静的単一代入形式上で通常形式部分冗長除去を実現する汎用的手法

今橋孝典[†1] 伊藤陽[†1,*1] 佐々政孝[†1]

部分冗長除去（PRE）は，部分的に冗長な式を除去する変換で，共通部分式除去とループ不変式移動の効果を含んだ効果的なプログラム最適化である．この PRE を，最適化に適した形式である静的単一代入（SSA）形式上で行おうという従来研究がいくつかあるが，これは一般には容易ではない．その理由は，SSA 形式の特徴である，変数名の一意性に関する規則が PRE の実装の妨げになるからである．たとえば，同じ名前であった変数どうしが SSA 形式化にともない異なる名前になり，変数名の同一性の判断ができなくなるといった問題があげられる．このような問題に対し，従来手法では，PRE を SSA 形式上で実現するために特別なデータ構造を用いるなど複雑な処理を行っていた．これに対し本論文では，SSA 形式でありながら通常形式にも近い性質を持つ CSSA 形式と phi congruence class というものを利用すれば，SSA 形式でも通常形式における変数名の同一性が判断できることに着目した．その事実に基づいて，通常形式の PRE アルゴリズムを SSA 形式に適用する手法を提案した．この手法は汎用的なものであり，挿入点の決定・式の挿入・式の置き換え（冗長性の除去）という通常の手順に従う PRE であれば，原則としてアルゴリズムによらず SSA 形式に対応させることができ，また元々の PRE のアルゴリズムの枠組みを変える必要もない．本手法を確かめる実験として，代表的な PRE アルゴリズムの 1 つである Lazy Code Motion を SSA 形式上のアルゴリズムに変換し，変換後でも部分冗長除去の効果が変わりなく発揮されていることを確認した．

A Generalized Method for Realizing PRE on SSA Form

Takanori Imahashi,[†1] Yo Ito[†1,*1] and Masataka Sassa[†1]

Partial Redundancy Elimination (PRE) is an effective optimization for eliminating partially redundant expressions and contains effects of common subexpression elimination and hoisting loop invariant. There have been some previous attempts to apply PRE to Static Single Assignment (SSA) form that is a suitable intermediate form for optimization. But such attempts have general difficulty because of the characteristics of variable names in SSA form. For example, variables which have the same name on normal form may have different names on SSA form. So, it becomes difficult to identify the same variables in SSA form. To handle such problems, previous methods perform complicated processing by using special data structures and so on. To deal with this problem, we pay attention to the so-called CSSA form and phi congruence class (pcc). With CSSA form and pcc, we can identify the same variables on SSA form. Based on this fact, we propose a method for transforming PRE algorithms for normal form to those for SSA form. This tranformation is a universal one. So, in principle, it can transform any PRE algorithm which has ordinary process (deciding insertion point, insertion of expression, and replacing expressions), and does not change the framework of the original PRE algorithm. Finally, as an experiment, we apply this method to Lazy code motion (LCM) that is one of representative PRE algorithms. We confirmed that the transformed LCM in SSA form performs PRE correctly and produces object code with the same efficiency as the one in normal form.

1. はじめに

1.1 背景

部分冗長除去（Partial Redundancy Elimination，以後 PRE と略称することもある）は，部分的に冗長な式を除去する変換であり，共通部分式除去とループ不変式移動の効果を含んだ効果的な最適化である．PRE は，Morel らによって最初のアルゴリズムが提案され[13]，その後も多くの改良されたアルゴリズ

†1 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
*1 現在，NEC
Presently with NEC Corporation

ムが提案されている[6)–8),11),12),15)]．

これらの研究の中で，PRE を**静的単一代入形式（Static Single Assignment Form，以後 SSA 形式と呼ぶ）**上で行おうという試みがある[10),19)]．SSA 形式は最適化に適したプログラムの表現形式であり，近年さかんに研究が行われている．しかし，PRE を SSA 形式上で行おうとすると，

- 通常形式上では同一の変数であったものが名前替えにより異なる変数名になることがある，
- SSA 形式での変数には満たすべき条件があるので，異なるブロックにコードを移動する際に変数をそのまま移動できない，

といった困難がある（詳細は次章で述べる）．

本研究では，これらの問題を解決し，PRE アルゴリズムを SSA 形式上で実現する汎用的手法（以後，本手法という）を提案する．

1.2 本手法の概要

本手法の概要を説明する．

一般的な PRE アルゴリズムの手順を図示すると**図 1**（左）のようになる．このうち，アルゴリズムの中心となるのは「挿入する式と挿入点の決定」の部分である．

一方，通常形式での PRE アルゴリズムを本手法によって SSA 形式に対応させると，その手順は図 1（右）のようになる．このうち，元の PRE アルゴリズムに対応している部分は「挿入する式と挿入点の決定」と「式の挿入」，「式の置き換え」の 3 つであるが，後者 2 つは「$t = a+b$」の挿入や，「$\ldots = a+b$」の「$\ldots = t$」への置き換えであって，どの PRE アルゴリズムでも同じである．よって，元の PRE アルゴリズムに依存するのは実質「挿入する式と挿入点の決定」のみとなる．この処理は，変数の字面の情報の代わりに pcc（後述）の情報を用いることで，アルゴリズムの本質を変えることなく SSA 形式に対応させることができる．よって，図 1（左）の手順に沿うような PRE アルゴリズムであれば本手法を用いて SSA 形式に対応させることができる．

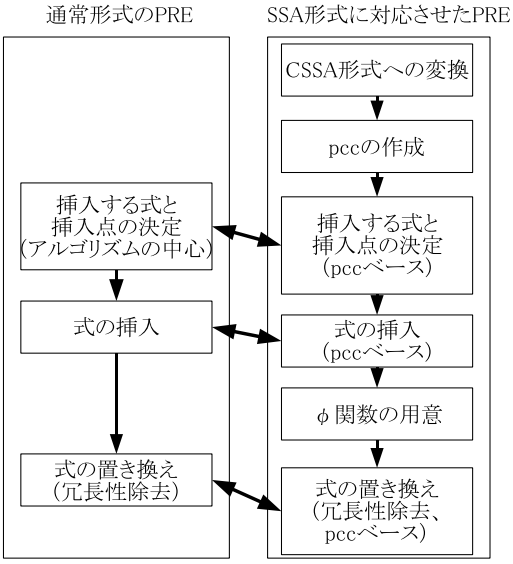


図 1 通常形式上の PRE と SSA 形式上の PRE
Fig. 1 PRE on normal form and PRE on SSA form.

