静的単一代入形式を用いた最適化（発展編）

滝本 宗宏　佐々 政孝

コンパイラでは，機械語の目的コードを生成するに際して，実行させたときにその目的コードが効率良く実行できるように，様々な変換を行う．これを「最適化」という．最適化の方法としては，従来はデータフロー解析と呼ばれる方法が使われていたが，最近は静的単一代入形式というものを用いた最適化の方法が注目を浴びている．静的単一代入形式（SSA 形式）は，すべての変数の使用に対して，その値を定義（代入）している場所がテキスト上 1 箇所しかないように変数の名前替えをした中間表現の形式である．この性質を利用することにより，いろいろな最適化が見通し良く，容易にできるようになる．これを静的単一代入形式最適化と呼ぶ．本稿（発展編）では，静的単一代入形式最適化のあらましについて，解説する．

Compilers not only generate target code but also apply to a program a lot of transformations, which are called code optimizations, because of efficient execution of the target code. Though, traditionally, most of them had been implemented using dataflow analysis, the techniques based on Static Single Assignment (SSA) form have been familiar these days. SSA form is an intermediate representation where each variable lexically has only one assignment through an entire program. Such a property makes traditionally complex code optimizations to be implemented easily. We call the code optimizations implementend based on SSA form SSA-optimization. In this paper, we are providing brief explanations of them.

1　はじめに

コンパイラでは，機械語の目的コードを生成するに際して，実行させたときにその目的コードが効率良く実行できるように，様々な変換を行う．これを「最適化」という．たとえば，ループの中で実行しなくても良い命令はループの外に出す，同じ計算は省略する，などの変換を行って目的コードの効率を向上させる手法がよく知られている．最適化は，現在のコンパイラで力が注がれている重要な技術の一つである．

最適化の方法としては，従来はデータフロー解析と呼ばれる方法が使われていたが，最近は静的単一代入形式を用いた最適化の方法が注目を浴びている．

静的単一代入形式（Static Single Assignment Form，以下 SSA 形式と呼ぶ）は，すべての変数の使用に対して，その値を定義（代入）している場所がプログラム上 1 箇所しかないように変数の名前替えをした中間表現の形式である．通常の中間表現では変数の使用に対してその値を定義している場所が複数個あり得るのだが，SSA 形式では，各変数の定義がプログラム上で 1 箇所しかない．この性質を利用することにより，いろいろな最適化が見通し良く，容易にできるようになる．実際の例は 2 節以降で述べる．

SSA 形式を利用した最適化を静的単一代入形式最適化（SSA 形式最適化）という．SSA 形式最適化では最適化の処理時間も多くの場合に短くなる．このためいくつかの最適化コンパイラ [15] [19] [16] [25] がその一部のパスで SSA 形式を採用するようになってきている．我々も，COINS [8] の一部として SSA 形式最適化を実装した．

Optimization in static single assignment form- Deployment Part

Munehiro Takimoto, 東京理科大学理工学部, Faculty of Science and Technology, Tokyo University of Science.

Masataka Sassa, 東京工業大学大学院情報理工学研究科, Graduate School of Information Science and Engineering, Tokyo Institute of Technology.

コンピュータソフトウェア, Vol.16, No.5 (2000), pp.78–83.
[解説論文] 2000 年 8 月 3 日受付.

導入編[32]では，SSA形式，SSA形式への変換と逆変換について述べたので，本稿（発展編）では，SSA形式上での最適化について，例を挙げながら述べる．

2 SSA形式最適化

この節ではいくつかの例を用いてSSA形式最適化の方法とその特徴を説明する．

2.1 共通部分式除去

ソースプログラムとして，たとえば

```
x = a + b;      // (1)
  ...
y = (a + b) * c;      // (2)
```

があったとき，

- (1) と (2) の「a + b」は同じ形の式であること
- (2) の計算の前に必ず (1) の計算が行われていること
- (1) の計算をしてから (2) の計算をするまでに a, b の値が変わらないこと

が確認できるとき,「a + b」を「共通部分式」という．(後の3節も参照)

このとき，(1) と (2) の間で x の値が変わらなければ，(2) を

```
y = x * c;
```

で置き換えることができる．この最適化を「共通部分式除去」という[37].

SSA形式における共通部分式除去[30]ではより高度な最適化が行える．その例として，図1(a)のソースプログラムを用いる．これをSSA形式中間表現に変換すると，図1(b)となる．SSA形式では，変数の定義が唯一になるように，a_1, b_1 のような下添字のバージョン名というものをつけることが多いが，以下では，SSA形式の変数のバージョン名は下添字でなくa1, b1等で示す．また本来はSSA形式は中間表現であるが，読みやすさのためソースプログラムの形式で示す．

図1(b)の上で共通部分式の解析を行うと，同図の「//」以降に書いた解析結果が得られる．たとえば，if文中の「z1 = a1 + b1」では，右辺の「a1 + b1」が上で計算した「y1 = a1 + b1」の右辺と「字面」が同じであるので，共通部分式であることがわかり，z1はy1と等しいことが分かる．従来の通常形式による方法では，字面が等しいだけでは共通部分式を認識できず，その2つの式の間でそれらのオペランドの値が変わらないことなどを解析する必要があるが，SSA形式は単一代入なので，字面が等しければ値が等しいことがすぐに分かるのが特長である．同様にして，z2もy1と等しいことが分かる．すると,「z3 = ϕ(z1,z2)」において，z1とz2はともにy1と等しかったので，z3がy1と等しいことが分かる．その次の行の「d1 = x1 + z3」では， z3がy1と等しかったので，d1は「x1 + y1」と等しいことがわかり，これは上で計算した「c1 = x1 + y1」の右辺なので，結局d1はc1と等しいことが分かる．

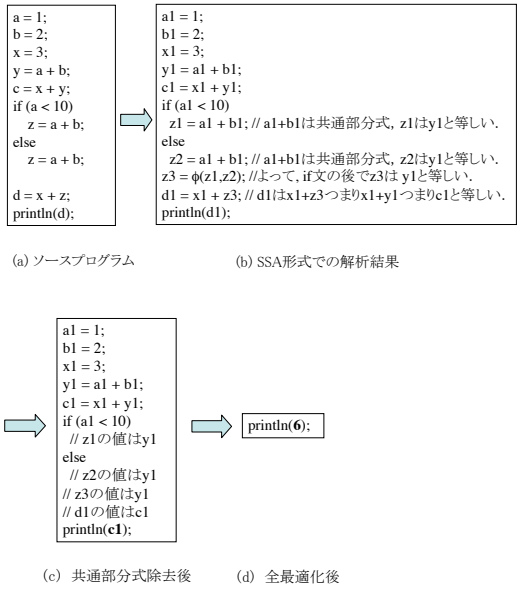


図1 共通部分式除去

これらを用いて共通部分式除去を行うと，図1(c)が得られる．SSA形式共通部分式除去のアルゴリズム[30][37]では，代入文の右辺が共通部分式であることが分かると，まず「その左辺の変数の値がその共通部分式を定義した変数にすでに格納されていることを表に登録して」，この文を除去する．たとえば図1(c)の例では,「z1 = a1 + b1」を見ると，z1の値は「a1 + b1」を計算したy1と同じなので,「z1の値がy1に

格納されていることを表に登録して」，その文「z1 = a1 + b1」を除去する．同様にして，「z2 の値が y1 にある」ことを表に登録して「z2 = a1 + b1」も除去する．以下同様に，「z3 の値が y1 にある」ことを表に登録して「z3 = ϕ(z1,z2)」も除去する．さらに，「d1 の値が c1 にある」ことを表に登録して「d1 = x1 + z3」も除去する．最終行の d1 は，表を見ると「d1 の値が c1 にある」と書いてあるので，c1 に置き換える．

図 1(c) にさらに定数伝播やコピー伝播，無用命令除去など SSA 形式最適化で行っている全最適化（後述の 2.2 節「条件分岐を考慮した定数伝播」や 4 節「COINS における SSA 形式最適化モジュール」参照）をかけると，図 1(d) が得られ，最終的に 1 行となる．

2.2 条件分岐を考慮した定数伝播

ソースプログラムとして，たとえば

```
x = 3;   // (1)
  ...
y = x;   // (2)
```

があったとする．(1) と (2) の間で x の値が変更されていない，などいくつかの条件が満たされるとき，(2) を

```
y = 3;
```

に置き換える最適化を「定数伝播」という．

SSA 形式における「条件分岐を考慮した定数伝播」[28] [3] [37] [30] では，より高度な最適化が行える．その例として，図 2(a) のソースプログラムを用いる．これは [3] の図 19.4 にある例である．

