「高水準中間表現HIRでの最適化」

渡邊　坦（COINSコンパイラ・インフラストラクチャ協会）　藤瀬哲朗（三菱総合研究所）

はじめに

最初の実用的高級言語であるFortranは，開発の当初から，高速実行できる機械命令列（目的コード）を生成することを目指していた．したがって，コンパイラ研究では，それが始まったときから現在まで，コード最適化が中心課題の1つであり続けている．前回までに説明したSSA最適化はCOINSでは機械語に近いレベルのLIRに対する変換として位置付けられているが，今回はソースプログラムの構造をかなりよく保存している中間表現であるHIRに基づく最適化（以下HIR最適化と略す）についてその概要を説明する．これはCOINSでは，本連載8月号の図-8に示すように，HIRに対する基本最適化として位置付けられている．紙数が限られているので，詳細については，一般的な最適化について体系的に解説した中田[1]や，最適化内容を具体的に記したCOINSのWebページ[2]を参照されたい．

HIR最適化の内容には，SSA最適化で行っていることと重複するものもあるが，HIRをCプログラムとして出力する場合や，最適化結果に基づいて並列化しOpenMPプログラム[5]として生成し並列実行させるような場合には，SSA最適化は適用されない．配列要素の表すメモリ領域の解析に基づく最適化・並列化や，インライン展開，ループ展開などの最適化はLIRよりもHIRで行う方がやりやすい．たとえば

```
c = aa[i] + aa[j];
func(c, aa[i] + aa[j]);
```

という文の列があったとすると，HIR最適化では2つの文の`aa[i]+aa[j]`の全体を同じ式と認識して前の文で計算した値を後ろの文で再利用する．一方，LIRでは`aa[i]`や`aa[j]`がそのアドレス計算を含むいくつもの命令列に分解されるため，分解された命令間の再利用はうまくでき，HIRでは対象としていなかった細かいレベルの再利用もできるが，`aa[i]+aa[j]`全体の再利用は，処理が分解されすぎていて少しむずかしくなる．1つのコンパイラだけを作るときは重複した機能を実装することはあまり行わないが，COINSはさまざまな使い方のできるコンパイラ・インフラストラクチャとするために，HIR段階とLIR段階の両方に，それぞれの特徴を活かした基本的な最適化機能を備えている．

用語や記号が定義されている個所に下線を付す．

HIR最適化

●最適化の必要性

コンパイラにおける最適化には，実行時間を短縮するものと，メモリ使用量を少なくするものがある．後者は組み込み用マイコンなどで重要であるが，それは主にLIRに対する最適化で行うものなので，ここでは実行時間短縮について述べる．その方法としては，

(1) コンパイル時に実行できる計算は実行する
(2) 同じ計算を何度も行わない
(3) 無用な計算は行わない
(4) 同じ結果をもたらすより高速な表現を選ぶ
(5) 先行命令の結果待ちを減らす

などがある．このうち，4番目，5番目の方法は機械語に近いLIRレベルで行う方が有効なので，HIR最適化では，3番目までを主として行うが，4番目に該当するものもある．

同じ計算や無駄な計算がプログラムに現れることは少ないと思われるかもしれないが，コンピュータでのプログラムの実行過程まで考えると，そうでもないことが分かる．いま，

```
if (a[i] > max)
    max = a[i];
```

というif文を見ると，`a[i]`が2カ所に現れている．ところで，配列要素`a[i]`の値を求めるには，一般に，

レジスタに配列`a`の先頭番地を設定する
`i`に配列要素長をかけた値をそれに加える

```
(assign   71 int            // _var1 = a[i]
 <var      72 int _var1>    // _var1 は int 型変数
 (subs     73 int           // 添字つき変数の表現
  <var      74 <VECT 100 0 int> a> // "a" は
                            // 要素数 100 の配列
  <var      75 int i>))     // 添字は i
(if       76 void line 17   // if (_var1 > max)
 (cmpGt    77 bool          // greater than,
                            // 結果の型は bool
  <var      78 int _var1>   // _varxx は一時変数
  <var      79 int max>)    // max は int 型変数
 (labeldSt 80 int           // ラベル付きの文
  (list 81
   <labelDef 82 _lab9>)     // ラベルは _lab9
  (block    83 void         // 文の列を表すブロック {
   (assign   84 int line 18 //   max = _var1;
    <var      85 int max>
    <var      86 int _var1>)) // } 外側閉じ括弧
                            // はブロックの終わり
```