2. 部分冗長除去と SSA 形式

本章では，本研究の前提となっている部分冗長除去と SSA 形式，および SSA 形式上での PRE の問題点について述べる．

2.1 部分冗長除去

プログラムの実行中には，ある式の値がすでに計算されているにもかかわらず，再びその式の値を計算し直すパスが存在することがある．PRE は，そのような冗長な計算をすでに計算した計算結果で置き換える最適化の手法である．**図 2** はその最も簡単な場合であり，図 2（左）に対して冗長性除去を行うと図 2（右）のようになる．

一般の部分冗長除去はその名が示すとおり，あるプログラムパスを通ってきた場合にのみ冗長となるような計算（部分冗長という）に対しても，冗長性を除去することができる．たとえば**図 3**（左）のようなプログラムでは，左上からのパスを通ってきたときは冗長であるが右上からのパスを通ってきたときは冗長ではない．よってこれは部分冗長である．

このようなプログラムに対しては，右上のブロックに対象となる計算を挿入することで，どのパスを通っても「$a+b$」の計算が冗長になる（全冗長という）ようにできる．その結果，冗長性が除去できるようになる（図 3 右）．以上のように，PRE の基本的な手順は，

(1) プログラムに式を挿入し，部分冗長な式を全冗長にする，
(2) 全冗長となった式を除去する，

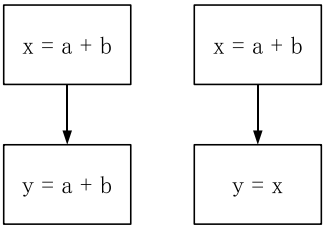


図 2 簡単な冗長性除去の例
Fig. 2 A simple example of PRE.

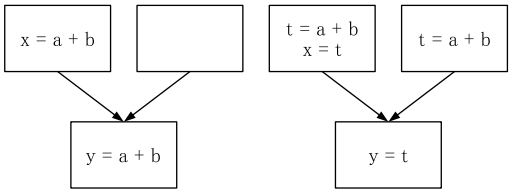


図 3 部分冗長除去（PRE）の例
Fig. 3 An example of PRE.

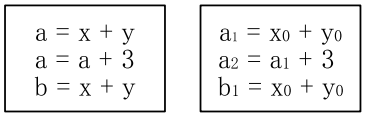


図 4 通常形式（左）と SSA 形式のプログラム（右）
Fig. 4 Normal form and SSA form.

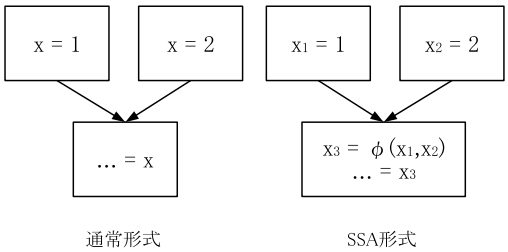


図 5 ϕ 関数の例
Fig. 5 An example of ϕ function.

である．この，挿入式と挿入点を求めることが PRE アルゴリズムの要であり，個々の部分の違いがさまざまなアルゴリズムの差であり，最適化効果の差となる．

2.2 SSA 形式

SSA 形式は，変数の定義がプログラムの字面上で唯一になるようにしたプログラムの表現形式である[1),2),5),14)]．静的とは，プログラムの字面上で，という意味である．変数の定義が唯一になるように変数の名前替えを行うが，これはふつう変数に添字をつけて表す．この結果，SSA 形式では，プログラムあるいは中間表現の上で，各変数の定義が 1 カ所だけになる（**図 4**）．

また，異なる定義の合流点には ϕ 関数という仮想的な関数を挿入することで定義を 1 つにまとめる．

図 5（右）での「$x_3 = \phi(x_1, x_2)$」は，制御の流れが左上の基本ブロックから来たときには x_3 に x_1 を代入し，右上の基本ブロックから来たときには x_3 に x_2 の値を代入する，という意味である．

SSA 形式は，プログラミング言語処理における最適化に適しているとされるプログラムの表現形式であり，近年さかんに研究が行われている．

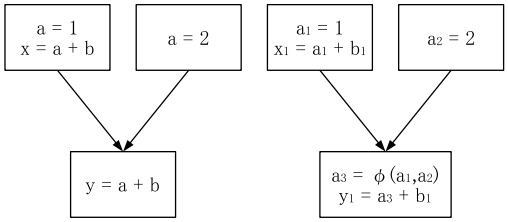


図 6 SSA 形式と PRE
Fig. 6 SSA form and PRE.

2.3 SSA 形式上での PRE

PRE は強力な最適化であるが，これを SSA 形式上で実現するのは以下に示す 2 つの理由のため容易ではない．

- 通常形式上では同一の変数として扱えたものが，SSA 形式上では添字付けのため異なる変数と認識され，「同一な式」の検出が困難になる．
- 単純に異なる基本ブロックにコード移動を行うと，SSA 形式が満たすべき条件が守られなくなる可能性がある．

たとえば，**図 6**（左）の「$a+b$」は PRE の対象となるものであるが，図 6（右）のように SSA 形式に変換すると，2 つの「$a+b$」が字面上では別々の形になってしまうため同一の式で冗長であることが判別できなくなってしまう．これが 1 つめの問題である．

また，仮に「a_1+b_1」と「a_3+b_1」が同じだと認識できても図 6（右）の下のブロックの「a_3+b_1」を右上のブロックに挿入することはできない．なぜなら，このままのコードを挿入すると，a_3 の使用が a_3 の定義よりも先にきてしまうからである．これが 2 つめの問題である．

