拡張値グラフに基づく効果的な部分冗長除去法

滝本宗宏[†] 原田賢一[†]

プログラミング言語処理系が行うコード最適化の中で，部分冗長除去の手法は，共通部分式の除去ばかりでなく，ループ不変コードのループ外移動をも統合的に行える優れたコード最適化である．しかし，通常の部分冗長除法は，構文上での同じ（構文等価）式を処理の対象にしているので，構文上は異なっていても，同じ値を計算する（意味等価）式を除去の対象にはできなかった．本論文では，意味等価な式の解析を行い，その後，意味等価な式に対して部分冗長除去を適用するという，効果的で効率の良い部分冗長計算の除去法を提案する．本手法では，拡張値グラフと呼ぶデータ構造を新たに導入する．拡張値グラフは，与えられたプログラム中の式を各計算点に移動したときの大域的な構造と等価関係を統一的に表現する．拡張値グラフの上では，計算順序を考慮することなく，意味等価な式を見つけることができる．部分冗長除去のためのデータフロー解析は，式どうしの依存関係を保ちながら行うことができるので，アルゴリズムを簡素化し，高速化することができる．さらに，拡張値グラフ上での定数畳込みによって，従来，原始プログラムに現れる式だけを対象にしていた定数畳込みを，新たに畳込み可能な式を算出する手法に拡張することができる．

Effective Partial Redundancy Elimination Based on Extended Value Graph

MUNEHIRO TAKIMOTO[†] and KENICHI HARADA[†]

The eliminating redundant computation is one of the powerful code optimization techniques used by optimizing compilers. It enables not only to remove common subexpressions but also to move loop-invariant out of loops. This paper proposes an efficient and effective algorithm for elimination of redundancy defined by semantic equivalency between expressions in a form of SSA (Static Single Assignment), while many algorithms proposed so far have been based on syntactic equivalency in lexical form. To detect semantic-equivalent expressions, we introduce an enhanced data structure, called an Extended Value Graph (EVG), whose node consists of a pair of nodes of the value graph proposed by B. Alpern, *et al.* An EVG represents semantic-equivalent relationships between expressions at each point on control flow graph, when an expression is hoisted from the original location. Since the evaluation order and equivalent relationships of each expression in a program are reflected on the EVG, we can efficiently find proper redundant expressions by dataflow analysis for EVG cooperating with control flow graph. Taking advantage of features of EVG, we also propose a new type of constant folding incorporated into partial redundancy elimination, which finds such expressions that produce a constant value if they are moved to backward basic blocks.

1. はじめに

冗長計算除去の手法は，コンパイラのコード最適化において強力な手法であり，以前から多くの研究がなされてきた．図1(a)に示す**制御フローグラフ**（control flow graph，以降**CFG**と呼ぶ）において，下側の式 $a+b$ のように，プログラムの開始点から式に到達するすべての経路上に同じ計算式が存在するとき，その式は**冗長**（redundant）であるという．このとき，冗長な式を除去することによってプログラムの意味を変えずに実行時の効率を上げることができる（図1(b)）．これを**共通部分式の除去**という．図**2**(a)の場合，左上の基本ブロックからの経路上では，$a+b$ が2回実行される．しかし，下側の式を除去してしまうと，右上からの経路上に $a+b$ の式がなくなってしまうので，下側の $a+b$ は冗長ではない．この場合には，右上の基本ブロックに $a+b$ を挿入することによって，下側の $a+b$ は冗長になり，除去可能になる（図2(b)）．このように上方に式を挿入することによって，冗長になる式は**部分冗長**（partial redundant）であるという．部分冗長な式は，直前に式を挿入することによって，

† 慶應義塾大学理工学部
Faculty of Science and Engineering, Keio University

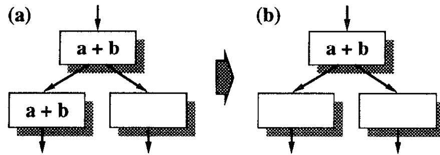


図1 冗長と冗長除去
Fig. 1 Redundancy and its elimination.

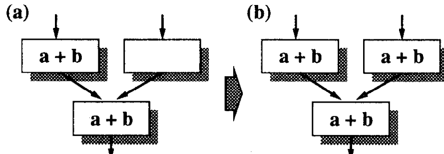


図2 部分冗長と冗長への変形
Fig. 2 Partial redundancy and transformation to redundancy.

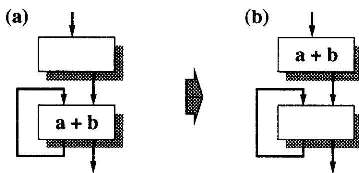


図3 ループ不変コード移動
Fig. 3 Loop-invariant code motion.

冗長なものに変えることができる．新たな式の挿入点と冗長な式の決定は，部分冗長の考えを基にしたデータフロー解析によって実現することができる．データフロー解析を用いて部分冗長を取り除く手法を**部分冗長除去**（partial redundancy elimination，以降**PRE**と呼ぶ）という[7),9),10),12)]．

PRE は共通部分式の除去に加えて**ループ不変コード移動**（loop-invariant code motion）をも統一的に扱うことができる．**図3**(a) で，ループの直前の基本ブロックに $a+b$ を挿入することによって，ループ内部の $a+b$ は冗長となる．すなわち，ループ内部の $a+b$ は部分冗長であり，PRE によってループ不変コード移動が実現できる．

上方に式を挿入し，冗長な式を除去するという操作は，**コード巻上げ**（code hoisting）を行うことと等しい．以降，挿入とそれにともなう除去をコード巻上げ，または単に巻上げと呼ぶ．

上述のように PRE は強力な手法であるが，従来は，字面が同じ式どうしの間で，それらのオペランドへの代入が存在しないという関係（以降，このような関係を**構文等価**と呼ぶ）を満たすものだけを冗長な式として扱ってきた．すなわち，構文上は異なっても，同じ値を計算する式（以降，このような式の関係を**意味等価**と呼ぶ）は，これまで部分冗長な式として扱われてこなかった．

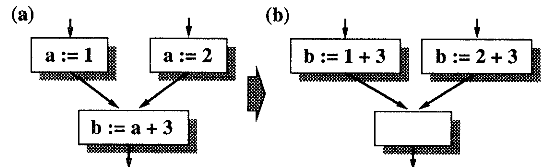


図4 PRE では得られない定数畳込みの効果
Fig. 4 Effective constant folding suppressed by PRE.

また，これまでの PRE では，変数の生存期間を短くするために，無用なコード巻上げを行わないようにしている．これは，無用なコード巻上げを行うと，式の値を保持しておくために一時変数の導入が必要となり，その変数の生存期間が長くなるので，レジスタからのスピルを頻繁に引き起こす可能性があるからである．しかし，その場合でも，**図4** の例に示すように有効な場合がある．図4(a) の下側の基本ブロックでは，$a+3$ を定数に畳み込むことはできないが，この式を巻き上げることによって，図4(b) に示すように変形でき，畳込みが可能になる．図4(a) の式 $a+3$ は，従来の PRE では冗長な式とはならないので，コード巻上げの対象にはならなかった．そのために，式の畳込みができなかった．

本稿では，**拡張値グラフ**（extended value graph，以降 **EVG** と呼ぶ）というデータ構造を導入することによって，意味等価な式を対象に定数畳込みを含む PRE の手法を提案する．

本手法の特徴は次のとおりである．関連研究については7章で述べる．

（1） EVG は，グラフの形で式の依存構造を表現するので，その依存構造を用いて，意味等価な式を発見することができる．この過程では，同時に，定数伝播の効果も含めた定数畳込みを行うことができる．

（2） EVG が表す式の依存構造は，単に原始プログラム中での位置における構造ばかりでなく，巻上げを行った際の各計算点での構造を統合的に表現する．すなわち，各計算点における式の依存構造が明らかになっているので，式をある計算点に移動させた際に意味等価となる式をグループ化しておくことができる．これによって，等価な式の発見と，計算式の挿入点の決定とを分離することができ，等価な式は，計算順序を考慮せずに発見できるようになる．この性質は，アルゴリズム全体の簡素化に貢献する．