これを SSA 形式に変換すると，図 2(b) が得られる．図 2(b) の SSA 形式プログラムに対して，条件分岐を考慮した定数伝播の解析を行うと，図 2(c) の情報が得られる．この解析の特徴は，定数である可能性があればさしあたり定数だと「楽観的」にみなしておき，その後，定数でないことがはっきりするまではそのまま定数だとみなすことである．このようなアルゴリズムは，否定されない限りは都合の良いように考えておくので，「楽観的なアルゴリズム」と言われる．たとえば，図 2(b) の「j2=ϕ(j1, j5)」を初めて調べるときは，j1 の値が 1 なので，さしあたり j2 の値も 1 だとみなす．「j3=i1」を調べるときも，i1

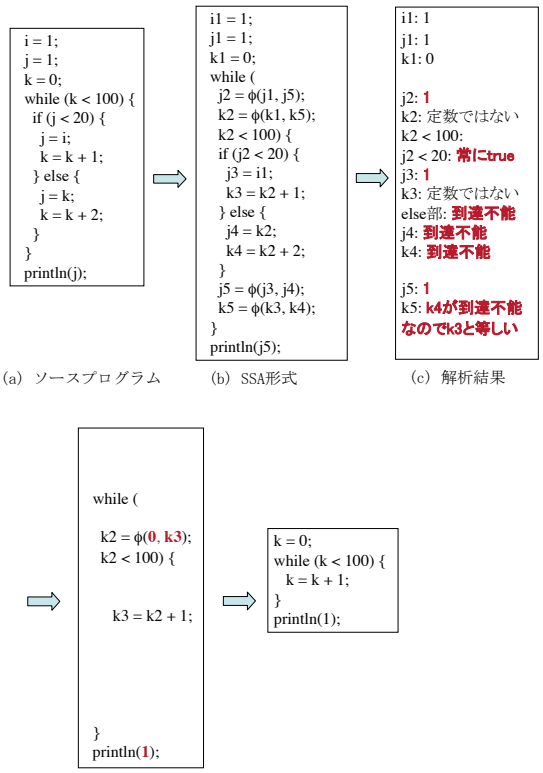


図 2 条件分岐を考慮した定数伝播

の値が 1 なので，j3 の値もさしあたり 1 だとみなす．同様に，「j5=ϕ(j3, j4)」を調べるときも，j3 が 1 なので，j5 も 1 だとみなす．

ループがあるので，その後ふたたび「j2=ϕ(j1, j5)」を調べることになるが，運良く j1 も j5 も 1 なので，最初の楽観的予想どおり，j2 は 1 とみなしておいて良かった，となる．j2 が 1 なので，「j2<20」はいつも true だと分かるので，else 部は到達不能だと分かる．このような解析を，仮にみなしておいた値が変化しなくなるまで繰り返す．結果として，

- j2, j3, j5 は常に 1
- 「j2<20」は常に true であるので，else 部には到達できない
- k4 への代入文が到達不能なので，k5 は k3 と等しい

であることが分かる．このような解析も，SSA 形式

で値が単一代入だからこそ容易である．

図 2(c) の解析結果を使い，図 2(b) の SSA 形式を最適化すると図 2(d) が得られる．

COINS での条件分岐を考慮した定数伝播の最適化は他の最適化も同時に行っているので，図 2(d) は次の最適化を行った結果である．(i) 定数であることが分かった変数は，その値を記憶して，その変数への代入文を除去し，その変数の使用は記憶しておいた定数で置き換える，(ii) ある変数 x の値が別の変数 y の値と等しいことが分かったら，x への代入文は除去し，以後の x の使用を y で置き換える，(iii) true あるいは false であることが分かった条件式の評価は省略する，(iv) 到達不能なコードを除去する．

図 2(b) の例で説明すると，図 2(c) の解析の結果，i1, j1, k1, j2, j3, j5 は定数であることがわかったので，図 2(d) ではそれらの変数への代入文は除去する．また，k1 の使用は 0 で置き換え，j5 の使用は 1 で置き換える．k5 への代入文は除去し，k5 の使用は k3 で置き換える．if 文の条件式は常に true なので else 部は除去する．詳しいアルゴリズムは [30] を見られたい．SSA 形式で行う最適化はここまでであるが，これを通常形式に戻すと，図 2(e) が得られる．

2.3 ループにおける演算の強さの軽減と判定の置き換え

「ループにおける演算の強さの軽減と判定の置き換えの最適化」とは，ループ内の帰納変数について，新たな帰納変数を導入し，乗算を加算で置き換え，終了判定を新たな帰納変数を用いて行い，元の帰納変数を除去するよう変換するものである [9] [30] [37]．ここで帰納変数とは，ループの制御変数のように，ループを回るごとに一定の値だけ変化する変数のことである．この最適化により，配列参照を含むループの実行時間を短縮することができる．

図 3，図 4，図 5 に演算の強さの軽減と判定の置き換えの例を示す．

図 3(a) は 2 つの配列 x, y を足して z にしまうソースプログラムの例である．図 3(b) は，その中心部分の SSA 形式中間表現である．(見やすさのため，C 言語風に示す，以下同じ．)「temp4i = 4 * i2」により

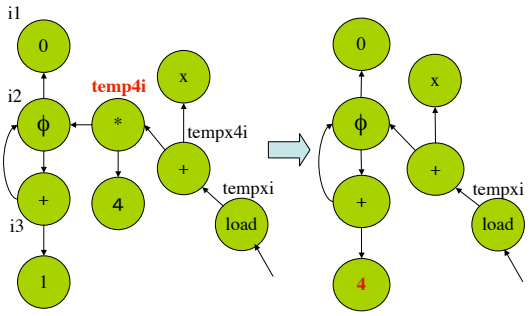
```
/* addvectosr.c -- add vector
for osr example */
void addvect
(int z[], int x[], int y[], int n) {
  int i;
  for (i = 0; i < n; i++) {
    z[i] = x[i] + y[i];
  }
}
```

(a) ソースプログラム

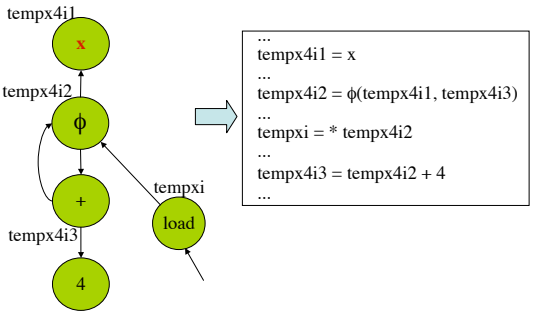
```
...
i1 = 0
L1:
  i2 = ϕ(i1, i3)
  if (i2 >= n1) goto L6
  temp4i = 4 * i2
  tempx4i = x + temp4i
  tempxi = * tempx4i
  ...
  i3 = i2 + 1
  goto L1
L6:
...
```

(b) 中間表現 (C言語風)

図 3　演算の強さの軽減と判定の置き換えのソースプログラム例



(a) 最初のSSAグラフ　(b) SSAグラフの変形 (その1)



(c) SSAグラフの変形 (その2)　(d) SSA形式中間表現に戻したもの

図 4　SSA グラフの変形

添字 i に配列要素の大きさ 4 を掛ける．「tempx4i = x + temp4i」により，x[i] の番地を計算する．そして，「tempxi = * tempx4i」により，x[i] の値を取り出している．

図 4(a) は，図 3(b) の SSA 形式を SSA グラフとい

う形式で表現したものである．SSA グラフは，ノードと有向辺からなる．ノードの中身は演算子，ノードにつけられたラベルは変数，有向辺は演算子のオペランドを表す．たとえば，SSA 形式「temp4i = 4 * i2」に対しては，temp4i というラベルがつけられたノードが取られ，そのノードの中身は演算子「*」で，オペランドである「4」と「i2」を表すノードへ有向辺（矢印）が張られている．また，「tempxi = * tempx4i」に対応するのは，tempxi というラベルがつけられたノードで，中身の「load」は C 言語でポインタの内容を取り出す演算子「*」を表し，オペランドである tempx4i へ有向辺が張られている．

図 4(b) は，図 4(a) で temp4i より左の部分を等価なグラフに変形したものである．temp4i のノードの演算は左の列のオペランドに 4 を掛けることであったので，図 4(a) の左の列の一番下のノードの 1 を 4 倍した 4 に置き換えることにより（左の列の一番上のノードの 0 は 4 倍しても変わらない），左の 2 列を図 4(b) の左の 1 列のように置き換えることができる．これにより，4 による乗算がなくなったので，演算の強さが軽減された．

さらに，図 4(b) のまん中の列を見ると，これは，左の列のオペランドに x を足す演算である．そこで，同様にしてこの左の 2 列もさらに置き換えができて，図 4(c) の左の 1 列が得られる．

この SSA グラフから SSA 形式を再構成すると，図 4(d) となる．ラベルづけられたノードごとに 1 つの文が作られる．文の順番は，オペランドを先に，演算子を後にする．

詳細は省略するが，以上の処理を，x[i], y[i], z[i] について行うと，図 5(b) が得られる．図 5(b) では，配列参照における 4 による乗算が消えていることが見てとれる．（これも本当は SSA 形式中間表現であるが，ϕ を省略して C 言語風に表した．）また，ここでは，ループ終了判定を i で行っていたものを temp4i を用いて行うような変形もしている．このように，ループの終了判定の変数を変えることをループの「判定の置き換え」という．

