OR並列Prolog(POPS)による構文解析

平川秀樹　古川康一
(財団法人 新世代コンピュータ技術開発機構)

1. はじめに

論理型言語Prologは、Horn logicをベースとし、unification とbacktrackingの機能を有している。Horn logicでは、述語のAND,ORは論理的な意味を持つのみであるが、Prologでは、このAND,ORの結合に関して順序づけを行ないoperational な意味を与えることにより、プログラムの実行を行なっている。すなわち、AND goalは記述の左から右に、OR goal はbacktrack 時に上から下に順次解釈される。このPrologのoperational な意味は、通常のsequentialなprogramming を可能としている反面、ある種の分野に於いては好ましくない結果をもたらす。例えば自然言語処理に於ける構文解析が挙げられる。自然言語の構文解析では、CFG(Context Free Grammar) の規則はHorn clause に対応づけられ、DCG [Pereira 80]に見られるように非常に素直な形で、かつ高記述性を持った形でPrologに埋め込まれる。しかしながら、自然言語処理の持つ特徴を考慮した場合に、次の様な問題点がある。

(1) 計算の重複

自然言語の構文解析には複数個の解が存在し、通常の文に対して構文解析結果が数十のオーダで出力されることはめずらしくない。この様な多義性はPrologのbacktrackingの機能によって得られるが、backtrackingは同じ計算を何度も繰り返し実行するという効率の悪さを持っている。例えば、"vp-->vt,np" , "vp-->vt,np,pp"という2つの規則があった場合には、"vt,np" の部分は完全に重複計算されてしまう。

(2) derivation cycle

自然言語は、本質的にrecursive な構造を有し、構文規則もそれを反映してcycle を含む規則となることが多い。たとえば、接続詞についての規則では"np-->np,and,np"があげられる。この様な規則をProlog流のtop down and depth firstのstrategyで解釈すれば、infinite loop に陥り、計算は終了しない。

(2) の問題点の解決法として、CFG の規則をbottom up serial に実行するPrologのプログラムに変換する方式 BUP [松本 83] が提案されている。BUP の場合には次の問題がある。

(3) ε規則の処理

自然言語では単語の省略等は頻繁に行なわれるが、これはCFG 規則ではε規則に対応する。ε規則をbottom up に解釈した場合には、無から有を生成する規則となり、単語の切れ目(無)すべてにおいて、ある文法カテゴリ(有)を仮定する結果となり、効率上問題となる。

以上の問題を解決する1つの方法として、筆者らは、Pure Prolog プログラムをOR並列に実行するグラフリダクションシステムPOPS(or-Parallel Optimizing Prolog System)をConcurrent Prolog 上にimplement した。POPSは、backtrackingに見られる様な同一計算の重複は行なわない、および、cycle を含むようなプログラムでも無限ループに陥らないという機能を、同一計算は高々一度しか実行しないというグラフリダクションメカニズムにより実現している。第2章では、グラフリダクションメカニズムとPOPSの計算モデル, 第3章ではConcurrent Prolog によるPOPSの記述について、第4章ではPOPSによる構文解析について述べる。

2. 基本計算モデル

本章では、グラフリダクションメカニズムとPOPSにおける計算モデルについて述べる。

2.1 グラフリダクションメカニズム

Prologプログラムの実行過程は、parent clausesからresolvant の生成の繰り返しであり、言葉を変えれば、goalが、ルールを適用し [Turner 79] 、自分自身をself-modify していく過程である。これはリダクションの定義と言える。すなわち、Prologプログラムの実行過程と、リダクションメカニズムとの間に親和性を見出すことができる。

リダクションには、いくつかの種類があるが、「同一の式の評価は、ポインタにより共有される。よって同一の式の評価は、高々一度しか実行されず、その結果は、共有しているものすべてに知らされる。」のが、グラフリダクションメカニズムである。グラフリダクションメカニズムをPrologプログラムの実行過程に適用すれば、同一termの評価は、ポインタにより共有されるので、同一termの重複計算を避けることができ、derivaration cycleを検出することができる。

POPSでは、後に詳細な説明をするが、同一termの検出をボード上で行い、同一termの評価はConcurrent Prolog の通行チャネルを通じて共有される。そしてそのtermのreduction 結果は、チャネルを通じて結果を待っているすべてのプロセスへ知らされる。

2.2 対象言語およびそのInterpretation

POPSが対象とする言語は、Pure Prolog であり、一般に次の

形式をしたDefinite Clause の集合である。

(a) H <-- G1,G2, ... ,Gn　(n >= 1)
(b) H <-- true.

