拡張値グラフを用いた部分無効コード除去法

滝本宗宏[†] 原田賢一[††]

基本的なコード最適化手法の1つである無効コード除去を，コード移動によって強化した部分無効コード除去は，代入文を移動することによって，新たに無効コードを生成して除去するばかりでなく，不変コードのループ外移動も同時に扱える強力なコード最適化手法である．しかし，従来の部分無効コード除去は，字面が同じ代入文を対象とし，制御フローグラフの結合点において，コードの移動が制限されていた．また，1回の部分無効コード除去の適用は，新たな無効コードを生成するという二次的効果を持つので，繰り返して適用する必要があり，計算コストが大きいという問題があった．本稿では，部分無効コード除去におけるコード移動について，字面による制限を取り除き，従来手法よりも広範囲にわたる部分無効コードを除去する手法を提案する．また，本手法では，部分無効コードの解析のために制御フローと代入文の構造を統合して，データフロー解析を適用する．この方法によって，二次的効果を一次効果として得ることができるので，繰返し適用の必要がなく，計算コストを低く抑えることができる．

Partial Dead Code Elimination Using Extended Value Graph

Munehiro Takimoto[†] and Kenichi Harada[††]

This paper presents an efficient and effective code optimization algorithm for eliminating partially dead assignments, which become redundant on execution of specific program paths. It is one of the most aggressive compiling techniques, including invariant code motion from loop bodies. Since the traditional techniques proposed to this optimization would produce the second-order effects such as sinking-sinking effects, they should be repeatedly applied to eliminate dead code completely, paying higher computation cost. Our technique proposed here can eliminate possibly more dead assignments regardless of the lexically same pattern, using an explicit representation of data dependence relations within a program in a form of SSA (Static Single Assignment). Such representation called Extended Value Graph (EVG), shows the computationally equivalent structure among assignments before and after moving them on the control flow graph. We can get the final result directly by once application of this technique, because it can capture the second-order effects as the main effects, based on EVG.

1. はじめに

コンパイラにおけるコード最適化手法の1つに**無効コード除去**（dead code elimination）[1)]がある．この手法を用いることによって，実行時に不必要な代入文の実行が避けられるので，プログラムの実行効率を高めることができる．

無効コード除去の効果を高めるために，コード移動と組み合わせた手法として，**部分無効コード除去**（partial dead code elimination）がある．本稿では，部分無効コード除去に新しいアルゴリズムを導入することによって，より効果的な最適化を，より効率良く行う手法を提案する．

変数の値が，あるプログラム点 p 以降のプログラム実行において使用されることがない場合，その変数は p において**無効**（dead）であるという．無効な変数を左辺に持つ代入文は不必要であり，**全無効**（totally dead）あるいは単に無効であるという．通常，無効コード除去においては，全無効な代入を対象としている．全無効に対して，**図1** (a) の節1のように，分岐の左側では無効であっても，右側では有効である代入文を**部分無効**（partially dead）[13)]であるという．この場合には，代入文 $y := a + b$ を節1から節2と3の各入口に**降下**（sink）させることによって，節1の代入

† 東京理科大学理工学部
Department of Information Sciences, Faculty of Science and Technology, Science University of Tokyo
†† 慶應義塾大学理工学部
Department of Computer Science, Faculty of Science and Technology, Keio University

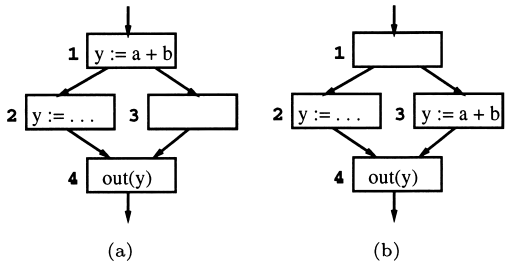


図 1 部分無効代入の除去
Fig. 1 Removing partially dead assignment.

文は節 2 において全無効になり，図 1 (b) のように除去することができる．ここで，降下とは，代入文の**前向き**（forward）の移動を意味する．このように，部分無効代入を適切なプログラム点に降下させることによって，全無効代入に置き換えて除去する手法を部分無効コード除去という．

部分無効コード除去において，代入文の降下できる範囲を大きくすることは，部分無効代入の除去の可能性を大きくすることになる．たとえば，**図 2** (a) において，節 4 の代入文 $y := x$ を節 5 に降下させることができれば，その文は，節 5 の入口で全無効になり，除去できる．その結果，$y := x$ で使用している x も無効になる．この場合，x を左辺に持つ文を節 5 の入口に降下させることができれば，さらにその文も無効になる．このような降下が可能になるためには，**結合点**（join point）である節 4 において，左側から降下してくる文と，右側から降下してくる文とで字面上のパターンが一致しなければならない．図 2 (a) の場合，節 2 の $x := a + b$ と，節 3 の $x := a + c * 3$ とはパターンが異なるので，このままでは，節 4 への降下は**ブロック**される．しかし，部分式 $c * 3$ の値を b に代入することができる場合には，両者の代入文は同じパターンになり，$x := a + b$ を節 46 へ降下させることができる．

本研究では，降下がブロックされる文を，オペランドの名前を付け替えることによって，さらに広範囲な降下を可能にする手法を提案する．広範囲な降下は，従来法に比べて多くの無効代入除去の機会を与える．本手法を実現させるために，変数の使用–定義関係に基づく依存構造を大域的レベルで表現する**拡張値グラフ**（Extended Value Graph，以降 EVG と呼ぶ）[19),20)] を用いる．EVG は，原始プログラム中に現れる計算の依存構造だけでなく，代入文が各プログラム点へ移動した場合に持つ依存構造を統合して表現する．文献 20) は，コードの**巻上げ**（hoisting）を目的として，各プログラム点における計算の依存構造を適切に表現するために EVG を定義しているが，本研究では，コード降下を行った際の依存構造を表現できるように EVG を定義しなおす[18)]．この EVG の性質を利用すると，変数名の付替えの効果は，EVG の変形によって得られる．

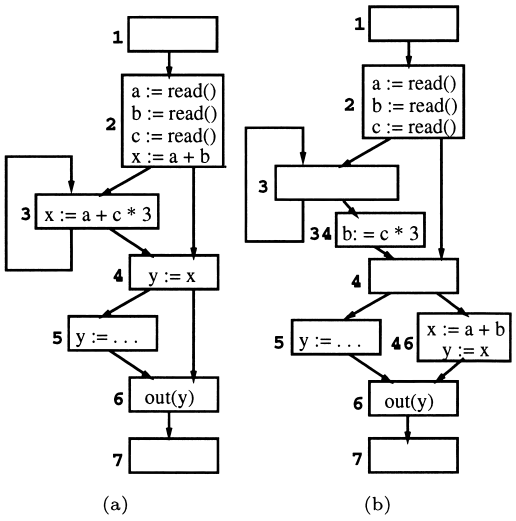


図 2 除去できない部分無効代入
Fig. 2 Missed optimization.