(3) EVGで表現した各計算点の式構造の中で，畳込み可能なものは，畳み込んでおくことができる．これは，式の実際の挿入点を決める以前で，畳込み可能な候補を見つけだしておくことを意味する．このEVG上での畳込み情報をPREのデータフロー解析に組み込むことによって，コード移動と同時に新たに定数畳込み可能な式を生成することができる．
(4) 挿入点の決定は，等価な式の発見における，式の処理順序（式の依存の深い順）に影響を受けないので，任意のCFGに対して適用が可能である．
(5) プログラム中の計算式の数を C，基本ブロックの数を N とすると，本手法の計算量は，悲観的に見ても，$O(CN)$ で抑えられる．

本手法による部分冗長除去の過程は次のステップからなる．

(1) EVGの作成
(2) EVGの変形
(3) EVGの等価な節の縮約
(4) EVGをデータフロー解析に反映させるための作業用グラフ（値フローグラフと呼ぶ）の作成
(5) 値フローグラフ上でのデータフロー解析による部分冗長除去
(6) 通常のコードへの変換

図5(a)のプログラムに対して，本手法を適用して得られる結果を，図5(b)に示す．本稿では，以降，図5(a)を共通の例として使用する．

本稿では，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．3章で，本手法の基礎となるデータ構造EVGの定義と作成法を述べる．

4章では，EVGに基づく意味等価な式の発見法を示す．次に，5章において，EVGの情報をデータフロー解析に反映させるための作業用グラフである**値フローグラフ**（value flow graph，以降**VFG**と呼ぶ）の定義と作成法を述べる．さらに，VFGに適用するPREのデータフロー方程式を示し，このデータフロー方程式によって，VFGとEVGがどのように統合され，効率の良い解析を行うことができるかについて述べる．

本手法の効果については，実験の結果を6章に示す．最後に結論を述べる．

2. 入力プログラムの表現

処理の内容を簡潔に述べるために，原始プログラムは，**静的単一代入形式**（Static Single Assignment Form，以降**SSA**形式と呼ぶ）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[5),15)]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（use）は，唯一の定義（definition）を持つ．すなわち，1つの変数は1度しか代入されない．
(2) 異なる制御フロー辺から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（本手法では，ϕ-関数を存在する基本ブロックによって区別する）．

プログラム例（図5(a)）に対するSSA形式の例を**図6**に示す．同じ名前で表現されている変数は，代入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．

本稿では，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つの基本ブロックに入ってくる辺は2つまでとする．

また，**図7**(a)に示すように，2つ以上の後続節を

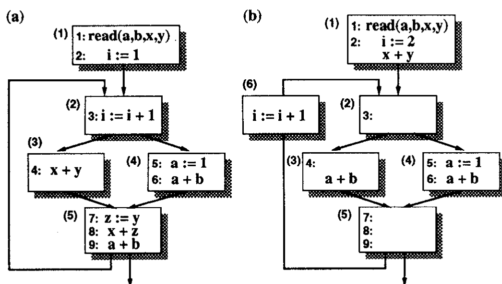


図5 本手法による効果
Fig. 5 Result of our approach.

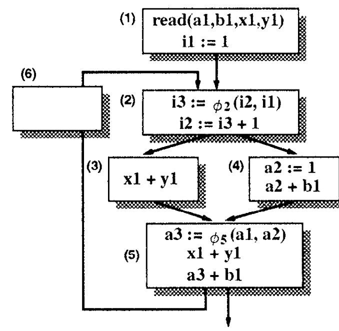


図6 SSA形式
Fig. 6 SSA form.

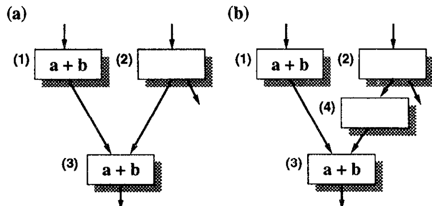


図 7　クリティカル辺の除去
Fig. 7　Critical edge and its elimination.

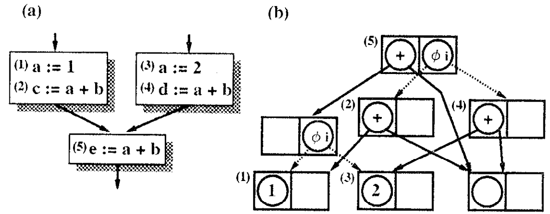


図 8　拡張値グラフ（EVG）
Fig. 8　Extended value graph.

持つ基本ブロック (2) から 2 つ以上の先行節を持つ基本ブロック (3) への辺（**クリティカル辺**（critical edge）[9),10)] と呼ぶ）は取り除かれているものとする．理由は，クリティカル辺がある場合には，有効な巻上げが阻止されるからである．たとえば，図 7 (a) において基本ブロック (3) の式 $a+b$ の巻上げは，ブロック (2) によって阻止されるので，この式はこれ以上，上方に移動することができない．クリティカル辺に対しては，図 7 (b) のように，新しい基本ブロックを挿入することによって冗長な式を除くことができる．図 5 (a) の場合，図 6 のように基本ブロック (6) が挿入される．

3. 拡張値グラフ（EVG）

EVG は，式の依存関係と等価関係を統合的に表すために Alpern，Wegman，Zadeck らによって提唱された**値グラフ**（Value Graph，以降 **VG** と呼ぶ）[2)] を拡張したプログラム表現である．**図 8** に例を示す．図 8 (a) の各式に対して，その値を表す図 8 (b) のグラフ節には，対応する式の番号が示してある．

EVG の 1 つの節は，2 つの副節からなる．左側の副節からの辺（実線矢印）は，式の依存関係を表す．右側の副節からの辺（点線矢印）は，制御フローに依存して等価になりうる（以降，**条件等価**と呼ぶ）節を表す．制御フローに依存せずに等価になる（以降，**無条件等価**と呼ぶ）式は，1 つの EVG 節で表現する．

この章では，まず EVG の定義を与え，次にその作成法を示す．EVG の作成は次の 3 つのステップからなる．

(1)　値グラフから初期 EVG の作成
(2)　初期 EVG の変形による変形 EVG の作成
(3)　変形 EVG の等価部分の縮約

(3) については，次の 4 章において，等価な式の発見とともに述べる．

3.1 EVG の定義

VG は，SSA 形式のプログラムに対して，演算子，関数，定数を節とし，それらの節を，定義と使用の関係によって，有向辺（使用から定義へ）で結んだ依存グラフである．ただし，コピー代入（ある変数の値を単に別の変数に代入すること）☆を介した間接的な使用は，もとの定義に相当する節と直接辺で結んで表す．

$\mathbf{C}$ を定数集合，$\mathbf{Op}$ を演算子集合，$\mathbf{\Phi}$ を ϕ-関数集合とし，$OP = \mathbf{C} \cup \mathbf{Op} \cup \mathbf{\Phi}$ とする．このとき，VG は三つ組 (N_{VG}, E_{VG}, L_{VG}) として定義される．ここで，

(1)　(N_{VG}, E_{VG}) は，節の集合 N_{VG} と辺の集合 $E_{VG} \subseteq N_{VG} \times N_{VG}$ による有向グラフである．
(2)　L_{VG} は $N_{VG} \rightarrow OP$ のラベル付け関数である．

図 6 における変数 $i2$ の値を表現する VG を**図 9** に示す．対応する変数名を節の横に記す．

EVG は，四つ組 $(N_{EVG}, E_{EVG}, \Pi^L_{EVG}, \Pi^R_{EVG})$ で表され，VG の記法を用いて次のように定義できる．

(1)　(N_{EVG}, E_{EVG}) は，節の集合 N_{EVG} と辺の集合 $E_{EVG} \subseteq N_{VG} \times N_{EVG}$ による有向グラフである．
(2)　N_{EVG} は組 $\langle N_{VG}, N_{VG} \rangle$ である．
(3)　Π^L_{EVG} は EVG 節の左側要素を抽出する関数である．すなわち，
$\langle n \in N_{VG}, n' \in N_{VG} \rangle \in N_{EVG} \rightarrow n$
(4)　Π^R_{EVG} は EVG 節の右側要素を抽出する関数である．すなわち，
$\langle n \in N_{VG}, n' \in N_{VG} \rangle \in N_{EVG} \rightarrow n'$
(5)　ある節 $x \in N_{EVG}$ は，次の条件を満たす．
$L_{VG}(\Pi^L(x)) \in \mathbf{C} \cup \mathbf{Op}$，$L_{VG}(\Pi^R(x)) \in \mathbf{\Phi}$

EVG 節の左側の副節は演算子または定数の節であり，右側の副節は ϕ-関数の節である．EVG 中の各有向辺は，左側の副節から出る辺と右側の副節から出る辺とで，次のように異なる意味を持つ．

左側の副節からの辺：式の依存関係（VG の辺に相当）
右側の副節からの辺：条件等価の関係（図 6 の式 $a2+b1$ と $a3+b1$）

3.2 初期 EVG の作成

初期 EVG は，次の手順で VG から作成する．

☆ または，自明な代入（trivial assignment）という．

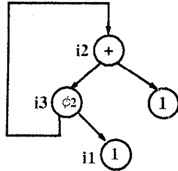


図9　図6における $i2$ の値を表現する値グラフ（VG）
Fig. 9　Value graph representing value of $i2$.

