ICOT Technical Report: TR-262

TR-262

並列論理型言語のコンパイル方式の誘導

神田陽治(富士通), 田中二郎

June, 1987

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

並列論理型言語の
コンパイル方式の誘導

神田陽治（富士通国際情報社会科学研究所）・　田中二郎（ＩＣＯＴ）

Concurrent Prolog のプログラムを、prologにコンパイルする方式は既に知られている。本論文では、prologで記述された Concurrent Prologインタプリタを基に、Concurrent Prolog プログラムを段階的に変換して行くことで、最終的にprologにコンパイルできることを示す。誘導の各段階はprolog処理系で実際に実行できる。例題を用いて、実行速度の改善される様子を調べた。

1．はじめに

Concurrent Prolog は、Shapiro によって提案された AND型並列論理型言語である（Shapiro 83）。あわせて開発された Concurrent Prologインタプリタは、prologで記述された動く仕様として意義深いものだったが、実行性能に難があった。上田と近山は、 Concurrent Prologのプログラムをprologへコンパイルする方式を開発し、この困難を解消した（Ueda 85 ）。これ以後、 Concurrent PrologをＣＰと記す。

彼等が開発したコンパイル方式は、Shapiro のインタプリタの考え方を素直に取り込んで実現した。よって、インタプリタの方式からコンパイルの方式へ、間をつないで見せうると期待できる。この論文では、その間の誘導を、appendプログラムを例題に用いて示す。出発点は、prologで記述されたＣＰのインタプリタと、ＣＰで記述されたappendプログラムである。これらを合わせて段階的に変換して行き、最後にprologで記述されたappendプログラムを得る。

さて、言語Ｌ１で記述された言語Ｌ２用のインタプリタと、言語Ｌ２で書かれたプログラムを組にして部分評価すると、言語Ｌ１にコンパイルされたプログラムが得られることは良く知られている。我々の場合は、Ｌ１がprolog、Ｌ２がＣＰの場合に相当する。もっとも、我々は部分評価の可能な道筋を示唆しただけであり、誘導の各段階の変換の正当性を保証してはいない。通常のプログラム変換では出発点のみが与えられるから、最終物がどのようなものになるかはわからず、元のプログラムに照らして正しいことの保証は重大である。しかしながら我々の場合には、「正しいと知れている」出発点と最終点が二つともに与えられている。正しく書かれたＣＰプログラムに対して、ＣＰインタプリタで実行したときと、コンパイルしてprolog処理系で実行したときの答えが同じになればよいのであって、それ以上の等価性は必ずしも要求されない。従って、バックトラック機構を持つprologでの、非決定性を持つ一般のプログラムを対象としたプログラム変換とは立場が異なる。「誘導」という言葉を用いたのは、そういう事情からである。

ところで、誘導法の存在は知識としてばかりでなく、実際に有用でもある。新しくコンパイルコードを設計する際のガイドラインを与えてくれる。誘導の各段階は、そのままprolog処理系で実行できるから、様々な方式の効果を実際に測定しうる。（Kursawe 86）に、このアイデアが明快に述べられている。彼はprologで記述したappendプログラムに次々とプログラム変換を施し、最後にWarren風のAbstract Prolog Machine codeが得られることを示した。

2．ＣＰのインタプリタ

(1)ＣＰで記述したＣＰインタプリタ

ＣＰの動作を、ＣＰ自身で記述することができる（Hirsch 86 ）。いわゆるメタインタプリタである。

```
(M0)   mcall(true).                                              % halt
       mcall((A,B)) :- mcall(A?),mcall(B?).                      % fork
       mcall(G) :-     system(G) | G.                            % system
       mcall(G) :-     nonsystem(G), G ≠true,  G≠(A,B) |         % reduce
                       reduce(G?,Body), mcall(Body?).
```

callは引数のゴールが、trueのとき終了し、複数のゴールに対しては全てが成功することを調べ、システム述語のときはすぐに実行し、そして、単独ゴールのときはreduceにより一回リダクションを施したのち、生成されたゴールを再帰的に解く。

(2)prologで記述したＣＰインタプリタ

prologで記述したShapiro のＣＰインタプリタを示す（Shapiro 83）。上記のメタインタプリタに類似した流れになっている。インタプリタの版は、solve/S0とreduce/R0 である。（種々の版を区別する必要があるときは、例えばsolve/S0のように、名前／識別子で区別する。）

```
(S0)   solve( ['$END'] ,-,-) :- !.                               % halt
       solve( ['$END'|H] ,  [] ,d) :- !, fail.
       solve( ['$END'|H] ,  ['$END'|T] ,nd) :- !,
              solve(H,T,d).
       solve( [ A|H ] ,T,-) :-                                   % system
              system(A), !, A,
              solve(H,T,nd).
```

```
        solve( [ A | H ] ,T,F) :-                    % reduce and then fork
              reduce(A,B,F,NF),
              schedule(B,H,T,NH,NT),
              !, solve(NH,NT,NF).
(R0)    reduce( A,B,-,nd ) :-                        % reduce
              guarded-clause(A,G,B),
              schedule(G,X,X,H,  ['$END' | T ] ),     % create a new goal queue
              solve(H,T,d),!.                         % solve guard G in it
        reduce( A, suspended(A),F,F ).
(C0)    guarded-clause(A,G,B) :-
              guarded-caluse(A,B1),find-guard(B1,G,B).
        guarded-clause(A,B) :-
              functor(A,F,N),functor(A1,F,N),clause(A1,B),
              unify(A,A1).                             % CP unify
        find-guard( (A | B),A,B) :-!.
        find-guard(A,true,A).
(C0)    schedule(true,H,T,H,T) :- !.                 % breadth-first
        schedule(suspended(A),H, [ A | T ] ,H,T) :- !.
        schedule((A,B),H,T,H2,T2) :- !,
              schedule(A,H,T,H1,T1),
              schedule(B,H1,T1,H2,T2).
        schedule(A, H, [ A | T ] ,H,T).
```