EVG に基づく部分無効コード除去はコンパイル効率の向上にも貢献する．コード降下および無効コード除去は，新たなコード降下あるいは無効コード除去の候補を生成する可能性がある．これらの効果は，次の 4 種類に分類することができる[13)]．

降下–除去 （Sinking-Elimination Effects）
代入文の降下によって，その代入文が無効になる．

降下–降下 （Sinking-Sinking Effects）
代入文を $s1$，$s2$ とするとき，$s1$ で定義する変数を $s2$ が使用するか，$s2$ で再定義する場合，あるいは $s1$ で使用している変数を $s2$ が定義する場合，$s2$ によってブロックされていた $s1$ が，$s2$ の降下によってさらに降下できるようになる．

除去–降下 （Elimination-Sinking Effects）
上記の $s1$，$s2$ について，$s2$ が除去されることによって，$s1$ が降下できるようになる．

除去–除去 （Elimination-Elimination Effects）
代入文 s の除去によって，s で使用している変数に対する代入文が除去できるようになる．

1 回の部分無効コード除去の適用によって，降下–除去効果を得ることができるが，残りの 3 つの効果を得るには，部分無効コード除去を再度適用する必要がある．これは，1 回の部分無効コード除去の適用が，別の無効代入を生み出し，さらに除去の機会を与えるか

らである．この効果を，部分無効コード除去の二次効果という．従来の方法は，二次効果を繰り返し利用するので，最適化には大きなコストを要していた．

本稿では，制御フローグラフと EVG の 2 つのグラフ上でデータフロー解析を行うことによって，上述の二次効果を一次効果として得る方法も示す．二次効果が一次効果として得られなかった理由は，次の 2 つである．

(1) 文 $s(x := t)$ の降下は，プログラムの意味を保存するために，x に対する定義 $s1$，x の使用 $s2$，または s で使用している変数に対する定義 $s3$ によってブロックされる（図 2 (a) において，節 3 の文は節 4 でブロックされる）．このとき，$s1$，$s2$，$s3$ が降下しても，s の降下はそれらの降下前のプログラム点でブロックされる．

(2) 変数 x の使用がなくなると，x に対する定義も除去できる可能性がある．しかし，x の定義が実際に除去されるのは，再度，部分無効コード除去を適用するときである．

本手法では，入力プログラムとして，**静的単一代入形式**（Static Single Assignment Form，以降 **SSA** 形式と呼ぶ）[2)~4),6),15)]を使用するので，上の (1) に関して，$s1$ と $s3$ によるブロックは起きない．$s2$ によるブロックについては，EVG の導入によって $s2$ の降下が s に反映されるので，s の降下がブロックされるのは，$s2$ の降下先になる．

(2) に関しては，変数 x を使用している文 us が，降下によって除去できる場合，その x を定義する文 ds が除去できるかどうかは，us に対する部分無効コード除去の結果を待たなければならない．この降下と除去の依存関係を取り除くために，本手法では，各文が，その位置で無効であるかどうかではなく，降下可能な各プログラム点において無効になるかどうかを計算する．すなわち，文が，各プログラム点において削除できるかどうかを，実際に文を降下させる前に計算しておくことによって解決する．

本稿の構成は次のとおりである．2 章で，前提となるプログラム表現について簡単に述べる．そして，3 章において，本手法の基礎となるデータ構造である拡張値グラフの定義と構成法を述べる．4 章では，制御フローグラフと拡張値グラフに適用するデータフロー方程式を示し，それに基づく部分無効コード除去の例を示す．本手法全体の計算量を 5 章で考察する．本手法の効果については，評価結果を 6 章に示す．本研究の位置を示すために 7 章で関連研究を述べ，最後に結論を述べる．

2. 入力プログラムの表現

処理内容を簡潔に述べるために，以下では，与えられた原始プログラムに関して次の集合を用いる．

- Var：変数集合
- Tmp：一時変数の集合
- C：定数集合
- OP：演算子集合

また，原始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）が作成されているものとする．CFG は，基本ブロックからなる節の集合 N，$E \subset N \times N$ の辺の集合，および特別な節である開始節 $\mathbf{s}$ と終了節 $\mathbf{e}$ からなる四つ組 $(N, E, \mathbf{s}, \mathbf{e})$ として表される．CFG 節 n の i 番目の先行節を $pred_i(n)$，i 番目の後続節を $succ_i(n)$ で表現する．

CFG 節に含まれる文は，次の 3 種類とする．

代入文 三番地コード $x := t$ の形で表現する．ここで，$x \in Var \cup Tmp$ であり，t は演算子をたかだか 1 つ含む式とする．

空文 操作を必要としない文．CFG 節に文が存在しない場合は，1 つの空文（skip statements）を含むものとする．

重要文 メモリ操作をともなう文，関数呼び出し，分岐文など，移動によってプログラムの意味が変わる文．重要文（relevant statements）は，元の CFG 節に固定のものとし，重要文で使用される変数は，すべて有効なものとして扱う．本稿では，重要文を $out(t)$ のような出力文の形で表す[13)]．

さらに，各文は，SSA 形式に変換されていることを前提とする．任意の原始プログラムに対する CFG から SSA 形式への変換については，効率的なアルゴリズム[3),7),16)]が知られている．SSA 形式とは，次の性質を満たすプログラムの表現形式である．

(1) すべての変数の使用は，唯一の定義を持つ．すなわち，1 つの変数は一度しか代入されない．

(2) 異なる辺から 2 つ以上の定義が使用に達する場合は，**ϕ-関数**と呼ぶ仮想関数を用いて，フローの合流点で定義の結合を明示する．本稿では，ϕ-関数を存在する CFG 節によって区別し，その集合を Φ とする．

図 2 (a) に対する SSA 形式を**図 3** に示す．同じ名前で表現されている変数は，定義ごとに変数名の後ろに数字を付けて区別する．定義が結合する箇所には，ϕ-関数による代入を挿入し，定義の結合を明示的に表現する．

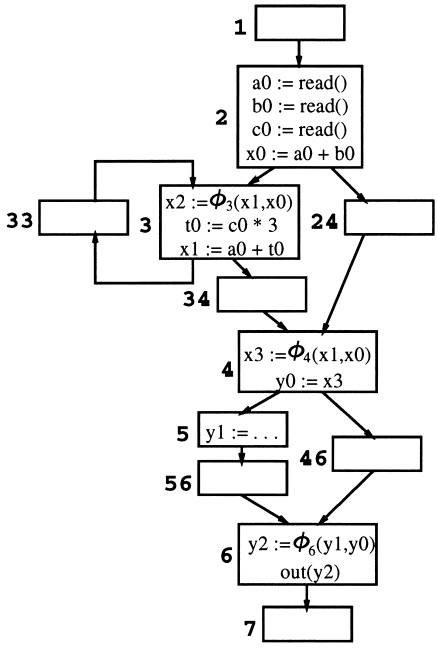


図 3 入力プログラムの表現
Fig. 3 Representation of input program.