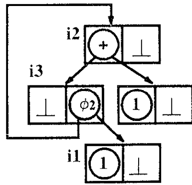


図10　$i2$ の値を表現する初期EVG
Fig. 10　Initial value graph representing $i2$.

(1)　各VG節に対応するEVG節を生成する．

(2)　VG節 n に対応するEVG節を x とすると，x を次のように設定する．

$$\begin{cases} L_{VG}(n) \in \mathbf{\Phi} \text{ のとき} & \Pi^L(x) := \bot \\ & \Pi^R(x) := n \\ L_{VG}(n) \in \mathbf{C} \cup \mathbf{Op} \text{ のとき} & \Pi^L(x) := n \\ & \Pi^R(x) := \bot \end{cases}$$

ここで，代入 “$:= n$” は，節 n が辺で結ばれた構造を保持したままで，節 n がコピーされることを意味する．また，$\bot$ は未定義であることを意味する．

(3)　VG節 n から出ている辺の先をVG節 n' とし，n' を含むEVG節を x とすると，辺 (n, n') を辺 (n, x) で置き換える．

図9のVGに対する初期EVGを図**10**に示す．ここで，VGの節と初期EVGの節は1対1に対応する．初期EVGの節を，特に**初期節**と呼ぶことにする．

3.3　変形EVGの作成

EVGがVGと異なる点は，VGのように，計算式がもともと存在した場所での式の構造を表現するだけでなく，計算式を移動したときの各計算点での構造を統一的に表現する点である．そのために，SSA形式における ϕ-関数を跨いだコード巻上げを考え，初期EVGに対して式の移動にともなう計算式の構造上の変形を行う．この変形を**条件等価変形**と呼ぶ．以下では，EVGに対する条件等価変形の例を示したあと，EVGに対する一般的な条件等価変形を示す．最後に変形のためのアルゴリズムを示す．

図**11**において，(a)の式 $x3+y1$ を直前の2つの基本ブロックに巻き上げようとすると，$x3$ への代入

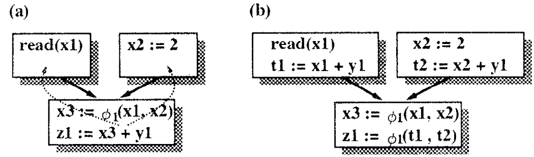


図11　SSA形式におけるコード巻上げ
Fig. 11　Code hoisting on SSA form.

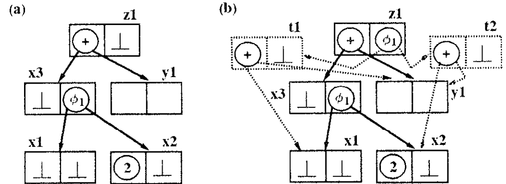


図12　図11に対するEVGの変形
Fig. 12　Transformation of EVG for Fig. 11.

を行っている ϕ-関数を跨ぐことになる．このときには，移動先の節に対応する ϕ-関数中の引数名を用いて，その式のオペランドを置き換えれば，プログラムの意味を変えずに巻上げを行うことができる．$x3+y1$ のオペランド $x3$ をそれぞれ $x1$ と $x2$ に付け換え，(b)のように変形することができる．$z1$ に代入される式 $x3+y1$（図11(a)）と $\phi(t1, t2)$（図11(b)）は同じ値を表している．

上述のSSA形式でのプログラム変形は，このあとに示すようにEVG上で直接行うことができる．EVGにおける条件等価変形の例を図**12**に示す（各節には値を保持する変数の名前を付加してある）．

この変形における特徴は，同じ値を表現する $x3+y1$ と $\phi(t1, t2)$ が1つのEVG節中の副節（VG節）として表現され，2つの計算式が意味等価であることを構造的に表現する点である．副節を導入することによって，1つの値は演算子をラベルに持つ節と ϕ-関数をラベルに持つ節の2つの形式で表現される．図12の(a)から(b)のように，条件等価変形を行ったあとのEVGを**変形EVG**と呼ぶ．

次に，一般的なEVGに対する条件等価変形の方法を示す．以下，$x, x', x'' \in N_{EVG}$，$(\Pi^L(x), x'), (\Pi^L(x), x'') \in E_{EVG}$，$\Pi^L(x) \neq \bot$，$\Pi^R(x) = \bot$ とする．このときEVG節 x に対する条件等価変形は，次の3通りの場合に分けられる．

(1)　**$\Pi^R(x') \neq \bot$ かつ $\Pi^R(x'') \neq \bot$ であり，$L_{VG}(\Pi^R(x')) = L_{VG}(\Pi^R(x''))$ の場合（図13(a)）**

ラベル $L_{VG}(\Pi^R(x'))$ を持つ副節を $\Pi^R(x)$ とする．そして，図13(b)の点線の箱で示すように2つのEVG節 $\langle n, \bot \rangle$（$L_{VG}(n) = L_{VG}(\Pi^L(x))$

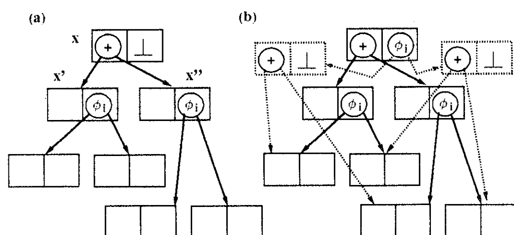


図 13　同じ ϕ-関数に依存する場合
Fig. 13　Node depeinding on two same ϕ-functions.

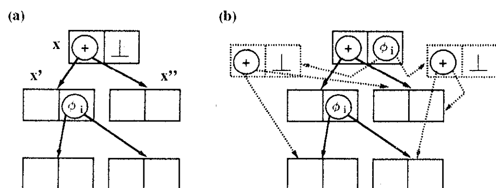


図 14　片方の辺だけが ϕ-関数に依存している場合
Fig. 14　Node depending on only one ϕ-function.

）を生成して，図 13 (b) の点線の辺で示すように辺でつなぐ．

(2) **$\Pi^R(x') \neq \bot$ かつ $\Pi^R(x'') = \bot$ の場合**（**図 14** (a)）
ラベル $L_{VG}(\Pi^R(x'))$ を持つ副節を $\Pi^R(x)$ とする．そして，図 14 (b) の点線の箱で示すように 2 つの EVG 節 $\langle n, \bot\rangle$ ($L_{VG}(n) = L_{VG}(\Pi^L(x))$) を生成して，図 14 (b) の点線の辺で示すように辺でつなぐ．x' と x'' が入れ換わっている場合も同様である．

(3) **$\Pi^R(x') \neq \bot$ かつ $\Pi^R(x'') \neq \bot$ であり，$L_{VG}(\Pi^R(x')) \neq L_{VG}(\Pi^R(x''))$ の場合**
$L_{VG}(\Pi^R(x'))$ と $L_{VG}(\Pi^R(x''))$ の ϕ-関数が存在する基本ブロックのうち，制御フロー上で，x の表す式が存在する基本ブロックに近いもの（巻上げを行った際に先に出会う方）を用いて，(2) と同様の変形を行う．

各条件等価変形は，1 つの ϕ-関数によって引き起こされることが分かる．$(\Pi^L(x), x') \in E_{EVG}$ かつ $\Pi^R(x') \neq \bot$ である EVG 節 x, x' が存在する場合，上述の変形が行われると，$L_{VG}(\Pi^R(x)) = L_{VG}(\Pi^R(x'))$ となる．このとき，x は x' から ϕ-関数 $L_{VG}(\Pi^R(x'))$ を**継承**したといい，x は ϕ-関数 $L_{VG}(\Pi^R(x'))$ によって**変形された**という．変形によって生成されたEVG節を初期節に対して**派生節**と呼び，x から**派生**したという．