H およびGi(1 <= i <= n) は、Prologで言うリテラルであり、trueは恒真を表わす特殊なリテラルである。Prologと同様に、(b) において'<-- true' は省略して記述してよい。また、Pure Prolog では、実行制御オペレータのカットや'not' 等のevaluable predicate は含まれない。POPSでは、Pure Prolog を、subgoal のAND 結合についてはserialに、OR結合についてはparallelに実行する。すなわち上記の(a),(b) をそのままプログラムとすれば、Gi,Gj(i < j)ではGiが計算（証明）されるまでGjは計算されないが、(a),(b) は同時に計算される。

2.3 構成要素

POPSは図１に示すように、プロセス、チャネル、ボード、およびホーンデータベース(HDB) の４つの要素から構成される。

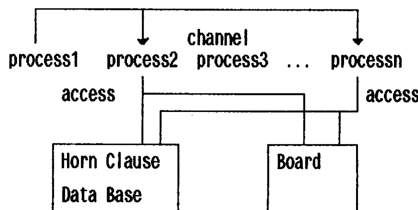


図１　POPSの構成

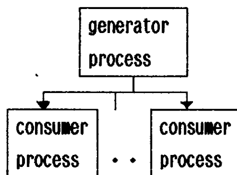


図２　プロセス／チャネル

プロセスは、計算の実行主体であり、任意個数存在できる。プロセスは、計算途中のclauseに対応し、例えばH<--G1,G2 といったclauseを内部に持っている。この際計算の環境は、clause の変数に実際の値がinstantiate されているという形式で保持されている。プロセスには　activeプロセスとwaiting process の２種があり、waiting プロセスは他のプロセスからデータを受取るまでwaitしている。チャネルはプロセス間の通信路であり、計算過程で動的に生成される。チャネルを通して送られるデータをメッセージと呼ぶ。メッセージの伝達方向は一方向とし、メッセージの送り手となるプロセスをgenerator 、受け手となるプロセスをconsumerと呼ぶ。この区別は相対的なものであり、１つのプロセスが同時にgenerator およびconsumer の２役を兼ねる事ができる。また、１つのgenerator プロセスは、１つのチャネルを用いて複数個のconsumerプロセスにメッセージを同時に送ることが可能である（図２）。また、逆に１つのconsumerが複数個のgenerator と接続可能である。ボードは、プロセスからアクセスされる記憶領域であり、その時点で計算が進行しているsubgoal すべてとそのsubgoal に対する計算結果（メッセージ）が送り出されてくるchannel を保存する役割を持っている。例えば　subgoal a(x)が呼び出されると、それに対し１つのチャネルC が生成されa(x)とC' が１組となりボードに登録される。このチャネルC からはsubgoal a(x)に対する解、たとえばa(1),a(2)..., が送られてくることになる。このchannel とsubgoal (term)を１組としたデータをチャネルヘッドペアと呼ぶこととする。チャネルヘッドペアは次の様に記述する。

Channel+Head

HDB は、Pure Prolog のclauseの集合であり、プロセスによってアクセスされる。

2.4 実行メカニズム

POPSでは、複数のプロセスがメッセージの交換を行ないながら計算が進行する。本節では、プロセスのより詳細な説明を行ない、簡単な例を示し、POPSの実行メカニズムを示す。なお、簡単のために、全ての述語は引数を持たないこととする。引数については 2.8で述べる。

プロセスは、'Status'，'Head'，'Goals '，'Input-Channel ' および 'Output-Channel' の５つの要素によって規定される。これを図式的に次の様に示す。

process(Status,Head,Goals,Input-Channel,Output-Channel)

'Status'は、プロセスの状態を表わし、'active' または 'waiting'のいずれかとなる。activeプロセスは、それ自身で計算を進めるが、waiting プロセスはメッセージを受けるまで何もしない。Headは１つの述語（term）であり、最終的にこのプロセスが計算すべきものを表わす。Goals は、φ、trueまたは述語の列であり、Headを計算するために計算すべき述語を示している。例えば、HDB に 'a <-- b,c ' があれば、

process(Status,a,(b,c),Input-Channel,Output-Channel)

というプロセスが存在して良い。さらに、述語b の計算が終了していれば、

process(Status,a,(c),Input-Channel,Output-Channel)

というプロセスが存在して良い。Channel は、既に述べたように、他のプロセスとメッセージの送受を行なうためのもので、プロセスは，Input-Channel に対してはconsumer,Output-Channel に対してはgenerator となっている。
次にプロセスの動作の定義を示す。
(A)　activeプロセス

Activeプロセスの動作は大別してderivation, terminationのいずれかである。Derivationはprologの推論規則の展開が進む場合で、動作の後にそのactiveプロセスは存続する。これに対し、termination は推論が恒真（true）にたどりついた場合または推論規則の適用が不可能（fail）になった場合で、いずれの場合もその時点でプロセスは消滅する。

derivation時の動作

process(active,H,G,I,0) において

R がφでもtrueでもなく、G がP または(P,...) の場合、

P がHDB 中に定義された述語である場合

チャネルヘッドペアI+P に関してボードへの参照／登録[*1]を行なう。

参照であった場合には、

プロセスの状態をwaiting に変える.