文献 10)，19) などでは SSA 形式上の PRE が提案されているが，これらのアルゴリズムでは制御フローグラフとは別に特別なグラフの作成とそのグラフにおける複雑な解析が必要になっている．また，文献 4) のアルゴリズムでは処理中にいったん SSA 形式から通常形式に戻ってしまう．立川は Lazy code motion [12)] のアルゴリズムをもとに，それらの問題を解決した SSA 形式上の PRE アルゴリズムを提案している[18)]．しかしこの方法は後述する CSSA 形式にすでになっているものにしか適用できないことのほか，各種の概念が整理されていない，アルゴリズムが煩雑，といった問題点がある．

3. TSSA 形式と CSSA 形式

CSSA 形式とは，すぐ後で述べる phi congruence class（以後，pcc と略称する）に属する変数をすべて

同じ代表変数に置き換えて ϕ 関数を除去すると，プログラムの意味の等しい通常形式が得られるような SSA 形式，と定義できる[17]．これは，元々Cytron らのアルゴリズム[5] で通常形式を SSA 形式に変換した直後の SSA 形式が有している性質である．この条件は，同じ pcc 内の変数どうしに干渉（生存区間が重複すること）がないこと，といい換えることもできる[⋆1]．

pcc とは，あらまし ϕ 関数内の変数（ϕ 関数の左辺を含む，以下も同じ）の集合のことである．ただし，異なる ϕ 関数内の変数どうしに同じものがあるときは，両方の ϕ 関数内の変数をマージする．

CSSA 形式における pcc は直感的には，同一の代表変数に置き換えてもかまわないような変数の集合を意味する．

しかし，SSA 形式での最適化変換を行うと，前述のような CSSA 形式の性質は一般には保たれない．つまり，pcc の変数間に干渉が発生する．このような SSA 形式を TSSA 形式という．本手法では PRE を行う前に，文献 17) の手法を用いて TSSA 形式から CSSA 形式への変換を行う．以下に，文献 17) の手法を簡単に示す．

図 7 (a) は，変数 a_3 と a_2 の生存区間に重なりがある（a_3 と a_2 が干渉する）．このような場合，コピー文を挿入して図 7 (b) あるいは図 7 (c) のように変形することで ϕ 関数内の変数間の干渉をなくすことができる．図 7 (b) は，ϕ 関数の左辺の生存区間が最小になるように変形した例であり，図 7 (c) は，ϕ 関数の引数の生存区間が最小になるように変形した例である[⋆2]．

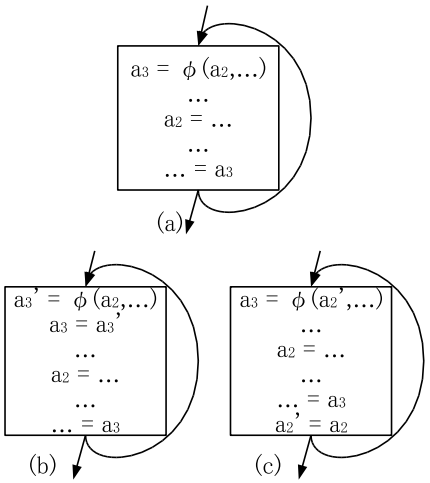


図 7 TSSA 形式から CSSA 形式への変換
Fig. 7 Transformation of TSSA form to CSSA form.

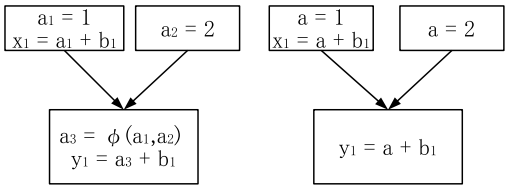
⋆1 この定義によれば，pcc の変数を同じ代表変数で置き換えても等価なプログラムになることが CSSA 形式の必要十分条件となる．

4. Phi congruence class

phi congruence class（以後 pcc と略称することもある）とは，ϕ 関数で結ばれた変数の集合である．たとえば**図 8**（左）では「$a_3 = \phi(a_1, a_2)$」によって a_1，a_2，a_3 が ϕ 関数で結ばれ，同一のクラスに属することになる．その結果，図 8 の phi congruence class は $\{a_1, a_2, a_3\}$ となる．

1 つの変数が複数の ϕ 関数に属する場合，phi congruence class は，プログラム中の各 ϕ 関数で結ばれた変数を同じクラスにし，最後に共通部分を持つクラスをマージしたものである．その簡単な例を**図 9** にあげる．ϕ 関数内の変数はそれぞれ $\{x, y, z\}$ と $\{u, x, w\}$ だが，x が双方に共通しているため，2 つの phi congruence class がマージされ，最終的な phi congruence class は $\{x, y, z, u, w\}$ となる．

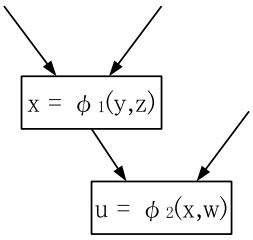
前章で述べたように，CSSA 形式上では同じ phi



a1、a2、a3、に干渉がないので、これらを
全て同じ変数aに置き換えてφ関数を除去すると
通常形式になる

{a1,a2,a3} → a

図 8 CSSA 形式の特徴
Fig. 8 A characteristic of CSSA form.



φ1のpcc → {x,y,z}
φ2のpcc → {u,x,w}
マージされたpcc → {x,y,z,u,w}

図 9 phi conguence class（pcc）のマージ
Fig. 9 Merge of phi congruence class (pcc).