solve(H,T,F)は、一本の差分リスト形式のゴールキューH、Tと、一個のデッドロックフラグFを持つ。サイクルマーカ'$END'がキューの一巡を知らせる。solve は、キューの先頭からゴールを一つ取り出しては、リダクションを試みる。

reduce/R0 は、ゴールのリダクションを行う。リダクションできたとき、すなわちヘッドとユニファイ可能なクローズがあり(guarded-clause で調べる。unify はＣＰ用に拡張されたユニファイである)、かつガードが解けたとき(solveで調べる)には、展開されたゴールを返す。できないときは、suspended(A)を返してリダクション不能だったことを教える。第三と第四の引数はデッドロックフラグである。第三の引数から第四の引数へデッドロック情報が伝達される。リダクション可能なときは、第四引数をndに強制設定し、リダクション不可能なときは第三引数から第四引数へコピーする。デッドロックフラグの初期値は dである。サイクルマーカを見てから次にサイクルマーカを見るまでの間、一回もndに落とされなかったことを調べることで、デッドロックの検出ができる。

schedule/C0 は、リダクションの結果をゴールキューの後尾にbreadth-first スケジューリングする。スケジューリング方式はdepth-first でもよく、そのときはschedule/C0 の代わりにschedule/C1 を用い、ゴールキューの先頭に入れる。ただし、suspended(A)とマークされているときは、ゴールAは常にキューの後尾にスケジューリングされるものとする。

```
(C1)    schedule(true,H,T,H,T) :- !.                 % depth-first
        schedule(suspended(A),H, [ A | T ] ,H,T) :- !.
        schedule((A,B),H,T,H2,T2) :- !,
              schedule(B,H,T,H1,T1),
              schedule(A,H1,T1,H2,T2).
        schedule(A, H,T, [ A | H ] ,T).
```

3．ＣＰプログラムのコンパイル

上田と近山は、ＣＰプログラムのprologへのコンパイル方式〔Ueda 85 〕を開発した。基礎になるアイデアは二つある。第一は、インタプリタ方式をコンパイル方式に直す方法である。 solveが持ち回っているゴールキューとデッドロック検出フラグを、各ゴールが持ち回ることにして、solve を経なければならない手間を消す。第二は、scheduleではbreadth-first あるいはdepth-first が基本であったのを、各ゴールがカウンタを持ち回ることにして、リダクション木の深さを抑制できるdepth-first スケジューリングを実現したことにある。これを彼等は、制限の基本単位の深さをNとして、N-bounded depth-first schedulingと呼んでいる。二つのアイデアは互いに独立である。そこでＣＰインタプリタに組み込む場合に、都合四つのヴァリエーションがありえる。

	インタプリタ方式	コンパイル方式
breadth-/depth-first	(i) Shapiro のＣＰインタプリタ	(ii) 単純なprolog実行コード
N-bounded depth-first	(iii) 改造されたＣＰインタプリタ	(iv) 上田のprolog実行コード

4．で、(i)から(ii)のコンパイル方式を、5．で(iii)から(iv)のコンパイル方式を誘導して見せる。5．1では(iii)のN-bounded depth-first を組み込んだＣＰインタプリタを示す。(ii)のコンパイルの実例が4．4に、(iv)のコンパイルの実例が5．2にある。

4. 単純なコンパイル方式の誘導

ＣＰで書かれたappendプログラムを、ＣＰインタプリタsolve/S0を使って変換して行くと、prolog版のコンパイルコードが得られることを順を追って示す。ここでは単純なスケジューリング方式を使う。

変換は大きく三つの段階からなる。第一段階は、ＣＰ版のappendプログラムをprolog版に変換する。第二段階はＣＰインタプリタsolve/S0の定義を使って、得られたappendプログラムを変形していく。第三段階では、呼び出しの形を変える。最後に本質的ではないが、補助述語を展開して目的のコードが得られる。例として用いるＣＰプログラムappend/A0 を示す。

```
(A0)   append( [ X | Xs] , Ys, [ X | Zs] ) :-
             append( Xs? , Ys?, Zs ).
       append( [] , Ys , Ys ).
```

4. 1 第一段階

ＣＰ版のappend/A0 を、prolog版のappend/A1 に直す。reduce/R0 をappend/A0 で特殊化する。この変換のアイデアは〔Hirsch 86 〕から得た。後の段階の変換と比べると、かなり複雑な変換操作である。

(1) reduce化

reduce/R0 をappend/A0 で特殊化する。reduce(append(L,M,N),B,F0,F1) をreduce/R0 で部分評価する。二つのreduceの定義クローズのうち、あとの場合は簡単で、

```
reduce(append(L,M,N),suspended(append(L,M,N)),F,F).
```

