1511 (1994).
14) Muchnick, S.S.: *Advanced Compiler Design Implementation*, Morgan Kaufmann (1997).
15) Sreedhar, V.C. and Gao, G.R.: A Linear Time Algorithm for Placing ϕ-Nodes, *Proc. Principles of Programming Languages* (*POPL'95*), pp.62–73, ACM (1995).
16) Weise, D., Crew, R.F., Ernst, M. and Steensgaard, B.: Value Dependence Graphs: Representation with Taxation, *Proc. Principles of Programming Languages* (*POPL'94*), pp.297–310, ACM (1994).

(平成 15 年 5 月 20 日受付)
(平成 15 年 7 月 8 日採録)

滝本　宗宏（正会員）

1994 年慶應義塾大学大学院理工学研究科計算機科学専攻修士課程修了．現在，東京理科大学理工学部情報科学科助手．工学博士．プログラミング言語およびその処理系に興味を持つ．ACM，IEEE，日本ソフトウェア科学会各会員．

武田　正之（正会員）

1977 年東京理科大学理工学部電気工学科卒業．1982 年東京工業大学大学院博士課程（電子物理工学専攻）修了．同年東京理科大学理工学部情報科学科助手となり，現在同大学助教授．工学博士．著書（共著）に『Prolog とその応用 2』（総研出版，1985）等がある．プログラミング言語の意味論，並列・分散システム，知識情報処理に興味を持つ．1982 年度情報処理学会論文賞受賞．電子情報通信学会，日本ソフトウェア科学会，人工知能学会，ACM 各会員．