図 5(b) に対し，さらに COINS で用意しているすべての SSA 形式最適化を適用すると，図 5(c) と

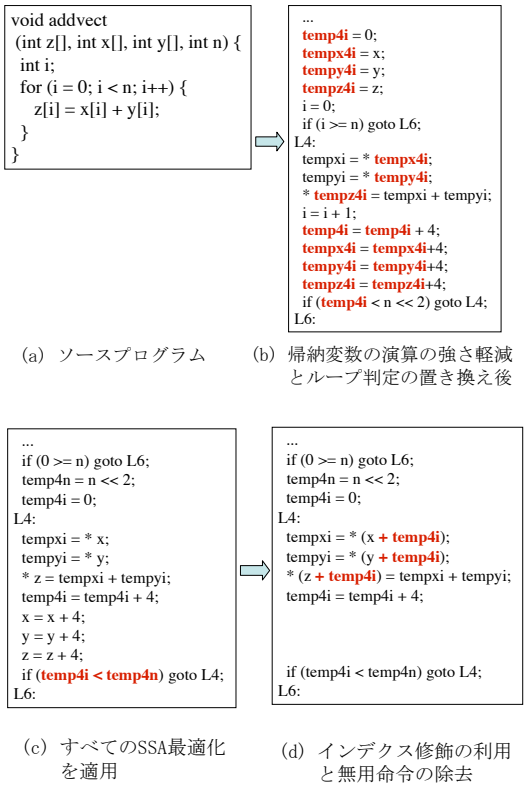


図 5 演算の強さの軽減と判定の置き換え後

なる．ここでは，コピー伝播により，tempx4i は x に，tempy4i は y に，tempz4i は z に置き換えられている．もはや使われなくなった i は除去されている．「n<<2」はループ不変式なので，ループの前に移動されて「temp4n = n << 2」が挿入され，「n << 2」は temp4n で置き換えられている．

これで変形は一段落であるが，SPARC のような，2 つのレジスタを用いたインデクス修飾ができるアーキテクチャでは，さらに最適化ができて，図 5(d) のようなコードが得られる．ただし，このような最適化は機械に依存する．

3 コード移動に基づく冗長除去

2.1 節で，冗長性を除去するコード最適化として，共通部分式の除去を紹介した．ここで，共通部分式の定義を確認すると，共通部分式 e の出現 e_c が除去可能であるとは，次の 2 つの**除去可能条件**を満たすこ

```
        ...
1: x = a+b;
2: if (p) y = a+b;
        ...
```

(a) 支配関係に基づく共通部分式

```
        ...
1: if (p) x = a+b;
2: else y = a+b;
3: q = a+b;
        ...
```

(b) 一般的な共通部分式

図 6　共通部分式

```
        ...
1: if (p) x = a+b;
2: else ;
3: y = a+b;
4: func(y);
        ...
```

(a) 部分冗長な式

```
        ...
1: do {
2:     x = a+b;
3: } while (p);
        ...
```

(b) ループ不変式

図 7　部分冗長

とを言う.

1. プログラムの開始点から，e_c までのすべての実行パス P 上に，同じ式 e の出現 e_i が存在する.
2. P の各実行パス上で，e_i と e_c との間に，e のオペランドを変更する文が存在しない.

通常形式の 2 つのプログラム図 6 において（以降，3.3 節までは，SSA 形式でなく通常形式で説明する.），図 6(a) 2 行目の `a+b` や，図 6(b) 3 行目の `a+b` は，いずれも除去可能な共通部分式である．このうち，図 6(a) で示した共通部分式は，除去可能な 2 行目の式が，1 行目の式に支配される関係にある．このように，除去可能条件の 1 は，e_c を支配する e_i が存在するという条件に置き換えることができる．支配関係は，支配木を用いて容易に検査できるので，効率的な共通部分式の除去が可能である [2] [7]．2.1 節で述べた共通部分式の除去法も，この支配関係の情報を用いている．また，SSA 形式を対象とする共通部分式の除去は，除去可能条件の 2 を考慮する必要がないので，実現も容易である.

一方，支配関係に基づく共通部分式の除去は，図 6(b)3 行目のように，除去可能な e_c が e_i に支配されない場合を扱うことができない．一般に，共通部分式を発見するためには，データフロー解析のような，制御フローに基づいた解析が必要である.

3.1　部分冗長な式

次に，共通部分式以外の冗長な式について考えよう．図 7(a) で，`p` が *true* の場合，if 文の then 部を通る実行パス上で，3 行目の `a+b` は冗長である．一方で，`p` が *false* の場合，`a+b` は 1 度しか実行されないので，冗長ではない．このように，少なくとも 1 つの実行パス上で冗長な式を**部分冗長**（partially redundant）であると言う．また，部分冗長な式のうち，全ての実行パス上で冗長な式を，特に**全冗長**（totally redundant）であると言う．ここで，共通部分式は，全冗長な式である.

```
        ...
1: if (p) {
2:     x = a+b;
3:     y = a+b;
4:     func(y);
5: }
6: else {
7:     y = a+b;
8:     func(y);
9: }
        ...
```

(a) 制御構造の変形

```
        ...
1: if (p) {
2:     x = a+b;
3:     y = x;
4:     func(y);
5: }
6: else {
7:     y = a+b;
8:     func(y);
9: }
        ...
```

(b) 冗長除去後のプログラム

図 8　制御構造の変更による部分冗長式の除去

部分冗長性について，別の例を示そう．図 7(b) は，ループの中に，ループ不変式が存在するプログラムを示している．2 行目の `a+b` は，do-while 文の 3 行目から 1 行目に戻る実行パス上で，それ自身に対して冗長と見なすことができる．一方，ループの外から本体に入る実行パス上では冗長でないので，`a+b` は部分冗長である.

部分冗長な式を単純に除去すると，冗長性が存在しなかった実行パスから，式の計算が無くなってしまう可能性があるので，共通部分式のように容易に除去することができない.

3.2　通常形式における部分冗長な式の除去

部分冗長な式を除去するには，プログラムの制御構造を変更する方法 [4] と，**コード移動**（code motion）を行う方法 [22] がある．図 7(a) を例に考えると，制

```
        ...
1: if (p) {
2:     t = a+b;
3:     x = a+b;
4: }
5: else t = a+b;
6: y = t;
7: func(y);
        ...
```

(a) 巻上げ後のプログラム

```
        ...
1: if (p) {
2:     t = a+b;
3:     x = t;
4: }
5: else t = a+b;
6: y = t;
7: func(y);
        ...
```

(b) 冗長除去後のプログラム

図 9 コード移動による部分冗長式の除去 (1)

```
        ...
1: t = a+b;
2: do {
3:     x = a+b;
4: } while (p);
        ...
```

(a) 巻上げ後のプログラム

```
        ...
1: t = a+b;
2: do {
3:     x = t;
4: } while (p);
        ...
```

(b) 冗長除去後のプログラム

図 10 コード移動による部分冗長式の除去 (2)

御構造を変更する方法では，3 行目の部分冗長な式を含む基本ブロックを複製し，if 文の then 部と else 部の下方に挿入する．結果として，図 8(a) に示すように，部分冗長であった `a+b` が，全冗長になる．この全冗長性は，図 8(b) に示すように，共通部分式の除去を適用することによって，除去することができる．制御構造を変更する方法は，複数の基本ブロックを複製する必要があるので，一般にプログラムコードが増加する傾向がある．図 8(b) では，`func(y)` が複製されている．

コード移動を行う方法では，部分冗長な式をプログラムの実行と逆向き（以降，**後向き**と呼ぶ）に移動させる．後向きのコード移動は，特に**巻上げ**と呼ばれる．図 9(a) は，図 7(a) の 3 行目を，if 文の then 部と else 部へ巻き上げた結果を表している．ここで，部分冗長だった式は全冗長になるので，図 9(b) に示すように，除去することができる．

図 7(b) も，2 行目の `a+b` と部分冗長の関係にあるそれ自身とを別物と考えると，同様に扱うことができる．2 行目の `a+b` をループの上に巻き上げると，

```
        ...
1: if (q) x = a+b;
2: else y = a+b;
        ...
```

(a) 元のプログラム

```
        ...
1: t = a+b;
2: if (q) x = t;
3: else y = t;
        ...
```

(b) 巻上げ後のプログラム

図 11 安全な巻上げ

ループ内の `a+b` は，図 10(a) に示すように，全冗長になる．続いて，共通部分式の除去を適用すると，図 10(b) に示すように，式 `a+b` のループ外移動が実現できる．

このように，巻上げに基づいて部分冗長性を除去する方法を**部分冗長除去**（partial redundancy elimination，以降 PRE と呼ぶ）[22] と言う．PRE を適用すると，図 9(b) の `x = t` や `y = t`，図 10(b) の `x = t` のように，コピー代入が生成される．このようなコピー代入は，コード最適化の効果を減じる可能性があるので，**コピー伝播**（copy propagation）を適用して除去しておくのがよい． また，多くのコピー代入は，レジスタ割付けの際に行う**変数合併**（coalescing）によって，取り除かれることが期待できる．