となる。最初の方は、reduceはguarded-clauseでユニファイ可能なクローズを選び、そのガード部をsolveで解く。guarded-clauseの処理の核心は、clauseで同じヘッド名を持つクローズを集め、ＣＰ用に強化されたunify でヘッドユニフィケーションを行い、候補クローズを絞ることである。append/A0 が二つの定義クローズから成っているので、clauseは二つの答えを返しうる。そこでreduceの第一クローズをappend/A0 で特殊化した結果は、次の二つのクローズになる。

```
reduce(append(L,M,N),append(Xs?,Ys?,Zs),F0,nd) :-
      unify(append(L,M,N),append( [ X | Xs] , Ys, [ X | Zs] )),
      schedule(true,X,X,H, ['$END' | T ] ),solve(H,T,d),!.
reduce(append(L,M,N),true,F0,nd) :-
      unify(append(L,M,N),append( [] , Ys, Ys)),
      schedule(true,X,X,H, ['$END' | T ] ),solve(H,T,d),!.
```

unify は、引数の構造が少しでもわかれば、計算を進めることができる。そのための補助述語、ulist 、unilは〔Ueda 85〕で使われたものである。ulist(A,H,T)はunify(A, [H | T]) と、unil(L) はunify(L, []) と同じである。また、解くべきガードはtrueであるので、常にガードを解くsolve は成功する。そこで、最終的に次の三つのクローズが、reduce/R0 のappend/A0 特殊化として得られる。

```
(A1)   reduce(append(A1,Ys,A3),append(Xs?,Ys?,Zs),-,nd) :-
            ulist(A1,X,Xs),ulist(A3,X,Zs),!.
       reduce(append(A1,A2,A3),true,-,nd) :-
            unil(A1),unify(A2,A3),!.
       reduce(append(X,Y,Z),suspended(append(X,Y,Z)),F,F).
```

4. 2 第二段階

append/A1 を、append/A4 まで変形して行く。solve/S0を見ると、reduceを呼び出した後は必ずscheduleとsolve が呼び出されているのがわかる。これを順次append/A1 に移していく。solve/S0も、solve/S2からsolve/S4まで変形される。

(1) scheduleの取り込み

solve/S0の続く二つの呼び出し、

```
reduce(A,B,F,NF), schedule(B,H,T,NH,NT)
```

を融合する。二つの呼び出しの間でのみ共通な変数、B が消去できて、solve/S2では、

```
reduce-schedule(A,H,T,NH,NT,F,NF)
```

に融合できる。

```
(A2)   reduce-schedule(append(A1,Ys,A3),H,T,NH,NT,-,nd) :-
            ulist(A1,X,Xs),ulist(A3,X,Zs),!,
            schedule(append(Xs?,Ys?,Zs),H,T,NH,NT).
       reduce-schedule(append(A1,A2,A3),H,T,NH,NT,-,nd) :-
            unil(A1),unify(A2,A3),!,
            schedule(true,H,T,NH,NT).
       reduce-schedule(append(X,Y,Z),H,T,NH,NT,F,F) :-
```

```
                schedule(suspended(append(X,Y,Z)),H,T,NH,NT).
(S2)    solve( ['$END'] ,-,-) :- !.
        solve( ['$END'|H] ,  [] ,d) :- !, fail.
        solve( ['$END'|H] ,  T, nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                solve(NH,NT,d).
        solve( [A|H] ,T, -) :-
                system(A), !, A,
                solve(H,T,nd).
        solve( [A|H] ,T, F) :-
                reduce-schedule(A,H,T,NH,NT,F,NF),
                !, solve(NH,NT,NF).
```

solve/S2の第3クローズは後の便宜を考えて、scheduleを用いるように書き直した。schedule/C0 あるいはschedule/C1 の定義で展開すれば、solve/S0のそれに一致することは容易にわかる。

(2) solve の取り込み

solve/S2の続く二つの呼び出し、

```
reduce-schedule(A,H,T,NH,NT,F,NF),!,solve(NH,NT,NF)
```

を融合する。二つの呼び出しの間でのみ共通な変数、NH,NT,NFが消去できて、solve/S3では、

```
$(A,H,T,F)
```

に融合できる。solve/S2において、二つの呼び出しに挟まれた「！」は、reduce-schedule が決定的な述語であるので、その前に移動でき、solve/S3に示す位置に持って行ける。しかも最後のクローズなので、実のところ、この「！」は除去できる。

```
(A3)    $(append(A1,Ys,A3),H,T,-) :-
                ulist(A1,X,Xs),ulist(A3,X,Zs),!,
                schedule(append(Xs?,Ys?,Zs),H,T,NH,NT),
                solve(NH,NT,nd).
        $(append(A1,A2,A3),H,T,-) :-
                unil(A1),unify(A2,A3),!,
                schedule(true,H,T,NH,NT),
                solve(NH,NT,nd).
        $(append(X,Y,Z),H,T,F) :-
                schedule(suspended(append(X,Y,Z)),H,T,NH,NT),
                solve(NH,NT,F).
(S3)    solve( ['$END'] ,-,-) :- !.
        solve( ['$END'|H] ,  [] ,d) :- !, fail.
        solve( ['$END'|H] ,  T, nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                solve(NH,NT,d).
        solve( [A|H] ,T, -) :-
                system(A), !, A,
                solve(H,T,nd).
        solve( [A|H] ,T, F) :-                  % !,
                $(A,H,T,F).
```