図 -1　if 文に対する HIR 最適化の結果

その結果で示される番地の内容を取り出す

というような操作が必要になる．このプログラムに対して COINS コンパイラで HIR 最適化を行うと，配列要素の参照に必要な操作を 2 回も行わなくてすむよう，この if 文は一時変数（コンパイラの導入した変数）_var1 を使って

```
_var1 = a[i];
if (_var1 > max)
     max = _var1;
```

に相当する表現に変換される．a[i] のような 1 つの記号に隠された操作まで考えてソースプログラムをこのような形に書くのは煩わしく，もとのプログラムのように同じ記号 a[i] を使う方が分かりやすい．

上記のような変換を行うために，HIR では a[i] は 1 つの記号ではなく，少し分解された形で表現する．HIR 最適化について説明するには，HIR の表現を少し説明しておいた方がよいので，上記の変換結果に対する HIR を**図 -1** に示す．ここで，// の右は注釈であり，assign 等のオペレータの次の番号（71, 72, ..., 86）は HIR のノードに付けられる一連番号である（具体的な HIR 表現はこのように長くなるが，HIR と C の表現のレベルは近いので，HIR を例示するための図 -1 以外では，HIR の変換結果をそれに相当する C の表現で説明することが多い）．

このように，ソースプログラムには一見無駄がないように見えても，最適化しないと同じ計算の繰り返しや無用な命令がしばしば現れる．したがって，最適化した場合としない場合では，実行速度が 2 倍以上違うこともよくある．

HIR 最適化のうち，本稿では主なものとして

- コンパイル時に実行できる計算を行う定数の畳み込みと伝播
- 同じ計算を何度も行わなくする冗長性の削除
- より高速な表現を選ぶインライン展開とループ展開

について述べる．ほかにも算出結果が使われてない無用な計算は削除するなど，いろいろな項目があるが，それらについては Web ページ [2] を参照されたい．

同じ計算を検出するなど，最適化に必要な処理を行うためには，プログラムの流れを調べる制御フロー解析や，データの流れを調べるデータフロー解析が必要になる．それらについても後で述べる．

● 定数の伝播と畳み込み

定数畳み込み (constant folding) では，コンパイル時に計算できる式をあらかじめ計算して定数に置き換える．定数伝播 (constant propagation and folding) では，先行する文で定数値をとることになった変数を後続の式で定数に置き換える．その結果さらに定数からなる式が現れると，それも計算して定数で置き換えることを繰り返す．いま，次のようなプログラムがあったとしよう．

```
a = 1;
b = 2;
if (a+1 < b-1) {
  x = a - b;
}else {
  x = a + b;
}
```

これを COINS で定数畳み込みと定数伝播の最適化を行う hirOpt=cf/cpf というオプションを指定してコンパイルすると，まず，a=1, b=2 なので if の条件式における a+1, b-1 は定数伝播の結果としてそれぞれ 1+1, 2-1 となり，then 節の a-b と else 節の a+b はそれぞれ 1-2, 1+2 となる．ついで，それらは定数畳み込みによってコンパイル時に演算されてそれぞれ 2, 1, -1, 3 となる．したがってこの if 文は次に相当する HIR に変換される．

```
if (2 < 1) {
  x = -1;
}else {
  x = 3;
}
```

このように変換されると，2 < 1 は常に false となり，後続の最適化処理で全体を

```
x = 3;
```

とすることも可能になる．HIRの定数畳み込みと定数伝播では制御構造を変更する処理は行っていないが，後続のLIRの最適化で全体を`x = 3;`とする変換を行っている．

上記のような変換を可能とするには，ある変数（`a`や`b`）の各参照個所に対して，その値を設定している文（後述の到達定義）がどれであるかを探す必要がある．到達定義がある1つの定数値を設定する文（`a=1;`や`b=2;`）しかないと保証できる場合は，その変数を定数に置き換えることができる．ソースプログラムに近いHIRの段階では定数の畳み込みや伝播の機会は多くないが，LIRの段階では配列要素や構造体要素の番地計算などでそのような機会が多く生ずる．他の最適化を行った結果としてその機会が生ずることもよくある．LIRにおけるこれらの最適化の方法は，本連載の9月号に掲載されたSSA最適化の中で説明されている[6]．

●冗長性の削除

分岐やジャンプを途中に含まない直線的な命令の列を基本ブロック(basic block)というが，1つの基本ブロックの中で同じ計算をしている場合は，その検出は比較的容易である．HIR最適化では，後述のデータフロー解析の段階で，同じ形の式に同じexpression identifier (ExpId)というものを付加しているので，それを見るだけで同じ形の式であることが判定できる．ところで，同じ形の式であってもオペランドの値が設定しなおされていれば値は同じといえない．したがって同じ形の式が出現している途中にオペランドを設定している文がないことを確認しなければならない．関数呼び出しがあれば大域変数の値は再設定されているかもしれない．このようなことをはっきりさせるため，HIR最適化では，代入文や関数呼び出しなど，値を設定する文に対して，データフロー解析のとき，値が変わる可能性のある変数を全部列挙しておく機能がある．