制御構造の変更に基づいた方法は，基本ブロックを複製する問題があるので，PRE ほど，頻繁に利用されることはないが，PRE で除去できない冗長性を扱うことができる．詳しくは，文献 [4] を参照されたい．

3.3 巻上げの条件

PRE は，最適化前に式 e が出現した実行パス上で必ず e が実行されることを保証するために，e の出現場所から始めて，隣接するすべてのプログラム点へ巻き上げることを繰り返す．

式の巻上げのようなデータのフローは，基本ブロックの出口の情報が入口に到達するかどうかどうか（あるいは，入口の情報が出口に到達するかどうか）を，前もって計算しておくことによって，基本ブロックごとに扱うことができる．すなわち，データのフローを扱う問題は，基本ブロックを節，基本ブロック間の制御の移動を有向辺とした**制御フローグラフ**（control flow graph，以降，CFG と呼ぶ）上で解くことがで

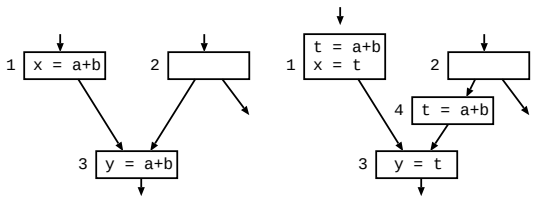


図 12　クリティカル辺の除去

```
        ...
1: while(p) {
2:      x = a+b;
3: }
        ...
```

(a) while ループ

```
        ...
1: if (p) {
       t = a+b;
2:     do {
3:          x = t;
4:     } while (p);
5: }
        ...
```

(b) 変形後の do-while ループ

図 13　while ループの変形

きる．ここで，CFG 節 $v1$，$v2$ として，辺 ($v1$,$v2$) が存在するなら，$v1$ は $v2$ の先行節と呼び，$v2$ は $v1$ の後続節と呼ぶ．以降の説明では，必要に応じて CFG を利用する．

PRE の巻上げを，CFG 上で考えると，節 v に巻き上げられた式 e は，さらに，v のすべての先行節へ巻き上げる必要がある．しかし，何も考慮せずに巻上げを行うと，最適化効果を減じる可能性がある．以降で，巻上げの際に考慮すべき点を述べる．

3.3.1　下向き安全

例えば，プログラム片 `if (p) x = a+b` の 式 `a+b` を，if 文の上に巻き上げると，`p` が $false$ になる実行パスに，新しく `a+b` の計算を導入することになり，実行効率を減じる可能性がある．このような問題を生じないように，コード移動に基づくコード最適化は，あるプログラム点 p への巻上げについて次の条件を満たすことが必要である．

- p からプログラムの終了点までのすべての実行パス上に，巻上げ対象の式の出現が存在する．

この条件は，**下向き安全性**（down-safety）[21] と呼ばれる．また，プログラム点 p が下向き安全性を満たすとき，p は下向き安全であるという．図 11(a) では，if 文の上のプログラム点を通るすべての実行パスに 式 `a+b` が出現するので，図 11(b) に示す，if 文の上への巻上げは，下向き安全である．

3.3.2　クリティカル辺と while ループ

次に，下向き安全の条件によって，有用な冗長性の除去が妨げられる場合を述べよう．例えば，図 12(a) の CFG において，節 3 の `a+b` は部分冗長である．そこで，この部分冗長性を除去するために，`a+b` を節 1 と節 2 に巻き上げると，節 2 から右下へ向かう実行パスに，新しく式を導入してしまう．すなわち，節 2 は，下向き安全でないので，巻上げは許されない．このように，2 つ以上の後続節をもつ節から 2 つ以上の先行節をもつ節への辺は，**クリティカル辺**（critical edges）[21] と呼ばれ，有用な巻上げを妨げる場合がある．図 12(a) では，節 2 から節 3 へ向かう辺がクリティカル辺である．クリティカル辺は，図 12(b) に示すように，新しい節（図の節 4）で分割することによって，除去することができる．以降，説明を簡単にするために，クリティカル辺は除去されているものとする．

また，ループ不変式のループ外への移動が妨げられる場合もある．図 13(a) に示す while ループには，ループヘッダからループ外への分岐が存在する．すなわち，2 行目を通らずに終了節に到達するパスが存在するので，ループの上方への巻上げは，下向き安全でない．このようなループは，図 13(b) に示すように，while の継続条件 `p` と同じ条件の if 文を導入することによって，do-while 型のループに変形することができる．do-while ループ内からの上方への巻上げは，下向き安全である．

3.3.3　不要な巻上げ

次に，不要な巻上げが，最適化効果を減じる可能性について述べる．図 14(a) 7 行目の `a+b` は，6-8 行目のループ（以降，ループ 1 と呼ぶ）から不変コードとしてループの外に巻き上げられたあと，1 行目の `a+b` に対する部分冗長性から，さらに，1 行目への巻上げを試みる．しかしながら，1 行目 の then 部と else 部は，`a+b` について下向き安全でないので，巻き上げた

```
        ...
1: if (p) x = a+b;
2: if (q) {
3:     do {
4:          ...
5:     } while (r);
6:     do {
7:        y = a+b;
8:     } while (s);
9: }
        ...
```

(a) 元のプログラム

```
        ...
1: if (p) x = a+b;
2: if (q) {
3:     t = a+b;
4:     do {
5:          ...
6:     } while (r);
7:     do {
8:        y = t;
9:     } while (s);
10: }
        ...
```

(b) 不要な巻上げ

図 14 不要な巻上げ

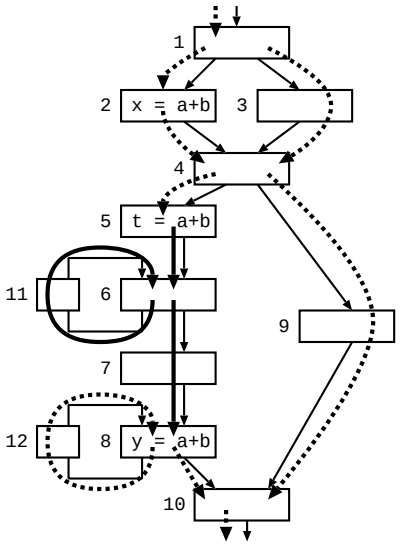


図 15 巻戻しの解析

式を，適切なプログラム点まで元に戻さなければならない（以降，巻き上げを元に戻すことを**巻戻し**と呼び，巻戻しを行うことを**巻き戻す**と言う）．この際，3-5 行目のループ（以降，ループ 2 と呼ぶ）内へ巻き戻したとすると，ループ不変式をループの外へ出す効果に反するので，図 14(b) に示すように，3 行目へ巻き戻すことが考えられる． ここで，`a+b` の巻上げは，ループ 1 の上へ行われれば十分なので，そのあとの 3 行目への巻上げは，不要な巻上げである．このような不要な巻上げは，巻き上げた式の値を使用に運ぶ変数の生存期間を長くし，変数がレジスタに割り付けられる機会を減ずる可能性がある．

実際，不要な巻上げは，1979 年に Morel と Renvoise が初めて PRE を発表 [22] して以来，初期の PRE において主要な問題であった†1 [11] [12] [22]．この問題は同様に主要な問題であった計算量の多さの問題とともに，1992 年に Knoop らの**なまけたコード移動**（lazy code motion，以降 LCM と呼ぶ）[21] によって，解決方法が示された． その後，さまざまな種類の PRE が提案されているが，現在，不要な巻上げを行わないことは，PRE の特徴の 1 つになっている．以降で，LCM が，どのように不要な巻上げを回避するのか述べよう．

LCM では，巻上げと別に，巻戻しの解析を行う．1 つの文を節とする CFG 上で，式 e を節 n の出口に巻き戻せるかどうかを，述語 $delay_n$ で表すことにすると，巻戻しは，おおよそ次のような手順になる．詳しくは文献 [21] を参照されたい．

1. $delay_n$ を節 n の種類によって，次のように初期化する．
   - 開始節： $false$
   - e が巻き上げられた節： $true$
   - 元から e が出現していた節： $false$
   - e のオペランドへの変更がある節 ：$false$
   - その他の節： $true$
2. 各節の値を後続節に伝え，後続節の値と論理積を計算し，その結果を元の値と置き換えていく．
3. すべての節について，述語の値が変わらなくなるまで，ステップ 2 を繰り返す．

最終的に，$delay_n = true$ である節 n に巻き上げられた式は，$delay_{n'} = false$ である節 n' の手前まで巻戻す．

以降，ある節の情報が，先行節あるいは後続節へ順に伝わることを**伝播**（propagation）と呼ぶ．

図 15 は，図 14(b) の巻戻しの解析結果を示している．CFG に重ねて示した太線は，伝播の様子を表しており，実線は $true$ の伝播，点線は $false$ の伝播を意味する．巻き上げられた節 5 から前向きに伝播し