以下，説明を簡単にするために，一般性を失うことなく ϕ-関数の引数は 2 つとする．すなわち，CFG において，1 つの節に入ってくる辺は 2 つまでとする．また，2 つ以上の後続節を持つ節から 2 つ以上の先行節を持つ節への辺（**クリティカル辺**，critical edge[11),12),20)]）は取り除かれているものとする．

本稿では，2 つ以上の先行節を持つ節に入ってくる辺は，すべて新しく節を挿入することによって分割されているものとする．この変形によって，クリティカル辺は除去でき，後の解析を単純にすることができる☆[11),20)]．

3. 拡張値グラフ

計算の依存構造を，字面パターンに制限されることなく扱うために，変数の定義と使用を辺で結んだグラフ表現がよく使用される．CFG の節では，そこに含まれる式を表現するために **DAG**（Directed Acyclic Graph）[1)]表現が使われる．DAG は，一部の計算の依存関係を表現するだけであり，プログラム全体における依存関係を表現することはできない．

一方，プログラム全体の定義と使用を結んだグラフである**定義–使用チェイン**（def-use chain）や**使用–定義チェン**（use-def chain）[1)]は，計算の依存関係全体をグラフで表現することができる．しかし，定義の結合を表現する方法を持たないので，制御フローに依存して変わる値を明示的に表現することができない．

これに対して，SSA 形式のプログラムにおける変数の定義と使用を辺で結んだ**値グラフ**（Value Graph，以降 **VG** と呼ぶ）[2)]や **SSA グラフ**（Static Single Assignment Graph）☆☆[6)]は，定義の結合（ϕ-関数）も合わせて，プログラム全体を表現できるので，制御フローと独立に計算構造を取り扱うことができる．

本手法では，この VG を拡張して，コード移動によるプログラム変形を表現できる EVG を導入する．EVG は，コード移動前後の計算の依存構造を表現できるので，コードの移動過程で，計算の依存構造の情報を利用することができるようになる．

この章では，コード降下と EVG の関係を示し，EVG の定義を与える．そして，その作成法を示す．

3.1 拡張値グラフの定義

通常形式のプログラムで，**図 4** (a) における節 1 と節 2 の文 $x := a + b$ を節 3 に降下させ，(b) のように変形することは，SSA 形式では，**図 5** (a) から (b) への変形に対応する．図 5 の節 3 において，(a) では，変数 $x3$ が ϕ-関数によって定義され，(b) では，$x3$ が + の計算によって定義されている．このような SSA 形式における変形は，ϕ-関数によって表現される値 $x_k := \phi_n(x_i, x_j)$ に対して，その引数 x_i，x_j を定義するすべての代入文を，x_k の定義と同じ CFG 節に降下させるときに生じる．

この SSA 形式で表されたプログラムの依存構造を表現するために VG を用いる．VG の定義は，次のとおりである．

定義 1（値グラフ） VG は，節の集合 V_{VG} と節から節への辺 A_{VG} からなる組 (V_{VG}, A_{VG}) である．各節は $OP \cup C \cup \Phi$ のラベルを持つ（定数値については，同一のラベルを持つ節は 1 つの節で表されているものとする）．

このとき，図 5 のプログラム (a) と (b) それぞれに対応する VG を**図 6** に示す．図 4 において，x への代入文が節 3 へ降下する前と後での x の値の表現を VG で考えると，図 6 では (a) の ϕ 節が (b) の + 節へ変化することが分かる．

通常形式のプログラムにおける代入文 s をその CFG 節 n へ降下させる場合を考え，元のプログラムに対

☆ 説明を簡単にするために，本稿の図において，不要な節は省略している．

☆☆ 値グラフは使用から定義へ向かう辺を持ち，計算の依存関係を表す．一方，SSA グラフは定義から使用へ向かう辺を持ち，値の流れを表現する．

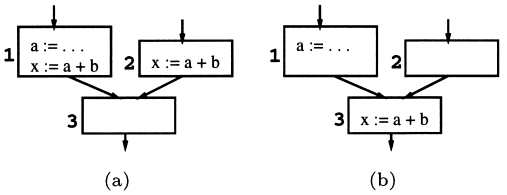


図 4 コード降下（通常形式）
Fig. 4 Code sinking in normal form.

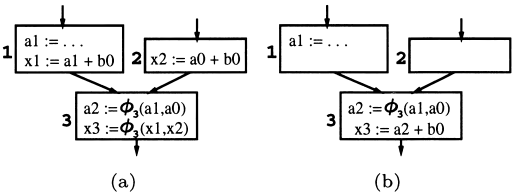


図 5 コード降下（SSA 形式）
Fig. 5 Code sinking in SSA form.

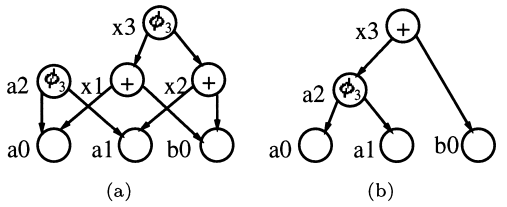


図 6 コード降下（VG 表現）
Fig. 6 Code sinking (Value Graph representation).

応する VG 表現が，その降下によってどのような変形を受けるかを示す．節 n における x の値を表す VG 節を v とする．

(1) v が ϕ 節でなければ，v は不変である．
(2) v が ϕ 節であり，SSA 形式において，その ϕ-関数があった CFG 節を n_ϕ とすると，次の 2 通りの場合がある．
  (a) $n \neq n_\phi$ の場合，v は不変である．
  (b) $n = n_\phi$ の場合，さらに次の 2 通りの場合がある．
    (i) v から出るすべての辺の先が同じラベル l を持つとき，l をラベルとして持つ節を新たに導入し，その節によって x の値を表現する．
    (ii) v から出る辺の先が異なるラベルを持つとき，n への降下はブロックされる．

本稿では，コード降下前後の依存構造を一度に表現するために，EVG と呼ぶグラフ表記を用いる（**図 7**）．このグラフの各節は，図 6 (a) の各節とそれに対応する同図 (b) の節を対にしたものからなる．1 つの EVG 節を構成する VG 表現の各節を，以降 EVG 節の**副節**と呼ぶ．

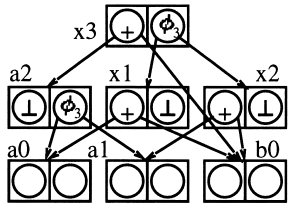


図 7 EVG 表現
Fig. 7 Extended Value Graph represetation.

EVG の定義は，次のとおりである．

定義 2（拡張値グラフ） EVG は，副節の集合 SV，副節の組の集合 V，副節から節への辺の集合 A からなる三つ組 (SV, V, A) である．

- $OP \cup C$ のラベルを持つ副節集合を SV_{op} とし，Φ のラベルを持つ副節集合を SV_ϕ とすると，節集合 V は，副節の組 $SV_{op} \times SV_\phi$ である．
- EVG に対する操作として，次の関数を定義する．
  $op : V \to SV_{op}$. $(sv_{op}, sv_\phi) \in V$ から sv_{op} を取り出す関数
  $phi : V \to SV_\phi$. $(sv_{op}, sv_\phi) \in V$ から sv_ϕ を取り出す関数
  $lb : SV \to OP \cup C \cup \Phi$. $sv \in SV$ からラベルを得る関数
  $loc : SV_\phi \to N$. $sv_\phi \in SV_\phi$ に対応する ϕ-関数のあった CFG 節を得る関数
  $child_i : SV \to V \cup \perp$. $sv \in SV$ の i 番目の後続節 v を得る関数．ここで，$(sv, v) \notin A$ ならば，$\perp$ とする．$sv_\phi \in SV_\phi$ のとき，$child_i(sv_\phi)$ は，対応する $pred_i(loc(sv_\phi))$ から到達する値を表現する．
- $Child_{op}(v) =_{def} \{v' | v' = child_i(op(v)) \wedge v' \neq \perp,\ for\ i = 0,\ 1\}$,
  $Child_\phi(v) =_{def} \{v' | v' = child_i(phi(v)) \wedge v' \neq \perp,\ for\ i = 0,\ 1\}$.
- $Parent_{op}(v) =_{def} \{v' | op(v') = child_i^{-1}(v),\ for\ i = 0,\ 1\}$,
  $Parent_\phi(v) =_{def} \{v' | phi(v') = child_i^{-1}(v),\ for\ i = 0,\ 1\}$.