もう1つ注意しなければならないのは，変数形式は異なるのに同じ実体を表す別名(alias)といわれるものの影響である．

```
x = aa[i] + 1;
aa[j] = b;
y = aa[i] + 2;
```

のような文の列があるとき，`aa[i]`の値は1番目と3番目の文では同じと思われるかもしれないが，`j`が`i`と同じ値を持つ場合，`aa[i]`と`aa[j]`の実体は同じなので，2つの`aa[i]`の値は異なると見なければならない．別名関係はポインタや引数によっても生ずることがある．別名の解析についても後で述べる．

基本ブロックにまたがった大域的な冗長性の削除を行うのは，少し複雑であり，データフロー解析をしっかりやる必要がある．HIR最適化では，大域的な冗長性削除は，E. MorelとC. Renvoiseが考え出した部分冗長性削除[3]という方法で行っている．これは1980年代までに書かれたコンパイラの専門書に載っているような共通部分式削除をカバーするだけでなく，ループ内で値の変わらないループ不変式をループ外に出すことや，if文のthen部またはelse部の一方を通ったときにだけ冗長性が生ずる場合の冗長性削除も同一の手続きで行うことのできる強力な方法である．まずその原理を簡単に説明する．

```
a = b + g;
if (a != 0) {
  x = a + b;
}else {
  x = b + g;
}
printf("%d ¥n", x);
y = (a + b) * (b + g);
printf("%d ¥n", y);
```

というプログラムでは，then部を通るときにはa+bが2度計算されるという冗長性がある．このように，どの経路を通るかによって冗長性が生ずるプログラムは，部分冗長性があるという．部分冗長性を削除する方法を追求してゆくと，どの経路を通っても冗長となる計算（完全冗長性）も削除できる．その説明のために，もう少し議論を精密化する．

ある式eの計算は，eのオペランドが計算済みであり，その後オペランドの値が変更されていない点にならば移動できる．このような点がいくつもあるならば，制御の流れをできるだけさかのぼったところや，それらの経路が合流しているところに移動すると，同種の移動を少なくできる．移動可能な点ですでにeが計算済みとなっていれば，その移動は削除できる．すなわち，その計算は削除できる．このような移動を試みることによって，eの計算を完全冗長にすることができると分かれば，eの計算を以前計算した値で置き換えることによって，冗長性を削除できる．これが部分冗長性削除の基本原理である．説明を聞くとなるほどと思われるが，この方法が提案されるまでに20年近くかかっている．古くから研究されていることにも大きな発見の機会がある例といえよう．

その実現方法の詳細は文献1）～4）に譲るとして，ここでは例を使って説明する．**図-2 (a)**は先の例に基本ブロック番号B1, B2, B3, B4を左端に付記したものであり，`a`, `b`, `x`, `y`は局所変数，`g`は大域変数であるとする．`b+g`はB1とB3, B4で計算され，`a+b`はifの条件式がtrueのときにのみ2回計算される．式`b+g`はB1, B3の