ϕ-関数を継承するEVG節 x は，x に依存するEVG節に対して同様の条件等価変形を引き起こす可能性がある．この場合には，変形のたびに派生節が生成される．派生節は，派生元をたどっていくと，唯一の初期節にたどり着くという性質を持つ．この性質は，後に述べるように，変形の範囲を限定するのに利用する．

EVG に対して複数回の条件等価変形を行う過程は，制御フロー上でコードをより上方へ移動していくことに相当する．変形 EVG は CFG を後向きにたどりながら，作成することができる．アルゴリズムの詳細は，次に示すとおりである．

EVG の条件等価変形

1 **forall** $v \in N_{EVG}$ **do**
2 　$v.localblock := v$ の表す式が存在する基本ブロック
3 　$v.src := v$, $v.visited := False$
4 　$v.start := v.localblock$, $suppress[v] := \emptyset$
5 $workset := N_{EVG}$
6 **while** $workset \neq \emptyset$ **do**
7 　**let** $x \in workset$
8 　$workset := workset \setminus x$
9 　**call** $visit(x)$
10
11 **where**
12 $visit(v)$ {
13 　**if** $v.visited = True$ **then return**
14 　$v.visited := True$
15 　**if** $\Pi^L(v) \neq \bot \wedge \Pi^L(v)$ *is pinned* **then**
16 　　**return**
17 　**if** $\Pi^R(v) \neq \bot$ **then return**
18 　**forall** $\{x \in N_{EVG} | (\Pi^L(v), x) \in E_{EVG}\}$ **do**
19 　　**call** $visit(x)$
20 　$blockset := v.start$
21 　**while** $blockset \neq \emptyset$ **do**
22 　　**let** $n \in blockset$
23 　　$blockset := blockset \setminus n$
24 　　**if** $n \in suppress[v.src]$ **then continue**
25 　　$suppress[v.src]$ に n を加える．
26 　　$v.ownarea$ に n を加える．
27 　　**if** $(\Pi^L(v), x'), (\Pi^L(v), x'') \in E_{EVG}$ **then**
28 　　　**if** $\Pi^R(x') \neq \bot$
29 　　　　$\wedge \Pi^R(x'') \neq \bot$ **then**
30 　　　　**if** $n = \Pi^R(x').phiblock$
31 　　　　　$\wedge$ $L_{VG}(\Pi^R(x')) = L_{VG}(\Pi^R(x''))$ **then**
32 　　　　　**図 13 の変形を適用**
　　　　　// 変形の際に生成した節を
　　　　　// y_1, y_2 とする．y_1, y_2 には対応
　　　　　// する n の先行節 $pred(n, 1)$,
　　　　　// $pred(n, 2)$ が存在する．
33 　　　　　**for** $i := 1$ **to** 2 **do**
34 　　　　　　$y_i.src := v.src$
35 　　　　　　$y_i.start := pred(n, i)$
36 　　　　　　y_i を $workset$ に加える．
37 　　　　　**continue**
38 　　　　**elseif** $n = \Pi^R(x').phiblock$ **then**
39 　　　　　x' を ϕ として**図 14 の変形を適用**
40 　　　　　**for** $i := 1$ **to** 2 **do**
41 　　　　　　$y_i.src := v.src$
42 　　　　　　$y_i.start := pred(n, i)$
43 　　　　　　y_i を $workset$ に加える．

```
44                continue
45            elseif n = Π^R(x'').phiblock then
46                x'' を φ として図14の変形を適用
47                for i := 1 to 2 do
48                    y_i.src := v.src
49                    y_i.start := pred(n, i)
50                    y_i を workset に加える.
51                continue
52        elseif Π^R(x') ≠ ⊥
53            if n = Π^R(x').phiblock then
54                x' を φ として図14の変形を適用
55                for i := 1 to 2 do
56                    y_i.src := v.src
57                    y_i.start := pred(n, i)
58                    y_i を workset に加える.
59                continue
60        elseif Π^R(x'') ≠ ⊥
61            if n = Π^R(x'').phiblock then
62                x'' を φ として図14の変形を適用
63                for i := 1 to 2 do
64                    y_i.src := v.src,
65                    y_i.start := pred(n, i)
66                    y_i を workset に加える.
67                continue
68    forall p ∈ n の先行節
69        p を blockset に加える.
}
```

関数 *visit* は，各 EVG 節を 2 度訪れないように(13，14 行目)，EVG 節の左側の副節から出る辺を深さ優先巡回する（19 行目）．EVG 節 v の条件等価変形では，v の右側の副節として ϕ-関数をラベルとする副節 v を生成する．この v は，v に依存する節にさらに変形を引き起こす可能性がある．この変形は，深さ優先巡回によって効率良く伝播される．巡回は，次のいずれかの節に出会ったとき終了する（ここで，(1) の条件は (2) の条件に含まれるが，説明を明確にするために分けて記述する.)．

(1) ϕ-関数をラベルとする副節を持つ節（17 行目），または

(2) 基本ブロックに固定（pinned）☆の計算式をラベルとする副節を持つ節（15 行目）

(1) は，EVG の上方に存在する ϕ-関数による条件等価変形を優先することを意味し，1 つの ϕ-関数による変形は帰りがけ順（post-order）に上の節に伝播される（ϕ-関数を上方に継承させていくことを意味する)．

(2) は，節の条件等価変形がその節に対応する式の ϕ-関数を跨いでの巻上げであることから，基本ブロックに固定であり，移動できない計算式は変形の対象にならないことを意味する．

☆ 関数呼出し（ϕ-関数を含む)，ロード・格納命令などメモリを介した値の受け渡しのために，順序を入れ替えるとプログラムの意味が変わってしまう性質を持つ計算式は，最初に存在した基本ブロックからは移動させない．

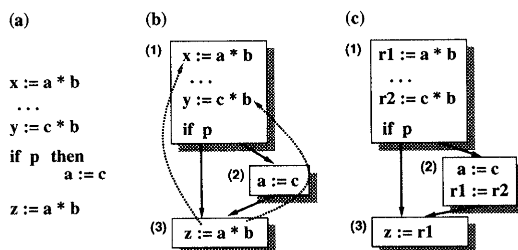


図 15 コピー代入を生成する巻上げ
Fig. 15 Introducing copy assignments.

関数 *visit* の中では，CFG 上を後向きに巡回して巻上げを行う．初期節は原始プログラムに存在する計算式の表現であるから，その計算式が含まれる基本ブロックから巻上げを開始する．派生節は，派生元の節に相当する計算式が ϕ-関数を跨いで先行節に移動した際に初めて，各先行節に対応する構造として生成されるので，変形を引き起したϕ-関数の固定の基本ブロックの先行節から始める（35，42，49，57，65 行目)．これらの巡回開始点は EVG 節 v の *v.start* に記録している（20 行目)．初期節は，もともと存在していた基本ブロックを *x.localblock* に記録しているので（2 行目)，*x.start* := *x.localblock* の初期化を行っている（4 行目)．

巡回中に訪れた CFG の節は *v.ownarea* に記録する（26 行目)．これは，v の計算式の構造によって表現される範囲を記録するためである．この範囲は，変形の出発点である初期節を共有する各派生節とその初期節とで共通部分を持たない．これによって，CFG に循環が存在した場合に起こる条件等価変形の無限適用を防いでいる．この範囲制限は，初期節とその派生節とで共通の *suppress* を参照することによって実現している（24，25 行目)．出発点の初期節が何であるかは，*v.src* によって派生節に伝播される（34，41，48，56，64 行目)．

CFG の巡回中に ϕ-関数に出会った場合には，32，39，46，54，62 行目でそれぞれの条件等価変形を行う．ϕ-関数が固定されている基本ブロックは，右の副節の属性 *phiblock* として保持している．

suppress によって同じ式構造で表現できる範囲を制限することに関しては，条件等価変形の無限適用を防ぐ以外に，次の理由が存在する．

図 15 (a) のプログラム例を考える．実行経路によって異なる値を持つ $z := a * b$ に対して，図 15 (b) の点線矢印は，それぞれの経路を通ったときに等価となる計算式が，上方の同じ基本ブロックに存在すること

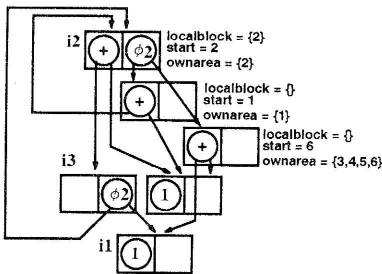


図 16 図 6 における $i2$ の値を表現する変形 EVG
Fig. 16 Transformed EVG representing value of $i2$.