(3) solve の消去

今度はsolve を消去する。solve はSを呼び、Sはsolve を呼ぶ。だから、介在物に過ぎないsolve は消去できる。solve の働きは、ゴールキューの先頭を取り出して場合分けする共通の仕事と、場合分け後の個々の仕事に分離できる。この共通する前半の仕事をappend/A3 に移す。まずsolve/S3を前後に分ける。

```
(S3')   '$2'('$END', [] , -, - ) :- !.
        '$2'('$END',  H, [] ,d ) :- !, fail.
        '$2'('$END',  H,  T,  nd) :- !,
                schedule(suspended('$END'),H,T,NH,NT),
                solve(NH,NT,d).
        '$2'(A, H, T, -) :-
                system(A), !, A,
                solve(H,T,nd).
        '$2'(A, H, T, F) :-
                $(A,H,T,F).
        solve( [A|H] ,T, F) :-
                '$2'(A,H,T,F).
```

solve クローズの展開により、solve(NH,NT,d)などの呼び出しを、NH = [G | NH2] , '$2'(G,NH2,NT,d) のように展開できる。ところで '$END' やシステム述語は、$の第一引数には出てくることはない。そこでユーザ定義の述語も、'$2'を介さず直接$を呼んでもよい。そこで、$と'$2'の呼び出しは区別する必要がなくなり、次のプログラムが得られる。同時に、システム述語を具体的に列挙するよう変更する。

```
(A4)   $(append(A1,Ys,A3),H,T,-) :-
            ulist(A1,X,Xs),ulist(A3,X,Zs),!,
            schedule(append(Xs?,Ys?,Zs),H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,nd).
       $(append(A1,A2,A3),H,T,-) :-
            unil(A1),unify(A2,A3),!,
            schedule(true,H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,nd).
       $(append(X,Y,Z),H,T,F) :-
            schedule(suspended(append(X,Y,Z)),H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,F).
(S4)   $('$END', [] , -, - ) :- !.
       $('$END', H, [] ,d ) :- !, fail.
       $('$END', H, T, nd) :- !,
            schedule(suspended('$END'),H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,d).
       $( unify(X,Y),H,T,-) :- !,                    % EXAMPLE
            unify(X,Y),                               %  such a clause for
            H = [ G | H2] , $(G,H2,T,nd).             %  each system-predicate
```

4．3　第三段階

append/A4 、solve/S4とも、すべてSクローズになった。$の呼び出しは、その第一引数で、どのSクローズが呼び出されるか決まってしまう。すなわち、$という名前は余分であって消去でき、第一引数を述語に昇格できる。この作業を二ステップに分けて行う。最初のステップは過渡的なもので、$をヘッドから消去する。第二ステップで、$をボディ部から消去する。

(1)　ヘッドからの$の消去

$クローズのヘッドを変形する。第一引数を述語に昇格させるが、その際、残りの引数も取り入れる。

```
append( 元の引数, H,T,F )
```

結局、appendの元々の引数に、solve が引き回していた引数が取り込まれたことになった。

しかし、$の呼び出しは元のままなので、この間を埋めるため、solve/S5に示す補助述語$が必要となる。これは次のステップで消去される。

```
(A5)   append(A1,Ys,A3,H,T,-) :-
            ulist(A1,X,Xs),ulist(A3,X,Zs),!,
            schedule(append(Xs?,Ys?,Zs,HO,TO,FO),H,T,NH,NT),
            NH = [ G | NH2 ] ,$(G,NH2,NT,nd).
       append(A1,A2,A3,H,T,-) :-
            unil(A1),unify(A2,A3),!,
            schedule(true,H,T,NH,NT),
            NH = [ G | NH2 ] ,$(G,NH2,NT,nd).
       append(X,Y,Z,H,T,F) :-
            schedule(suspended(append(X,Y,Z,HO,TO,FO)),H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,F).
(S5)   '$END'( [] , -, - ) :- !.
       '$END'( H, [] ,d ) :- !, fail.
       '$END'( H, T, nd) :- !,
            schedule(suspended('$END'(HO,TO,FO)),H,T,NH,NT),
            NH = [ G | NH2 ] , $(G,NH2,NT,d).
        unify(X,Y,H,T,-) :- !,                       % EXAMPLE
            unify(X,Y),                               %  such a clause for
            H = [ G | H2] , $(G,H2,T,nd).             %  each system-predicate
( * )  $(G,H,T,F) :-
            functor(G,-,A),
            arg(A,G,F),
            A1 is A-1, arg(A1,G,T),
            A2 is A-2, arg(A2,G,H),
            call(G).
```

(2) ボディ部からのSの消去

前ステップで導入した、solve/S5の補助述語Sを消去する。その役目は、ゴールに拡張部分の引数の値を引き渡す事にあった。Sの呼び出しの直前は、必ずゴールキューからゴールを取り出してくる操作である。ゴールキュー要素の構造を工夫することで、ゴールキューからのゴールの取り出しと、拡張部分への値の設定を同時に行える。このアイデアは〔Ueda 85〕による。

ゴールキューに入れる形を、次のように拡張されたゴールと拡張引数の組とする。

$(head(A1,...,Am,B1,...,Bn), B1,...,Bn)

ここで、B1,...,Bnの引数部分は仮引数の働きをし、全体でλ式に似た構造である。外側のB1,B2 等に値をユニファイすることで、head内のB1,B2 等に値を設定できる。

今回の例では、

$(append(元の三つの引数、H0,T0,F0), H0,T0,F0)

こうすると、これがゴールキューから取り出されたとき、次のようにして値を与えられる。

H = [$(G, H2, T値, F値) | H2]