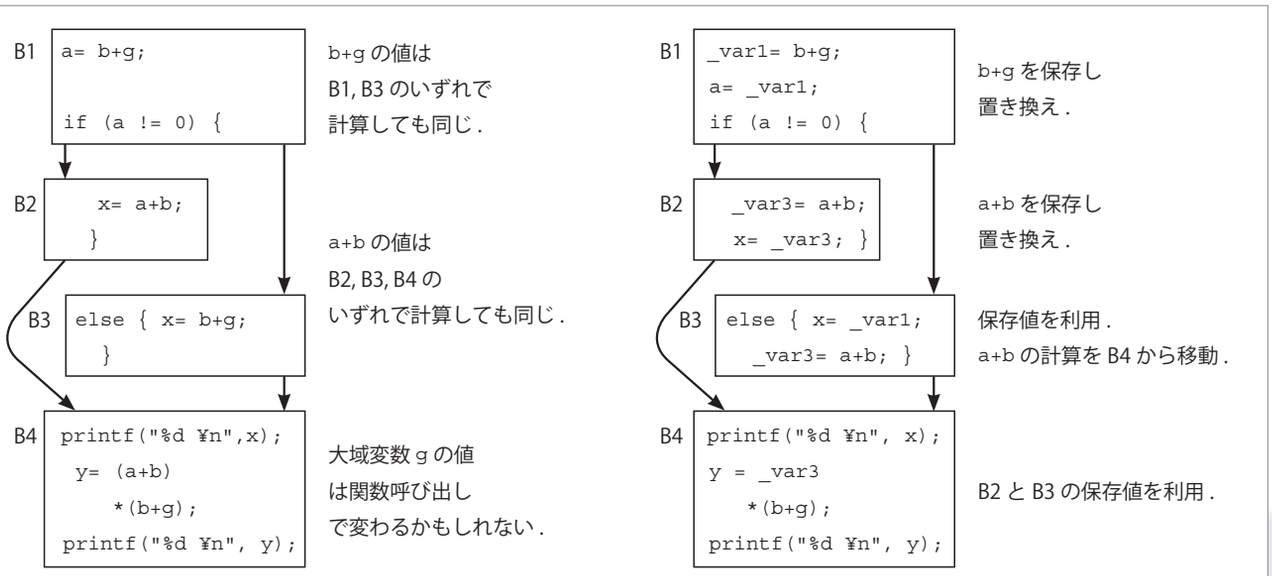

図 -2　冗長性の削除

いずれで計算しても同じ値となるが，B4 に関数呼び出しがあるので，その後では値が異なるかもしれないと判定する．

a+b は大域変数を含まず関数呼び出しの影響を受けないので，B2, B4 のいずれで計算しても同じ値となる．

HIR で部分冗長性削除（Partial Redundancy Elimination）を行う hirOpt=pre というオプションを指定すると，上記の解析結果に基づいて，HIR を変換する．通常は HIR 最適化をした結果が LIR に変換され，そこでの最適化変換を受けた後に機械語に変換されるのであるが，COINS には HIR 最適化をした結果を C プログラムに変換する機能もある（hir2c=opt というオプションでそれを指定する）．ここでは，図 -2(a) に対して HIR 最適化を適用した結果を分かりやすくするため，変換後の HIR を hir2c で C プログラムに変換したものを **図 -2(b)** に示す（hir2c による変換結果では，暗黙の型変換をすべて明示的に表現し，すべての部分式を括弧でくくるので少し見づらくなるが，図 -2(b) では見やすいように少し整形している）．これで見ると，最適化前では if の条件式が真のときは a+b が 2 度計算されたのに対し，最適化後では条件式の真偽にかかわらず，1 回しか計算されない．b+g の計算は，printf で g には値を設定しないことまではコンパイラで判断していないため，printf の後でもう一度計算している．

次に，ループ不変式の扱いを見るために，**図 -3** のプログラムをとりあげる．ここでも左端の B1, B2, ..., B5 は基本ブロックの番号である．

```
B1 for (i = 0;
B2      i < 10;
B4      i++) {
B3   c = c + (a + b);
   }
B5 d = a + b;
```

図 -3　冗長性削除の例題 2

```
for ( i = 0, _var1 = a + b; i < 10; i = i + 1) {
    c = c + _var1;
}
_lab4:;
d = _var1;
```

図 -4　冗長性削除後の HIR を C で表現した結果（図 -3 に対応）

式 a+b の値は B2, B3, B4, B5 のいずれで計算しても同じ値となるので，それらの共通の上流点 B1 の末尾で計算してもよい．これに対して部分冗長性削除を行った結果がどうなるかを示すため，変換後の HIR から hir2c で C に変換し整形したプログラムを **図 -4** に示す．a+b の計算は，ループの初期化部分の末尾で 1 回行われるだけになっている．

```
1) int main()
2) {
3)   int a=1, b=3, c;
4)   c = absdiff(a, b);
5)   printf("c=%d¥n", c);
6)   return 0;
7) }
8) int absdiff( int p1, int p2 )
9) {
10)   if (p1 > p2)
11)     return p1 - p2;
12)   else
13)     return p2 - p1;
14) }
```

図 -5　インライン展開前のプログラム

```
1) int main( )
2) {
3)     int a, b, c;
4)     int _var1,_var3,_var5;
5)     {  a = (int )1;
6)        b = (int )3;
7)     }
8)    _var1 = a;
9)    _var3 = b;
10)    {
11) _lab8:; { if ((_var1) > (_var3)) {
12) _lab9:;    { _var5 = ((_var1) - (_var3));
13)               goto _lab12;
14)            }
15)          } else {
16) _lab10:; { _var5 = ((_var3) - (_var1));
17)             goto _lab12;
18)          }
19)        }
20)      }
21) _lab12:;
22)    }
23)    c = _var5;
24)    (&(printf))( (("c=%d¥n")),c);
25)    return (int )0;
26) }
```

