18. ソースプログラムの高速化†

石　畑　　清††

1. ソースプログラムの高速化とはなにか

プログラムの開発や保守の手間を少なくするために，プログラムをわかりやすく書くことが重要である．しかし，わかりやすいプログラムが効率の面でも優れているとは限らない．効率に対する配慮を優先して，複雑なわかりにくいプログラムを書かなければならないことも多い．

ソースプログラムの高速化（**プログラム変換** (program transformation) とも呼ぶ）は，プログラムの明快さと効率を両立させるための手法である．最初にわかりやすいプログラムを書き，これに変換を施して効率的なプログラムを作ろうというのがその基本的な考え方である．つまり，プログラムの最適化の一種と言ってもさしつかえない．ただ，最適化がオブジェクトプログラムを対象とするのに対して，ソースプログラムのレベルで変換を進めようとする点が違うだけである．

ここで重要なのは，変換規則を機械的に適用することによって変換を行うという点である．変換規則は，プログラムの働きを変えないことがあらかじめ証明されているものとする．変換後のプログラムの機能は変換前と同じはずだから，プログラマが変換の過程や変換結果の細部を気にする必要はない．変換前のわかりやすい方のプログラムにだけ努力を集中すればよいわけである．効率改善法のメニューを変換規則という形で提供することによって，効率に対する配慮を不要にすると言うこともできる．

プログラム変換の例として，ここではループの高速化と再帰呼出しの除去を取り上げ，それぞれのごく簡単な例を示す．なお，本稿中のプログラムは Pascal を一部拡張した言語を使って記述する．

2. ループの高速化

ループの高速化は，プログラミングの常識として日常よく使われる技法の集積と考えてよいだろう．その意味で，どちらかと言えば古典的な話題である．

この分野では，コンパイラによる最適化の研究が進んでおり[12)~15)]，その結果をソースプログラムレベルの変換に応用することができる．

（1） ループの展開 (loop unrolling)

ループの繰り返しの回数だけ，その本体を並べて書けば，繰り返しの制御が不要になる．そこまで極端でなくとも，

```
for i:=1 to 100 do
    a[i]:=a[i]+b[i]
            ⇩
for i:=1 by 2 to 100 do
    begin a[i]:=a[i]+b[i];
          a[i+1]:=a[i+1]+b[i+1] end
```

と変換するだけで，繰り返しの回数が半分になり，その分だけ高速化される*.

（2） ループの融合 (jamming, loop fusion)

同じ繰り返し形式を持つ二つのループを一つにまとめるやり方である．たとえば，

```
for i:=1 to 100 do a[i]:=0;
for i:=1 to 100 do b[i]:=c[i]
            ⇩
for i:=1 to 100 do
    begin a[i]:=0;
          b[i]:=c[i] end
```

これも繰り返しのオーバヘッドを減らす効果がある．

この変換は，ループの中の文の実行順序を変えるので，いつでも使えるというわけにはいかない．変換によってプログラムの働きが変わらないことを確認する必要がある．上の例では二つのループの中の文が互い

† Performance improvement of source programs by Kiyoshi ISHIHATA (Department of Information Science, Faculty of Science, University of Tokyo).

†† 東京大学理学部情報科学科

* $i+1$ の計算は配列要素を参照する命令に吸収されるためオーバヘッドにはならないと考える．

に独立なので大丈夫である.

一般に，変換規則には制約条件がついていて，この条件が成立する時にだけ適用できる．上の例のように条件を満たすことが簡単に示せることもあるが，より一般的には，データフロー解析などによって検証しなければならない.

(3) ループ外への計算の移動

値の変わらない式の計算をループの外に追い出す方法である．たとえば，

```
for i:=1 to 100 do a[i]:=x*y
        ⇩
tmp:=x*y;
for i:=1 to 100 do a[i]:=tmp
```

また条件判断をループの外へ出すことも考えられる.

```
for i:=1 to 100 do
    if switch then a[i]:=0
        ⇩
if switch then
    for i:=1 to 100 do a[i]:=0
```

いずれも，式や条件の値がループの中で変わらないことと評価の際に副作用を持たないことが制約条件となる.