登録であった場合には、

P に関してHDB よりselection [*2]を行ないクローズの集合S を得る。S の全要素に対してactiveなプロセスを生成し、その各プロセスとチャネルI で接続する。（各プロセスがproducer)

プロセスの状態をwaiting に変える。

*1　ボードへの参照／登録　：　ボードはチャネルヘッドペアC1+H1, C2+H2, ..., Cm+Hm を保存している。これに対し、C+H を参照／登録するとは、H=Hi(1≦i ≦m) であればC=Ciとし（参照）、そうでなければボードにC+H を追加する（登録）ことである。

*2　selection　：　P に関するselection とは、HDB 中でHeadがP とunify できる全てのclauseを取り出すことである。

termination 時の動作

termination には、success termination とfailure termination の2つの場合がある。success termination は導出がtrueに到達した場合であり、failure termination は、HDB に対するselection が失敗した場合であり、Prologのfailに相当する。

success termination

R がφまたは、trueである場合、H をチャネル0 に送信し、プロセス自身は消滅する。

failure termination

プロセス自身が消滅する。

(B)　waiting プロセス

Channel I よりメッセージM(term) を受信した場合、waiting プロセスのGoals G のコピーG'（形式はP'または(P',P1...)となる）を作り、その先頭要素(P')とM をunify する。（計算結果の伝達）そしてR'より先頭要素を除いたものをNewGとする。ただし、R'がP'だけの時はNewGはtrueとする。そして次のactiveなプロセスを生成する。

process(active,H,NewG,I',0)

・I'は新たなチャネル

waiting プロセス自身はそのまま存続する。

この計算メカニズムの全体の停止条件は、存在するプロセスが全てwaiting になる事であり、これをdeadlock terminationと呼ぶ。

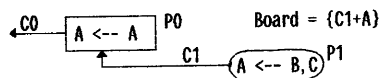
2.5　計算例

以下では、POPSの実行状況を示す簡単な例を示す。図においてactiveプロセスP は ⬭ P　waiting　processは □ P 、チャネルC は→C で示す。(P,Cは省略する場合もある）また、Head HおよびGoals G はH <-- G の形で表わす。ボードBoardの状態は {} でくくって表わす。今、HDB を次のように与える。

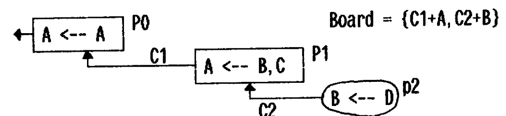
HDB={A<--B,C B<--D D<--true C<--true}

ここで、述語A を計算するため、初期プロセスとして次のプロセスを生成する。

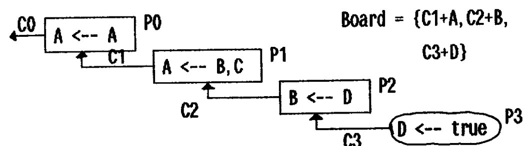
ここで'A <-- A' は、意味的には'<--A'であり、HeadのA　はこの初期プロセスが返すべきメッセージを示している。C0より出力されるメッセージが解である。P0は、active processでGoals A を持つので、selection が行なわれ新たなプロセスが生成され、P0はwaiting となる。また、Board にはチャネルヘッドペアが登録される。



同様に、P1が動作し、P2が生成される。

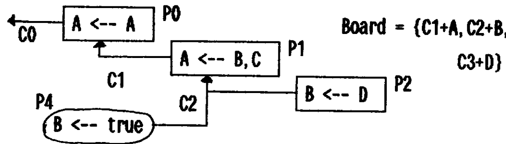


さらに、

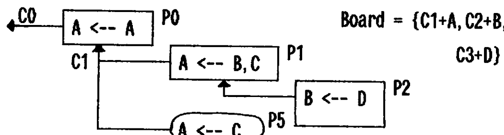


P3は、Goals がtrueでありactiveプロセスのtermination にあたり、チャネルC3にメッセージD を送る。このメッセージによ

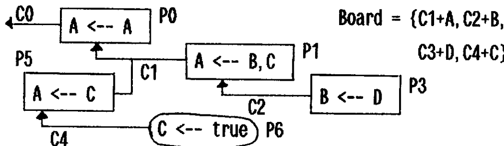
りwaiting プロセスP2は新たなプロセスP4を生成する。なお、このメッセージは、プロセスP3が消滅した後にも、ボード内のチャネルヘッドペアに保存されている。