⋆2 ϕ 関数の引数の変数の生存区間は，その ϕ 関数のある基本ブロックの先行ブロックの最後までである．また ϕ 関数の左辺の変数の生存区間は，その ϕ 関数のある基本ブロックの先頭からである．

cougruence class に属する変数を同じ代表変数に置き換えるとプログラムの意味が等しい通常形式となる．よって，「CSSA 形式上で変数どうしが同じ phi congruence class に属することは，通常形式上で同じ字面の変数であること」に対応する．この事実を利用すれば，PRE を SSA 形式上で行うときの問題の 1 つであった「同一な式の検出」が可能になる．つまり，同一の式かどうかを判断する場合に通常形式上においては同じ変数かどうかの判断を行っていたが，CSSA 形式上では代わりに同じ phi congruence class に属する変数かどうかを判断すればよい．

以下に，フローグラフ fg に対する pcc を作成するアルゴリズムの概略を示す．

アルゴリズム 4.1（pcc の作成）

```
makePhiCongruenceClass(FlowGraph fg){
  for(各 φ 関数 phi ∈ fg){
    phi で結ばれた変数を同じ pcc にする;
  }
  共通部分を持つ pcc をマージする;
}
```

また，pcc を使用して，与えられた 2 つの式 $expr1$ と $expr2$ の同一性を識別する方法を次に示す．

アルゴリズム 4.2（pcc による式の識別）

```
same(expr1, expr2){
  if(式 expr1, expr2 の 演算子,
    型がそれぞれ等しい){
    if(expr1, expr2 の オペランドが属する
      pcc がそれぞれ等しい){
      return true;
    }
  }
  return false;
}
```

5. 挿入する式と挿入点の決定

挿入する式と挿入点の決定は，実装する PRE アルゴリズムが行う．ただしその際 CSSA 形式では，前章で述べたように，同じ変数かどうかを判断する代わりに同じ phi congruence class に属するかどうかを判断するよう，PRE アルゴリズムを変更する．

6. 式 の 挿 入

部分冗長な式を発見すると，PRE アルゴリズムは

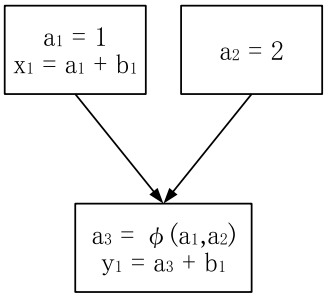


図 10 PRE 適用前のプログラム
Fig. 10 The program before PRE.

前章で求められたプログラムの挿入点（**図 10** の場合，右上のブロック）に文「一時変数 = 式」を挿入し，部分冗長な式を全冗長にする．また後の処理のために，計算結果を一時変数に格納しておく必要があるため，もう片方の先行ブロック（この場合，左上のブロック）の元の計算の直前にも，やはり同様の文を挿入する．

このような式の挿入が行われる際に，SSA 形式の満たすべき条件が問題になる．つまり，単純に式を挿入すると，変数の使用が定義に先行してしまう恐れがある．そのため，挿入する際には，挿入しても問題のない変数名に変数を書き直す必要がある．本手法では式の挿入を以下の例のように行う．

ここでは，図 10 のプログラムで「a_3+b_1」を左上のブロックの「$x_1 = a_1+b_1$」の直前に挿入する場合を考える．まず，式のそれぞれの変数に対して，挿入点からその点を支配する点を上向きにたどっていく．そしてその変数と同じ phi congruence class に属する変数の定義を見つけたら，その見つかった変数を挿入する式の変数とする．この例だとまず変数 a_3 について，同じ phi congruence class に属する変数の定義を探すことになる．その結果，a_3 と同じ phi congruence class に属する a_1 の定義「$a_1 = 1$」が見つかる（**図 11**）．よって，式を実際に挿入する際には a_3 を a_1 と書き直すことになる．

また，b_1 については図の中には記されていないが，上方に b_1 の定義文があるとする．すると，b_1 が挿入すべき変数となる．以上から，左上のブロックには，書き直された式「a_1+b_1」を挿入すればよいことが分かる．

ここまでの例では左上のブロックにのみ挿入を行ったが，通常の PRE アルゴリズムだと，右上のブロックにも式を挿入しようとする．そこで右上のブロックの「$a_2 = 2$」の直後に挿入点があるとして同様の処理を適用すると，最終的にプログラムは**図 12** のようになる．なお，一時変数は，まだ使用されていない t_1，

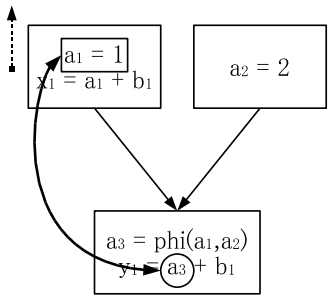


図 11 同じ pcc に属する変数の定義の探索
Fig. 11 Search of definition of variable which belongs to the same pcc.

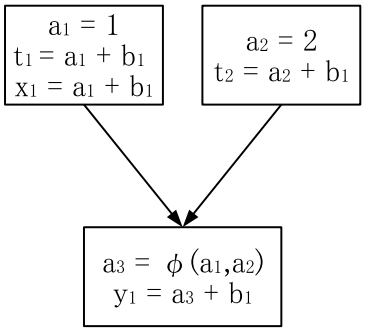


図 12 式を挿入した直後のプログラム
Fig. 12 The program after insertion of expression.

t_2 を使った．

以下に，式 *expr* をプログラムポイント p に挿入するアルゴリズムを示す．*expr* は a *op* b とする．*op* は演算子，a，b はオペランドである．

アルゴリズム 6.1（式の挿入）

```
insertExpr(expr, p){
  p を支配する点を上向きにたどり,
  a と同じ pcc に属する変数 a′の定義を探す;
  p を支配する点を上向きにたどり,
  b と同じ pcc に属する変数 b′ の定義を探す;
  まだ定義されてない変数 t_i を作成する;
  代入文 t_i = a′ op b′を p に挿入する;
}
```

7. ϕ 関数の用意

一時変数の挿入の後は，その挿入した一時変数のための ϕ 関数を用意する．

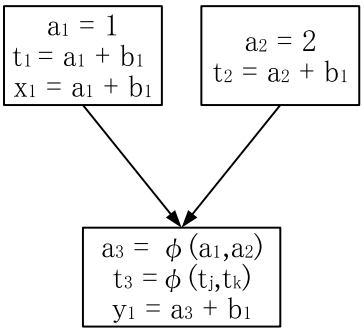


図 13 一時変数の ϕ 関数を作成したプログラム
Fig. 13 The program with ϕ function for temporary variable.