を意味する．$z := a * b$ の右辺を冗長と見なして，右辺を一時変数 $r1$ で置き換えたとすると，z が後に使用される場合を考慮して，基本ブロック (2) にコピー代入が必要である（図 15 (c)）．このような状況が重なると，冗長除去によって逆の効果を生じる可能性がある．本手法における EVG 変形の制限は，このような過多なコピー代入の導入を防ぐ効力も持つ．

図 10 の初期 EVG に対する変形 EVG を図 **16** に示す．$i2$ の初期節とその派生節には，*localblock*, *start*, *ownarea* もあわせて示す．

初期節と派生節が同じ基本ブロックを訪問しないので，このアルゴリズムの計算コストは，悲観的に評価しても，$O(CN)$（計算式の数を C，基本ブロックの数を N とする）である．

4. 等価な式の発見

この章では，EVG に基づいて等価な式を発見する手法を述べる．本手法は，Click の**大域値番号づけ** (global value numbering)[4] と同様に，ハッシュ表に基づくボトムアップ型の**大域一致発見**（global congruence detection)[2),4)] を行う．大域一致発見は，式の依存関係だけで等価な式を見つけるので，もとのプログラムにおける式どうしの関係を無視することができる．したがって，アルゴリズムの簡素化と効率化が可能になる．大域一致発見をボトムアップで行うと，トップダウン型の Alpern, Wegman, Zadeck の方法[2]（以降 **AWZ 法**と呼ぶ）とは異なり，グラフにループが存在するものどうしの一致を見つけることはできない☆．しかし，同時に定数畳込みや，代数的性質を用いたより多くの等価な式の発見が可能になる．

前の節で述べた EVG の条件等価変形は，ある計算点に式を移動させた場合の EVG 表現を，前もって作っておくことに等しい．したがって，変形 EVG にボトムアップ型の大域一致発見を適用することによって，ϕ-関数の節の位置☆☆に関係なく，等価な EVG を発見することが可能になり，定数畳込みが行える可能性も大きくなる．

EVG に基づく等価な式の発見の手順は次のとおりである．

(1) 各 EVG 節を深さ優先順序の逆順で訪れる（同じ節は 2 回訪れない）．
(2) 依存先が定数の場合は，定数畳込みを行う．
(3) ハッシュ表を検査して，等価な式を発見する（ハッシュ表は EVG 節をそのまま登録するものとする）．

畳込みによって，新たに定数となった節から出ている依存辺は取り去るようにする．

EVG 節 x のハッシュ表による等価な式の発見は，$(\Pi^L(x), x')$, $(\Pi^L(x), x'')$, $(\Pi^R(x), y')$, $(\Pi^R(x), y'')$ を E_{EVG} として，次の 3 通りの場合に分けることができる．

(1) $\boldsymbol{\Pi^L(x) \neq \bot \wedge \Pi^R(x) = \bot}$ **の場合**
$(L_{VG}(\Pi^L(x))\ x'\ x'')$ をキーとして，ハッシュ表を検査する．もし，表中に等価節 v が存在すれば，EVG 節 x を v で置き換える．存在しなければ，x をハッシュ表に登録する．

(2) $\boldsymbol{\Pi^L(x) = \bot \wedge \Pi^R(x) \neq \bot}$ **の場合**
$(L_{VG}(\Pi^R(x))\ y'\ y'')$ をキーとして，ハッシュ表を検査する．もし，表中に等価節 v が存在すれば，EVG 節 x を v で置き換える．存在しなければ，x をハッシュ表に登録する．

(3) $\boldsymbol{\Pi^L(x) \neq \bot \wedge \Pi^R(x) \neq \bot}$ **の場合**
$(L_{VG}(\Pi^L(x))\ x'\ x'')$ と $(L_{VG}(\Pi^R(x))\ y'\ y'')$ の両方をキーとして検査し，少なくとも片方の検査で，表中に等価節 v が存在すれば，両方のキーに対して，表中の v を x で置き換える．存在しなければ，x をハッシュ表に登録する．

(1), (2) では EVG 節 x を表中の v で置き換え，(3) では表中の v を x で置き換えているのは，$(L_{VG}(\Pi^L(x))\ x'\ x'')$ と $(L_{VG}(\Pi^R(x))\ y'\ y'')$ の 2 つのキーで検査したときに表中の EVG 節が一致している必要があるからである．

等価な節 x と x' を 1 つにして，表中に x だけを残す場合，*x.ownarea* と *x.localblock* は，それぞれ $x.ownarea := x.ownarea \cup x'.ownarea$,

☆ 本手法は AWZ 法を前処理として実行しておくことができる．

☆☆ 大域一致発見は VG（辺の向きが逆の SSA グラフを使用する場合もある）に適用するので，同じ値を表す 2 つの VG が ϕ-関数の位置で異なった構造になり，その等価性が発見できない場合がある[13]．

$x.localblock := x.localblock \cup x'.localblock$ とする必要がある．

5. 値フローグラフ（VFG）

最後のステップとして，EVG の情報を利用して部分冗長除去を行うためのデータフロー解析の枠組みとして，VFG について述べる．

VFG は，同じ値の流れを制御フローグラフに沿って辺で結んだグラフであり，等価な式を制御フロー上で結び合わせる働きを持つ．本手法で用いる VFG は Steffen, Knoop, Rüthing によって提唱されたもの[16]の変形である．この VFG と EVG 上でデータフロー解析を行うと，原始プログラム中でのコードの場所に縛られることなく，コードの依存関係を保存した解析が実現できる．

この章では，まず，VFG の定義を与え，VFG の作成について述べる．そして，VFG 上でのデータフロー解析を，データフロー方程式を中心に述べる．最後に，EVG 表現で表されていたプログラムを CFG に戻すための方法を述べる．

5.1 値フローグラフの定義

VFG は，CFG に代わって，データフロー解析に枠組みを与える作業用のグラフである．VFG の各節には，対応する唯一の CFG 節が存在する．さらに，VFG 節は対応する唯一の EVG 節を持つ．辺で結ばれた VFG 節どうしは，同じ EVG 節に対応するか，EVG 節の右の副節から出ている辺（条件等価な節をつなぐ）で結ばれた節に対応する．VFG の定義の詳細は次のとおりである．

VFG を $VFG = (N_{VFG}, E_{VFG}, \mathbf{s}_{VFG}, \mathbf{e}_{VFG})$ の組で表す．

(1) N_{VFG} は，VFG の節集合である．

(2) E_{VFG} は，$N_{VFG} \times N_{VFG}$ の組で表される VFG の辺集合である．

(3) $\mathbf{s}_{VFG}$ と $\mathbf{e}_{VFG}$ は，それぞれ N_{VFG} に含まれる，データフロー解析を行うための開始節と終了節である．

(4) $\mathcal{V}$ を $N_{VFG} \rightarrow N_{EVG}$ の関数，$n \in N_{VFG}$，$v \in N_{EVG}$ として，$\mathcal{V}(n) = v$ となる唯一の v が存在する．

(5) $\mathcal{N}$ を $N_{VFG} \rightarrow N_{CFG}$ の関数，$n \in N_{VFG}$，$M \in N_{CFG}$ として，$\mathcal{N}(n) = M$ となる唯一の M が存在する．

(6) $n, n' \in N_{VFG}$ かつ $(n, n') \in E_{VFG}$ とすると，$\mathcal{V}(n) = \mathcal{V}(n')$ または，$\mathcal{V}(n) \in CHILDREN(\Pi^R(\mathcal{V}(n')))$，ここで，EVG 節内の VG 節から，その i 番目の依存先（EVG 節）への変換関数を $child_i$ として，$CHILDREN = \bigcup_i child_i$ である．