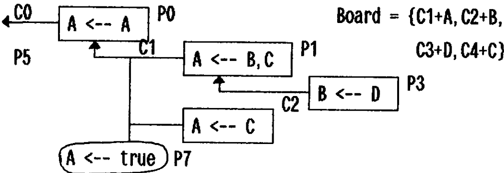


P4もtermination に相当するので、C2にメッセージB を送る。これによりwaiting プロセスP1は新たに、activeプロセスP5を生成する。

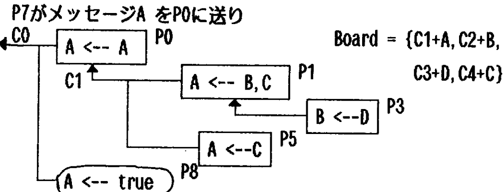


以下同様に計算が進行してゆく

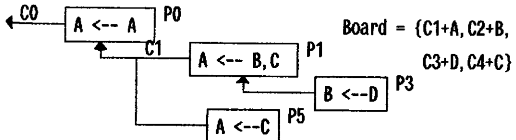




P7がメッセージA をP0に送り



P8はtermination しC0にメッセージA を送る。この事は、A の解の１つが得られた事に対応し、全体の状況は次のようになる。



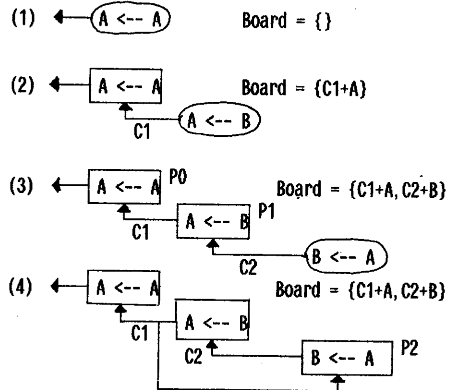
存在する全てのプロセスはwaiting であり、deadlock termi nationにより計算は停止する。この例では、OR並列となる述語が無いため、常にactiveプロセスは１つであるが、OR並列性が存在する時はactiveプロセスは複数個となる。

2.6 Derivation cycle

通常のPrologでは、cycle を含むような導出パスが存在する場合には、無限ループに陥ってしまう。これに対しPOPSでは、１度計算を開始した述語は、ボードに登録され、再度その述語が導出の過程に現れた場合には、ボードへの参照となり、新たなプロセスを生成することはない。このため、無限の導出に陥ることなくシステム全体はdeadlockを検出し、deadlock termi nationとなる。例えば、ホーンデータベースHDB が

{A<--B B<--A... }

であり、A →B →A...といったcycle を含んでいる場合、計算過程は次の様になる。



(3) から(4) への過程で、P2はA に関してボードへの登録／参照を行なう。

この場合C1+Aが既に登録されているため、参照の場合となり、P2のInput-Channel はC1となる。(4) はdeadlockの状態であり計算は停止する。但し、無限個数の解が存在する場合には当然のことながらdeadlockにはならず計算は無限に継続する。たとえば上記の例で'A <-- true'がHDB 中に存在すれば無限個数のderivation path すなわち無限個数の解が存在し、システムはtermination しない。また、変数を含む場合には、goalの呼び出しごとに状況を変えるプログラムを記述可能であり、この場合もtermination しない。たとえば、'a(X) <-- a([s | X])' はtermination しない。

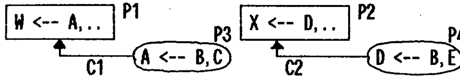
2.7 計算の共有

ボードの存在は、cycle となるクローズを含むプログラムにおける計算の停止を保障することの他に、同一の計算はくり返し行なわないという機能を提供している。説明のため次のようなプログラムを考える。

HDB={....A<--B,C D<--B,E...}

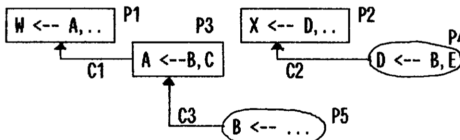
ここでA およびD は、そのsubgoal としてB を共通に持っている。ある述語を計算する過程においてA およびD の両者の計算が行なわれるとする。通常のtop down serial なstrategyでは、A におけるB の計算とD におけるB の計算は別々に実行さ

れる。これに対し、POPSでは、B に関する計算は全体で１回しか行なわれない。つぎに、その実行の様子を示す。いま、A およびD をGoals の先頭要素として持つ２つのactiveプロセスP1とP2が存在したとすると、次のようにP1, P2 はそれぞれP3, P4を生成する。



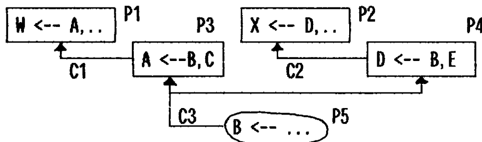
Board = {..,C1+A,C2+D,..}

この状態で、P3とP4のいずれもがB に関してボードへの参照／登録を行なうが、P3の方が先に動作したとするとボードへの登録となり状態は次のようになる。



Board = {..,C1+A,C2+D,...,C3+B}

次にP4の行なう参照／登録は、参照となり次のようにP5とP4がC3で接続される。



Board = {..,C1+A,C2+D,.....,C3+B}

この後に、P5より送られるメッセージB はP3, P4 の２つのプロセスに送られ、それぞれ対応する新しいプロセスを生成する。P5のメッセージの送信がP4とP5のC3による接続の後の場合でも、メッセージ自身はC3に残っているため、この接続と同時にP4はメッセージを受けることになる。このように、B に関する計算はP5が行なうのみであり、計算のsharing が行なわれる。