†1 Morel と Renvoise のオリジナルのアルゴリズムでは，ループ 2 が存在しなくても，1 行目の下まで巻き上げられる．

```
        ...
1: x = a+b;
2: y = a+b;
3: p = x*2;
4: q = y*2;
        ...
```

(a) 元のプログラム

```
        ...
1: x = a+b;
2: y = x;
3: p = x*2;
4: q = y*2;
        ...
```

(b) 新しい冗長性の発見

図 16　冗長除去の副次的効果

た *true* は，節 6 と節 11 のループ 2 を通過し，節 8 まで到達する．節 8 は，元から式が出現していることから前述の初期化によって $false$ なので，$false$ の値が，ループ 1 の戻り辺を辿って，ループヘッダへ到達する．結果として，ループヘッダは $false$ になり，巻戻しは，ループ 1 のヘッダの先行節までとなる．

一連の巻き戻しで，不要な巻上げを行ったループについては巻き戻すが，ループ内に元からあった式のループ外への巻上げは，巻き戻さないことが分かる．

3.4　SSA 形式に基づく冗長除去

SSA 形式のプログラムから冗長性を除去する方法には，大きく分けて 2 種類ある．1 つは，SSA 形式上で PRE のようなデータフロー解析を行うものであり，もう 1 つは，**大域値番号付け**（global value numbering，以降 GVN と呼ぶ）[24] に基づくものである．

3.4.1　SSA 形式に基づく部分冗長除去

データフロー解析を用いる方法は，冗長除去の効果を向上させる目的をもつものと，解析効率を向上させる目的をもつものとに分けられる．効果の向上を目的とするものとしては，各式が移動しやすいようにプログラム中の依存構造を変形したのち，PRE を適用する手法 [5] が挙げられる．しかし，その多くは，SSA 形式の定義と使用を直接結びつける性質を用いたり [20]，支配辺境を利用しながら同じ値を生成する式同士を SSA 形式のように関連づけたりして，PRE のデータフロー情報を必要なプログラム点だけに伝播させ，解析効率を向上させることを目的としている [6] [14].

3.4.2　大域値番号付け

冗長性を除去すると，新たな冗長性が見付かる場合がある．これを，冗長性除去の**副次的効果**（second order effects）[26] と言う．例えば，図 16(a) は，共通部分式 `a+b` を含んでいるので，図 16(b) のように除去することができる．さらに，除去によって生じたコピー代入 `y = x` を，4 行目にコピー伝播すると，4 行目は `x*2` となり，3 行目との共通部分式が新たに見付かる．PRE が，副次的効果を反映するためには，PRE とコピー伝播の反復適用が必要である [26]

この副次的効果を効率的に反映させることを目的とした手法が，GVN である．元々，値番号付けとは，式と，式の値を保持する変数を管理する表を用いて，「基本ブロック内」の共通部分式を変数で置き換えていく手法である．このとき，変数は，代入が生じるたびに，値番号という唯一の番号を与えて区別する．これに対して，GVN は，SSA 形式の変数名を値番号として利用し，式の管理にはハッシュ表を用いることによって，値番号付けの対象をプログラム全体に拡張したものである．2.1 節で紹介した手法は，GVN で共通部分式の除去を実現した例である．

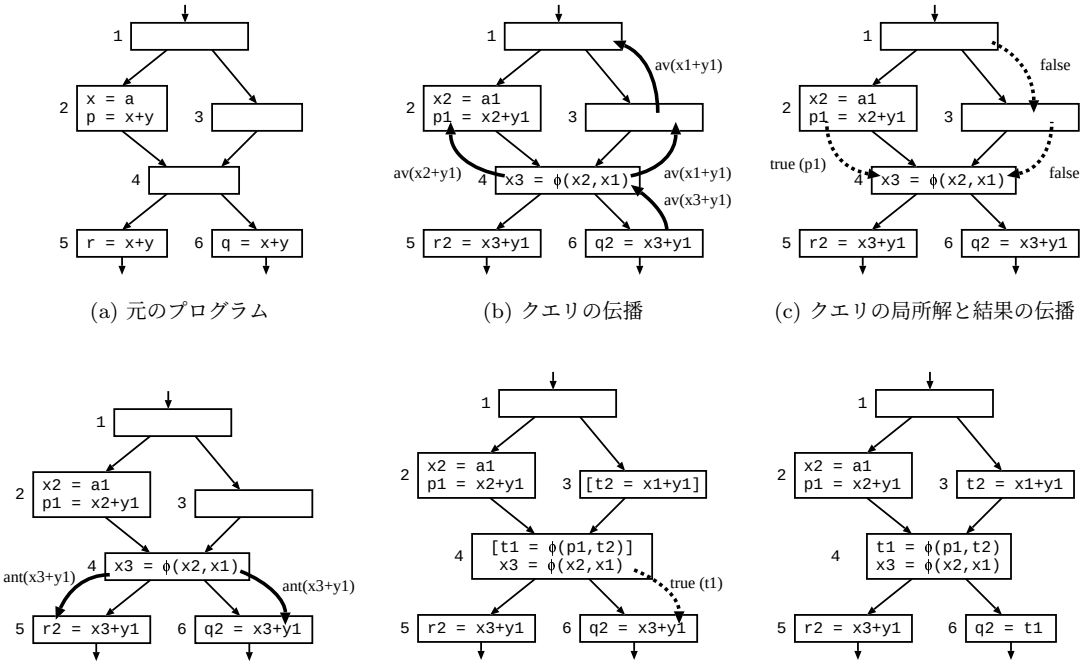
部分冗長性については，初期の GVN から考慮されており，適用できる制御構造に制限があったり，限られた部分冗長性だけを対象にしていたりという制限[†2]があるものの，多くの手法が，データフロー解析を反復適用する方法より解析コストが小さい．

3.4.3　SSA 形式の問題

PRE と GVN のいずれを用いる場合でも，通常形式と比べて，SSA 形式の方が扱いが難しくなる場合がある．

図 17(a) の通常形式で見ると，6 行目の式 `a+b` は，3，5 行目の `a+b` に対して冗長であることが分かる．一方，同じプログラムを，図 17(b) の SSA 形式で見ると，3，5，7 行目は，互いに異なる式になっている．このように，字面上，通常形式では同じなのに，SSA 形式では異なる式を扱う手法は，幾つか提案されている [24] [35] [27] が，アルゴリズムが複雑になる傾向がある．

†2　最近では，PRE と GVN の両方の特徴をもつ手法 [27] も提案されている．



(a) 元のプログラム (b) クエリの伝播 (c) クエリの局所解と結果の伝播

(d) 下向き安全性の検査 (e) 最終結果の伝播 (f) プログラムの変形

図 18 質問伝播に基づく部分冗長除去の概略

```
        ...
1: if (p) {
2:     a = 3;
3:     x = a+b;
4: }
5: else y = a+b;
6: z = a+b;
        ...
```

(a) 通常形式

```
        ...
1: if (p0) {
2:     a1 = 3;
3:     x1 = a1+b0;
4: }
5: else y1 = a0+b0;
6: a2 = φ (a1,a0)
7: z = a2+b0;
        ...
```

(b) SSA 形式

図 17 SSA 形式の問題

3.5 COINS の GVN

COINS には，PRE の効果をもつ GVN が組み込んである．コンパイラ・インフラストラクチャのコード最適化として，次の 2 点を考慮している．

1. 他のコード最適化との組合せ
2. 変更を容易にする理解しやすいアルゴリズム

他のコード最適化との組合せとして，例えば，**大域命令スケジューリング**（global instruction scheduling）との組合せが考えられる．最近，**投機的移動**（speculation）を含む命令スケジューリングは，冗長性を増す可能性があるので，共通部分式の除去を組み合わせると効果的であることが知られるようになった[23]．さらに，命令スケジューリングと PRE を組み合わせて，補償コードを自動的に挿入したり[17]，ループスケジューリング[10][18]を同時に実現したり[36]する手法も提案されている．命令スケジューリングとの組合せでは，各命令をスケジュールするたびに，網羅的な解析を行うとコストがかかるので，必要な範囲だけ解析を行う方が有利である．

そこで，ある式 e の出現 e_o を対象に，e_o が冗長であるかどうか検査し，冗長であれば除去する**要求駆動型**（demand-driven）の手法を採用した．e_o が冗長であるかどうかを検査するために，「e は冗長か」というクエリ $av(e)$ を後向きに伝播させる．伝播の際，式 e のオペランドを定義する ϕ 関数 $\phi(a_1, a_2, ..., a_3)$ があれば，オペランドを各引数 a_i で置き換え，a_i に対応する先行節へさらに伝播させる．最終的に，すべてのクエリ $av(e')$ が，e' の出現に到達するか，既に同じクエリが伝播済みの節に到達した場合，式 e は，共通