(7) $n, n' \in N_{VFG}$ かつ $(n, n') \in E_{VFG}$ とすると，$\mathcal{V}(n) = \emptyset \wedge \mathcal{V}(n') \neq \emptyset$ のとき，$n = \mathbf{s}_{VFG}$ また，$\mathcal{V}(n) \neq \emptyset \wedge \mathcal{V}(n') = \emptyset$ のとき，$n' = \mathbf{e}_{VFG}$ である．

5.2 値フローグラフの作成

VFG は，EVG によって表現される等価関係を使って作成することができる．EVG 節 x において，無条件等価の関係は，$x.ownarea$ に保持している．$x.ownarea$ は，x の EVG 表現のままで，移動できる CFG 上の範囲を表しており，この範囲においては，実行経路に関係なく同じ値を持つことを表す．また，条件等価の関係は，$\Pi^R(x)$，すなわち x の式に対する ϕ-関数表現の形で保持されている．$\Pi^R(x)$ からの EVG 辺は，その辺に対応する方向の先行節から制御が移ってきたときに限って，同じ値を持つことを表す．すなわち，VFG は，$x.ownarea$ の各基本ブロック対応する VFG 節を CFG に沿って結び合わせた VFG の副グラフどうしを $\Pi^R(x)$ の依存情報によってつなぎ合わせた構造を持つ．VFG の作成手順は次のとおりである．

(1) EVG 節 x の $x.ownarea$ 中の各基本ブロックに対応して，VFG 節を作る（図 **17** (a)）．

(2) $\Pi^R(x) \neq \perp$ である EVG 節 x の右側の副節 v から出る辺を用いて，v 固定の基本ブロックに対応する VFG 節と条件等価な先行節に対応する VFG 節との間に辺を作る（図 17 (b)）．

(3) 各 EVG 節 x について，$x.ownarea$ 内の基本ブロックに対応する VFG 節を辺でつなぐ

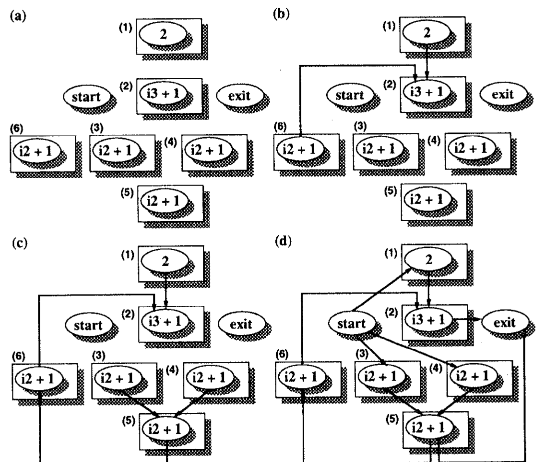


図 **17** 図 6 における $i2$ の値に対する VFG の作成
Fig. 17 Creation of VFG for value of $i2$ in Fig. 6.

（図 17 (c)）．

（4） もし，CFG の節に対応する EVG 節が存在しなければ，開始節（先行節において，EVG 節が存在しないとき），または終了節（後続節において，EVG 節が存在しないとき）と辺でつなぐ（図 17 (d)）．

VFG の作成過程の例を図 17 に示す．楕円と実線矢印は VFG の節と辺を表す．長方形の箱は CFG の基本ブロックであり，辺は省略してある．

5.3 値フローグラフ上のデータフロー解析

最後に，前節で作成した VFG 上で，PRE のデータフロー解析を行う．ここで用いる PRE 法は，Knoop, Rüthing, Steffen らの手法[9),10)]を変形したものである．PRE 法の多くが双方向のデータフロー解析であるのに対し，本手法は，巻上げと遅延（delay）をそれぞれ，後向きと前向きの単方向のデータフロー解析で行う．遅延とは，不要なコード巻上げを防ぐために 1 度巻き上げた式をプログラムの意味を変えない範囲で逆に終了点に向かって移動する操作をいう[9),10)]．双方向の解析を単方向にしたことによって，すべてのコードはまず，プログラム開始点に最も近いところへ巻き上げられるので，巻上げによる効果（定数畳込みなど）を確実に得ることができる．

データフロー解析で用いるデータフロー方程式を**図 18～図 21** に示す．方程式中のすべての述語は，*true* か *false* かのいずれかの値をとる．

各方程式中の述語 $\mathsf{COMP_n}$ は，原始プログラムにおいて，VFG 節 n に対応する基本ブロック $\mathcal{N}(\mathsf{n})$ に EVG 節 $\mathcal{V}(\mathsf{n})$ に対応する式が存在するときだけ *true*，それ以外の場合は *false* となる．

図 18 は，どの経路にも新しい式を導入しないで，巻上げ可能な最大範囲と，冗長になる範囲を求めるための方程式である．$\mathsf{N\text{-}HOISTABLE_n}$, $\mathsf{X\text{-}HOISTABLE_n}$ は，それぞれ後向きのデータフロー解析で決定する述語である．VFG 節 n の入口，出口においてそれぞれ巻上げ可能である場合に *true* になる．また，$\mathsf{N\text{-}REDUNDANT_n}$, $\mathsf{X\text{-}REDUNDANT_n}$ は，前向きのデータフロー解析で決定する述語である．VFG 節 n の入口，出口において，それぞれ式が存在するときに限って冗長になる場合に *true* となる．

$\mathsf{DEFINED_n}$ は，対象としている式のオペランドの定義が基本ブロック $\mathcal{N}(\mathsf{n})$ で使用できることを表す述語である．対象とする式 $\mathcal{V}(\mathsf{n})$ の依存先 $CHILDREN(\Pi^L(\mathcal{V}(\mathsf{n})))$ が，1 つでも基本ブロック $\mathcal{N}(\mathsf{n})$ に対応する VFG 節 w で $\mathsf{N\text{-}HOISTABLE_w} = false$ なら $\mathsf{DEFINED_n} = false$ になる．この $\mathsf{DEFINED_n}$ によって，EVG の構造が VFG に反映され，式の依存による計算順序を壊すことなくすべての式の解析を同時に行うことができる．

図 18 において，巻上げに関するデータフロー解析は，$\mathbf{e}_{VFG}$ と基本ブロックに固定（pinned）のコードが存在する節に限って *false*，他のすべての節を *true* とした初期状態に対する最大解の計算である．冗長に関する解析は，$\mathbf{s}_{VFG}$ だけ *false* で，他のすべての節を *true* とする初期状態における最大解の計算である．

次に，$\mathsf{N\text{-}HOISTABLE_n}$，X-HOISTABLE, N-REDUNDANT，X-REDUNDANT の情報を用いて，式の最大巻上げ点のうちで，冗長にならないもの，すなわち最低限挿入の必要な場所を決定する．この挿入を表す述語が，**図 19** の $\mathsf{EARLIEST_n}$ である．この値は，各 VFG 節についての局所的な計算によって求めることができる．

最後に，巻上げすぎた式を遅延させるためのデータフロー解析（**図 20**）を行う．述語 N-DELAYED, X-DELAYED は，入口，出口のそれぞれにおいて遅延可能な場合 *true* である．また，式の遅延は，巻上げによって得られる効果を損なわないようにして行われる（ループの外に出た式は元に戻さない）．この遅延において，PRE に拡張を加えているのが，$\mathsf{EFFECTVE_n}$ である．図 19 に示すように，$\mathsf{EFFECTVE_n}$ は，巻上げによって定数の畳込みができるようになる効果と，異なる値の式が巻上げによって 1 つになるため，片方を冗長にできるという効果を表現する．この情報は EVG から得られる．$\mathsf{EFFECTVE_n}$ が *true* のとき，巻上げによって得られる効果を損なわないように，遅延は阻止される．

また，$\mathsf{USED_n}$ は，$\mathsf{DEFINED_n}$ と同様に，EVG をデータフロー解析に反映させるためのもので，対象とする式の値が基本ブロック $\mathcal{N}(\mathsf{n})$ において使用されているかどうかを表す．もし，基本ブロック $\mathcal{N}(\mathsf{n})$ に式 e をオペランドとして使用している式が遅延されてこなかったとしたら，e の遅延も *false* となる．ここで，$PARENTS(v)$ は，EVG 節 v に依存する EVG 節集合を表している．