2.8 変数の処理

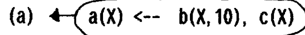
変数を含むprologプログラムをOR並列に実行する場合には、OR-clausesとのunification における変数の独立性を保証する必要がある。例えば、次の様なプログラムにおいて、subgoal b(X,10) の計算を行なう場合を考える。

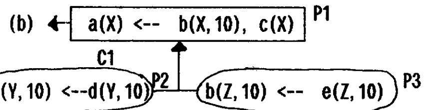
(1)　a(X)　<-- b(X,10),　c(X)

(2)　b(Y,U)<-- d(Y,U)

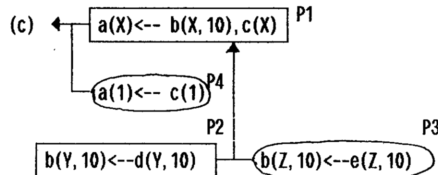
(3)　b(Z,V)<-- e(Z,V)

b(X,10)は(2),(3) を用いてderiveされX=Y,10=UおよびX=Z,10=Vとunify される。このderivationに関する変数の束縛による変数の同一化は、同一clause内の同じ名前を持つ変数の同一性と同様に保証されなければならない。一方、OR並列における変数Y とZ に関しては、全く独立の変数とする必要がある。POPSでは、copy*3とunification の機能により、上記の条件を満足させている。すなわち、subgoal のOR並列のderivation時に、（その時点の）subgoal のコピーを生成してderivationを行ない、計算結果であるメッセージを受けた時点で、そのsubgoalを含むHeadとGoals のコピーを生成し、そのコピー中のsubgoal とメッセージのunification を行なうことである。上記 (1) - (3)を例に取れば、計算過程は、次の様になる。





ここで、P2が先に計算が進み、メッセージb(1,10) をC1に送信したとすれば、P1はHeadとGoals のコピーa(X') <-- b(X',10), c(X') を生成し、Goals の先頭要素b(X',10)とメッセージb(1,10) とのユニフィケーションを行ない、新たなプロセスP4を生成する。



以上の様に、POPSでは必要に応じてcopyおよびunification を行なう事で同一clause間の変数の同一性、derivationにおける変数の束縛、OR-clause 間の独立性を保証している。

*3 あるstructure S のcopy S' とは、S 内の変数を全て別の変数に置換したものである。但し、もとのtermで等価の関係にある変数に対応するcopy中の変数は、やはり等価である。

3．POPS in Concurrent Prolog

本章では、Concurrent Prolog [Shapiro 83]によるPOPSについて述べる。

3.1 Concurrent Prolog

Concurrent Prologはand-parallelism とor-parallelismを導入し、前者を並列プロセスの記述に、後者をnon-deterministic プロセスの記述に用いている。Prologではor-parallelismはbacktrackingにより実現されるが、Concurrent Prolog では一度一つのclauseが選ばれると、他のchoiceは捨てられてしまう。この点でPrologは"Don't know determinism"であり、Conc

urrent Prolog は"Don't care determinism"であると言える。Concurrent Prolog では、並行して走るプロセス間で共有される変数をプロセス間通信用に用いている。Concurrent Prolog の詳細については［Takeuchi 83]を参照することとし、ここでは、簡単なプログラムの例により後の説明に必要な部分の説明を行なう。

```
(a) opposite-sex-child(X):-man(X) |
                    daughter(X,Y), output(X,Y?).
(b) opposite-sex-child(X):-woman(X) |
                    son(X,Y), output(X,Y?).
```

このプログラムに表われる '|' および'?' はConcurrent Prolog 固有の記号である。また、',' はPrologとは異なった意味を持っている。 '|' はguard bar と呼ばれguard sequence とgoal sequence を区切っている。guard bar はcut symbolと同様な意味を持ち、他のalternative clauseをcut する。',' はparallel-ANDシンボルであり、論理的には通常のAND シンボルと同じであるがoperational には異なっており、'P, Q'はP とQ を並列に処理する事を示している。Concurrent Prolog ではserial-ANDは記号'&' で表わす。

'?' の意味は「? の付いている変数をnon-variable term に instantiate してはならない」という事であり、(a) で言えば Y をinstantiate するのは'daughter'プロセスであり、'output'プロセスはY を読み出し専用として扱う事を意味する。この '?' のことを'read only annotation'、'?' の付く変数を'read only variable'と呼ぶ。この機能により、２つの並列プロセス間の共有変数による通信が可能となっている（プロセス間通信）。また、Concurrent Prolog では、read only variableを持つプロセスは、その変数に他のプロセスにより値がinstantiate されるまでwaitする（プロセス同期）。

