TM-0104

KL0機械語へのif-then-else制御構造の導入による プログラム実行効率の向上について

高木茂行，近山　隆
横田　実，瀧　和男

April, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

１．始めに

Ｐｒｏｌｏｇの実行はユニフィケーションと自動バックトラックによって実現される。自動バックトラックを制御するための機構としてｏｒノードを枝刈りするカット操作が用いられる。実行時には，自動バックトラックのための情報をｏｒノード毎に保存することが必要となるが，これは実行上の大きなオーバーヘッドとなっている。実行が決定的である部分については，これを行なわなくとも，正しい実行順序が保証されるので，機械語レベルでこれをサポートすれば実行効率の向上が期待できる。逐次型推論マシンＰＳＩ上でこの実験をおこなった結果，予期した効率向上が実現できる見通しが得られたのでここに報告する。

２．ＫＬＯの実行

逐次型推論マシンＰＳＩ[1)]の機械語であるＫＬＯ[2)]は，ｐｒｏｌｏｇの実行を高速化することを目的として設計され，それに適した多くの組込み述語を持っている。この上に構築されたシステム・プログラミング言語ＥＳＰ[3)]は，ｐｒｏｌｏｇの処理にオブジェクト指向の考え方を取入れている。ＰＳＩのオペレーティング／プログラミング・システムであるＳＩＭＰＯＳ[4)]は全てＥＳＰで記述されており，その実行効率はＫＬＯの実行効率に依存している。

Ｐｒｏｌｏｇプログラムの実行においても，処理が決定的であり，バックトラックを要しない部分が存在する。特にＯＳの内部処理では，決定的な条件判定が多数出現するので，この部分の実行効率を向上させることは，ＯＳ全体の実行効率を上げるのに役立つと考えられる。これらの判定はＸ＜Ｙのような大小比較や，ａｔｏｍ（Ｘ）のような型の判定に代表される単純なものの組合せであり，しかもそのうちの多くがif-then-elseの連鎖として見ることができる場合が多い。この処理は決定的であり，バックトラックを必要としないので，バックトラック用の情報を保存しておく必要はなく，そのためのオーバーヘッドを削減することが可能である。

Ｐｒｏｌｏｇではif-then-elseの処理を行なう場合，条件判定の述語の呼出しの後に，不必要な選択肢をカットするという操作を行なう（図２－１）。ＫＬＯの実行時には，他の選択肢にバックトラックするための情報をスタックに積んだ後に条件判定をせざるを得ないので，わざわざスタック上にカットすることになる情報を積み，条件判定の直後にそれをカットすることがしばしば起こる。従って，このｏｒノードの作成とカットの処理をいかに少なくあるいは効率良く行なうかが速度向上の鍵となる。

① t :- c, !, p.　　クローズを分けたｏｒの記法
　t :- q.

② t :- (c, !, p; q).　　クローズ内ｏｒの記法

図　２－１　ｏｒとｃｕｔによるif-then-elseの実現
　　ｃ：条件判定，ｐ：ｔｈｅｎ部，ｑ：ｅｌｓｅ部

一方，一般に実行時間の９０％がコードの数％の部分に集中していることが知られており，Ｆｏｒｔｒａｎ等のプログラムでは実測も行なわれている[5]。それによれば，ごくわずかの部分のコードを書きなおすことで，全体の実行効率を向上させることが期待できる。そこで，ＫＬＯの条件判定用組込み述語に手を加えて，if-then-elseを実現し，速度向上を測定した。以下の説明ではここで実現したif-then-elseを示す記法として，

③　t :- (c -> p; q).　　if-then-elseの記法

と書くことにする。DEC-10 prolog [6]でもこの記法は採用されているが，現在のところインタプリタでのみサポートされ，コンパイラではサポートされていない。

３．組込み述語の変更

３．１　変更対象と改造量

ｏｒノードのための情報をスタックに積まずにすませることが可能な組込み述語は，次の制限条件を満足するものでなければならない。

1) 未定義変数に値をユニファイしない。
2) その命令の実行中に例外／割込みを発生しない。
3) 副作用を持たない。

この制限に合う条件判定の組込み述語は以下の１４個であった。

atom,	atomic,	equal,	floating_ point,
identical,	integer,	less_than,	location,
not __equal,	not __identical,		not __less__then,
number,	structure,	unbound	