図 -6　インライン展開後の main プログラム（hir2c で変換後整形）

●インライン展開

関数呼び出しは，レジスタの待避・回復やスタック領域の確保・解放などを伴う重い処理であり，短い関数ではこれらのオーバーヘッドの比率が高くなる．インライン展開は，小さい関数の本体を，それを呼び出している（call している）ところに展開することによって，オーバーヘッドを減らそうとするものである．展開すると呼び出し元との間で共通部分式が増えて最適化の機会が増えるという利点もある．オブジェクト指向プログラミングなどでは，小さい関数が多数作られることが多く，これは実行速度向上に有効である．ただし，展開すると目的コードは大きくなることが多い．

COINS では hirOpt=inline というオプション指定があると，インライン展開をする．ただし，次のような場合は展開しない．

- 関数が大きい．
- call が if 文やループ文の条件式の中，あるいは switch 文の選択式の中にある．

関数の大きさの限界値はコンパイル・コマンドで変更できる．展開によってプログラムの構造が不正になったり，記号の衝突が起こったりしないようにするためには，プログラムの構造に合わせた展開位置の選択や，一時変数の導入が必要になる [2]．

たとえば，**図 -5** のプログラムは**図 -6** のように展開される．ここで，左端の番号は説明用につけた一連番号である．下線 (_) で始まる変数は展開時に生成された変数であり，図 -5 の行 4 の

```
c = absdiff(a, b);
```

は，図 -6 では行 8 から 23 に示す形に展開されている．これを見ると，実引数 a と b の値をそれぞれ _var1，_var3 で置き換えて absdiff 関数の仮引数 p1 と p2 の代わりに使い，return 値を受け取る変数を _var5 として計算している．

COINS では，上例のように関数の定義が参照個所より後ろにある場合も展開する．また，再帰的な関数を単純に展開しようとすると無限に展開が繰り返されるが，そのような場合はある定められた回数（2 回）までで展開をやめる [2]．

●ループ展開

プログラムで長い実行時間を費やすのはループ部分であるため，多くのループ最適化手法が開発されてきた．その 1 つにループ展開がある．たとえば

```
for (i = 0; i < 100; i = i + 1) {
  a[i] = b[i];
}
```

を

```
for (i = 0; i < 100; i = i + 2) {
  a[i]   = b[i];
  a[i+1] = b[i+1];
}
```

```
_var5 = 1*7;
_var7 = 1*8;
for (i = 0; i < 100 - _var5; i = i + _var7) {
  sum1 = sum1 + i + i + 1 + i + 2 + i + 3 + i + 4
     + i + 5 + i + 6 + i + 7;
  sum2 = sum2 + a[i] + a[i+1] + a[i+2] + a[i+3]
    + a[i+4] + a[i+5] + a[i+6] + a[i+7];
}
for (; i < 100; i = i + 1) {
  sum1 = sum1 + i;
  sum2 = sum2 + a[i];
}
```

図 -7 ループ展開の例

とすると，判定とジャンプの回数が半分になり，少し速くなる．また，b[i] と b[i+1] などで共通部分式削除の機会が増えるかもしれない．このようなループ展開は，目的コードが大きくはなるが，他の最適化との相乗効果もあって，小さいループに対しては高速化に有効な場合が多い．

COINS では，hirOpt=loopexp というオプション指定があると，ループ本体が短い最内側の for ループを展開する．展開の回数は，対象機種のレジスタ数とループ本体の複雑度によって決めるが，コンパイル・コマンドでも調整できる．ただし，最内側ループであっても，展開によって意味が変わる可能性がある場合や展開操作が複雑になる場合は展開しない [2]．

ループ内で移動しても支障のない文は，他の最適化との相乗効果をねらって，同型演算をまとめるために移動や統合を行う．たとえば

```
for (i = 0; i < 100; i++) {
  sum1 = sum1 + i;
  sum2 = sum2 + a[i];
}
```

をループ展開すると，**図 -7** のように，i, i+1, ... を加算する 8 つの代入文を 1 つの代入文にまとめ，a[i], a[i+1], ... を加算する 8 つの代入文を 1 つにまとめる．

● HIR 最適化の効果

HIR 最適化をいくつかの例題プログラムに適用したときの Sparc における実行時間（秒）を示す．* 印は同じ行に並んでいる COINS の測定値のうち最も速いものを示す．参考までに Sparc gcc -O2 の測定値を最後の列にあげ，COINS より gcc が速いものを ^ 印で示す．