3.2 Concurrent Prolog によるPOPSプロセスの記述

Concurrent Prolog によるPOPSでは、プロセスは次のtermで表現される。

```
process(Status,OutputChannel+InputChannel+Head<--Goals,
        Board)
```

プロセスの生成は、"process :- process1, process2" のようにparallel-ANDで行なわれ、プロセスの消滅はそのプロセスのtermination "process :- true" で表現されている。チャネルはConcurrent Prolog の並列プロセス間の共有変数、プロセス同期はread only annotationにより実現されている。また、すべてのプロセスからaccessされるボードは、全てのプロセスが共有する変数'Board' により実現されている。'Board' は、チャネルヘッドペアの参照／登録の機能を有すれば、その構成法は任意であるが、ここではbinary tree を用いて実現している。[Warren 80] また、ここでは表現されていないが、HDB はConcurrent Prolog の内部データベースに'ax(horn clause)'のメタ表現を用いて構成している。

次にConcurrent Prolog によるPOPSのprocess の記述を示す。

```
(c1) process(active,OutCh+_+Cls,Brd) :-
         call((derivationp(Cls,RemTop,InCh,Brd,Alt_or_not_yet),
               new(RemTop,NextGoal) )) |
                 process(wait,OutCh+InCh? +Cls,Brd) ,
                 process_fork(Alt_or_not_yet,InCh+NextGoal,Brd).

(c2) process(active,OutCh+_+Cls,Brd) :-
         terminatep(Cls,Mess) |
                 sendmess(Mess,OutCh).

(c3) process(active,_+_+Cls,Brd) :-
         call( (write('process killed  '),
                portray(Cls),
                nl) ).

(c4) process(wait,OutCh+[Terminated_Goal|C1]+Cls,Brd) :-
         newcls(Cls,Terminated_Goal,NewC) |
                 process(wait,OutCh+C1? +Cls,Brd) ,
                 process(active,OutCh+_+NewC,Brd).
```

(c1) - (c3) は、activeプロセスの動作を、(c4)はwaitingプロセスの動作を定義している。(c1)は、derivationを行なう場合に相当する。述語'derivationp' はGoals の先頭要素がHDB に定義されているかをチョックし、定義されている場合には、さらにその要素に関してボードへの参照／登録を行なう。参照の場合は'Alr-or-not-yet'にalready が、登録の場合はnot-yet がinstantiate される。この情報は、述語'process-fork'に必要である。先頭要素がHDB に無い場合は述語'derivatiop'はfailする。'new' は、copyを行なう述語である。このGuard 部分がsuccess すると、goal部分の２つの述語'process' と'process-fork'が並列に実行される。'process' は、もとのプロセスがwaiting となったものであり、変数'InCh'には'?' が付加される。'process-fork'は、もとのプロセスのGoals の先頭要素のcopyに関してselection を行ない、新たに取られたclauseのおのおのに対応させて新たなactiveプロセスを生成する。但し、'Alr-or-not-yet'がalready の場合は、すでにそれらのプロセスが存在するため新しいプロセスを生成する必要は無く何もしない。

(c2)は、termination の場合に相当する。述語'terminatep' は'Cls' が'X <-- true'の形式かをチェックする。述語'sendmess'はメッセージをOutCh (D-List)に送る（追加する）。また、プロセス自身は消滅する。

(c3)は、(c1),(c2) のいずれにも相当しない場合、すなわち、それ以上のderivationが不可能となった場合であり、'process killed'のメッセージを出力し、プロセス自身は消滅する。

(c4)は、waiting プロセスの動作を定義している。(c4)では、Input-Channel がread only variableであり、ここに値がinstantiate された場合（メッセージの受信）に動作する。述語'newcls'は、waiting しているプロセスのHead,Goalsのcopy H,G を生成し、G の先頭要素とメッセージをunify し、さらに、その先頭要素を除いた残りをG'として、新たに生成すべきプロセ

スのHeadとGoals を作る。プログラムのゴールの部分では、、元のプロセスのcopyと新たなactiveなプロセスとにforkしている。このプログラムの実行例は付録に示す。

以上で示したように、POPSはConcurrent Prolog で記述可能である。これは言い替えれば、AND-parallelism によりOR-parallelismを実現していることになり、"Don't care nondeterminism" でPrologが記述可能であることを意味する。

4. POPSによる構文解析

popsはPrologの並列処理系であり、DEC-20 Prolog に組み込まれているDCG を実行することにより構文解析を行なうことができる。すなわち、DCG のsyntaxで記述された文法をDCGのトランスレータでPrologのプログラムに変換し、そのプログラムをpopsの処理系で実行することで構文解析を行なう。DCG記述は、CFG の記述をベースにして、それに引数とPrologの述語呼び出し機能を追加した形式をしている。DCG の特徴として、