ここでは、先頭の要素 $(G,H0,T0,F0)が取り出されると同時に、新しいゴールキューの先頭 (H=[- | H2] のH2) が、仮引数 (H0) を通じて、G 内のH0に伝えられている。、また、T0を通して T値が、F0を通して F値が、G 内のT0とF0に伝えられている。

二つのステップの結果を通して眺めてみると、Sクローズが消去される一方、それに代わる構造体がゴールキューに移されたように見える。

```
(A6)    append(A1,Ys,A3,H,T,-) :-
              ulist(A1,X,Xs),ulist(A3,X,Zs),!,
              schedule($(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0),H,T,NH,NT),
              NH = [ $(G,NH2,NT,nd) | NH2 ] ,
              call(G).
        append(A1,A2,A3,H,T,-) :-
              unil(A1),unify(A2,A3),!,
              schedule(true,H,T,NH,NT),
              NH = [ $(G,NH2,NT,nd) | NH2 ] ,
              call(G).
        append(X,Y,Z,H,T,F) :-
              schedule(suspended($(append(X,Y,Z,H0,T0,F0),H0,T0,F0)),H,T,NH,NT),
              NH = [ $(G,NH2,NT,F) | NH2 ] ,
              call(G).
(S6)    '$END'( [] , -, - ) :- !.
        '$END'( H, [] , d) :- !, fail.
        '$END'( H, T, nd) :- !,
              schedule(suspended($('$END'(H0,T0,F0),H0,T0,F0)),H,T,NH,NT),
              NH = [ $(G,NH2,NT,d) | NH2 ] ,
              call(G).
         unify(X,Y,H,T,-) :- !,                    % EXAMPLE
              unify(X,Y),                            %  such a clause for
              H = [ $(G,H2,T,F) | H2] ,              %  each system-predicate
              call(G).
```

4．4 最終段階

スケジューリング方式を決めれば、スケジューリングされる対象が常に具体的なので、scheduleを消去できる。breadth-firstスケジューリングに固定して展開したのが、append/A7 とsolve/S7である。なお、システム述語用のクローズはsolve/S7とsolve/S6で同じなので省略した。得られた結果は、breadth-first を組み込んだコンパイルコードであり、3．での分類の、ヴァリエーション（ii）にあたる。

```
(A7)    append(A1,Ys,A3,H,T,-) :-
              ulist(A1,X,Xs),ulist(A3,X,Zs),!,
              T = [ $(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0) | NT] ,
              H = [ $(G,H2,NT,nd) | H2] ,
              call(G).
        append(A1,A2,A3,H,T,-) :-
              unil(A1),unify(A2,A3),!,
              H = [ $(G,H2,T,nd) | H2] ,
              call(G).
        append(X,Y,Z,H,T,F) :-
              T = [ $(append(X,Y,Z,H0,T0,F0),H0,T0,F0) | NT] ,
              H = [ $(G,H2,NT,F) | H2] ,
```

```
                call(G).
(S7)    '$END'( [] , -, - ) :- !.
        '$END'( H, [] , d) :- !, fail.
        '$END'( H, T, nd) :- !,
                T = [ $('$END'(H0,T0,F0),H0,T0,F0) ¦NT] ,
                H = [ $(G,H2,NT,d) ¦H2] ,
                call(G).
```

今度は、scheduleをdepth-first スケジューリングに固定して展開する。その結果がappend/A8 である。append/A7 と異なるのは第一クローズだけであり、solve はsolve/S7と同じである。ところがキューの先頭に入れ、直ちに出すのは無駄であるので、さらにappend/A8'のように変換できる。得られた結果は、depth-first を組み込んだコンパイルコードであり、3．での分類の、ヴァリエーション（ii）にあたる。

```
(A8)    append(A1,Ys,A3,H,T,-) :-
                ulist(A1,X,Xs),ulist(A3,X,Zs),!,
                NH = [ $(append(Xs?,Ys?,Zs,H0,T0,F0),H0,T0,F0) ¦H ] ,
                NH = [ $(G,NH2,T,nd) ¦NH2 ] ,
                call(G).
(A8')   append(A1,Ys,A3,H,T,-) :-
                ulist(A1,X,Xs),ulist(A3,X,Zs),!,
                append(Xs?,Ys?,Zs,H,T,nd).
```

5．上田、近山のコンパイル方式の誘導

N-bounded depth-fisrt スケジューリングをＣＰインタプリタに組み込みたい。そのようなインタプリタは実行情報を操作できなくてはならず、それぞれのゴールの実行属性を陽に出す必要がある。ゴールを一つのプロセスと見れば、実行属性とはオペレーティングシステムでいうＰＣＢ (Process Control Block)のことである。インタプリタはゴールに付与されたＰＣＢ内の情報を参照しつつ、プロセスであるゴールをスケジューリングする。