図 7 において，$x3$ の値を表現する EVG 節 v は，x への代入文が CFG 節 $loc(phi(v))$ に降下する前は $phi(v)$ の構造を持ち，その節に降下した後は，$op(v)$ の構造を持つことを意味する．

3.2 拡張値グラフの作成

VG から，元の計算の依存構造だけを表現する EVG

を直接的に作成することができる．このEVGを**初期EVG**と呼ぶ．最終的に用いるEVGは，次の手順によって作成する．

(1) 初期EVGの作成
(2) 初期EVGの変形

初期EVGはVGと同じ構造持ち，それを変形することによって，コード移動の情報を含むEVGが得られる．

VGから初期EVGを作成する方法は次のとおりである．まず，各VG節を副節 sv とし，新しく導入した副節 $sv' \in SV$（$lb(sv') = \bot$）と次のように組にする．

- $lb(sv) \in OP \cup C$ ならば，(sv, sv') を生成する．
- $lb(sv) \in \Phi$ ならば，(sv', sv) を生成する．

この後，VG節を指している辺の先を，そのVG節を副節として持つEVG節に付け替える．

コード降下による計算の依存構造の変化を初期EVGに加えるには，降下によって構造が変化する部分についてだけ，初期EVGを変形する．その影響を受けるのは，CFGにおいて，2つ以上の先行節を持つ結合点に文を降下させるときである．結合点 n にその先行節から変数 x（通常形式における変数名）への代入文が降下させるための条件をSSA形式で考えると，次のように示すことができる．ここで，SSA形式におけるそれらの代入先を x_i（$i = 0, 1, \ldots$）とする．

(1) n の先行節から n に降下させる代入文の代入先 x_i がすべて同じである場合（x の定義が唯一である場合），または
(2) n の先行節から n に降下させる代入文の代入先 x_i は異なっているが，すべての x_i が n における ϕ-関数の引数である場合．

(1)については，計算のパターンは変わらないので，EVG表現も同じである．(2)については，代入先として，ϕ-関数の代入先と同じ変数名を持つ1つの代入文で表現される．n の各先行節から降下する代入文を1つの代入文で表現するには，降下させる代入文の右辺についても，次の条件が満たされなければならない．

(1) 降下させる代入文は，すべて同じ演算子あるいは関数名を持たなければならない．さらに，
(2) 降下させる代入文の対応するオペランドあるいは引数は，通常形式においてすべて同じでなければならない．

(2)については，SSA形式において，代入先と同じ条件が必要である．以上から，EVG節 $v = (sv_{op}, sv_\phi)$ は次の変形条件を持つ．

変形条件：

$sv_i = op(child_i(sv_\phi))$（$i = 0, 1$）として，

(1) $lb(sv_{op}) = \bot \wedge lb(sv_\phi) \neq \bot$.
(2) すべての sv_i について，$l = lb(sv_i)$ が同じであること．
(3) すべての sv_i について，$child_j(sv_i)$ が同じ節 v_j であるか，あるいは $child_j(sv_i) \in Child_\phi(v_j) \wedge loc(phi(v_j)) = loc(sv_\phi)$ を満たす v_j が存在すること（$j = 0, 1$）．

変形条件を満たす節 v に対する変形は，合体した代入文を表現する sv_{op} のラベルを $\bot$ から代入文の演算子に変更し，必要なEVG辺を付加する操作からなる．このEVG辺の付加は，ϕ-関数 $x_k := \phi_n(\cdots)$ の引数になっている変数への代入文が n に降下する際に，その代入先を x_k に変更する操作に対応する．v に対する変形操作は，変形条件の記号を用いると，次のように表現できる．

変形操作：

(1) $lb(sv_{op})$ を l に付け替える．
(2) 辺 (sv_{op}, v_j) を生成して，A に加える．

実際のEVGに対する変形は，降下してくる文の右辺に着目し，**図8**に示すように4パターンに分類でき

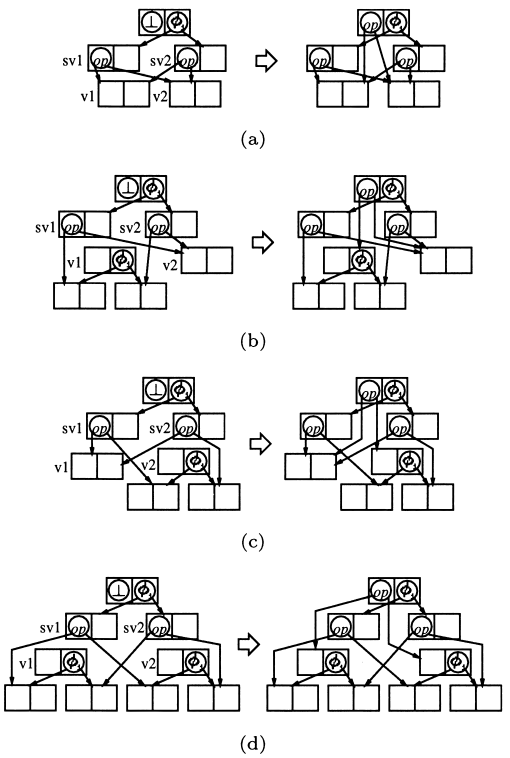


図8 変形パターン
Fig. 8 Transformation patterns.

る．それらは次の場合に対応する．

(1) 対応するすべてのオペランドが，SSA 形式において同じ変数名である場合（図 8 (a)）．

(2) 対応する第 2 オペランドは，SSA 形式において同じ変数名であるが，第 1 オペランドが同じ ϕ-関数の引数になっている場合（図 8 (b)）．

(3) 対応する第 1 オペランドは，SSA 形式において同じ変数名であるが，第 2 オペランドが同じ ϕ-関数の引数になっている場合（図 8 (c)）．

(4) 対応するすべてのオペランドが，同じ ϕ-関数の引数になっている場合（図 8 (d)）．

3.3 変形条件の緩和

図 3 から初期 EVG を作成すると，**図 9** (a) に示す結果が得られる．$x2$ の値に対応する節や $x3$ の値に対応する節は，それぞれ辺の先が同じラベル $+$ を持つが，それらのオペランドは変形条件 (3) を満たさないので，変形の対象にならない．その理由は次のとおりである．$x0$ に対応する節と $x1$ に対応する節に関して，左から出る辺の先はそれぞれ同じ $a0$ に対する節であるが，右から出る辺の先は異なっていて，それらを結合させる ϕ 節もないからである．図 3 からも，$x1 := a0 + t0$ と $x0 := a0 + b0$ は結合できないことが分かる．

この場合，図 9 (b) の点線で示した節のように，一時変数 $tb0$, $tb1$ を表す節を挿入する．これらの節をそれぞれ (sv, sv') とし，$x2$ に対応する節を $vx2$ とすると，(sv, sv') は $lb(sv) = \bot \wedge lb(sv') = lb(phi(vx2)) \wedge loc(sv') = loc(phi(vx2))$ の条件を満たす．それぞれの sv' からは $b0$ と $t0$ に対応する節へ辺を生成する．この操作は，通常形式の代入文において，異なる代入先の変数名を同じ名前に付け替えることに対応し，さらなる変形が可能になる．