最終的な挿入点の計算は，図 21 の方程式で与えられる．LATEST は巻上げを最も遅延できる場所を表していて，最大で原始プログラム中の式と同じ場所になる．

データフロー解析における VFG 節の巡回は，述語に変化があった VFG 節に隣接する節に限られる．さらに，述語は *true* から *false* に変更されるだけであり，*false* になった後は変化することがない．したがっ

$$
\begin{aligned}
\text{N-HOISTABLE}_{\rm n} &= \text{DEFINED}_{\rm n} \cdot \text{X-HOISTABLE}_{\rm n} \\
\text{X-HOISTABLE}_{\rm n} &= \text{COMP}_{\rm n} + \begin{cases} \mathit{false} & \mathit{if}\ {\rm n} = {\rm e}_{VFG} \\ \prod\limits_{M \in succ(\mathcal{N}({\rm n}))} \sum\limits_{\substack{{\rm m} \in succ({\rm n}) \\ \mathcal{N}({\rm m}) = M}} (\text{N-HOISTABLE}_{\rm m}) & \mathit{otherwise} \end{cases} \\
\text{DEFINED}_{\rm n} &= \begin{cases} \mathit{false} & \mathit{if}\ \mathcal{V}({\rm n})\ \mathit{is\ pinned\ on}\ {\rm n} \\ \prod\limits_{\substack{\mathcal{V}({\rm w}) \in CHILDREN(\Pi^L(\mathcal{V}({\rm n}))) \\ \mathcal{N}({\rm w}) = \mathcal{N}({\rm n})}} \text{N-HOISTABLE}_{\rm w} & \mathit{otherwise} \end{cases} \\
\text{N-REDUNDANT}_{\rm n} &= \begin{cases} \mathit{false} & \mathit{if}\ {\rm n} = {\rm s}_{VFG} \\ \prod\limits_{{\rm m} \in pred({\rm n})} (\text{COMP}_{\rm m} + \text{X-REDUNDANT}_{\rm m}) & \mathit{otherwise} \end{cases} \\
\text{X-REDUNDANT}_{\rm n} &= (\text{COMP}_{\rm n} + \text{N-REDUNDANT}_{\rm n})
\end{aligned}
$$

図 18 巻上げ部のデータフロー方程式

Fig. 18 Dataflow equation for hoisting.

$$
\begin{aligned}
\text{EARLIEST}_{\rm n} &= \text{N-HOISTABLE}_{\rm n} \cdot \prod_{{\rm m} \in pred({\rm n})} \overline{(\text{X-REDUNDANT}_{\rm m} + \text{X-HOISTABLE}_{\rm m})} \\
&\quad + \text{X-HOISTABLE}_{\rm n} \cdot \overline{\text{DEFINED}_{\rm n}} \\
\text{EFFECTIVE}_{\rm n} &= \sum_{M \in \mathrm{succ}(\mathcal{N}({\rm n}))} \Bigg(\sum_{\substack{{\rm m} \in succ({\rm n}) \\ \mathcal{N}({\rm m}) = M}} L_{VG}(\Pi^L(\mathcal{V}(n))) \in \mathbf{C} \wedge L_{VG}(\Pi^L(\mathcal{V}(m))) \notin \mathbf{C} \\
&\quad + \begin{cases} \mathit{true} & \mathit{if\ more\ than\ one}\ \text{X-HOISTABLE}_{\{{\rm m} | \mathcal{N}({\rm m}) = M\}}\ \mathit{are\ true} \\ \mathit{false} & \mathit{otherwise} \end{cases} \Bigg)
\end{aligned}
$$

図 19 効果的な巻上げ

Fig. 19 Effective hoisting.

$$
\begin{aligned}
\text{N-DELAYED}_{\rm n} &= \text{EARLIEST}_{\rm n} \\
&\quad + \begin{cases} \mathit{false} & \mathit{if}\ {\rm n} = {\rm s}_{VFG} \\ \prod\limits_{{\rm m} \in pred({\rm n})} \text{X-DELAYED}_{\rm m} & \mathit{otherwise} \end{cases} \\
\text{X-DELAYED}_{\rm n} &= \text{N-DELAYED}_{\rm n} \cdot \overline{\text{COMP}_{\rm n}} \cdot \text{USED}_{\rm n} \cdot \overline{\text{EFFECTIVE}_{\rm n}} \\
\text{USED}_{\rm n} &= \prod_{\substack{\mathcal{V}({\rm w}) \in PARENTS(\mathcal{V}({\rm n})) \\ \mathcal{N}({\rm w}) = \mathcal{N}({\rm n})}} \text{X-DELAYED}_{\rm w}
\end{aligned}
$$

図 20 遅延部のデータフロー方程式

Fig. 20 Dataflow equation for delay.

$$
\text{LATEST}_{\rm n} = \text{N-DELAYED}_{\rm n} \cdot \overline{\text{X-DELAYED}_{\rm n}} + \text{X-DELAYED}_{\rm n} \cdot \sum_{{\rm m} \in succ({\rm n})} \overline{\text{N-DELAYED}_{\rm m}}
$$

図 21 挿入点の計算

Fig. 21 Insertion point.

て，データフロー解析の計算量は，たかだか VFG の節の数で抑えられる[8),16)]．VFG は，EVG の各初期節に対して，重ならない CFG の範囲に作成されることから，VFG 上のデータフロー解析の計算量は，変形 EVG の作成と同様に $O(CN)$ で抑えられる．

5.4 CFG 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制御フローに変換しなければならない．この変換は，EVG が依存グラフの性質を持つことから，基本ブロックに対応する VFG 節を n と

すると，基本ブロックごとに，深さ優先順序の逆順で $\mathsf{LATEST_n} = ture$ を満たすEVG節 $\mathcal{V}(\mathsf{n})$ に対応するコードを出力していけばよい．あるEVG節 v に対する三番地コード（three address code）は，各EVG節が区別できるように番号がふられているとして次のように表される．ここで，v から，その唯一の番号を添字とする変数を生成する関数を id とする．

$id(v)$ ":= "
$id(child_1(\Pi^L(v)))\, L_{VG}(\Pi^L(v))\, id(child_2(\Pi^L(v)))$

ϕ-関数をラベルとする副節を持つ場合には，ϕ-関数固有の基本ブロック（ϕ-関数は基本ブロックに固定である）の i 番目の先行節の出口に次の式を挿入する．

$id(v)$ ":=" $id(child_i(\Pi^R(v)))$

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[3]で使われる変数融合（variables merge）[3]によって，除かれることが期待できる．

以上の変換の際，無効コード除去（dead code elimination）の効果を同時に得ることができる．無効コードとは，使用されることのない変数への定義を行うコードをいう．関数の引数や，配列の添字に用いられる式のコード，または必ずメモリに格納しなければばらないコード（レジスタに置けないコード）などの確実に使用されるコードに対応するEVG節について，それを根として上述の深さ優先順序の逆順でコードを出力することによって，無効コードの生成自体を防ぐことができる．

6. 評 価

本手法の効果を評価するために，C言語のサブセットを設定してコンパイラを作成し，実験を行った．このコンパイラは，目的コードとして，仮想機械[4]に対する機械語を出力する．目的コードはインタプリタ方式の仮想機械によって実行される．この仮想機械は，各命令を1サイクルで実行できるものとし，すべての計算に必要なだけの個数のレジスタを持つものとしている．また，機械語のオペランドとしてメモリへのアクセスができるのは，ロード命令と格納命令だけとし，他の命令のオペランドはレジスタへのアクセスだけとしている[4]．

評価は次の最適化の比較で行った．

最適化1: 定数伝播 → 定数畳込み → PRE → 無効コード除去 → 変数融合

従来の構文等価な式に対するPREと，定数伝播，定数畳込み，無効コード除去，変数融合を組み合わせたもの．

最適化2: 定数伝播 → 定数畳込み → 大域値番号づけ → PRE → 無効コード除去 → 変数融合

最適化1に加えて，PREの前に，VG上で意味等価な式を発見し，等価な変数を同一の変数名に付け替えるための大域値番号づけ（大域一致発見を用いている）を行ったもの．

本手法： EVGによるPRE → 変数融合

EVGによるPREを適用した後，変数融合を行ったもの．

実験結果として，各最適化を行った目的コードの実行サイクルを**表1**に，最適化1と最適化2を基準とした高速化の割合を**表2**に示す．

結果から，本手法は通常のPREによる最適化1よりも大きな効果があることが分かる．最適化2と比べても，10%前後の高速化が行われている．この3通り

表1 実行サイクルの比較
Table 1 Execution cycle counts of sample programs.