ＰＣＢ情報を顕在化には、ゴールキューの要素の形を、「ゴール @ PCB」の形とする。@ の後ろの部分がＰＣＢを表す。ＰＣＢを扱えるＣＰインタプリタの概略を示す。

```
(R1)    reduce( Head @ PCB, Body, -, nd) :-
            firmware( ポリシー,PCB, GuardPCBs, BodyPCBs),
            guarded-clause(Head,Guard0,Body0),
            attach(Guard0,GuardPCBs,Guard),
            schedule(Guard,X,X,H, ['$END' ¦T ] ),  % create a new queue
            solve(H,T,d),!,                        % solve Guard in it
            attach(Body0,BodyPCBs,Body).
        reduce( Goal @ PCB, suspended(Goal @ SuspendedPCB),F,F) :-
            firmware( ポリシー,PCB, SuspendedPCB ).
```

solve は、オリジナルのＣＰインタプリタのsolve/S0と同じでよい。reduce/R1 が、ゴール@ PCB の形式を扱う。ここでfirmwareが、指定されたポリシーを実現するためのメカニズムであり、親ゴールのPCB から、子供ゴールのPCB リストを計算する。attachは、単に整合をとるための補助述語で、第一引数のゴール並びと第二引数のPCB リストを順に組み合わせる。その目的は新しいゴールに新しいPCB を与え、ゴール@ PCB の形式に戻すことにある。

5．1　N-bounded depth-firstスケジューリングを組み込んだインタプリタ方式

PCB を、リダクション木の深さ制限用のカウンタB と、その上限値BCの、(B,BC)の形の二つ組とする。N-bounded depth-fisrt スケジューリングを組み込んだreduce/R2 を示す。solve/S0とあわせ、3．でのヴァリエーション（iii）にあたる。

```
(R2)    reduce( Head@(B,BC), Body, -, nd) :-
            B>0, NB is B-1,
            guarded-clause(Head,Guard0,Body0),
            attach(Guard0,(BC,BC),Guard),
            schedule(Guard,X,X,H, ['$END' ¦T ] ),  % create a new queue
            solve(H,T,d),!,                        % solve Guard in it
            attach(Body0,(NB,BC),Body).
        reduce( Goal@(-,BC), suspended(Goal@(BC,BC)),F,F).
```

この場合のポリシーは、「カウンタB がまだ残っているときはリダクションを続けてもよいが、そうでないときはサスペンド扱いとする。その際はカウンタをBCに戻す。」というものである。子供ゴールが持たされるPCB は皆同一なので、attachは簡単になる。なお、solve 内で呼ばれるscheduleは、depth-first スケジューリングに固定する。breadth-first スケジューリングを行うことは無意味だからである。

5．2　N-bounded depth-firstスケジューリングを組み込んだコンパイル方式

append/A0 に、reduce/R0 の代わりにreduce/R2 を用い、4．と同じプログラム変換を施して行くと、N-bounded depth-first スケジューリング組み込みのコンパイルコードappend/A10が得られる。これは3．でのヴァリエーション（iv）にあたる。

配慮がいるのは、第一段階のreduce化のときである。第一段階では、reduce/R2 に合わせて、append/A0 にreudce化を施す。その結果、次のappend/A9 が得られる。

```
(A9)    reduce(append(A1,Ys,A3)@(B,BC),append(Xs?,Ys?,Zs)@(NB,BC),-,nd) :-
              B>0,ulist(A1,X,Xs),ulist(A3,X,Zs), !, NB is B-1.
        reduce(append(A1,A2,A3)@(B,BC),true,-,nd) :-
              B>0,unil(A1),unify(A2,A3), !.
        reduce(append(X,Y,Z)@(-,BC),suspended(append(X,Y,Z)@(BC,BC)),F,F).
```

ところでＰＣＢ情報は必ずゴールと一緒に扱われるものだから、ＰＣＢ情報を各ゴールの引数に取り込んでしまっても問題は生じない。

```
(A9')   reduce(append(A1,Ys,A3,B,BC),append(Xs?,Ys?,Zs,NB,BC),-,nd) :-
              B>0,ulist(A1,X,Xs),ulist(A3,X,Zs), !, NB is B-1.
        reduce(append(A1,A2,A3,B,BC),true,-,nd) :-
              B>0,unil(A1),unify(A2,A3), !.
        reduce(append(X,Y,Z,-,BC),suspended(append(X,Y,Z,BC,BC)),F,F).
```

第二段階以降は、4．と全く同じ手順で変換して行けばよい。そして最終結果として、次が得られる。

```
(A10)   append(A1,Ys,A3,B,BC,H,T,-) :-
              B>0,ulist(A1,X,Xs),ulist(A3,X,Zs), !, NB is B-1,
              append(Xs?,Ys?,Zs,NB,BC,H,T,nd).
        append(A1,A2,A3,B,BC,H,T,-) :-
              B>0,unil(A1),unify(A2,A3), !,
              H = [ $(G,H2,T,nd) |H2] ,
              call(G).
        append(X,Y,Z,-,BC,H,T,F) :-
              T = [ $(append(X,Y,Z,BC,BC,H0,T0,F0),H0,T0,F0) |NT] ,
              H = [ $(G,H2,NT,F) |H2] ,
              call(G).
```

6．誘導法の利用

appendプログラムは、ガード部を持たないＦＣＰプログラムであった。ＦＣＰとはガード部に組み込み述語しか許さないＣＰのサブセットである。〔Ueda 85〕で用いられた例題もまた暗黙に、ＦＣＰプログラムであった。ガード部にどんな呼び出しが現れてもよいＣＰプログラムのコンパイルコードを、誘導法を利用して設計してみる。例題として、メタインタプリタmcall/M1をコンパイルしてみる。mcall/M0のreduceを、具体的にプログラムしてみせたのが、mcall/M1である。リダクションできそうなクローズを集めるclausesと、ひとつにクローズを絞り込むresolve で実現している。Shapiro は読出記法「？」を普通の項として実装した。そこで余分な「？」適宜とり去る必要があり、system2,exec2,clauses はそれを行う。

```
(M1)    mcall(true).
        mcall((A,B)) :- mcall(A?), mcall(B?).
        mcall(A) :- system2(A) | exec2(A).
        mcall(A) :- clauses(A,Clauses) |
              resolve(A,Clauses,Body), mcall(Body?).
(#)     resolve(A, [(Head:-Guard | Body) |Cs] ,Body) :-
              unify(A,Head),mcall(Guard) | true.
        resolve(A, [(Head:- Body) |Cs] ,Body) :-
              unify(A,Head),Body ≠ (- | -) | true.
        resolve(A, [ C | Clauses ] ,Body) :-
              resolve(A,Clauses,Body) | true.
```