以上から，一連の変形アルゴリズムは，次のように要約できる．

(1) 上述の変形条件を満たす節，あるいは新しい節を挿入して変形条件を満たす節を変形する．

(2) 変形候補がなくなるまで (1) を繰り返す．

(2) については，節 v への変形は $lb(op(v)) \neq \bot$ にするので，$(sv, v) \in A \wedge lb(sv) \in \Phi$ のような sv を持つ節が，新たに変形候補になる可能性がある．また，新しく生成された節も，変形候補になる場合が考えられるが，変形対象は元の V の要素だけとする．したがって，変形アルゴリズムの計算量は ϕ-節の数に比例する．

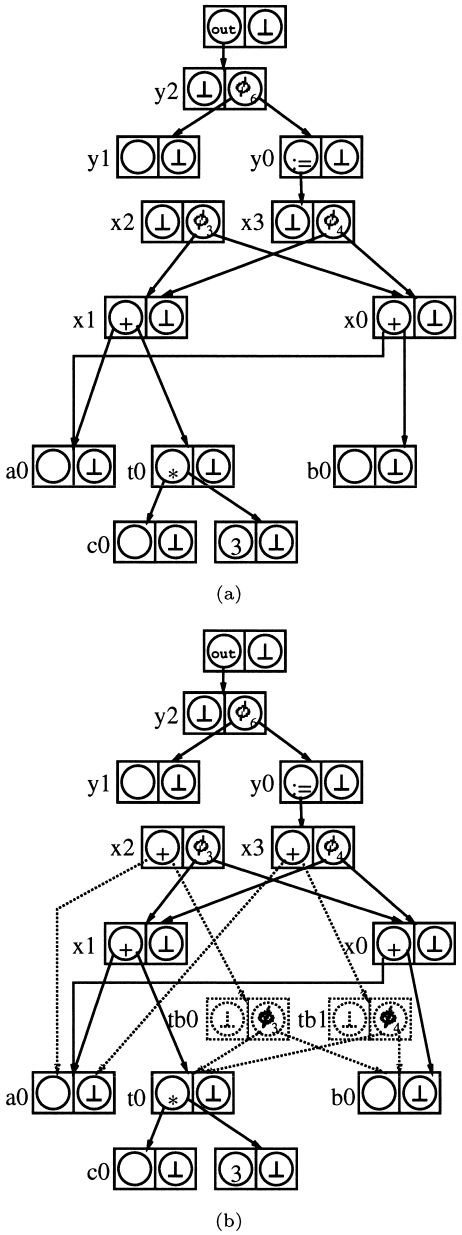


図 9 図 2 (a) の EVG
Fig. 9 EVG of Fig. 2 (a).

4. 部分無効コード除去

この章では，従来の部分無効コード除去が繰返し適用によって得ていた二次効果を，3 章で導入した EVG を用いて，一次効果として見付け出す部分無効コード除去について述べる．

まず，無効な代入文と降下可能な代入文が二次効果として生じるという相互関係を取り除くために，絶対無効変数と絶対無効代入という概念を導入する．これ

によって，最も降下できるプログラム点と，無効になる代入を独立に決定することが可能になる．

次に，絶対無効変数とコード降下のデーターフロー方程式を示し，部分無効コード除去の実現法を例を用いて示す．

4.1 絶対無効

代入文 $x := t$ の除去によって，t の中で使用している変数への代入文が無効になる可能性がある．このような，二次的に無効コードとなる代入文を，特に**弱体代入**（faint assignment）[13]という．

弱体代入は，すべての重要文から走査を始めて，使用–定義関係によって到達できる各文の中で使用されることのない変数への代入をいう．ここで，弱体代入は，変数を使用している文と定義している文とが，元のプログラムの位置にあるものとして計算されるので，コード降下の効果を反映することができない．コード降下によって，新たに弱体となった文の除去は，次の部分無効コード除去☆の適用まで待たなければならない．

この問題を解決するために弱体代入を拡張した，**絶対無効代入**（absolute dead assignment）という概念を導入する．絶対無効代入の定義を示す前に，次の 2 つの用語を導入する．

定義 3（関係する重要文） 代入文 $s \equiv x := t$ に関係する重要文 r とは，s で定義する値が，最終的に使用される重要文のことである．

s と r には，対応する EVG 節が存在し，それぞれを v と vr とすると，vr は v の祖先である．

定義 4（最下点） 代入文 $s \equiv x := t$ の文 s' に対する最下点を次のように定義する．s，s' が存在する CFG 節を，それぞれ n，n' として，

(1) もし，n から終了節 **e** へのパス上に，x を使用している文 us が存在すれば，us の s' に対する最下点が s の s' に対する最下点である．

(2) もし，n から n' へのパス P 上に，x を引数とする ϕ-関数 s_ϕ が現れれば，s_ϕ の存在する CFG 節 n_ϕ の P 上の先行節が最下点である．

(3) (1)でも(2)でもなければ，n' が最下点である．

s の s' に対する最下点は，s に対応する EVG の構造を保存したままで降下可能なプログラム点のうち，s' に最も近いプログラム点である．

EVG を用いて，絶対無効変数と絶対無効代入を次のように定義する．

定義 5（絶対無効変数と絶対無効代入） 絶対無効変数とは，各重要文を r として，r に対応する EVG 節 vr から下向きに到達可能な節に対応する文だけが，r に対する最下点で使用されていると仮定した場合の無効変数である．絶対無効変数を左辺に持つ代入文を，絶対無効代入という．

絶対無効代入は，原始プログラムに現れる各文が，関係する重要文に最も近い点に降下されていると仮定して弱体代入を計算したものに相当する．これによって，コード降下が行われたときに無効となるべき文とプログラム点とを，降下に関係なく，計算しておくことができる．

4.2 データフロー解析

4.1 節で定義した絶対無効変数を計算し，その後，代入文を降下させて，部分無効代入を除去するデータフロー解析を示す．

データフロー解析には，各データスロットごとにデータを伝播させる，スロットワイズ[8),10),17)]と呼ぶ方法を用いる．入力プログラムは SSA 形式を仮定しているので，スロットは，対応する代入文を表現する EVG 節と，プログラム点に対応する CFG 節の組で表現できる．このスロット集合を $SL \subset V \times N$ と表す．通常形式のプログラムにおいては，同じスロットで表現されるデータも，SSA 形式においては，次のように異なったスロットによって表現される場合がある．部分無効コード除去は，変数が有効であるか無効であるかという性質に注目しているので，通常形式のプログラムにおいては，代入先の変数ごとにスロットを用意すればすむ．しかし，SSA 形式においては，定義ごとに新しい変数名が導入されているので，後に ϕ-関数によって結合される定義に複数のスロットが割り付けられることがある．スロット $sl1$ の情報を次にスロット $sl2$ に伝播させなければならないとき，$sl1$（あるいは $sl2$）を $sl2$（あるいは $sl1$）の**隣接スロット**と呼ぶ．

スロット $sl \equiv (v, n)$ の隣接スロットは，sl の先行スロット $pred_{SL}(sl)$ と後続スロット $succ_{SL}(sl)$ を定義することによって与えることができる．まず，sl を含む CFG 節の i 番目の先行節に含まれる先行スロット $pred_{SLi}(sl)$ と後続節に含まれる後続スロット $succ_{SLi}(sl)$ を次のように定義する．