部分式であることが分かる．もし，クエリ $av(e')$ が，１つでも，開始節に到達するか，e' のオペランドを変更する ϕ 関数以外の文に到達するか，異なるクエリが伝播済みの節に到達したなら，e は冗長でない．

このクエリの伝播は**質問伝播**[24]（question propagation）と呼ばれ，通常形式上で一般化したものは，**要求駆動型データフロー解析**（demand-driven dataflow analysis）[13][34] と呼ばれる．

COINS の GVN では，一旦，クエリに解が得られると，その解を，クエリが伝播してきた経路上を逆向きに伝播させるという点で，質問伝播を拡張している．この逆向きの伝播によって，不要な巻上げを巻き戻す効果を実現する．以降，本手法を**質問伝播に基づく部分冗長除去**（Partial Redundancy Elimination based on Question Propagation，以降，PREQP と呼ぶ）と呼ぶ．PREQP は，質問伝播を用いる他の GVN[24] と同様に，図 17(b) のような字面の異なる式の冗長性も扱うことができる．

また，アルゴリズムを直観的に理解しやすくするために，式 e' を挿入する際に必要な下向き安全の検査を，別の質問伝播として，クエリ $ant(e')$ を前向きに伝播させるようにした．

PREQP が，ある式 e の出現 e_r について，冗長を除去する手順は次のとおりである．

1. 冗長性の検査： e_r が冗長かどうか検査する．
2. プログラムの変形： 1 の検査の結果，冗長であることが判明した場合，必要な ϕ 関数と式の挿入を行ったあと，e_r を適切な変数で置き換える．

以降で，PREQP の詳細を，冗長性の検査，プログラム変形の順に説明する．また，副次的効果を反映した網羅的な冗長除去の構成法についても述べる．

3.5.1 冗長性の検査

図 18(a) を SSA 形式に変換した図 18(b) の CFG を例に，PREQP の振舞いを説明する．今，図 18(b) 節 6 の式 `x3+y1` の冗長性を考える．まず，節 6 から始めて，クエリ `av(x3+y1)` をすべての先行節に伝播させていく．この過程で，もし，オペランド x をもつ式 $e(x)$ のクエリ $av(e(x))$ が，x を定義する ϕ 関数 $x = \phi(a_1, a_2, ..., a_i)$ に到達したなら，クエリを $av(e(a_1))$，$av(e(a_2))$，... ，$av(e(a_i))$ に変更して，対応する先行節にさらに伝播させる．

例では，クエリ `av(x3+y1)` を，節 4 の `x3 = φ(x2,x1)` によって，`av(x2+y1)` と `av(x1+y1)` と変形し，節 2 と節 3 にそれぞれ伝播させる．

(1) クエリの局所解

「式 e が冗長か」というクエリ $av(e)$ は，同じ e の出現する節 n に到達すると，n でのクエリの解が $true$ であることが分かる．また，クエリが開始節に到達したり，e のオペランドを変更する文を含む節に到達した場合，e の冗長な出現が無かったことが分かるので，クエリの解が $false$ であることが分かる．これらの解は，到達した節から直接求まるので，**局所解**と呼ぶ．

図 18 の例では，図 18(c) に示すように，節 2 で $true(p1)$，節 1 で $false$ の結果がそれぞれ得られる．$true$ に続く括弧内の変数は，後述するように，式の値を運ぶ変数である．

PREQP の局所解は，既に伝播済みの節に到達した場合にも求まる．まず，伝播済みのクエリ q_{old} が，今回伝播してきたクエリ q と異なる場合，同じ式の出現を期待できないので，解を $false$ にする．q_{old} と q が同じ場合，q_{old} の解によって，次のように分けられる．

1. q_{old} の解 a_{old} が求まっている場合：a_{old}
2. q_{old} の解が求まっていない場合：$true$

2 の場合は，クエリが，先にループに沿って伝播した場合に生じる．図 19 で，節 3 から，先にループに沿って伝播したクエリは，再度節 3 に到達する．節 3 は，訪問済みなので，クエリに対する解が必要であるが，節 1 と節 2 の解が求まっていないので，節 3 における解は得られない．このように，解が求まっていない場合に $true$ を採用する理由は，$false$ を採用する場合よりも多くの解が得られる可能性があるからである．この点については，文献 [1] のデータフロー解析の解説を参照されたい．

一旦局所解が求まると，得られた解を前向きに伝播することによって，他の節の解も計算していくことができる．

(2) 値を運ぶ変数の決定

PREQP では，$true$ の解を伝播する際に，到達し

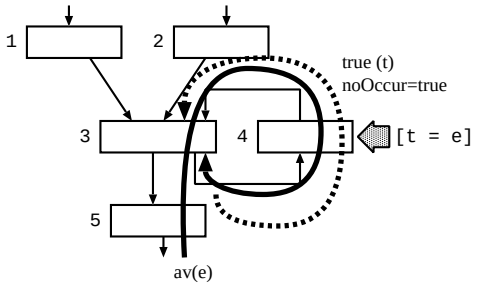


図 19 解が求まっていない節への再訪問

た式の値を保持する変数も一緒に伝播させる．この変数は，最終的に，冗長性の検査を適用した式を置き換えるために利用する．伝播させる変数は，次のように決まる．

1. e を含む代入文 $y = e$ に到達した場合：y
2. q_{old} の解 a_{old} が求まっている場合：a_{old} とともに伝播済みの変数
3. q_{old} の解が求まっていない場合：新しいテンポラリ変数 t を用いて，$t = e$ を生成し，q が，伝播済みの節 n に到達する前に訪問した節 p の入口に仮に挿入する．伝播させる変数は，t になる．

以降，解 $true$ とともに伝播する変数 t は，$true(t)$ のように，$true$ に続く括弧の中に表記することにする．

3 の場合，伝播済みのクエリの解が $true$ になるとは限らないので，プログラムの意味を変えないように，e を右辺とする代入文を仮に挿入する（図 19）．以降，仮の挿入を図に示すときには，図 19 の `[t = e]` のように，`[`と`]` で囲って示す．

このような理由で挿入される文は，**補償コード**（compensation code）と呼ばれる．PREQP の補償コードは，伝播済みのクエリの解が $true$ になれば，除去できる．また，そうでなくても，多くの場合，プログラム変形の際に挿入されない．例えば，図 19 の節 3 の解が $false$ なら，これは，ループに沿って伝播したクエリの解に関係なく，節 3 の解が求まったことを意味するので，プログラム変形の際に節 4 のコードは考慮されない．

(3) 解の伝播

節 n が複数の先行節をもつ場合，クエリの n における解は，先行節 p_i における複数の解 a_i から計算することになる．すべての a_i が $true$ の場合と，すべての a_i が $false$ の場合，n での解は，それぞれ，$true$ と $false$ になる．

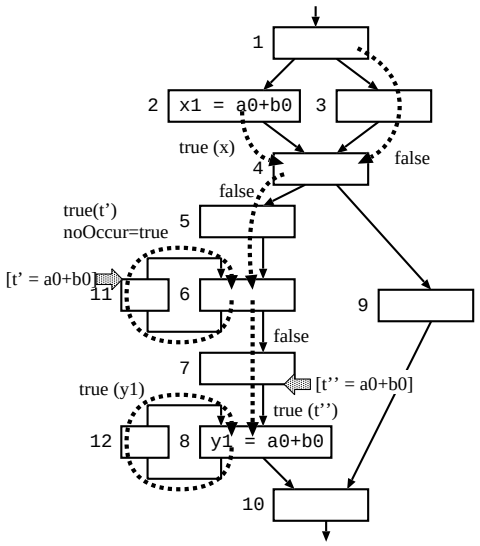


図 20 PREQP における不要な巻上げの抑制

a_i が $true$ の先行節と，$false$ の先行節がある場合，$false$ の先行節にクエリの式を挿入して，部分冗長性を取り除くことを試みる．クエリの式の挿入は，次の条件を満たす必要がある．

1. n の入口で，式 e が下向き安全である．
2. 先行節の解 a_i の幾つかは，「式の出現」に到達して得られた解である．

条件 1 は，$a_i = false$ であった先行節 p_i の下向き安全を検査するべきであるが，クリティカル辺を除去してあるプログラムでは，n の入口について検査すれば十分である．下向き安全は，前向きの質問伝播を用いて検査できる．詳細は，次の節で述べる．

また，条件 2 は，3.3.3 節で示した不要な巻上げを避けるための条件に相当する．ループ l において，この条件が満たされない場合，l がループ不変式を含まないことを意味する．すなわち，l の上方への巻上げは，不要である．

例えば，図 14(a)7 行目の式 `a+b` に PREQP を適用することを考える．図 20 は，このときの解の伝播を示している．まず，節 4 には，開始節から $false$，節 2 から $true$ が伝播してくる．しかし，節 4 で下向き