例題	hirOpt	hirOpt + ssa etc.	ssa etc.	no opt	gcc -O2
ISort	474.4	281.0	252.9*	474.5	194.5 ^
SSort	674.5	396.5	396.2*	561.9	336.9 ^
heap	115.9	104.6*	111.6	122.0	94.4 ^
komachi	208.7	163.1*	177.0	206.7	206.6
matmult	12.1	5.7*	7.0	13.0	7.2
prime	507.8	237.0*	243.6	503.0	320.5
queen	176.7	147.3	107.3*	157.7	146.2
shell	160.9	117.1	111.5*	157.2	104.7 ^
soukan	283.2	183.1*	214.1	388.0	222.8

```
target=sparc-v8 (UltraSPARC-IIs 450MHz)
hirOpt: hirOpt=inline/loopexp/cs/cpf/pre
ssa etc.: schedule,regpromote,ssa-opt=
     prun/divex/cse/cstp/hli/osr/hli/cstp/
     cse/dce/srd3
```

COINS の測定値で見ると，HIR 最適化だけで十分な効果があるとはいえないし，逆効果となる例もあるが，他の最適化と組み合わせることによって，効果が上がる場合が多い．

フロー解析

最適化や並列化では，種々の変換に対してそれを適用する条件が満たされているかどうか調べなければならない．たとえばある式の計算を冗長であるとして削除するには，その計算を行っている点に到達する経路上に同じ計算があって，しかもオペランドの値が変更されていないことを確かめる必要がある．このような判断をするためには，プログラムの実行経路を調べる制御フロー解析と，変数の値がどこで設定されてどこで利用可能かなどを調べるデータフロー解析が必要になる．

● 制御フロー解析

基本ブロックの中ではプログラムは一直線に走るので，制御フローは基本ブロックを単位として考えればよい．すなわち，基本ブロックを頂点，その先行，後続関係を表す矢印を辺とする有向グラフとして制御フローを表現すればよい．これを制御フローグラフ（CFG, control flow graph）という．HIR では，制御文（if 文，ループ文, jump 文, switch 文, return 文）とラベルに注目すれば，制御フローグラフを作ることができる．

プログラムの解析や変換にあたっては，制御フローグ

```
int printf(char*, ...);
int a[10] = {0, 2, 1, 4, 3, 6, 5, 8, 7, 9 };
int main()
{
  int i, n, max;
  n = 10;
  max = a[0];
  for (i = 1; i < n; i=i+1) {
    if (a[i] > max) {
        max = a[i];
    }
  }
  printf(" %d", max);
}
```

図 -8　フロー解析の対象例

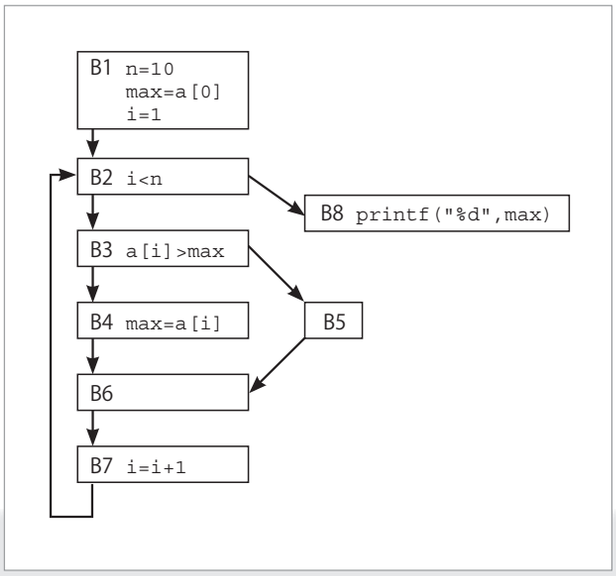

図 -9　制御フロー情報（図 -8 に対応）

ラフをある順序でたどる，基本ブロックの中の文や式を順次見る，などの操作が必要になる．文の区切りと基本ブロックの区切りは必ずしも一致しないので，その操作は少し複雑になるが，COINS はこれらの操作を簡単に行うための各種のメソッドを備えている [2]．

例として，**図 -8** のプログラムを制御フロー解析すると，trace=Flow.2 と指定することにより，**図 -9** に相当する制御フローグラフの情報が得られる．ここで，B1, B2, ..., B8 は基本ブロック番号を表す．HIR では 1 つの文がいくつかの基本ブロックに分かれたり，文の一部が前の基本ブロックに入ったりする．この例では `for(i=1; i<n; i=i+1)` の初期化部分 `i=1;` は基本ブロック B1 に，`i=i+1` はブロック B7 に入っている．