- $pred_{SLi}(sl)$:
  - もし，$loc(phi(v)) = n$ ならば，$(child_i(phi(v)), pred_i(n))$ である．
  - もし，$\exists(v' \neq v).loc(phi(v')) = n \wedge v \in Child_\phi(v')$ ならば，存在しない．
  - さもなければ，$(v, pred_i(n))$ である．

☆ 弱体コードの概念を用いたものを，特に部分弱体コード除去（partial faint code elimination）と呼ぶ．

- $succ_{SLi}(sl)$:
  - もし，$\exists(v' \neq v).v' \in Parent_{\phi}(v) \wedge loc(phi(v')) = succ_i(n)$ ならば，$(v', succ_i(n))$ である．
  - もし，$succ_i(n) = loc(phi(v))$ ならば，存在しない．
  - さもなければ，$(v, succ_i(n))$ である．

$pred_{SLi}(sl)$ と $succ_{SLi}(sl)$ によって，$pred_{SL}(sl)$ と $succ_{SL}(sl)$ は，次のように与えられる．

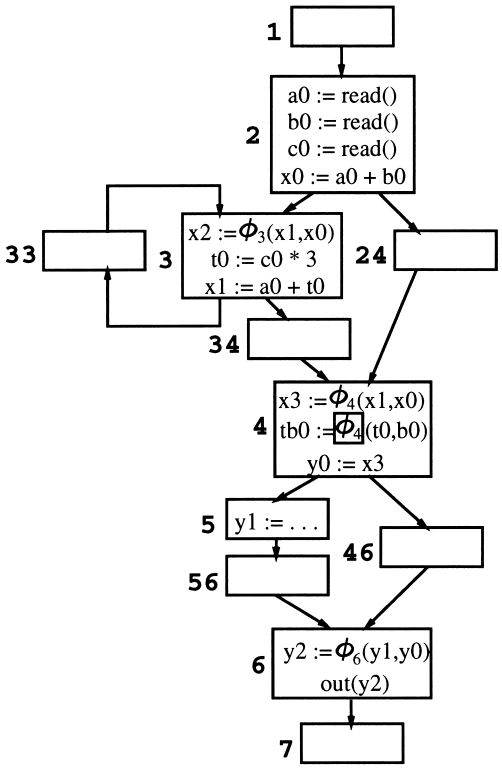


図 10 変形後の SSA 形式プログラム
Fig. 10 Transformed program.

- $pred_{SL}(sl) =_{def} \{sl' | sl' \equiv pred_{SLi}(sl),\ for\ i = 1, 2\}$.
- $succ_{SL}(sl) =_{def} \{sl' | sl' \equiv succ_{SLi}(sl),\ for\ i = 1, 2\}$.

さらに，$pred_{SL}(sl)$ と $succ_{SL}(sl)$ を用いて，開始スロット $SL_{\mathbf{S}}$ と終了スロット $SL_{\mathbf{E}}$ を定義する．$SL_{\mathbf{S}}$ と $SL_{\mathbf{E}}$ は次の 2 つの集合となる．

- $SL_{\mathbf{S}} =_{def} \{sl \mid pred_{SL}(sl) = \emptyset\}$.
- $SL_{\mathbf{E}} =_{def} \{sl \mid succ_{SL}(sl) = \emptyset\}$.

図 10 に対するスロットの一部を図 14 に示す．図 14 において，点線の矩形は CFG 節を表し，点線矢印は CFG 辺を表す．実線矢印はスロットの隣接関係を表す．ただし，それ以外の隣接スロットは，隣接 CFG 節上の同じ変数名で表現し，実線矢印は省略してある．

このスロット上のデータフロー解析は，次の 3 つのステップからなる．

(1) 絶対無効変数の計算
(2) コード降下の計算
(3) 挿入点の決定

(1) の絶対無効変数 $DEAD$ の計算に対するデータフロー方程式を**図 11** に示す．$DEAD$ の初期値はすべて $true$ であり，その値が $false$ に変更された場合，その影響は CFG 上を後向きに伝播する．重要文から，SV_{op} の辺だけを用いて到達できる文は，重要文と同じ CFG 節で使用されていることを $USED$ によって伝播する．また，スロットを後向きにたどる際に，$DEAD = false$ が，異なった EVG 節を持つスロットに伝播したとき，その情報は，やはり $USED$ を用いて伝播される．この $USED$ による $DEAD = false$ の伝播によって，各文の関係する重要文に使用される範囲が決定し，無効な範囲はその他の部分となる．

(2) のコード降下 $SUNK$ の計算に対するデータフロー方程式を**図 12** に示す．$SUNK$ の初期値は，開始スロットと，EVG 上で変形が起こらなかった ϕ-関数

- すべての $DEAD_{(v,n)}$（$USED_{(v,n)}$）を $true$（$false$）に初期化する（指定したものを除く）．

$$USED_{(v,n)} =_{def} \begin{cases} true & if\ lb(op(v))\ is\ relevant \\ & \wedge\ loc(op(v)) = n \\ \displaystyle\sum_{\substack{(v',n') \in succ_{SL}(v,n) \\ \wedge\ v' \neq v}} \neg DEAD(v', n') \vee \sum_{v' \in Parent_{op}(v)} USED_{(v',n)} & otherwise \end{cases}$$

$$DEAD_{(v,n)} =_{def} \neg USED_{(v,n)} \wedge \prod_{(v',n') \in succ_{SL}(v,n)} DEAD_{(v',n')}$$

図 11 絶対無効変数の解析
Fig. 11 Dead variable analysis.

- $COMP(v,n)$ ：v に相当する代入文が n 上に存在する．
- すべての $SUNK_{(v,n)}$（$BLOCKED_{(v,n)}$）を $true$（$false$）に初期化する（指定したものを除く）．

$$BLOCKED_{(v,n)} =_{def}$$

$$\begin{cases} true & if\ lb(op(v))\ is\ relevant \\ & \wedge loc(op(v)) = n \\ \sum_{v' \in Parent_{op}(v)} (\neg DEAD_{(v',n)} \wedge SUNK_{(v',n)} & \\ \quad \wedge \sum_{(v'',n') \in succ_{SL}(v',n)} \neg SUNK_{(v'',n')}) & otherwise \end{cases}$$

$$SUNK_{(v,n)} =_{def}$$

$$\begin{cases} false & if\ (v,n) \in SL_{\mathbf{s}} \vee \\ & lb(op(v)) = \bot \wedge loc(phi(v)) = n \\ COMP_{(v,n)} \vee \prod_{(v',n') \in pred_{SL}(v,n)} (SUNK_{(v',n')} & \\ \quad \wedge \neg DEAD_{(v',n')} \wedge \neg BLOCKED_{(v',n')}) & otherwise \end{cases}$$

図 12 可能な降下の解析
Fig. 12 Possible sinking analysis.

$$INSERT_{(v,n)} =_{def} \neg DEAD_{(v,n)} \wedge SUNK_{(v,n)} \wedge \sum_{(v',n') \in succ_{SL}(v,n)} \neg SUNK_{(v',n')}$$

図 13 挿入点
Fig. 13 Insertion point.