以上，ループに関する変換は，次に述べる再帰呼出しに比べて単純である．しかし，ループはプログラムの中で一番多く実行される場所なので，その効果は大きい.

なお，これらの変換の価値は，使用する計算機やコンパイラによって変わることに注意しなければならない．変換をしなくても，高性能のコンパイラが同じような最適化をしてくれることがあるし，パイプライン計算機やベクトル命令を持つ計算機では，変換したためにかえって性能が悪くなることすら考えられる.

3. 再帰呼出しの除去

再帰呼出しの除去とは，再帰呼出しを繰り返しの形式に変換して高速化を図ることである.

最も一般的なやり方としては，スタックを使った実現法がある．これについては，本特集「再帰呼出しの実現法」を参照のこと．普通，再帰呼出しの除去と言う場合は，問題の性質を利用して，もっとうまく変換することを指す．このうち，表を利用する方法については，本特集「再帰呼出しの索表計算法」で解説されている．ここでは，それ以外の方法を取り上げる.

(1) 末端再帰 (tail recursion) の除去

最も基本的な方法は，末端再帰の除去である．再帰的な呼出しが手続きの実行の最後でしか行われない場合に，呼出しを goto 文で置き換えるやり方である.

再帰的な呼出しが手続きの最後にあれば，呼出しの後にこの手続きがする仕事は何も残っていない．したがって，この手続き用のスタック領域はもはや不要である．呼出しを goto 文で置き換えるのは，新しい手続きにこの領域をそのまま使わせることにほかならない．メモリが節約でき，スタック管理のオーバヘッドがなくなる.

たとえば，与えられた値がリストの中にあるかどうかを調べる関数 *member* は末端再帰の形をしている.

```
function member (x : item ; y : list) : boolean ;
begin
  if y=nil then member:=false
  else if x=y↑.value then member:=true
  else member:=member (x, y↑.next)
end
```

再帰呼出しを取り除くと

```
function member (x : item ; y : list) : boolean ;
begin
L1: if y=nil then member:=false
    else if x=y↑.value then member:=true
    else begin y:=y↑.next ; goto L1 end
end
```

となる．関数呼出しにはパラメータを渡す機能があるが，goto 文にはないので，代入文を使ってパラメータの値をセットしている.

複数の手続きが互いに再帰的に呼び合っている場合でも，それぞれの最後で呼び出す形になっていれば同様に変換できる．一般に，スタックの類をまったく使わず，意味情報も使わずに再帰呼出しを取り除けるのは末端再帰の場合だけであることが証明されている[21].

(2) プログラムの意味情報を利用した変換

末端再帰の除去は，制約条件を持たず，プログラムの形にしかよらないが，適用範囲は狭い．もっと複雑な形を処理するためには，制約条件をつける，言い換えればプログラムの意味に関する情報を活用することが必要である.

階乗を求める関数を再帰的に書くと，

```
function f (n : integer) : integer ;
begin
```

```
if n=0 then f:=1
       else f:=f(n-1)*n
end
```

となる．これは末端再帰ではない．fの呼出しより後に乗算をしなければならないからである．ところが，この場合は補助関数gを使えば，次のような末端再帰の形に簡単に変換することができる．

```
function f(n: integer): integer;
begin f:=g(n, 1) end;
function g(n, r: integer): integer;
begin
  if n=0 then g:=r
         else g:=g (n-1, r*n)
end
```

これを goto 形式に直すと，

```
function g(n, r: integer): integer;
begin
L1: if n=0 then g:=r
           else begin r:=r*n;
                      n:=n-1;
                      goto L1 end
end
```

が得られる．ここで，gを導入して末端再帰の形にした変換は次の一般形によった．

```
function f(x: t1): t2;
begin
  if P(x) then f:=K
          else f:=A(f(B(x)), C(x))
end
          ⇩
function f(x: t1): t2;
begin f:=g(x, K) end;
function g(x: t1; y: t2): t2;
begin
  if P(x) then g:=y
          else g:=g(B(x), A(y, C(x)))
end
```

Kは定数，A, B, C, P は関数である．ただし，この変換が成立するためには，Aが

$A(A(\alpha, \beta), \gamma)=A(A(\alpha, \gamma), \beta)$

という関係（右交換則）を満たすことが必要である*．階乗の例では，プログラムの形のほかに，乗算が右交換則を満たすという性質を利用して，再帰呼出しを取り除いたわけである．