安全の質問伝播を行うと $false$ なので，節 4 から節 5 へ，$false$ を伝播する．このあと，節 6 には，戻り辺に沿って，同じクエリが伝播済みで解が求まっていない節から $true$ が伝播し，節 5 から $false$ が伝播してくるので，節 6 の結果は，$false$ になる．結果として，節 5 への不要な巻上げを抑制できる．

一方，節 8 には，戻り辺に沿って解析対象の式自身に到達した結果 $true$ と，節 7 から $false$ が伝播してくる．節 8 の入口は，節 8 の `a0+b0` によって，下向き安全なので，節 7 に式を挿入することができる．

条件 2 を満たすかどうかは，条件 2 の「式の出現に到達したか」を反転した「式の出現に到達しなかった」という情報 $noOccur = true$ を，補償コードを挿入した節から，クエリの解とともに伝播させることで検査する．ここで，$noOccur$ は，すべての先行節が $noOccur = true$ の場合にだけ $true$ になる述語である．条件 2 は，先行節に，$noOccur = false$ のものが含まれれば満たされる．以降，説明を簡単にするために，$noOccur$ について触れない．

最終的に，条件 1 と 2 が満たされれば，節 n での解は $true$ であり，さもなければ，$false$ になる．

(4) 下向き安全の検査

節 n の入口の下向き安全の検査は，従来の前向きの質問伝播で実現できる．式 e の節 n の入口における下向き安全を検査したい場合，n から始めて，クエリ $ant(e)$ をすべての後続節に伝播させていく．この過程で，e のオペランドを引数としてもつ ϕ 関数に到達した場合は，そのオペランドを ϕ 関数の代入先で置き換える．

最終的に，すべてのクエリが対象の式の出現に到達して $true$ になるか，既に同じクエリが訪問済の節に到達して $true$ になったなら，節 n の下向き安全が保証される．終了節に到達するか，対象式を変更する文に到達するか，異なるクエリが訪問済の節に到達したクエリがあるなら，節 n は下向き安全でない．

図 18(d) の節 4 は，式の挿入が可能な場合を示している．節 4 から `ant(x3+y1)` のクエリを前向きに伝播すると，すべてのクエリは，節 5，6 で，`x3+y1` の出現に到達して $true$ になるので，節 4 は，`x3+y1` について下向き安全であることが分かる．

(5) 値を運ぶ変数の伝播

節 n に伝播したクエリの解が $true$ であった場合，式の値を運ぶ変数も決定しなければならない．まず，解が $false$ の先行節 p_i に，クエリの式 e_i を右辺とし，テンポラリ変数 t_i を代入先とする $t_i = e_i$ を仮に挿入する．この結果，各先行節 p_i の解は $true$ になり，式の値を運ぶ変数は t_i になる．次に，各 p_i から伝播してきた変数 var_i を引数とし，新しいテンポラリ変数 t に代入する ϕ 関数 $t = \phi(var_1, var_2, ..., var_i)$ を，n の入口に，仮に挿入する．結果として，n から伝播させる変数は，t となる．

例えば，図 18(e) に示すように，テンポラリ変数 `t2` を代入先とした `t2 = x1+y1` を，節 3 の出口へ仮に挿入し，次に，`p1` と `t2` を引数とし，テンポラリ変数 `t1` を代入先とする ϕ 関数を，節 4 の入口に仮に挿入する．節 4 の結果は，$true(t1)$ である．

最終的に，$true(t1)$ は，節 6 まで伝播し，`x3+y1` は冗長であることが分かる．

(6) ϕ 関数の抑制

変数の伝播の過程で行う ϕ 関数の挿入は，多くの不要な ϕ 関数を導入する可能性がある．実際は，n でクエリの式に変更がない場合，$\phi(a_1, a_2, ..., a_j, ..., a_i)$ の a_j の定義が，n を支配するなら，n から伝播する変数として，a_j を用いることができる．

例えば，図 20 のループ 1（節 8，12）に注目すると，前述の通り，解の伝播によって節 7 に，式を挿入することが可能であることが分かる．そこで，節 8 に ϕ 関数の挿入が必要になるが，節 7 が，節 8 を支配しているので，ϕ 関数は生成せず，単に，節 7 の出口に `t'' = a0+b0` を仮に挿入する．最終的に，節 8 の結果は，$true(t'')$ となる．

3.5.2 プログラムの変形

PREQP は，冗長性の検査によって，式 e の節 n 上の出現 e_r が冗長であると判明した場合にだけ，プログラムを変形する．プログラムの変形の手順は，次のとおりである．ここで，節 n におけるクエリ $av(e)$ の解が $true(x)$ であったとする．

1. 変数 x から始めて使用から定義へ順に辿りながら，定義に到達するたびに，その文を実際に挿入する．

2. 節 n 上の e_r を x で置き換える．
3. コピー伝播を適用して，不要なコピー代入を除去する．

最終的に，図 18(a) 節 6 の `x+y` に対する PREQP の適用によって，図 18(f) の結果を得る．

PREQP の適用は，冗長の除去や，補償コードの除去によって，コピー代入を生じる可能があるので，手順 3 のコピー伝播によって除去する．PREQP を 1 回適用するたびに，コピー伝播を適用することによって，以降の PREQP の適用において，より多くの副次的効果を反映することができる．

3.5.3 網羅的な冗長除去

PREQP は，個々の式の出現に対して適用するコード最適化なので，プログラム中のすべての式の出現に対して，適切な順序で適用することによって，網羅的な冗長除去を効果的に実現することができる．

前述した冗長除去の副次的効果は，冗長な式を除去することによって，その式の値を使用する別の式の冗長性を発見できるようにする．すなわち，冗長な式の除去が，後続の式に影響する．

この点を考慮して，COINS では，CFG 節を**ポストオーダの逆順**（reverse postorder）で訪問し，基本ブロック内を入口から出口に向かって到達した式に対して，PREQP を適用するようにしている．この方法によって，多くの副次的効果を反映できる網羅的な冗長除去を実現している．

4 COINS における SSA 形式最適化モジュール

4.1 実装した SSA 形式最適化と変換

紙面の関係で省いたが，COINS の SSA 形式最適化部には，次のような SSA 形式最適化と変換を行うものが備えられている．これらは主要な SSA 形式最適化をほぼ網羅している．詳しくは [30] を参照されたい．後述のお勧めの最適化の説明のために，括弧内に最適化の略称を記した．

（以下，SSA 形式最適化）

- 共通部分式除去 (cse)：2.1 節で前述
- 条件分岐を考慮した定数伝播 (cstp)：2.2 節で前述
- 質問伝播大域値番号付けと部分冗長性除去 (preqp)：3.5 節で前述
- 演算の強さの軽減と判定の置き換え (osr)：2.3 節で前述
- コピー伝播 (cpyp)：コピー文があったら，以後その左辺の変数の使用を右辺の変数で置き換える．
- ループ不変式移動 (hli)：ループ内で値が変わらない式の計算をループの前に移動する．
- 無用命令除去 (dce)：以後使われない変数の計算や実行されないコードを除去する．
- 大域的再結合 (gra)：式の分配規則を用いて式を分け，ループ不変式を増やす．

（以下，SSA 形式最適化に有用な変換や副アルゴリズム）

- クリティカル辺の除去 (esplt)：クリティカル辺（複数の後続ブロックを持つ基本ブロックから複数の先行ブロックを持つ基本ブロックへ向かう辺）に空の基本ブロックを挿入する．
- 無用 ϕ 除去 (rpe)：無用な ϕ 関数を除去する．
- 空ブロック除去 (ebe)：中身が空の基本ブロックを除去する．
- 基本ブロックの連結 (cbb)：必ず連続して実行される複数の基本ブロックを一つにまとめる．
- SSA グラフの作成 (ssag)：SSA 形式をグラフ表現したものを作成する．
- 式の分割 (divex)：右辺に複数の演算子を持つ式を右辺に 1 つの演算子だけを持つ式の列に分ける．

4.2 最適化の組合せ

一般に，最適化の効果的な組合せおよび適用順序（以下たんに「組合せ」と呼ぶ）を一意的に決めることは難しい．COINS では，一般ユーザの便宜のためにお勧めの最適化の組合せを決めており，「-O3」などのコンパイル時オプションの指定により対応する最適化を適用できる．例えば「-O3」のオプションが指定されると，SSA 形式最適化では次の「/」で区切られた最適化がこの順に適用される．

divex/cse/cstp/hli/osr/hli/cstp/cpyp/preqp/
cstp/rpe/dce

同じ最適化が 2 回以上適用されることにも注目されたい．

しかし，この最適化の組合せの効果は対象とするプログラムによって異なる．文献[33] では，SSA 形式最適化の組合せを変えたときに，種々のベンチマークでその効果がどのように変わるかについて述べている．[33] では，SPEC CPU2000 の C および F77 の 14 個のベンチマークについて，SSA 形式最適化の 4 つの組合せで実験を行った．その結果，平均して良い結果を与えるような SSA 形式最適化の組合せを一つ決めたとしても，4 個，5 個あるいは 6 個のベンチマークについては他の SSA 形式最適化の組合せを採用した方が目的コードの実行時間が数パーセント程度短くなる，という逆転現象が起こることが示された．ただ，COINS の SSA 形式最適化はその後も改良が行われているので，最新版ではこれほどの逆転は起こらないかもしれない．これらを考慮して，COINS では，ユーザがソースプログラムに応じて，オプションでの指定により，それぞれの最適化を任意の順序で任意の回数適用することができるようになっている．これにより，いろいろな最適化の適用順序を変えて，効果を試してみることができる．