（図 10 において，四角で囲んでいない ϕ-関数）のスロットを除いて，$true$ とする．$false$ の影響は CFG 上を前向きに伝播する．$BLOCKED$ は，コード降下の際，定義が使用を追い越して降下し，プログラムの意味を変えてしまうのを防ぐため，降下をブロックする働きをする．重要文に対しては，$BLOCKED = false$ であり，重要文を移動させない働きもする．(1) に現れる $\neg DEAD$ は，定義側が無効になった際に，ブロックを行わない効果を与える．実際の $SUNK$ の計算は，図 11 中の $\neg DEAD$ によって，絶対無効代入になったところで降下は終了する．

すべてのデータフロー解析が終了した後，**図 13** に示したように，$INSERT$ が $true$ である CFG 節に文を挿入する．これによって，最も降下したプログラム点での，無効でない文の挿入が実現できる．

図 10 に対するデータフロー解析の結果の一部を**図 14** に示す．この図は，$x3$ および $y0$ の定義を CFG 節 46 に降下させることを示している．

4.3 通常形式プログラムへの変換

最適化の後処理として，EVG で表現されているプログラムを通常のプログラム形式に変換しなければならない．これは，CFG 節 n ごとに，$INSERT_{(v,n)} = true$ の EVG 節 v を，深さ優先順序で訪問し，対応する文を生成すればよい．三番地コードを生成するのであれ

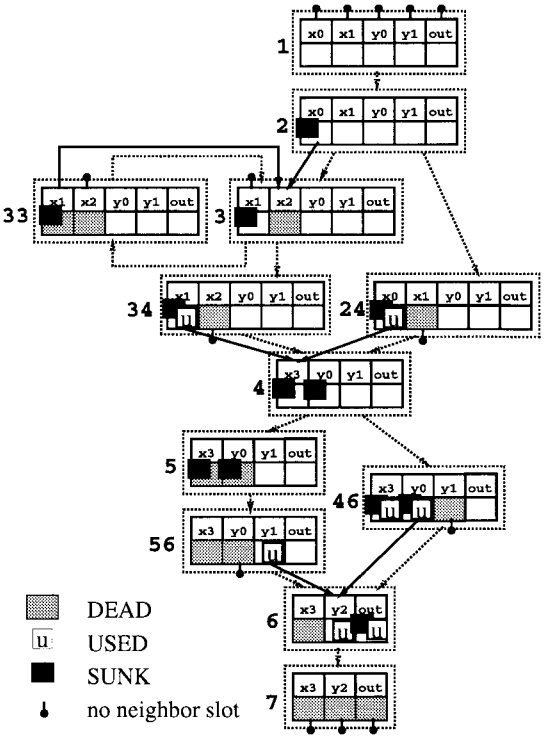


図 14 データフロー解析の結果
Fig. 14 Result of data flow analysis.

ば，各 EVG 節 v に唯一の変数 $id(v)$ が割り当てられているとして，次の文を挿入する．

$id(v)$
:= $id(child_1(op(v)))$ $lb(op(v))$ $id(child_2(op(v)))$

v が ϕ-関数をラベルとする副節を持つ場合には，$loc(phi(v))$ の CFG 節（ϕ-関数固定の CFG 節）の先行節の出口に次の文を挿入する．

$id(v)$:= $id(child_i(phi(v)))$

これらのコピー代入の多くは，後に**レジスタ彩色アルゴリズム**（register coloring）[5]で使われる**変数融合**（variables merge）[5]によって除かれることが期待できる．

5. 計 算 量

本手法の計算量を次のパラメータを用いて示す．

- I：通常形式プログラムの命令数
- I_{SSA}：SSA 形式プログラムの命令数
- I_ϕ：SSA 形式プログラム中の ϕ-関数の数
- V_{SSA}：SSA 形式プログラム中の変数の数
- N：基本ブロック数

3 章で示したように，EVG の変形にかかるコストは I_ϕ に線形である．変形にともなって，新たに ϕ-関数が挿入されるが，これは単なる変数名の変更にすぎないので，通常形式で考えると，プログラムサイズは変化しない．以降の議論に影響がないことから，EVG の変形後の命令数，ϕ-関数の数をそれぞれ I_{SSA}，I_ϕ とおくことにする．

次に，データフロー解析のコストを考える．スロットのデータが $true$ と $false$ の 2 値しか持たず，1 方向（$true$ から $false$，または $false$ から $true$）にしか変更されないので，各スロットはたかだか 1 回の評価ですむ．したがって，データフロー解析にかかるコストはスロットの数で抑えられる．スロットは各変数ごとに CFG 節上にあるから，その数は $V_{SSA} \cdot N$ である．したがって，全体の計算量は $O(V_{SSA} \cdot N)$ である．

ここで，SSA 形式の変数は，定義ごとに区別されるので，V_{SSA} は I_{SSA} に線形である．また，I_ϕ はループの最大の入れ子の深さに線形である[7]から，N で抑えられる．したがって，I_{SSA} は I に線形なので，全体の計算量は，$O(I \cdot N)$ となる．

この計算量は悲観的なものである．実践的にデータフロー解析に関わるスロット数は一部であり，その中で評価されるスロットの数はさらに小さい．

6. 評 価 結 果

本手法の効果を示すために，従来の部分無効コード除去法と本手法を最適化部に持つ C コンパイラを実装し，評価を行った．このコンパイラは，**中水準中間表現**（Medium-Level Intermediate Representation，以降 MIR と呼ぶ）[14]を生成するフロントエンドプロセッサと，MIR を C プログラムに変換するバックエンドプロセッサからなる．生成された C プログラムは，MIR 命令の実行回数を記録する操作を含んでいる[4),6)]ので，生成されたプログラムをコンパイルして，実行することによって，MIR 操作に対する実際のコストを知ることができる．

評価は，2 つの有名なベンチマークと 4 つの簡単なアルゴリズム[21]を用いて，プログラム中に現れる各関数について評価を行った．**表 1** に実行結果を示す．表の各列の意味は次のとおりである．

non-PDE： 最適化を施さなかった場合の命令実行回数

PDE： 従来の部分無効コード除去法を適用した場合の命令実行回数

new： 本手法を適用した場合の命令実行回数

PDE/non-PDE： non-PDE に対する PDE の割合

new/PDE： PDE に対する new の割合

new/non-PDE： non-PDE に対する new の割合

new/PDE の結果から，従来の部分無効コード比べて，性能を低下させることはなく，最大で実行時間は 0.933 の高速化が得られているのが分かる．non-PDE と比べると，最大で 0.825 の高速化が得られた．

7. 関 連 研 究

部分無効コードの除去の重要性は，Feigen らによって指摘された[9]．Feigen らの手法では，代入文 s が実行されるときのすべてのパスに対して，s の代入先が使用されるパスの割合が高いプログラム点に s を移動させ，文の移動がブロックされるときは，制御フローグラフの分岐構造ごと移動させる．このため，プログラム構造を変更してしまう可能性がある．また，文の移動先が 1 つのプログラム点なので，除去できない部分無効コードが存在し，ループ外コード移動の効果は持たない．これに対して，Knoop らの手法[13]は，制御構造を変形しない範囲で，ループ外コード移動も含めて，Feigen らの手法を一般化している．Knoop らの手法は，データフロー解析を基に，繰返し適用を必要とするので，最悪の場合には，プログラムサイズの 5 乗の計算量，合理的な過程を用いても 3 乗の計算量

表 1 実験結果
Table 1 Experimental results.