（3） スタックを使った変換

スタックを使って再帰呼出しを実現する場合でも，その一部操作を省略できれば，再帰呼出し除去と呼ぶに値する[10),17)]．

木探索の手続きを再帰呼出しを使って書くと次のようになる．

```
procedure traverse (t: tree);
begin
  if t<>nil then
    begin traverse (t↑.left);
          process (t↑.value);
          traverse (t↑.right) end
end
```

まず，末端再帰を取り除く．

```
procedure traverse (t: tree);
begin
L1: if t<>nil then
      begin traverse (t↑.left);
            process (t↑.value);
            t:=t↑.right;
            goto L1 end
end
```

次に，スタックと goto 文を使って呼出しを展開する．この時，一般には戻り番地もスタックに入れるが，*traverse* の再帰的な呼出しは一箇所しかないので，この場合は不要である．局所変数 t だけをスタックに入れればよい．

```
procedure traverse (t: tree);
begin
L1: if t<>nil then
      begin push (stack, t);
            t:=t↑.left;
            goto L1;
      L2: t:=pop (stack);
            process (t↑.value);
            t:= t↑.right;
            goto L1
      end;
  if not empty (stack) then goto L2
end
```

---

* たとえば，$P(B(B(x)))$ が真だとすると，変換前のfの値は
$A(A(K, C(B(x))), C(x))$
変換後のfの値は
$A(A(K, C(x)), C(B(x)))$
である．

スタックが空でないのは，再帰的に呼ばれた場合に相当する．再帰的呼出しは一箇所しかないので，その場所 (L2) に飛べばよい．

4. プログラム変換システム

以上，プログラム変換の簡単な例をいくつか紹介した．この程度の例では，ありがたみが感じられないと思うが，これは変換規則が単純すぎたためである．変換規則のレパートリを増やし，組み合わせて使用できるようにすると，非常に強力なプログラミングの道具となる．ごく単純なアルゴリズムに変換を繰り返し適用することによって，有名な高速アルゴリズムを導けることがある[18]のがその一つの例証となるだろう．

追加すべき変換規則にはきわめて多くの種類がある．一例をあげると，データ構造に関するものなどである．また，Hanoiの塔などの特殊なパターンをうまく解決する手法[19),20]も含まれる．

プログラム変換を実際のプログラム作りに応用するために，プログラム変換システムを作ることが考えられている[8]．

プログラマは，最初に簡潔なプログラムをシステムに与える．そして，コマンドによって変換法を指示し，変換の結果を得るという操作を繰り返す．システムと会話しながら効率を向上させていくわけである．

システムは変換規則のライブラリを持っている．コマンドによる指示を受けると，まず変換規則のプログラム形式と実際のプログラムのパターンマッチングを行う．次に，制約条件が満たされていることを確認し，最後に変換してその結果を表示する．変換そのものは簡単な記号処理にすぎない．むしろ，制約条件の確認の方が一般には難しい．適当に条件を追加することによって，人間が確認の手助けをすることも必要になるだろう．

上記のような会話型のシステムでは，どの変換を使えば効率が上がるかを判断するのはプログラマである．一方，この部分も自動化しようという試みもある[7]．原理的には，変換法を片端から試すという力ずくのやり方である．もちろん，プログラムの性能評価の基準をあらかじめシステムに教えておくことが必要である．

この種のシステムでは，変換規則が多いのは好ましくないので，少数の基本規則しか用意しない．7)では，手続き呼出しをその本体で置き換えたり (unfolding)，逆に文をくくり出して手続き呼出しにする (folding) などの六つの規則によって変換を進める．今まで述べたような変換例は，これらの規則の組み合わせで説明できるのである．

5. その他の応用

関数型プログラミング言語上でのプログラム変換の研究[2]が盛んである．関数型言語には副作用などの問題がないため，従来の言語に比べて変換がやりやすい．

プログラム変換をプログラムの高速化以外に応用しようという研究もある．たとえば，プログラムの検証[4]やプログラミング言語の意味の記述[4),6]などである．

さらに，超高級言語の上でプログラム変換をしようという試み[5),9]もある．超高級言語で書くプログラムは，問題の解法というよりも問題の仕様に近い．仕様から変換を使って具体的なプログラムを作ろうというわけである．変換の過程に人間の手助けが必要となるが，プログラム合成の一つのやり方と言えよう．