(a) 自然言語の構文規則の記述として明瞭性に勝れ、また、[Pullum 82] に見られるように記述能力として適当である。

(b) CFG の拡張として、DCG は引数の記述と述語呼び出し機能を提供しているが、これらは、拡張のワクぐみとして非常にsimpleであり、また、記述力及び一般性に優れている。

POPSによるDCG プログラムの実行過程は (pseudo) parallel strategy に対応し、基本的には、Chart Parser[Kay 80]と等価であることが示される。[Hirakawa 83] 既に述べたように、POPSでは同一計算のsharing が行なわれ、backtrackingが有する同一計算の再試行は行なわれない。また、2章で述べたように、derivation cycleのチェックを行なうため、cycle ないしは左再帰規則を取り扱うことが可能であり、また、解析自体はTop Downに行なわれるため、イプシロン規則を含む文法もそのまま取扱うことが可能である。計算のsharing による効果の有無を調べるため、DCG で書かれた文法 (DIAGRAM [Robinson 80] をベースにしている) に現れる述語のclauseの数とその述語の出現頻度を調べたところ、たとえば、名詞句である'np'については63個のclauseがあり、文法中で93か所から呼び出されている。この解析自身は充分なものではないが、POPSによる処理の高速化がかなり期待できることが言えよう。
付録に、DCG 記述の例とPOPSによる構文解析の実行例を示す。

5. ディスカッション

POPSは、PrologのOR並列をparallelに処理するメカニズムを提供する。POPSに於いては、プログラムの解に多義性がある時には、その解集合の全体を求めるように動作する。例えば'man(X)'とすればX は全ての人間に相当する。この意味で変数はuniversal であると言える。これに対して、existential な変数を考えることができる。例えば、“子供があれば結婚している“という知識を'married(X) <-- has-child(X)'としたときには、X に子供が何人いても、このclause自体は、子供がある場合にtrueの解を一つ出力するだけでよい。このような知識を処理するためにunversal predicateとexistential predicate の区別を導入することを考える。universal predicate は、解集合全体を出力し、existential predicate は、解集合の1つの解しか出力しないものである。プログラム記述上では、この2つの区別は、述語の記述に属性タグを付加したり、あるいは、メタな定義述語を導入するなどして指示する。POPSの計算メカニズムにおいてはChannel のclose の概念を導入し、existential predicate のメッセージをチャネルに送ると同時に、そのチャネルをclose し、それ以降のメッセージの送信をdisable することにより実現できる。Concurrent Prolog では、D-List(Channel) の変数に'[]'をinstantiate することでimplement が可能である。

現在、POPSはProlog上にimplement されたConcurrent Prolog 上にimplement されているため、速度的には速くはない。POPSモデルは基本的にparallel動作を行なうため、将来、parallel processor上でimplement することにより、parallel parsingが可能となり、高速の処理が期待できる。

6. 結論

メッセージパッシングとマルチプロセスに基づくPure Prolog の処理系POPSについて報告した。POPSは、PrologのOR並列を実現するメカニズムを有し、同一計算は複数回行なわない、derivation cycleを含むプログラムを取り扱える等の特徴を持っている。POPSは、AND 並列、プロセス間通信、プロセス同期等の機能をもつConcurrent Prolog 上にimplement されている。POPSにより構文解析を行なう場合には、同一計算の共有による効率かがはかれ、また、イプシロン規則や左再帰規則を含む文法を直接扱うことが可能である。

謝辞

本研究の機会を下さりご指導いただいた渕一博ICOT研究所長に感謝いたします。本研究を進めるにあたり、討論いただいたICOT第2研究室の皆様、Concurrent Prolog とPOPSに関して有用な助言をいただいた竹内研究員、引数処理に関して討論いただいた坂井研究員に感謝します。また、本論文の作成にあたりアドバイスして下さった近山研究員に感謝します。