これらの述語の引数として，判定がfailした場合の飛び先きオフセットを追加し，failの場合はそこへjumpするように仕様を変更した。また，ＥＳＰ，ＫＬＯのコンパイラもそれに合せて改訂した。

改造量はファームウェアが約２０ステップ，ＥＳＰコンパイラが約２０ステップ，ＫＬＯ

コンパイラが約１００ステップであった。

３．２　ファームウェアからの効果予測

例として次のようなものを考える。

```
or & cut        t(X):- (__ X<10, !, Then__; __Else ), Next.
                        ①  ②  ③      ④  ⑤
if-then-else    t(X):- ( X<10 -> Then __; Else ), Next.
                          ②          ⑥
```

この例におけるThen,Else,Nextの処理を除いた，if-then-elseの実行の制御のためのマイクロ・プログラムの実行ステップ数は，それぞれ以下のようになる。

①ｏｒ	４８（フレームの生成と環境の退避）
②Ｘ＜１０	３３／２７／３４／３６
	（従来のsuccess/fail，if-then-elseのsuccess/failに対応）
③ｃｕｔ	２９
④ｏｒからＮｅｘｔへ	２０
⑤ｏｒのバックトラック	３３
⑥Ｎｅｘｔへ分岐	３

従ってこの処理にかかるマイクロ・プログラムのステップ数は

	success	fail
if-then-else	３７（②＋⑥）	３６（②）
or & cut	１３０（①＋②＋③＋⑤）	１０８（①＋②＋④）

となる。マイクロ・プログラムの１ステップは200ns なので，この例ではsuccess の場合19.4μ秒，failの場合14.4μ秒の利得があると予想される。

４．評価プログラム

評価には，既に多くのｐｒｏｌｏｇのベンチマークで使われているｑｕｉｃｋ　ｓｏｒｔ，ＳＩＭＰＯＳの内部で処理の重いと考えられるプロセスの割込み制御部分，アプリケーションの例として，ＦＧＣＳ´８４で使用したハーモナイザの主処理部分の３本を用いて実行時間の計測を行なった。

4．1 quick sort

quick sortはリストのソートを行なうプログラムである。ここで用いたプログラムは，以下のようなものである。

```
        qsort([X | L],R,R0) :- !,
                p(L,X,L1,L2), qsort(L2,R1,R0), qsort(L1,R, [X | R1]).
        qsort([],R,R).

a :     p([X | L],Y, [X | L1],L2) :- X<Y, !, p(L,Y,L1,L2).
        p([X | L],Y,L1,[X | L2])  :- !, p(L,Y,L1,L2).
        p([], _,[],[]).          %  クローズを分けたｏｒとｃｕｔによる
```

テストでは，このｐのみを次のように書きかえた。

```
b :     p([X | L],Y,LL1,LL2) :- !,
                ( X<Y -> LL1= [X | L1], LL2=L2; LL1=L1, LL2= [X | L2]] ),
                p(L,Y,L1,L2).
        p([], _,[],[]).          %  if-then-elseによる

c :     p([X | L],Y,LL1,LL2) :- !,
                ( X<Y, !, LL1=[X | L1], LL2=L2; LL1=L1, LL2= [X | L2]] ),
                p(L,Y,L1,L2).
        p([], _,[],[]).          %  クローズ内ｏｒとｃｕｔによる

d :     p([X | L],Y,LL1,LL2) :- !,
                ( X<Y -> LL1= [X | L1], p(L,Y,L1,LL2);
                         LL2= [X | L2], p(L,Y,LL1,L2) ).
        p([], _,[],[]).          %  if-then-elseによる

e :     p([X | L],Y,LL1,LL2) :- !,
                ( X<Y, !, LL1=[X | L1], p(L,Y,L1,LL2);
                          LL2=[X | L2], p(L,Y,LL1,L2) ).
        p([], _,[],[]).          %  クローズ内ｏｒとｃｕｔによる
```