● データフロー解析

〈データ表現〉

最適化のために，オペランドとなる変数の値がどこで計算されたか，ある式の値が計算済みであるか否かなどを調べるには，まず変数，式，定義点などを識別できるようにしなければならない．変数に値を設定する（定義する）ものは，HIR では代入文 (AsignStmt) か関数呼び出し (call) なので，それらの HIR 上の位置（プログラム点）で定義点を表現できる．また，先に述べたように，同じ形の式には同じ ExpId を対応させているので，式は ExpId で表現できる．変数も式もプログラム点もインデックスをつけることによって，番号で表現できる．番号付けの仕方は，プログラム点と変数に対しては独立につけるが，変数と ExpId に対しては合わせて 1 つの番号付けを行う．そうすることによって，処理を共通にすることができる場合が多くなるからである．たとえば，配列要素や構造体要素などの複合変数も，それに ExpId を対応させることによって単純変数の場合と同じように処理することができる．ExpId を導入してこのようなことを可能にしている点が COINS における HIR のデータフロー解析の特徴である．

データフロー情報は，各基本ブロックにおいて，ある条件を満たす変数や式，プログラム点としてはどのようなものがあるか，という形で表現する．これは各対象に対して 1 ビットで表現できるので，ある条件を満たす変数や式の集合を表すビットベクトルを基本ブロックごとに用意し，単純変数や複合変数，演算式の (ExpId の) インデックスでその情報のビット位置を表す．同様に，ある条件を満たすプログラム点の集合は，プログラム点を表すビットベクトルによって表現する．

〈データフロー情報〉

データフロー情報は，まず基本ブロック内だけ見て決めることのできる情報を求め，ついで隣接基本ブロックにその情報を渡してゆく処理を繰り返すことによって，制御フローグラフ全体に対する情報を求める．以下，B は基本ブロックを表し，変数は単純変数や複合変数を表すとして，それらについて説明する．変数 x の定義点とは，x へ値を設定する点のことである．基本ブロック内だけ見て決めることのできる情報には次のようなものがある．

```
BBlock 3 _lab9
 exposed   max a i
 availIn   a[0], i<n
 availOut  a[0], i<n, a[i], a[i]<max
BBlock 4 _lab5
 defined   max
 exposed   a, i
 availIn   a[0], i<n, a[i],  a[i]<max
 availOut  a[0], i<n, a[i]
```

図 -10 データフロー情報の算出例

Def(B) : B における変数の定義点の集合
Defined(B) : B で値の設定される変数の集合
Exposed(B) : B で値設定をする前に使っている変数の集合
Used(B) : B で使われている変数の集合
EGen(B) : B で計算されたあと，B の中ではそれ以降，そのオペランドに対する値変更がない式の集合
EKill(B) : B でオペランドに値設定があるため，以前に計算された値が B の出口では無効となっている式の集合

制御フローグラフ全体を見て求める情報としては次のものがある．

Kill(B) : 変数 x への値設定のうち，B での x への値設定によって無効にされる定義点の集合
Reach(B) : 変数の定義点のうち，B の入り口まで無効にされずに到達するものの集合（到達定義）
AvailIn(B) : 計算された値が無効にされずに B の入り口まで伝わってきている式の集合
AvailOut(B) : 計算された値が無効にされずに B の出口まで伝わってきている式の集合
LiveIn(B) : 設定された値が B か B の後ろで使われる変数の集合
LiveOut(B) : 設定された値が B の後ろで使われる変数の集合
DefIn(B) : どの経路を通っても B の入り口で値が設定済みとなっている変数の集合
DefOut(B) : どの経路を通っても B の出口で値が設定済みとなっている変数の集合
DefUseList(x の定義点) : その定義点で設定した値を使う参照点のリスト
UseDefList(x の参照点) : x の参照点まで無効にされずに伝わる x の定義点のリスト

これらの情報があるとこれまでに述べた最適化の処理が可能になる．その正確な定義と計算仕方は，文献 1)，2) を参照されたい．

図 -8 における

```
if (a[i] > max) {
    max = a[i];
}
```

の部分は基本ブロック 3 と 4 になるが，それに対するデータフロー情報のうち，主なものは**図 -10** のようになる（実際の表示は assign 22 とか _xId12 のようになるが，図ではそれに対する文や式として示す）．if の条件判定部 (`BBlock 3`) では，max, a, i を定義済みとして使い (`exposed`)，条件式の計算が終わると `availOut` に示した式が計算済みとなるので，それらは then 部では `availIn` として列挙される．Then 部 (`BBlock 4`) では `max` が `defined` となり，`a[i]<max` は `max` の値が変わるので `availOut` からは削除されている．