7.1 ϕ 関数の作成

まず，最小 SSA を作成するアルゴリズム[5] に従って，前章で $t_i = a_j + b_k$ の形の式を挿入したブロックの支配辺境に，一時変数のための ϕ 関数を作成する．図 12 のプログラムに対して，一時変数のための ϕ 関数を作成すると**図 13** のようになる．

なお，ϕ 関数の左辺には，まだ使われていない一時変数の名前（図 13 の例では t_3）を付ければよい．また，右辺の変数にはこの時点では仮の名前を付けておく．なぜなら，図 13 の場合だと ϕ 関数の右辺が指すべき変数は t_1，t_2 でありそれらはすでに存在しているが，t_3 のような「この手続きで挿入される，他の ϕ 関数の左辺の一時変数」を指すべき場合も存在しうるためである．その場合は ϕ 関数の作成順序によっては，「本来指すべき変数」がこの時点ではまだ存在しないことになり，ひととおり ϕ 関数を挿入した後でないと右辺を正しく決めることができなくなる[9]．

以下に，ϕ 関数を作成するアルゴリズムを示す．

アルゴリズム 7.1（ϕ 関数の作成）

```
insertPhi(){
  for(式の挿入を行った各ブロック blk){
    if(blkの支配辺境 dfにまだ φ 関数が
      挿入されていない){
      まだ定義されていない変数 t_i を作成する;
      ダミーの変数として任意の変数 t_j, t_k を
      作成する;
      φ 関数 t_i = φ(t_j, t_k) を dfに挿入する;
    }
  }
}
```

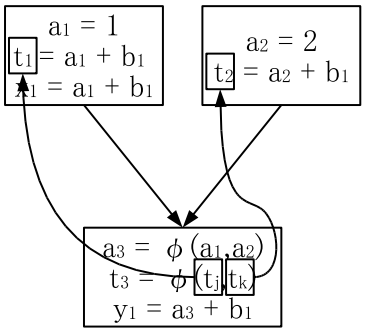


図 14 ϕ 関数内の変数の名前替え
Fig. 14 Change of name of variable in ϕ function.

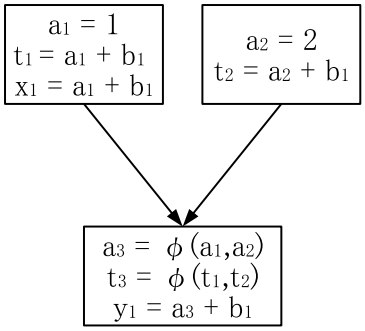


図 15 ϕ 関数の用意が完了したプログラム
Fig. 15 The program with ϕ function ready.

7.2 ϕ 関数内の変数の名前替え

一時変数のための ϕ 関数をひととおり挿入した後で，ϕ 関数の右辺の一時変数を適切な名前に書き換える処理を行う．

たとえば図 13 の式「$t_3 = \phi(t_j, t_k)$」の t_j を置き換える場合を考える．この場合，問題の ϕ 関数の左上のブロックから，支配するブロックをたどっていき，前節で挿入した式か，本節前半で作成した ϕ 関数を探す（探すべきこれらの式や ϕ 関数の情報は，あらかじめ記憶しておく）．そしてどちらかが見つかるとその左辺の一時変数で ϕ 関数内の変数を置き換える．この例だと，左上のブロックの式「$t_1 = a_1 + b_1$」が前節で挿入した式であるので，左辺の t_1 で t_j を置き換える．t_k を置き換える場合は，右上のブロックから同様に探索を行う．この例の場合は式「$t_2 = a_2 + b_1$」が見つかるのでその左辺 t_2 で t_k を置き換える．これらの探索の様子を図示すると**図 14** のようになり，名前替えの結果，最終的に得られるプログラムは**図 15** のようになる．

以下に名前替えのアルゴリズムを示す．ただし，ここでは ϕ 関数を正確に表すため「$t_i = \phi(t_j : L_a, t_k : L_b)$」と記している．これは，ブロック L_a から来たときは t_j を使用し，ブロック L_b から来たときは t_k を使用する，ということを示している．

アルゴリズム 7.2（ϕ 関数の引数の名前替え）

```
rename(){
  for(挿入した各 φ 関数 t_i =
      φ(t_j : L_a, t_k : L_b)){
      ブロック L_aからそのブロックを支配する
      ブロックを上向きにたどり，
      挿入した式か挿入した φ 関数を見つけたら，
      その左辺で t_jを置き換える;
      ブロック L_bからそのブロックを支配する
      ブロックを上向きにたどり，
      挿入した式か挿入した φ 関数を見つけたら，
      その左辺で t_kを置き換える;
      名前替えを行った φ 関数の引数の変数を
      同じ pcc にする;
  }
}
```

8. 冗長となった式の除去

冗長な式の除去は，実際には式の，今導入した一時変数への置き換えである．その際，置き換える一時変数を決める必要がある．対象とする式 1 つに対し，挿入された一時変数はすべて 1 つの phi congruence class に属している．たとえば今回の例では式「$a+b$」の結果を格納している変数は t_1，t_2，t_3 の 3 つであり，これらは同じ pcc $\{t_1, t_2, t_3\}$ に属している．この「$a+b$」と $\{t_1, t_2, t_3\}$ のような，式とそれに対応する一時変数の pcc との組は，それぞれ保存されているものとする．そのため，「$a+b$」の結果を一時変数で置き換えたい場合は，pcc $\{t_1, t_2, t_3\}$ に属する一時変数を探せばよい．具体的には，置き換えたい式の存在する点から，その点を支配する点を上にたどっていき，上述の pcc に属する変数の定義を見つけたら，その変数で式を置き換える．

以下にプログラムポイント p の右辺の式を一時変数で置き換えるアルゴリズムを示す．点 p はすでに求まっており，一時変数は t と同じ pcc に属しているとする．

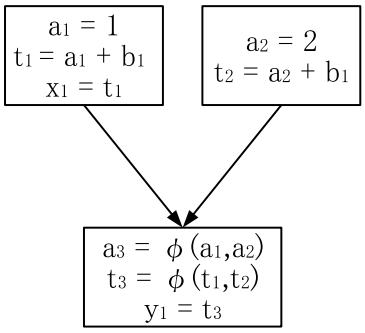


図 16 SSA 形式上の PRE の結果
Fig. 16 PRE on SSA form.