ａが最も一般に現在のｐｒｏｌｏｇやＫＬＯで用いられている書きかたである。ｃ，ｅはそれをクローズ内ｏｒの型式に書きなおしたものであり，現在のｐｒｏｌｏｇやＫＬＯで実行できる。ｂ，ｄはそれをさらにif-then-elseに書きなおしたもので，今回の実験のために作成した。

プログラムａからｅの計５個を，重複を含む５０個の整数から成る同じデータを入力として，それぞれ１０００回ループさせ，ｃｐｕ時間を計ったのが表４－１である。

表　4－1　ｑｕｉｃｋ　ｓｏｒｔ

ａ：	１８．１７４　秒	
ｂ：	１８．５０４　秒	
ｃ：	２２．９４７　秒	要素数：５０
ｄ：	１６．５７５　秒	ループ回数：１０００
ｅ：	２０．１５２　秒	ループ・オーバーヘッド：０．１５２秒

このプログラムの実行では，X<Y の判定にのみif-then-elseを用いている。この判定を通過してsuccess するのは 106回，failするのが 123回あり，合計の実行回数は 229回である。また，終端条件としてp の第１引数が[]で呼出される場合がデータの個数分（50回）ある。また，qsort の呼出し回数は 101回であり，p の呼出し回数を含めた総ユニフィケーション回数（述語呼出し回数）は 609回である。従って呼出しの37.6 %が条件判定となっている。

もとのプログラムａと最も高速化されたｄとを比較した場合，全体の利得は 8.8 %であり，判定の呼出し１回当たりの平均利得は，(18.270-16.575)/(106+123)=6.97 μ秒となる。また，クローズ内ｏｒとカットの組合せによるｃ，ｅとそれに対応するif-then-elseのｂ，ｄを比べると，利得はそれぞれ19.4 %，17.8 %，判定１回当たりで19.40 μ秒，18.39 μ秒となり，ファームウェアのステップ数からの推定値に良く一致している。 6.97 μ秒と19.40 μ秒，18.39 μ秒との差は，ヘッド部のユニフィケーション・パターンの違い，ボディ部のユニフィケーション回数の違い，クローズ・レベルでのｏｒノードの情報をスタックに積むか否かの違い等によるものであると考えられる。

４．２　プロセスの割込み処理ルーチン

ＳＩＭＰＯＳでは通常の場合，１回のウィンドウ出力を行なう毎に２回ずつユーザ・プロセスとウィンドウ・プロセスとの間のプロセス切替が行なわれ，この処理の間に何回も割込み禁止状態にする必要がある。また，ウィンドウのロック等の処理にも割込み禁止にする必要があり，そのたび毎に割込みの禁止と許可の処理が行なわれる。

ウィンドウに次々に文字を出力していくような場合，これが大きなオーバーヘッドになっており，ユーザ・インタフェースの改良のためには避けて通れない部分であることがわかっている。

割込みの禁止と許可を行なうスーパバイザの手続きの中の各々１箇所をif-then-elseに書きかえ，ウィンドウに１００文字出力するのに要した経過時間を測定した。この場合，プロセスの切替に要する経過時間が短縮されることが期待できる。結果は表４－２のようになった。

表　４－２　ウィンドウへの１００文字出力

if-then-else	３１９３　ミリ秒
or & cut	３２３０　ミリ秒

全体での利得は 1.2 %，ウィンドウへの１字出力１回当たりの利得は 400μ秒である。

４．３　ハーモナイザ

このプログラムはメロディをデータとして与え，それに対する和音のハーモニーを生成するプログラムである。数十の和音生成規則の中から制限条件に合致するものをバックトラックで探索し，規則に合ったハーモニーを生成する。処理の中には大きな構造体を引数でひきまわす部分や，１０個を越える引数を持つ手続きが多く存在する。評価にはこのプログラムのデータ処理の部分のみを取出してｃｐｕ時間を計測した。