〈フロー解析の使い方〉

フロー解析は，通常はそれ自体を目的とするのではなく，最適化や並列化を行うときなどに，ある条件に適合する変数や式にはどのようなものがあるかを調べるために使う．その解析にあたっては，データフロー解析を行うには制御フロー解析ができていなければならないとか，Reach 情報の計算には Def 情報を必要とするというように，解析項目間に依存関係による順序づけが要求される場合がある．項目 Z は項目 Y の結果を利用し，項目 Y は項目 X の結果を利用する場合，まず X を計算してから Y を計算し，それから Z を計算する必要がある．フロー情報を利用するのにこのような順序関係をいちいち考えるのは煩わしいので，COINS では必要とする Z を要求すれば自動的にそれに必要な X, Y の計算を行う機構が備えてある．また，データフロー情報は種類が多いので，要求に応じて算出する．X という情報が要求されれば，もしそれが算出されていなければ X とその計算に必要な情報を計算するだけで，要求されない情報は計算しない．要求された情報が計算済みであればそれを再利用し，計算を省略する．

● 別名解析

別名解析は，ソースプログラムでは相異なる変数であっても同じ記憶領域を表す可能性はないかということを調べる．2 つの変数間で対応する記憶領域に重なりがある可能性があると，それらは互いに別名関係にあるという．別名情報は，ある変数に値が設定されたとき，それと別名関係にある変数にも値が設定された可能性があるとみなし，ある変数の値が参照されたとき，それと別名

関係にある変数の値が参照された可能性があるとみなして，誤る可能性のある最適化変換は抑制する等の目的で利用される．

COINS の別名解析には，楽観的解析と悲観的解析がある．楽観的解析では，(a) 相異なる仮引数は互いに別名関係にない，(b) 配列要素の添字値は配列宣言で指定された範囲内にあるなどと仮定して解析する [2]．悲観的解析では，絶対に確実なこと以外は成り立たない場合があると想定して解析する．うっかりミスはもう含まれていないとか（変数間で意図的にメモリを重ね合わせるような）裏技は用いていないというようなプログラムに対しては楽観的解析でもよいのであるが，コンパイラではそこまで判断できないので，-coins のオプションとして alias=opt を指定しなければ，悲観的解析を行う．HIR の最適化では，別名解析の結果を利用して変換を制限する．ポインタを多用する C のプログラムは，配列を主たるデータ構造とする Fortran のプログラムに比べ，別名解析がむずかしいので最適化が制限される場合が多い．

おわりに

HIR を対象とする最適化について述べた．それは多くのコンパイラでも実装されているものであるが，COINS ではそれを部品化して，新しい機能を実現するようなときに利用しやすい形にしている．次回は COINS における並列化について述べる．

謝辞　COINS 開発にご尽力いただいた方々とご支援いただいた方々，ならびに貴重なコメントをいただいた閲読者の方々と中田育男教授に感謝します．

参考文献

1) 中田育男：コンパイラの構成と最適化，朝倉書店（1989）．
2) 並列化コンパイラ向け共通インフラストラクチャ COINS : http://www.coins-project.org/COINSdoc/
3) Morel, E. and Renvoise, C. : Global Optimization by Suppression of Partial Redundancies, CACM, Vol.22, No.2, pp.96-103 (Feb. 1979).
4) Dhamhere, D. M. : E-path_PRE - Partial Redundancy Elimination Made Easy, ACM SIGPLAN Notices, Vol.37, No.8, pp.53-65 (Aug. 2002).
5) Chandra, R. et al. : Parallel Programming in OpenMP, Morgan Kaufmann Publishers (2001).
6) 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, pp.1032-1038 (Sep. 2006).

（平成 18 年 10 月 10 日受付）

▶渡邊　坦（正会員）
tan@watanabe.ai.to
1962 年京都大学理学部数学科卒業．日本 IBM（株），（株）日立製作所，電気通信大学を経て，現在電気通信大学名誉教授．工学博士．主として言語処理系に興味を持つ．ACM，日本ソフトウェア科学会各会員．

▶藤瀬哲朗（正会員）
fujise@mri.co.jp
1984 年電気通信大学大学院修了．同年（株）三菱総合研究所入社．1992 年（財）新世代コンピュータ技術開発機構出向．現在（株）三菱総合研究所主任研究員．並列記号処理，並列化コンパイラ技術に興味を持つ．