アルゴリズム 8.1（冗長となった式の除去）
replace(p,t)\{
p を支配する点を上向きにたどり,
t と同じ pcc に属する変数 t'の定義を探す;
p に存在する文の右辺の式を t'で置き換える;
\}

たとえば，図 15 の下のブロックの「a_3+b_1」を一時変数に置き換えたいときは，ここから支配関係を上にたどると一時変数と同じ phi congruece class に属する変数 t_3 の定義「$t_3=\phi(t_1,t_2)$」が見つかるので，t_3 で置き換える．左上のブロックの「a_1+b_1」を置き換えたいときは，同様にして t_1 の定義「$t_1=a_1+b_1$」が見つかるので t_1 で置き換える．

以上の処理によって SSA 形式上での PRE は達成され，最終的に得られるプログラムは**図 16** のようになる．

9. 実　　験

本研究では実験として，代表的な PRE アルゴリズムである Lazy code motion [12] を SSA 形式に対応させた．実験には COINS の SSA 形式最適化コンパイラ用モジュール[16] を用い，Sun Blade 1000（UltraSPARC III）で実行時間を測定した．

最適化としての効果を確認するためのベンチマークプログラムとして，SPEC CINT2000 および SPEC FP2000 のベンチマークを使用して，各最適化を行った．

9.1 適用した最適化列

本手法の効果を評価するため，本研究ではまず SSA 形式 PRE 単体での効果を調べるための実験，次に他の最適化と組み合わせたときの効果を調べる実験の，2 つの実験を行った．以下では，それぞれの実験で使用した最適化列について解説する．ただし No-opt 以外はいずれも，「式の 3 アドレス命令への変換」を最初に，「不要命令除去」を最後に行っている．これは，一般的に最適化が，これらの処理を前提として設計されているためである．

9.1.1 実　験　1

まず 1 つめの実験として，本手法で実装した SSA 形式上の Lazy code motion の単体での最適化効果を調べるため，以下の最適化列を適用し，実行時間を計測した．

- No-opt：最適化なし
- SSALCM：SSA 形式 Lazy code motion
- NormalLCM：通常形式 Lazy code motion

最適化を何も適用しない「No-opt」と，SSA 形式 Lazy code motion のみを行う「SSALCM」，また LCM の効果が発揮されていることを確認するため，通常形式での LCM との比較を行った．

9.1.2 実　験　2

次に 2 つめの実験として，Lazy code motion を他のさまざまな最適化と組み合わせたときの効果を調べるため，COINS で良い結果が得られるとされている最適化列「O2」をベースにした 3 つの最適化列の効果を調べた．

- O2：式の 3 アドレス命令への変換 → 共通部分式除去 → 定数伝播 → ループ不変式移動 → 演算の強さの軽減 → ループ不変式移動 → 定数伝播 → コピー伝播 → preqp → 定数伝播 → 不要 ϕ 関数除去 → 不要命令除去
- O2-：「O2」から「preqp」を取り除いたもの
- SSALCM+：「O2-」の最適化列の preqp があった位置に SSA 形式 LCM を加えたもの
- NormalLCM+：「O2-」の最適化列に通常形式 LCM を加えたもの

O2 の中にある preqp とは，滝本の質問伝播に基づく大域値番号付けと部分冗長除去である[16]．preqp は LCM と効果が重複するため今回の比較実験では使用せず，O2 から preqp を取り除いた O2-と，それらに SSA 形式と通常形式の LCM を加えた SSALCM+と NormalLCM+の 3 つの最適化列で目的コードの実行時間を測った．

9.2 結果と評価

実験結果は**図 17** および **図 18** のようになった．

1 番目の実験結果（図 17）では，SSA 形式 Lazy code motion を単体で適用した LCM が，最大で約 10%の効果があることが分かった．

また 2 番目の実験結果（図 18）により，他の最適化と組み合わせたときに，SSA 形式の LCM は通常形式の LCM とほぼ同等の結果を出していることが分か

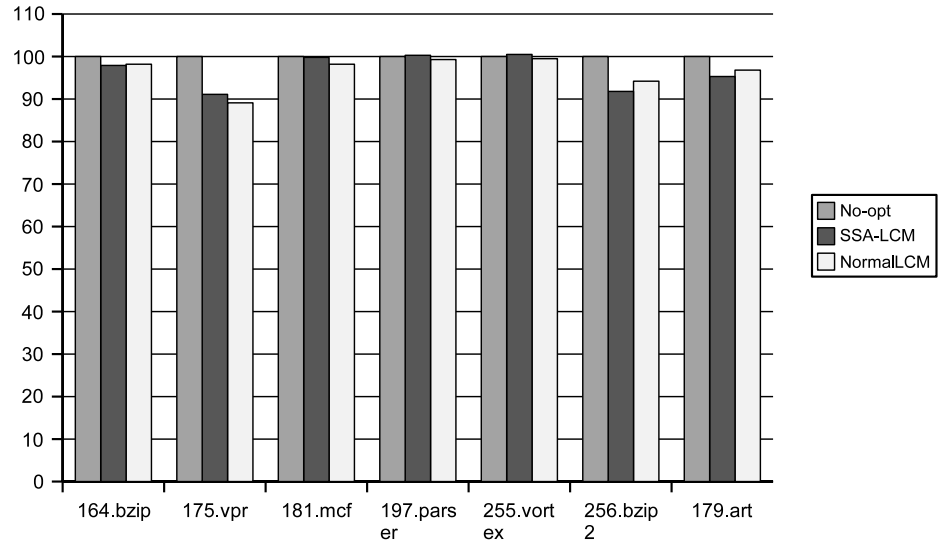


図 17 実験 1 の目的コードの実行時間（No-opt との比率）
Fig. 17 Execution times of the object codes of the first experiment compared with No-opt.