プログラムの修正部分を図４－１に示す。書きかえた述語は５個である。入力データとしては，ＦＧＣＳ゛８４のデモに用いた荒城の月，赤とんぼ，もみじの３曲を用いた。結果を表４－３に示す。

```
gen_harmony(_, error, _, _, _, _, _, _, _, [-1], illegal_format) :- !;
gen_harmony(-1, _, _, _, _, _, _, _, _, [-1], normal_end) :- !;
gen_harmony(_, _, Generation_type, Key_scale, Prev_scale,
    Prev_harmony,Pre_pre_bas,I_pnt, Prev_harmony_scale,
    [1, Time, Sound_length, 0, Sop, Velocity, Alt,
     Velocity, Ten, Velocity, Bas, Velocity|O_list], Status) :-
            ... ... ...
gen_harmony(Stopper, Estatus, Generation_type, Key_scale, Prev_scale,
    Prev_harmony,Pre_pre_bas,I_pnt, Prev_harmony_scale, Olst, Status) :-
    ( Estatus==error -> Olst=[-1], Status=illegal_format;
    ( Stopper== -1 -> Olst=[-1], Status=normal_end;
        Olst=[1, Time, Sound_length, 0, Sop, Velocity, Alt,
              Velocity, Ten, Velocity, Bas, Velocity|O_list],
        ... ... ... ) );

distribution_test(0, [], _, _) :- !;
distribution_test(0, [Ss,Aa,Tt,Bb], [Sop,Alt,Ten,Bas], _) :-
    Ss-Tt>12, Sop-Ten>12, !;
distribution_test(0, [Ss,Aa,Tt,Bb], [Sop,Alt,Ten,Bas], _) :-
    Ss-Tt<12, Sop-Ten<12, !;
distribution_test(0, [Ss,Aa,Tt,Bb], [Sop,Alt,Ten,Bas], _) :-
    Ss-Tt=:=12, !;
distribution_test(0, [Ss,Aa,Tt,Bb], [Sop,Alt,Ten,Bas], _) :-
    Sop-Ten=:=12, !;
distribution_test(0, [Ss,Aa,Tt,Bb], [Sop,Alt,Ten,Bas], _) :-
    Pre_dst is Ss-Tt,  Dst is Sop-Ten,
    ( Pre_dst>12, Dst>12 -> true;
    ( Pre_dst<12, Dst<12 -> true;
    ( Pre_dst==12        -> true;
      Dst    ==12 ) ) ), !;
distribution_test(1, _, _, _);

:progression_test(Class,0,[],_,_,_,_,_) :- !;
:progression_test(Class,0,Pre_harmony,Harmony,
    Pre_hscale,Hscale,Gen_type,Key_scale) :- !,
            ... ... ...
:progression_test(Class,1,[],_,_,_,_,_) :- !;
:progression_test(Class,1,Pre_harmony,Harmony,
    Pre_hscale,Hscale,Gen_type,Key_scale) :-
            ... ... ...
:progression_test(Class,Times,Pre_harmony,Harmony,
    Pre_hscale,Hscale,Gen_type,Key_scale) :-
    ( Times==0 -> ( Pre_harmony==[] -> true;
            ... ... ... );
    ( Times==1 -> ( Pre_harmony==[] -> true;
            ... ... ... );
      fail ) );

sop_bas_progression([Ss,Aa,Tt,Bb],[Sop,Alt,Ten,Bas]) :-
    samekey(Ss,Aa,Tt,Bb,Sop,Alt,Ten,Bas), !;
sop_bas_progression([Ss,Aa,Tt,Bb],[Sop,Alt,Ten,Bas]) :- Ss>Sop, !, Bb<Bas;
sop_bas_progression([Ss,Aa,Tt,Bb],[Sop,Alt,Ten,Bas]) :- Ss<Sop, Bb>Bas;
sop_bas_progression([Ss,Aa,Tt,Bb],[Sop,Alt,Ten,Bas]) :-
    ( samekey(Ss,Aa,Tt,Bb,Sop,Alt,Ten,Bas), !;
    ( Ss>Sop -> Bb<Bas; Ss<Sop, Bb>Bas ) );

test_seventh_note([],_) :- !, fail;
test_seventh_note([Note|Rest],Key_scale) :-
    ( (Note-Key_scale) mod 12=:=5, !; test_seventh_note(Rest,Key_scale) );
test_seventh_note([Note|Rest],Key_scale) :-
    TTT is (Note-Key_scale) mod 12,
    ( TTT==5 -> true; test_seventh_note(Rest,Key_scale) );
```