これはＦＣＰプログラムではない。なぜならresolve の第一クローズにはmcall(Guard)の、第三クローズには resolve(A,Clauses,Body)の再帰呼び出しがあるからである。ここで、誘導法を知り、誘導の各段階に沿って考え行けば、ＣＰのユーザ定義のガード述語をコンパイルする方法が機械的に得られる。

誘導の第一段階ではreduce/R0 をmcall/M1について特殊化する。以後、(#) のクローズで誘導過程を説明する。reduce(resolve(L,M,N),B,F0,F1)を、(#) で特殊化した中間結果は、次のようになる。

```
(#1)    reduce(resolve(L,M,N),true,F0,nd) :-
              unify(resolve(L,M,N),resolve(A, [(Head:-Guard | Body) |Cs] ,Body),
              schedule((unify(A,Head),mcall(Guard)),X,X, ['$END' | T ] ),
              solve(H,T,d),!.
```

appendの場合と同様に、unify の計算をさらに進めることができる。一方、ガードの方は、unify が組み込み述語であり、かつsolve によりすぐに実行されてしまうので、単独に解いても同じである。結局、

```
(#2)    reduce(resolve(A, Arg2, Body),true,-,nd) :-
                ulist(Arg2,(Head:-Guard | Body),Cs),unify(A,Head),
                schedule(mcall(Guard),X,X, ['$END' | T ] ),
                solve(H,T,d),!.
```

あとは、appendプログラムと同様な変換を施す。最終的に得られるmcall/M1の(#) のクローズに対するコンパイルコードを示す。単純なスケジューリング方式の、3．のヴァリエーション（ii）の場合のコンパイルコードである。

```
(#3)    resovle(A, Arg2, Body, H,T,-) :-
                ulist(Arg2,(Head:-Guard | Body),Cs),unify(A,Head),
                NH = [ $('$END'(HO,TO,FO),HO,TO,FO) | NT] , mcall(Guard,NH,NT,d),!,
                H = [ $(G,H2,T,nd) | H2] , call(G).
```

7．実行性能の段階的改善

(1) 例題

インタプリタからコンパイルコードへのプログラム変換の各段階は、実際に実行することができる。各段階の実行速度を調べることで、各変換がどの程度強く、最終的な性能向上に役立ったのかを知りうる。

ここでは性能を総実行時間で計る。測定に使ったprolog処理系は、VAX UNIX 4.2BSD 用のQuintus prolog release 1.5であり、結果を表1に示す。出発点のＣＰプログラムから最終目標のprologまでの計８段階のそれぞれについて、以下に述べる三つの例題の実行性能を測定した。簡単なスケジューリング方式でdepth-first を使う場合、breadth-first を使う場合、さらに 10-bounded depth-first （木の深さ制限の単位が10) を使う場合に分けて測定した。さらに、prolog処理系の処理方式による違いを調べるために、Quintus prolog処理系のコンパイラを使う場合と使わない場合にも分けた。表の値は、第一段階については実行時間そのものを、その他の段階については、第一段階との実行時間の比で示している。

なお、インタプリタ方式のprolog処理系であるC-prologを使って測定したところ、表1のインタプリタの場合と同様な結果が得られた。

測定に用いた例題は、三つで、

[nrev] ナイーブな方法で逆順リストを求めるプログラムreverse を、3 段のパイプラインで動かす。

```
        nreverse(3, [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16] ,S).
```

すなわち、reverse([1,2,3,...] ,X),reverse(X?,Y),reverse(Y?,S).のこと。

[qsrt] クィックソート法でランダムリストをソートする。答えは差分リストで集める。

```
        qsort( [17,26,13,21,5,1,20,9,3,27,15,25,11,30,24,8,2,
                28,29,4,23,19,16,22,31,6,10,14,32,12,7,18] ,S).
```

[mcal] メタコール (mcall/M1) 上で、qsort([4,2,3,5,1] ,S) を実行する。

```
        mcall( qsort( [4,2,3,5,1 ] ,S)).
```

(2) 結果

prolog処理系の方式が、大きく影響することがわかる。インタプリタの場合、大まかに言って、誘導を進めるに応じて実行速度も改善されて行く。（ただし、第三段階のヘッドからＳを消去する段階は、全体からＳを消去するための過渡的段階であり、考慮に入れるべきではない。）しかしコンパイルした場合は、第三段階で一回揺れ返しがある上に、例題によりスケジューリング方式により、改善の様子にバラツキが目立つ。prologコンパイルの最適化戦略にうまく乗れるかどうかが効くのだと推察される。