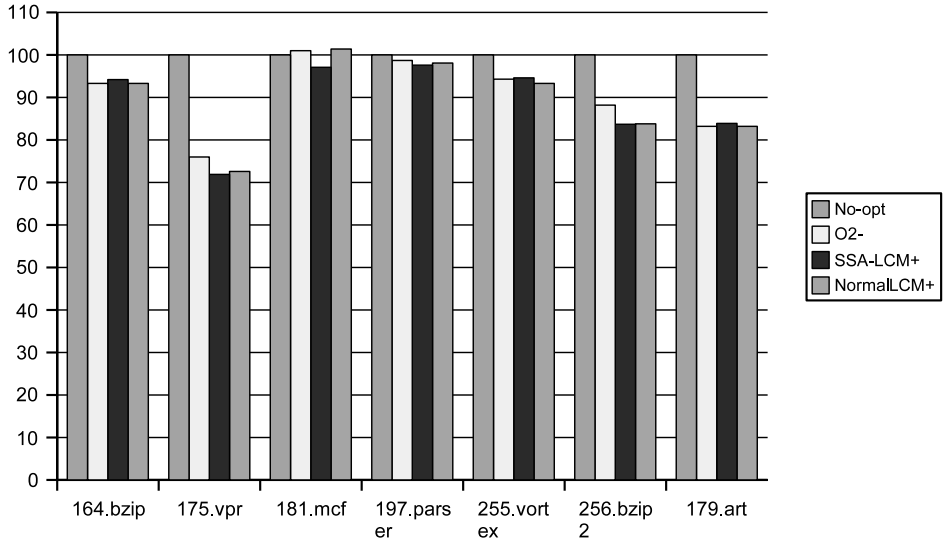


図 18 実験 2 の目的コードの実行時間（No-opt との比率）
Fig. 18 Execution times of the object codes of the second experiment compared with No-opt.

る．なお，SSA 形式，通常形式ともに LCM の導入によって格段の向上が見られない場合があるのは，類似の最適化である「共通部分式除去」「ループ不変式移動」がすでに O2-に含まれていることが原因と考えられる．

以上から，本手法で実装した SSA 形式 Lazy code motion が正しく部分冗長除去の効果を発揮できていることが確認できた．

10. 議　　論

本章では，本手法の汎用性および寄与について述べる．

10.1 汎 用 性

本研究では，実験の際に PRE アルゴリズムとして LCM（Lazy code motion）を採用した．これは，LCM が PRE アルゴリズムの代表的なものであり，また初期の素朴な PRE に比べて効果が高く，複雑な処理を行っているためである．この LCM への本手法の適用

結果および実験結果より，本手法の汎用性は示されたと考える．

その他の PRE アルゴリズムに関していうと，文献 6)–8)，15) はいずれも図 1（左）の枠組みを使っているので，LCM と同様に本手法が適用できる．

また，文献 3) のアルゴリズムには，コードの複製などによってフローグラフを変形させるといった複雑な部分があるが，このようなアルゴリズムも究極的には図 1（左）の枠組みに収められ，本手法が適用可能である．ただし，コードの複製によって CSSA 形式が崩れ，複製後に CSSA 形式への再変換が必要になる可能性もある．

そのほかに，本提案手法を用いた，実行時情報を利用した SSA 形式での PRE が存在する[9]．

10.2 本研究の寄与

SSA 形式の枠組みの中で，既存の PRE と同じアルゴリズムを提供できるようにしたことが本手法の貢献である．これを，SSA 形式を通常形式に戻して，既存の PRE アルゴリズムを適用する場合と比べると，次がいえる．

- SSA 形式上の PRE の最適化能力については，基本的には通常形式と同じ最適化を行っているので，ほとんど差はないと考えられる（効果の実験結果については 9 章参照）．
- 一連の SSA 形式上での最適化の間に，SSA 形式から通常形式への変換を行い，通常形式上の PRE を行い，再び SSA 変換をしてから，SSA 形式最適化を続ける，という方法は，本提案に比べて最適化処理に時間がかかる．
- 定数伝播など，SSA 形式と相性の良い処理と PRE を組み合わせることが容易になる．たとえば最近の滝本らの研究[16] は大域値番号付けと PRE を組み合わせようとした例といえる．本手法を用いることで，SSA 形式の利点を利用した新たな PRE アルゴリズムの開発も期待できる．

本手法では，PRE を SSA 形式上で実現できるので，逆変換のオーバヘッドなしに SSA 形式上での最適化列の中に PRE を組み込むことが可能になる．本手法は，新しい PRE アルゴリズムを開発し，その PRE を SSA 最適化列内に組み込みたい開発者にとって有益なものとなるだろう．

従来の SSA 形式 PRE の実行では，ふつうは疎な依存関係を利用して高速化を図るしかなく，ビットベクトル法は使えない．しかし，本手法で SSA 形式化したアルゴリズムでは，PRE の大部分の処理に通常形式と同じ手順が使える．そのほとんどはデータフロー方程式の解を求めているので，通常形式と同様のビットベクトル法を用いることができる．これにより SSA 形式 PRE の高速化が期待できる．

10.3 SSA 形式上の最適化列での PRE の位置

SSA 形式上で PRE を行った場合は，コピー文が残ることが多いので，その後にコピー伝播を行うとよい．この結果不要命令が生じることが多いので，さらに不要命令除去の最適化を行うことが望ましい．