Fig. 4-1 If-then-else in Harmonizer program

表　4－3　ハーモナイザのif-then-elseによる高速化

	if-then-else	or & cut	利得
荒城の月	２１．５１４　秒	２４．１１９　秒	１０．８％
赤とんぼ	０．５９８　秒	０．６５７　秒	９．０％
もみじ	１．７１０　秒	１．８７９　秒	９．０％

if-then-elseを用いることによって複数のクローズを１個にまとめることができたため，クローズ・ヘッドのユニフィケーション回数も減らすことができ，結果として上記の高速化が達成された。

プログラムの変更した部分を見直すと，次のようなことがわかる。

1) gen _harmony の第１クローズは入力データのエラーを検出した時の処理を行なうもので，通常はfailするパスである。
2) gen _harmony の第２クローズは，処理が完了した時の終了条件の判定を行なう部分であり，success するのは，１曲について１回だけである。
3) :progression_testの第２引数にはclause indexing が適用され，実際の判定は第３引数が[]かどうかの判断で行なわれる。
4) distribution_testでは，第２から第５クローズを１個のクローズにまとめることにより，ヘッド部のリストのユニフィケーションを減少させる効果がある。
5) sop _bas _progression の第２及び第３クローズにも同様の効果がある。
6) sop _bas _progression の第１クローズは，クローズ内ｏｒにするオーバーヘッドとヘッドのユニフィケーションの再実行によるオーバーヘッドの得失によってまとめるか分けたままにするかを考えるべきである。
7) test_seventh _noteはクローズ内ｏｒをif-then-elseに書きかえただけなので純粋にこの書きかえの効果のみが現われる。

8) 以上の点から見て，gen _harmony と:progression_testの書きかえは，高速化にはほとんど貢献せず，むしろ遅くなる方向への書きなおしになっている可能性が高いと考えられる。
9) if-then-elseの使用は，残りの３個の述語で効果を現わしていると考えられ，それによるクローズ内ｏｒの減少の他に，クローズ・ヘッドにおける構造体のユニフィケーション回数の減少にも良い効果を与えていると考えられる。

5. 終りに

逐次型推論マシンＰＳＩの組込み述語の一部にif-then-elseの制御構造をサポートするファームウェアを導入することによって，ｐｒｏｌｏｇプログラムの実行を高速化することが可能であることを示した。ファームウェア・ソフトウェアの改造量がそれほど多くなく，高速化が実現できるので，このサポート機構を用いることがｐｒｏｌｏｇプログラムの実行に適していると考えられる。今後このリポートを標準のシステムに採用し，システム全体の高速化をはかることを予定している。

謝辞

ファームウェアの改造に御協力いただいた，ＰＳＩ開発担当グループの諸氏に感謝します。

参考文献

1) 森 他, Hardware Design and Implementation of Personal Sequential Inference Machine (PSI), Proc. FGCS'84, pp. 398-409, (1984)
2) 横田 他, A Microprogrammed Interpreter for the Personal Sequential Inference Machine, Proc. FGCS'84, pp. 410-418, (1984)
3) 近山, Unique features of ESP, Proc. FGCS'84, pp. 292-298, (1984)
4) 横井 他, Sequential Inference Machine:SIM -Its Programming and Operating System-, Proc. FGCS'84, pp. 70-83, (1984)
5) D.Knuth, An impirical study of Fortran programs, Software-Practice and Experience, vol 1, no. 1, pp. 105-133, (1971)
6) D.L.Bowen et.al. DECsystem-10 PROLOG USER'S MANUAL, Univ. of Edinburgh, p.101, (1983)