programs	functions	non-PDE	PDE	new	PDE/non-PDE	new/PDE	new/non-PDE
linpack	main	7190	6661	6256	0.926	0.939	0.870
	print_time	368	368	360	1.000	0.978	0.978
	epslon	23	20	19	0.869	0.950	0.826
	dmxpy	147774	146361	146361	0.990	1.000	0.990
	dgesl	213252	192504	187330	0.902	0.973	0.878
	dgefa	6685068	6286202	6154928	0.940	0.979	0.920
	daxpy	184404012	181715066	181715066	0.985	1.000	0.985
	dscal	1791452	1751698	1751698	0.977	1.000	0.977
	idamax	2071602	1932138	1803438	0.932	0.933	0.870
	matgen	12219498	10853028	10853028	0.888	1.000	0.888
whetstone	main	1419378	1371296	1341777	0.966	0.978	0.945
	p0	190960	190960	190960	1.000	1.000	1.000
	p3	206770	188790	188790	0.913	1.000	0.913
n-queen	main	280	233	231	0.832	0.991	0.825
	backtrack	483671	450759	447875	0.931	0.993	0.925
quicksort	main	90403	84147	84146	0.930	0.999	0.930
hilbert curves	main	219	202	191	0.922	0.945	0.872
	C	2076	2028	1916	0.976	0.944	0.922
	B	3172	3091	2902	0.974	0.938	0.914
	A	4612	4486	4192	0.972	0.934	0.908
	D	3172	3091	2902	0.974	0.938	0.914
shortest path	main	2035	1942	1939	0.954	0.998	0.952
	init	896	812	812	0.906	1.000	0.906

と見積もられている．

制御構造を変更することを前提に，Feigen らの手法を一般化したものに，Bodik らの手法がある．これは，スライシングの技術を使って，効果的な部分無効コード除去を実現している．しかし，その計算量はプログラムサイズに対して指数的である．

本手法は，Knoop らと同様に，制御構造を変更しないことを前提にしている．計算量は，プログラムサイズの 2 乗のオーダであり，過去の研究よりも低いコストの部分無効コード除去を実現した．また，変数名の付替えによって，コード移動がブロックされないようにする方法の提案は，本手法が初めてである．変数名の付替えは，他の手法と併用することも可能である．

以上に述べた以外にも，冗長コードの除去の際，過剰なコード巻上げを抑えるために行うコード降下によって，副次効果として得られる部分無効コード除去がある[4),6)]．しかし，これらの部分無効コード除去は，ϕ-関数に出会うまで適用されるだけで，コード降下の範囲が制限されたものとなっている．

コード最適化における EVG の利用には，部分冗長除去（partial redundancy elimination）を高速にかつ効果的に行う研究として，滝本らの手法がある[20)]．これは，EVG 上に依存情報をフローさせることによって，効率的なコード移動を実現する点で，本手法と共通している．しかし，本手法では，コード移動だけでなく，部分無効コード除去において大きなコストを占める無効代入の解析を同時に行うことを可能にした点で異なる．

8. 結　　論

本稿では，EVG に基づく効果的な部分無効コード除去法を提案した．EVG は，計算が移動する過程の構造を適切に表現するので，複数の先行節から降下してくる代入文の右辺が，同じオペランドを持たなかったとしても，プログラムの意味を変えることなく，計算を合体させ，さらに降下させることができる．この性質を利用して，従来法よりも広い範囲での部分無効コード除去法を実現した．

EVG は，変数の定義–使用関係を持つので，文が無効になったという情報と降下したという情報を，使用側の文から定義側の文に伝播させることができる．これによって，従来繰り返し適用しなければ除去できなかった無効コードを一度に除去できるようになった．

本手法の効果を評価するために，本手法による最適化を行う C コンパイラを試作し，実験を行った．結果として，従来法に比べ，さらに効果が得られることを示した．

参 考 文 献

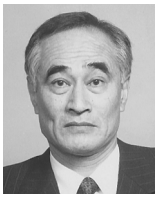
1) Aho, A.V., Sethi, R. and Ullman, J.D.:

Compilers: Principles, Techniques, and Tools, Addison Wesley (1986).
2) Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting Equality of Variables in Programs, *POPL*, pp.1–11, ACM (1988).
3) Appel, A.W.: *Modern Compiler Implementation in ML*, Cambridge University Press (1998).
4) Briggs, P. and Cooper, K.D.: Effective Partial Redundancy Elimination, *PLDI*, pp.159–170, ACM (1994).
5) Chow, F.C. and Hennessy, J.L.: The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.4, pp.501–536 (1990).
6) Click, C.: Global Code Motion Global Value Numbering, *PLDI*, pp.246–257, ACM (1995).
7) Cytron, R., Ferrante, J., Rosen, B.K. and Wegman, M.N.: Efficiently Computing Static Single Assignment Form and Control Dependence 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, *PLDI*, pp.212–223, ACM (1992).
9) Feigen, L., Klappholz, D., Casazza, R. and Xue, X.: The Revival Transformation, *POPL*, pp.421–434, ACM (1994).
10) 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).
11) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *PLDI*, pp.224–234, ACM (1992).
12) Knoop, J., Rüthing, O. and Steffen, B.: Optimal Code Motion: Theory and Practice, *ACM Trans. Prog. Lang. Syst.*, Vol.16, No.4, pp.1117–1155 (1994).
13) Knoop, J., Rüthing, O. and Steffen, B.: Partial Dead Code Elmination, *PLDI*, pp.147–158, ACM (1994).
14) Muchnick, S.S.: *Advanced Compiler Design Implementation*, Morgan Kaufmann (1997).
15) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global Value Numbers and Redundant Computations, *POPL*, pp.12–27, ACM (1988).
16) Sreedhar, V.C. and Gao, G.R.: A Linear Time Algorithm for Placing ϕ-Nodes, *POPL*, pp.62–73, ACM (1995).
17) Steffen, B., Knoop, J. and O. Rüthing: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *Proc. 3rd ESOP*, Copenhagen, Denmark, pp.389–405, Springer-Verlag (1990).
18) Takimoto, M. and Harada, K.: Partial Dead Code Elimination Using Extend Value Graph, *Proc.Int.Static Analysis Symposium* (*SAS '99*), LNCS, Vol.1694, Venice, pp.179–193, Springer-Verlag (1999).
19) 滝本宗宏，原田賢一：ϕ-関数移動による効率的な部分冗長計算除去，プログラミング—言語 · 基礎 · 実線研究会報告，Vol.20, No.3, pp.21–30 (1995).
20) 滝本宗宏，原田賢一：拡張値グラフに基づく効果的な部分冗長除去法，情報処理学会論文誌，Vol.38, No.11, pp.2237–2250 (1997).
21) 石畑　清：アルゴリズムとデータ構造，岩波書店 (1989).

(平成 10 年 12 月 21 日受付)
(平成 11 年 10 月 7 日採録)

滝本　宗宏（学生会員）

1967 年生．1994 年慶応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，プログラミング言語およびその処理系に興味を持つ．



原田　賢一（正会員）

1940 年生．1966 年慶応義塾大学大学院工学研究科管理工学専攻修士課程修了．1967 年同大学工学部助手．1970〜1989 年同大学情報科学研究所助手，専任講師，助教授，教授．1989 年 4 月より同大学理工学部計測工学科教授．同大学大学院計算機科学専攻教授兼任．この間，1973〜1975 年米国メリーランド大学訪問研究員．工学博士．ソフトウェア工学，プログラミング言語およびその処理系の研究に従事．ACM，IEEE，ソフトウェア科学会各会員．