COINS の SSA 最適化で，「O2」としているものは 9.1.2 項に記載したものであるが，このうちの preqp は別途開発した PRE の一種であり，この最適化列は上記の考察を満たしている．最適化列 SSALCM+も同様に上記を満たしている．

ちなみに，最適化列 O2 は上記のような考察で選んだ 4 種類の最適化の列を SPEC ベンチマークでの実験により比較した結果，最も目的コードの性能が良いものとして得られたものである[20]．

なお，（通常形式の）最適化の適用順序については，最近では，上のような考察のほかに，膨大な数の最適化列を適用し，実験により効果を確かめるさまざまな研究がなされている．

11. ま と め

従来，静的単一代入形式（SSA 形式）上で部分冗長除去（PRE）を行うことは困難であったが，本研究では PRE を SSA 形式上で実現する手法を提案した．それは，SSA 形式上での変数名の同一性の判断や式の移動などに，CSSA 形式における phi congruence class（pcc）の特徴を利用するというものである．

この手法は汎用的なものであり，今回実装した Lazy code motion 以外の多くの通常形式上の PRE アルゴリズムにも適用可能である．特に，図 1（左）の手順に沿ったアルゴリズムならば，「挿入する式と挿入点の決定」の処理さえ正しく pcc ベースのものに変換できれば，SSA 形式に対応させることができる．本手法を用いれば，新しい PRE アルゴリズムの開発者や，PRE を SSA 最適化列に組み込みたいコンパイラの開発者が容易に SSA 形式上での PRE を実現できるようになる．

謝辞 本研究の一部は日本学術振興会科学研究費補助金の補助を受けた．

参 考 文 献

1) Alpern, B., Wegman, M.N. and Zadeck, F.K.: Detecting equality of variables in programs, *POPL '88: Proc. 15th ACM SIGPLAN-*

SIGACT symposium on Principles of programming languages, pp.1–11, ACM Press, New York, NY, USA (1988).
2) Appel, A.W. and Palsberg, J.: *Modern Compiler Implementation in Java*, 2nd ed, Cambridge University Press (2002).
3) Bodík, R., Gupta, R. and Soffa, M.: Complete removal of redundant expressions, *SIGPLAN Not.*, Vol.39, No.4, pp.596–611 (2004).
4) Briggs, P. and Cooper, K.D.: Effective partial redundancy elimination, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.159–170 (1994).
5) Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N. and Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.4, pp.451–490 (Oct. 1991).
6) Dhamdhere, D.M.: A fast algorithm for code movement optimisation, *SIGPLAN Not.*, Vol.23, No.10, pp.172–180 (1988).
7) Dhamdhere, D.M.: Practical adaption of the global optimization algorithm of Morel and Renvoise, *ACM Trans. Prog. Lang. Syst.*, Vol.13, No.2, pp.291–294 (1991).
8) Dhamdhere, D.M.: E-path_pre: Partial redundancy elimination made easy, *SIGPLAN Not.*, Vol.37, No.8, pp.53–65 (2002).
9) 伊藤　陽，佐々政孝：実行時情報を利用した部分冗長除去とSSA形式への適用，日本ソフトウェア科学会第8回プログラミングおよびプログラミング言語ワークショップ (PPL2006) 論文集，pp.170–181 (2006).
10) Kennedy, R., Chan, S., Liu, S., Lo, R., Tu, P. and Chow, F.: Partial redundancy elimination in SSA form, *ACM Trans. Prog. Lang. Syst.*, Vol.21, No.3, pp.627–676 (1999).
11) Knoop, J., Rüthing, O. and Steffen, B.: Lazy code motion, In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp.224–234, San Francisco, CA (June 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 (July 1994).
13) Morel, E. and Renvoise, C.: Global optimization by suppression of partial redundancies, *Commun. ACM*, Vol.22, No.2, pp.96–103 (1979).
14) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
15) Paleri, V.K., Srikant, Y.N. and Shankar, P.: A simple algorithm for partial redundancy elimination, *SIGPLAN Not.*, Vol.33, No.12, pp.35–43 (1998).
16) 佐々研究室：静的単一代入形式最適化システム外部仕様書 (2006). http://www.is.titech.ac.jp/~sassa/coins-www-ssa/japanese/ssa-external-japanese.pdf.
17) Sreedhar, V.C., Ju, R.D., Gillies, D.M. and Santhanam, V.: Translating out of static single assignment form, In *SAS '99: Proc. 6th International Symposium on Static Analysis*, pp.194–210, Springer-Verlag, London, UK (1999).
18) 立川　英：静的単一代入形式上の部分冗長除去，修士論文，東京工業大学大学院情報理工学研究科数理・計算科学専攻 (2004).
19) 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).
20) 佐々政孝，福岡岳穂，滝本宗宏：コンパイラ・インフラストラクチャにおける静的単一代入形式最適化部の実現，情報処理学会論文誌：プログラミング，Vol.47, No.SIG 2 (PRO 28), pp.30–43 (2006).

(平成19年 7月 9日受付)
(平成19年10月16日採録)

今橋　孝典

1984年生．2006年東京工業大学理学部情報科学科卒業．同年同大学大学院情報理工学研究科数理・計算科学専攻入学．コンパイラのプログラム最適化に興味を持つ．

伊藤　　陽

1981年生．2004年東京工業大学理学部情報科学科卒業．2006年同大学大学院情報理工学研究科数理・計算科学専攻修了．同年NEC入社，現在に至る．

佐々 政孝（正会員）

1948 年生．1970 年東京大学理学部物理学科卒業．1974 年同大学院博士課程中退，東京工業大学理学部情報科学科助手．1981 年筑波大学電子・情報工学系．1992 年東京工業大学理学部．現在同大学大学院情報理工学研究科数理・計算科学専攻教授．理学博士．プログラミング言語，コンパイラ，プログラミング環境に興味を持つ．著書『プログラミング言語処理系』（岩波書店）．日本ソフトウェア科学会，ACM，IEEE 各会員．