4.3 SSA 形式最適化の利点と限界

SSA 形式を用いた最適化を実装した経験から，SSA 形式による最適化と通常形式による最適化を比べてみると，SSA 形式による最適化は，

- 変数が単一代入であるため，解析が容易である．
- 条件分岐を考慮した定数伝播（2.2 節）のような通常形式で行うと複雑な最適化が容易に行える．
- PREQP のように式を移動させる際に，必要に応じて，式のオペランドを ϕ 関数の代入先変数から引数に（あるいは引数から代入先変数に）付け替えることによって，プログラムの意味を容易に保存することができる．

といった利点があり，結果として実装コストも少なかった．一方，ループにおける演算の強さの軽減と判定の置き換え（2.3 節）は，通常形式とまったく異なるグラフを用いたアルゴリズムであるので，実装はやや面倒であった．

なお，コンパイラの最適化には様々なものがあり，SSA 形式最適化は万能ではない．たとえば，配列の扱いやポインタ解析による別名の処理などは SSA 形式最適化の技法がまだ確立されておらず，今後の研究が待たれる分野である．また，SSA 形式にするよりはソースプログラムにより近い中間コード上で行ったほうが良いループ展開などの最適化や，命令スケジューリングのようにコード生成部分で行う最適化もある．

5 おわりに

以上，（導入編）（発展編）にわたって SSA 形式の概要，SSA 形式への変換，逆変換，SSA 形式での最適化の例を紹介した．本稿で例示した SSA 形式最適化のほとんどは COINS コンパイラ・インフラストラクチャ [8] 上で実装してある．動作させてみたい場合は，[31] [29] を参照されたい．

参 考 文 献

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools Second Edition*, Addison Wesley, 2007.

[2] Alpern, B., Wegman, M. N., and Zadeck, F. K.: Detecting Equality of Variables in Programs, *Proc. Principles of Programming Languages (POPL'88)*, ACM, 1988, pp. 1–11.

[3] Appel, A. W.: *Modern Compiler Implementation in Java, second ed.*, Cambridge University Press, 1998.

[4] Bodik, R., Gupta, R., and Soffa, M. L.: Complete Removal of Redundant Expressions, *Proc. Programming Language Design and Implementation (PLDI'98)*, ACM, 1998, pp. 1–14.

[5] Briggs, P. and Cooper, K. D.: Effective Partial Redundancy Elimination, *Proc. Programming Language Design and Implementation (PLDI'94)*, ACM, 1994, pp. 159–170.

[6] Choi, J. D., Cytron, R., and Ferrante, J.: Automatic Construction of Sparse Data Flow Evaluation Graphs, *Proc. Principles of Programming Languages (POPL'91)*, ACM, 1991, pp. 55–66.

[7] Click, C.: Global Code Motion Global Value Numbering, *Proc. Programming Language Design and Implementation (PLDI'95)*, ACM, 1995, pp. 246–257.

[8] COINS: 並列化コンパイラ向け共通インフラストラクチャ, http://www.coins-project.org/.

[9] Cooper, K. and Torczon, L.: *Engineering a Compiler*, Morgan Kaufmann, 2003.

[10] Darte, A. and Huard, G.: Loop Shifting for Loop Compaction, *Proc. Int. Conf. Workshop on*

Languages and Compilers for Parallel Computing (LCPC'99), Berlin, Heidelberg, Springer-Verlag, 2000, pp. 415–431.

[11] Dhamdhere, D. M.: Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2(1991), pp. 291–294.

[12] 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(1993), pp. 321–336.

[13] Duesterwald, E., Gupta, R., and Soffa, M. L.: A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis, *ACM Trans. Prog. Lang. Syst.*, Vol. 19, No. 6(1997), pp. 992–1030.

[14] F. Chow, S. C., Kennedy, R., Liu, S. M., and Lo, R.: A New Algorithm for Partial Reundancy Elimination based on SSA Form, *Proc. Programming Language Design and Implementation (PLDI'97)*, ACM, 1997, pp. 273–286.

[15] Fitzgerald, R., Knoblock, T. B., Ruf, E., Steensgaard, B., and Tarditi, D.: Marmot: An Optimizing Compiler for Java, *Software —Practice and Experience*, Vol. 30, No. 3(2000), pp. 199–232.

[16] GCC Homepage: http://gcc.gnu.org/.

[17] Gupta, R.: A Code Motion Framework for Global Instruction Scheduling, *Proc. Int. Conf. Compiler Construction (CC'98), LNCS*, Lisbon, Springer-Verlag, 1998, pp. 219–233.

[18] Gupta, S., Gupta, R., Dutt, N., and Nicolau, A.: *SPARK: A Parallelizing Approach to The High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, 2004.

[19] IBM: *Jikes Research Virtual Machine*, http://jikesrvm.sourceforge.net/.

[20] Johnson, R. and Pingali, K.: Dependence-Based Program Analysis, *Proc. Programming Language Design and Implementation (PLDI'93)*, ACM, 1993, pp. 78–89.

[21] Knoop, J., Rüthing, O., and Steffen, B.: Lazy Code Motion, *Proc. Programming Language Design and Implementation (PLDI'92)*, ACM, 1992, pp. 224–234.

[22] Morel, E. and Renvoise, C.: Global Optimization by Suppression of Partial Redundancies, *Comm.ACM*, Vol. 22, No. 2(1979), pp. 96–103.

[23] Nicolau, A., Dutt, N. D., Gupta, R., Savoiu, N., Reshadi, M., and Gupta, S.: Dynamic Common Sub-Expression Elimination during Scheduling in High-Level Synthesis, *Proc. of the 15th International Symposium on System Synthesis (ISSS 202)*, IEEE, IEEE Computer Society, 2002, pp. 261–266.

[24] Rosen, B. K., Wegman, M. N., and Zadeck, F. K.: Global Value Numbers and Redundant Computations, *Proc. Principles of Programming Languages (POPL'88)*, ACM, 1988, pp. 12–27.

[25] Scale Compiler Group: Scale homepage: http://www-ali.cs.umass.edu/Scale/.

[26] Steffen, B., Knoop, J., and O. Rüthing: The Value Flow Graph: A Program Representation for Optimal Program Transformations, *Proc. Int. Conf. European Symposium on Programming (ESOP'90)*, Copenhagen, Denmark, Springer-Verlag, 1990, pp. 389–405.

[27] VanDrunen, T. and Hosking, A. L.: Value-Based Partial Redundancy Elimination, *Proc. Int. Conf. Compiler Construction (CC'04), LNCS*, Berlin, Springer-Verlag, 2004, pp. 167–184.

[28] Wegman, M. N. and Zadeck, F. K.: Constant Propagation with Conditional Branches, *ACM Trans. Prog. Lang. Syst.*, Vol. 13, No. 2(1991), pp. 181–210.

[29] 佐々政孝: デモの仕方, http://www.is.titech.ac.jp/%7Esassa/coins-www-ssa/japanese/michishirube-ipsj-ssa/. 辿れない場合は, http://www.coins-project.org/ より「静的単一代入最適化部」を辿り, さらに,「21 世紀のコンパイラ道しるべ - COINS をベ ースとして 情報処理学会誌の連載 - SSA 最適化部」を辿る.

[30] 佐々政孝: 静的単一代入形式に基づく最適化に関する研究, http://www.is.titech.ac.jp/%7Esassa/coins-www-ssa/japanese/index.html.

[31] 佐々政孝: コンパイラ・インフラストラクチャCOINSを用いた SSA 最適化 (その 2), 情報処理, Vol. 47, No. 9(2006), pp. 1032–1038.

[32] 佐々政孝, 滝本宗宏: 静的単一代入形式を用いた最適化（導入編）, コンピュータソフトウェア, 日本ソフトウェア科学会, Vol. ?, No. ?(2008), pp. ?

[33] 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)(2006), pp. 30–43.

[34] 滝本宗宏, 福岡岳穂, 佐々政孝: 疎な要求駆動型データフロー解析, 情報処理学会論文誌プログラミング, Vol. 46, No. SIG 11 (PRO 26)(2005), pp. 16–26.

[35] 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(1997), pp. 2237–2250.

[36] 勝原達也, 滝本宗宏: 部分冗長除去に基づく大域命令スケジューリング, 情報処理学会論文誌プログラミング, Vol. 48, No. SIG 12 (PRO 34)(2007), pp. 52–65.

[37] 中田育男: コンパイラの構成と最適化, 朝倉書店, 1999.