プログラム名	最適化1 X	最適化2 Y	本手法 Z
8 女王問題	448863	400134	372057
ガウス消去法	1739	1249	1173
クイックソート	89490	79275	68379
マージソート	18998	18638	16731
ヒープソート	27145	25149	21205
最短距離問題	2249	2054	1833
Pascal（サブセット）フロントエンド☆	77195	62019	57042

表2 高速化の割合
Table 2 Speedup ratio.

プログラム名	$\frac{Z}{X} \times 100$ (%)	$\frac{Z}{Y} \times 100$ (%)
8 女王問題	82.8	92.9
ガウス消去法	67.4	93.9
クイックソート	76.4	86.2
マージソート	88.0	89.7
ヒープソート	78.1	84.3
最短距離問題	81.5	89.2
Pascal（サブセット）フロントエンド	73.8	91.9

☆ 入力として最大公約数を求めるプログラムを用いた．

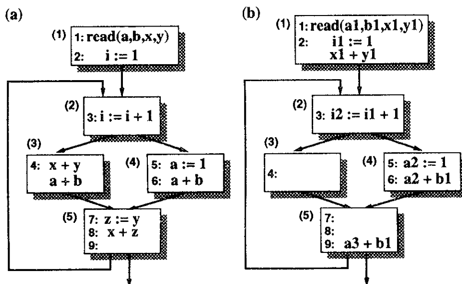


図 **22**　最適化1と最適化2の結果
Fig. 22　Results of optimization 1 and 2.

の手法について，図5(a)を例に効果の相違を述べる．

まず，通常のPREの適用である最適化1の場合は，構文等価な式だけを対象とするので，部分冗長である $a+b$（9行目）だけが除去される．結果は図 **22**(a)のようになる．

これに対して，最適化2では，前処理として行った大域値番号づけによって，$x+y$（4行目）と $x+z$（8行目）が構文的に同じになるので，これらの計算式は除去され，ループの外へ移動される．一方，最適化1で構文等価と見なされた $a+b$ は，1つの定義しか到達しない6行目と，2つの定義が到達する9行目とで構造が異なるので，最適化2では構文等価とはならない．したがって，9行目の式は除去されず，図22(b)の結果になる．

本手法では，図5(b)の結果が得られる．この図からは，最適化1で得られる効果と最適化2で得られる効果の両方が得られていることが分かる．さらに，3行目の $i := i+1$ は，ループの外側の先行節に移動したものが畳み込まれて定数になる．

本手法が，もとのPREの効果を損うことなしに，意味等価の発見とPREを組み合わせていること，さらに，定数の畳込みを効果的に統合していることが示された．

7. 関連研究

ループ不変コード移動については，不変コードをループの直前の基本ブロックに移すためのアルゴリズムがAho，Sethi，Ullmanによって示されている[1]．このアルゴリズムはSSA形式を使用していないので，1つの使用に対して，複数の定義が到達する計算式は巻き上げることができない．

PREは，巻上げによって効果がある場合に有効な手法であり，ループ不変コード移動も含まれる．Morel，Renvoise[12]によって初めて示され，後に多くの改良がなされてた．その中には，双方向のデータフロー解析を単方向の組合せにすることによって，データフロー方程式の双方向依存をなくし，ビットベクトルを用いたデータフロー解析の計算量を単方向と同等にする手法[7]や，同様の方法で，無用な巻上げをまったく行わないことを保証する手法[9),10]などが提案されている．また，PREが式を対象にしているのに対して，代入文自体を移動する手法も提案されている[6),11]．これらが除去の対象としているのは，構文等価な計算だけである．PREとコピー伝播を組み合わせて複数回の適用を行うと，意味等価な式が順次構文等価なものに変わり，除去の対象になることがある．しかし，その場合の計算量は，計算式の数 C，基本ブロックの数 N，式の依存の深さ（ランク）R[14]として，双方向データフロー解析によるものが $O(N^3R)$，単方向にしたものが $O(N^2R)$ である．

基本ブロック単位のハッシュ表を用いた値番号づけ[1]を大域的な範囲に拡張したRosen，Wegman，Zadeckの手法（以降，RWZ法と呼ぶ）[14]は，SSA形式の利用によって，意味等価な計算式どうしを冗長除去の対象にしている．しかし，この手法は，ループの認識など入力プログラムの構造に大きく依存し，アルゴリズムに必要なデータ構造も複雑である．また，その計算量は，$O(CN^2)$ である．

RWZ法を効率化したものに，Clickの手法がある．プログラムの大きさに対してほぼ線形の計算量ですむ．しかし，計算式が存在しない経路に式を挿入することを許しているため，もとのプログラムよりも実行時間が長くなる可能性を含んでいる．また，巻上げの範囲が，ϕ-関数を跨いでさらに上方に及ぶことがないので，RWZ法に比べ制限されたものになっている．

Steffen，Knoop，Rüthingの手法[16]は本手法と同様にVFGを用いる手法である．彼らの手法は，計算式を閉路なし有向グラフ（directed acyclic graph）で表現し，CFGに沿って，伝播させることで各計算点の計算式表現を作成する．この方法による計算量は，$O(N^4)$ である☆．

本手法は，以上の研究と比べると，計算量が $O(CN)$ と効率が良く，かつ，簡単なデータ構造とアルゴリズムで実現できる．さらに，計算式の移動によって新たに可能になる定数畳込みをPREに統合している点で，従来提案されている手法よりもさらに大きな効果が期待できる．

☆ この場合の N は，並行代入だけを1つの基本ブロックとする制御フローグラフの節数である．1つの計算式に対して，ほぼ1つの基本ブロックが対応する．

8. 結　論

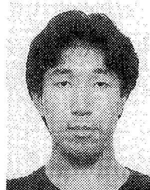
本稿では，EVGに基づく効果的な部分冗長除去法を提案した．拡張値グラフは，計算式が各計算点に移動したときの等価関係を構造的に表現することで，意味等価に基づく部分冗長除去を可能にした．また，EVG上での定数畳込みは，実際に計算式を移動した際に畳込みの効果が得られるかどうかの情報を部分冗長除去に組み込むことを可能にした．計算量の点でも，悲観的にみて $O(CN)$ であり，従来の方法に比べて効率が良い．

本手法の効果を評価するために，C言語のサブセットに対して，本手法による最適化を行うコンパイラを試作し，実験を行った．結果として，従来の最適化の効果的な組合せに比べて，さらに，効果が得られることを示した．

参考文献

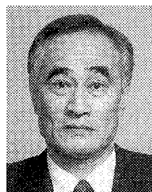
1) Aho, A.V., Sethi, R. and Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA (1986).
2) Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting equality of variables in programs, *POPL*, ACM, pp.1–11 (1988).
3) Chow, F.C. and Hennessy, J.L.: The Priority-Based Coloring Approach to Register Allocation, *ACM Trans. Prog. Lang. Syst.*, Vol.12, No.4, pp.501–536 (1990).
4) Click, C.: Global Code Motion Global Value Numbering, *PLDI*, pp.246–257, ACM (1995).
5) 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).
6) Dhamdhere, D.M.: Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.2, pp.291–294 (1991).
7) Dhamdhere, D.M. and Patil, H.: An Elimination Algorithm for Bidirectional Data Flow Problems Using Edge Placement, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.2, pp.321–336 (1993).
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) Knoop, J., Rüthing, O. and Steffen, B.: Lazy Code Motion, *PLDI*, pp.224–234, ACM (1992).
10) 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).
11) Knoop, J., Rüthing, O. and Steffen, B.: The Power of Assignment Motion, *PLDI*, ACM, pp.233–245 (1995).
12) Morel, E. and Renvoise, C.: Global optimization by suppersion of partial redundancies, *Comm. ACM*, Vol.22, No.2, pp.96–103 (1979).
13) 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・実践—研究会報告，Vol.20, No.3, pp.21–30 (1995).
14) Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Global value numbers and redundant computations, *POPL*, pp.12–27, ACM (1988).
15) Sreedhar, V.C. and Gao, G.R.: A Linear Time Algorithm for Placing ϕ-Nodes, *POPL*, pp.62–73, ACM (1995).
16) 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).

(平成8年11月29日受付)
(平成9年9月10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，同大学大学院後期博士課程在籍，プログラミング言語と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会員．