謝辞 本稿に関する助言をいただいた都立大・疋田輝雄氏に感謝する．

参 考 文 献

プログラム変換一般

1) Arsac, J. J.: Syntactic source to source transforms and program manipulation, Comm. ACM, Vol. 22, No. 1, pp. 43-54 (1979).
2) Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, Comm. ACM, Vol. 21, No. 8, pp. 613-641 (1978).
3) Balzer, R.: Transformational implementation: an example, IEEE Trans. Softw. Eng., Vol. SE-7, No. 1, pp. 3-14 (1981).
4) Bauer, F. L. and Broy, M. (eds.): Program construction, Lecture Notes in Computer Science, Vol. 69, p. 651, Springer, Berlin-Heidelberg-New York (1979).
5) Bauer, F. L. et al.: Programming in a wide spectrum language: a collection of examples, Sci. Comput. Program., Vol. 1, No. 1, 2, pp. 73-114 (1981).
6) Broy, M. and Pepper, P.: Program development as a formal activity, IEEE Trans. Softw. Eng., Vol. SE-7, No. 1, pp. 14-22 (1981).
7) Burstall, R. M. and Darlington, J.: A transformation system for developing recursive programs, J. ACM, Vol. 24, No. 1, pp. 44-67

(1977).
8) Darlington, J. and Burstall, R. M.: A system which automatically improves programs, Acta Inf., Vol. 6, No. 1, pp. 41-60 (1976).
9) Dewar, R. B. K., Sharir, M. and Weixelbaum, E.: Transformational derivation of a garbage collection algorithm, ACM Trans. Prog. Lang. Syst., Vol. 4, No. 4, pp. 650-667 (1982).
10) Knuth, D. E.: Structured programming with go to statements, Comput. Surv., Vol. 6, No. 4, pp. 261-301 (1974).
11) Loveman, D. B.: Program improvement by source-to-source transformation, J. ACM, Vol. 24, No. 1, pp. 121-145 (1977).

コンパイラによる最適化

12) Aho, A. V. and Ullman, J. D.: Principles of compiler design, p. 604, Addison-Wesley, Reading, Mass. (1977).
13) Allen, F. E. and Cocke, J.: A catalogue of optimizing transformations, Design and optimization of compilers, R. Rustin (ed.), Prentice-Hall, Englewood Cliffs, N. J. pp. 1-30 (1972).
14) Cocke, J. and Markstein, P. W.: Measurement of program improvement algorithms, Information Processing 80, S. H. Lavington (ed.), North-Holland, Amsterdam-New York-Oxford, pp. 221-228 (1980).
15) 中田育男：コンパイラ，p. 278, 産業図書，東京 (1981).

再帰呼出しの除去

16) Arsac, J. and Kodratoff, Y.: Some techniques for recursion removal from recursive functions, ACM Trans. Prog. Lang. Syst., Vol. 4, No. 2, pp. 295-322 (1982).
17) Bird, R. S.: Notes on recursion elimination, Comm. ACM, Vol. 20, No. 6, pp. 434-439 (1977).
18) Bird, R. S.: Improving programs by the introduction of recursion, Comm. ACM, Vol. 20, No. 11, pp. 856-863 (1977).
19) Hikita, T.: On a class of recursive procedures and equivalent iterative ones, Acta Inf., Vol. 12, No. 4, pp. 305-320 (1979).
20) Partsch, H. and Pepper, P.: A family of rules for recursion removal, Inf. Process. Lett., Vol. 5, No. 6, pp. 174-177 (1976).
21) Strong, H. R., Jr.: Translating recursion equations into flow charts, J. Comput. Syst. Sci., Vol. 5, No. 3, pp. 254-285 (1971).
22) Walker, S. A. and Strong, H. R.: Characterizations of flowchartable recursions, J. Comput. Syst. Sci., Vol. 7, No. 4, pp. 404-447 (1973).

(昭和57年12月14日受付)