第一段階の変換はreduce/R0 を特殊化する過程である。この変換では、コンパイルしない場合には 5倍から 7倍、コンパイルした場合は、 5倍から20倍の改善が見られた。第二段階はsolve/S0内の呼び出しを少しずつ外に分散していく過程であり、全体の実行性能には影響を与えないと推察される。実際、第二段階は、コンパイルするしないに関わらず改善効果がほとんど見られない。少し良くなるケースもあれば少し悪くなるケースもあるという具合である。第三段階でＳを完全に消去する操作は、インタプリタを使う場合には改善となるが、コンパイラを使った場合には逆に悪くなってしまった。補助述語の展開には、常にまあまあの改善効果がある。

8．課題

誘導の各段階を特徴付ければ、第一段階は部分評価による特殊化、第二段階は呼び出しの並行移動、第三段階は、引数項の、述語への昇格、最終段階は述語の展開となる。このうち第三段階の変換は、多分に技巧的である。汎用のプログラム変換の技法を補う方法として、この種の変換技法も開発すべきと思える。

本論文ではＣＰを AND型並列論理型言語の例として選んだが、その他の AND型並列論理型言語でも、同様な結果が示しうる。実際、上田〔Ueda 85 〕では、ＧＨＣのコンパイルも扱っている。

通常のメタインタプリタをprologへコンパイルできることを7．で示した。興味ある話題は、機能強化したメタインタプリタから、機能強化されたコンパイルコードを得る手続きの自動化である。〔Hirsch 86 〕は、ＣＰの上で、この問題を扱っている。

9. まとめ

上田と近山によって開発されたＣＰプログラムのprologへのコンパイル技法は、一見すると複雑に見える。それはいくつかの工夫が複合して現れているからである。この論文では、Shapiro のＣＰインタプリタから彼等のコンパイルコードまで、順次変換して行く道筋を示すことにより、個々の工夫を分離して見せることに成功した。具体的には、ＣＰで記述されたappendプログラムに、prologで記述したＣＰインタプリタを組み込んでいくことで、最終的にprologにコンパイルされたappendプログラムが得られることを示した。

誘導段階	例題	Quintus prolog インタプリタで実行 Depth	Breadth	10-Depth	Quintus prolog コンパイラで実行 Depth	Breadth	10-Depth
初期段階 ＣＰ プログラム	nrev	7.15	5.01	6.05	20.4	14.4	8.89
	qsrt	7.67	4.59	5.44	22.2	15.3	8.74
	mcal	5.32	5.38	5.05	4.79	4.84	4.35
第一段階 reduce化	nrev	14.83 sec	25.97 sec	18.10 sec	0.77 sec	1.35 sec	1.70 sec
	qsrt	10.42 sec	41.67 sec	16.22 sec	0.50 sec	1.78 sec	1.52 sec
	mcal	13.90 sec	14.05 sec	15.12 sec	2.82 sec	2.83 sec	3.20 sec
第二段階 schedule の取り込み	nrev	1.03	1.04	1.02	1.00	0.98	0.99
	qsrt	1.01	1.01	1.04	1.03	0.87	0.99
	mcal	1.01	1.01	1.02	0.99	0.99	1.02
第二段階 solve の取り込み	nrev	0.84	0.84	1.06	0.89	0.93	0.86
	qsrt	0.91	0.87	1.08	0.93	0.76	0.87
	mcal	0.96	0.97	1.04	0.98	0.99	0.99
第二段階 solve の消去	nrev	0.88	0.88	1.09	0.78	0.77	0.80
	qsrt	0.96	0.94	1.12	0.80	0.68	0.83
	mcal	0.99	1.00	1.06	0.98	0.99	0.99
第三段階 ヘッドからの Ｓの消去	nrev	0.94	1.00	1.07	2.37	2.64	1.24
	qsrt	0.83	0.81	0.87	1.97	2.13	0.98
	mcal	0.94	0.97	0.95	1.11	1.12	1.01
第三段階 ボディからの Ｓの消去	nrev	0.60	0.61	0.78	2.24	2.43	1.12
	qsrt	0.63	0.52	0.69	1.87	1.95	0.91
	mcal	0.83	0.84	0.85	1.08	1.10	0.99
最終段階 補助述語 の展開	nrev	0.36	0.53	0.56	0.74	2.23	0.45
	qsrt	0.46	0.48	0.56	0.93	1.78	0.60
	mcal	0.77	0.80	0.80	0.99	1.06	0.91

表１　誘導の各段階での実行性能の比較
第一段階の値は実行時間、それ以外の段階の値は、第一段階に対する実行時間の比を表す。

謝辞
本研究は、第５世代コンピュータ・プロジェクトの一環として実施しています。研究の機会を提供して頂いた、ICOT第１研究室、富士通国際研の北川会長ならびに榎本所長に感謝します。なお、査読者ならびにプログラム委員の貴重なコメントを論文の内容に十分反映できなかった責は、すべて著者らにあります。

参考文献

〔Hirsch 86〕 Hirsch, M., Silverman, W. and Shapiro, E.: Layers of Protection and Control in the Logix System, CS86-19, Weizmann Instit., 1986.

〔Kursawe 86〕 Kursawe P.: How to invent a Prolog machine, in Proc. of 3rd Inter. Conf. on Logic Prog., 1986, pp.134-148.

〔Shapiro 83〕 Shapiro, E.: A subset of Concurrent Prolog and its Interpreter, Tech. rep. TR-003, ICOT, 1983.

〔Ueda 85〕 Ueda, K, and Chikayama, T.: Concurrent Prolog Compiler on top of Prolog, In Proc. of Symp. on Logic Prog., 1985, pp.119-126.