付録 - GRAMMAR AND EXECUTION -

```
% very simple English grammar

s(s(NP,VP)) --> np(NP), vp(VP).

np(np(NOUN)) --> noun(NOUN).
np(np(and(NP1,NP2))) -->      /* Left Recursive rule */
        np(NP1), [and], np(NP2).
np(np(NP,NPMOD)) --> np(NP), srel(NPMOD).

vp(vp(V)) --> verb(V).
vp(vp(V,NP)) --> verb(V), np(NP).

srel(srel(RP,RPP)) --> rp(RP), s(RPP).

rp(rp(who)) --> [who].
rp(rp('Empty')) --> [].       /* epsilon rule */

noun(noun(john)) --> [john].
noun(noun(mary)) --> [mary].
noun(noun(lucy)) --> [lucy].

verb(verb(loves)) --> [loves].
verb(verb(hates)) --> [hates].

'C'([X|Y],X,Y).  /* Connect definition */


%
%   SIMPLE PARSING EXAMPLES
%

| ?- do(s(K,[john,loves,mary],[])).

Killed  noun(noun(mary),[john,loves,mary],X)
           <-- C([john,loves,mary],mary,X)
Killed  noun(noun(lucy),[john,loves,mary],X)
           <-- C([john,loves,mary],lucy,X)
Killed  np(np(and(np(noun(john)),X)),[john,loves,mary],Y)
           <-- (C([loves,mary],and,Z),np(X,Z,Y))
Killed  rp(rp(who),[loves,mary],X)
           <-- C([loves,mary],who,X)
Killed  verb(verb(loves),[loves,mary],[])
           <-- C([loves,mary],loves,[])
Killed  verb(verb(hates),[loves,mary],[])
           <-- C([loves,mary],hates,[])
Killed  verb(verb(hates),[loves,mary],X)
           <-- C([loves,mary],hates,X)
Killed  noun(noun(john),[loves,mary],X)
           <-- C([loves,mary],john,X)
Killed  noun(noun(mary),[loves,mary],X)
           <-- C([loves,mary],mary,X)
Killed  noun(noun(lucy),[loves,mary],X)
           <-- C([loves,mary],lucy,X)
Killed  noun(noun(john),[mary],[])
           <-- C([mary],john,[])
Killed  noun(noun(lucy),[mary],[])
           <-- C([mary],lucy,[])
Killed  noun(noun(john),[mary],X)
           <-- C([mary],john,X)
Killed  noun(noun(lucy),[mary],X)
           <-- C([mary],lucy,X)
Killed  np(np(and(np(noun(mary)),X)),[mary],Y)
           <-- (C([],and,Z),np(X,Z,Y))

Killed  rp(rp(who),[],X)
           <-- C([],who,X)
Killed  noun(noun(john),[],X)
           <-- C([],john,X)
Killed  noun(noun(mary),[],X)
           <-- C([],mary,X)
Killed  noun(noun(lucy),[],X)
           <-- C([],lucy,X)
Killed  np(np(and(np(noun(mary)),X)),[mary],[])
           <-- (C([],and,Y),np(X,Y,[]))

Message = s(s(np(noun(john)),vp(verb(loves),
           np(noun(mary))))),[john,loves,mary],[])

Execution Time = 15197 ms

        % PARSING A SNTENCE USING
        % A EPSILON RULE.

| ?- do(s(S,[john,loves,mary,lucy,hates],[])).

Killed  noun(noun(mary),[john,loves,mary,lucy,hates],X)
           <-- C([john,loves,mary,lucy,hates],mary,X)
                  :
                  :
Killed  np(np(and(np(np(#),srel(#,#)),X)),[mary,lucy,hates],Y)
           <-- (C([],and,Z),np(X,Z,Y))

Message = s(s(np(noun(john)),vp(verb(loves),
             np(np(noun(mary)),srel(rp(Empty),
             s(#,#))))),
           [john,loves,mary,lucy,hates],[])

Execution Time = 35926 ms

        % PARSING A SENTENCE USING
        % A LEFT RECURSIVE RULE.

| ?- do(s(S,[john,loves,mary,and,lucy],[])).

Killed  noun(noun(mary),[john,loves,mary,and,lucy],X)
           <-- C([john,loves,mary,and,lucy],mary,np)
           :
           :
Killed  np((and(np(and(#,#)),X)),[mary,and,lucy],Y)
           <-- (C([],and,Z),np(X,Z,Y))

Message = s(s(np(noun(john)),
             vp(verb(loves),np(and(np(#),np(#))))),
           [john,loves,mary,and,lucy],[])

Execution Time = 27389 ms
```

参考文献

[Turner 79] D.A.Turner :A New Implementation Technique for ApplicativeLanguages, software-practice and experience, No.1,vol9.(1979)

[Shapiro 83] Shapiro,E,Y: "A Subset of Concurrent Prolog and Its Interpreter" ,ICOT Technical Report TR-003, (1983)

[Takeuchi 82] Takeuchi,A: "Let's Talk Concurrent Prolog", ICOT TechnicalMemo TR-008,(1982)

[Warren 80] Warren,D.H.:Logic Programming and Compiler Writing", DAI Research Paper No.128

[Pereira 80] Pereira,F. and Warren,D.:"Definite Clause Grammar for Language Analysis -Survey of the Formalism and a Comparison with Augmented Transition Networks", A rtificial Intelligence, 13, pp231-238,(1980)

[Matsumoto 83] 松本, 田中, 平川, 三吉, 安川, 向井, 横井: "Prologに埋め込まれたボトムアップパーサ:BUP",Proc. of Logic Programming Conference 83,(1983)

[Pullum 82] Geoffrey K. Pullum and Gazdar, "Natual Languages and Context-Free Language",Linguistics and Philosophy 4, 1982

[Robinsoin 80] Robinson,J.,J:"DIAGRAM: A Grammar for Dialogues",SRI Tec. Note 205,1980

[Kay 80] Kay,M:"Algorithm Schemata and Data Structures in Syntactic Processing",Xerox Tec. Rep., 1980

[Hirakawa 83] Hirakawa,H.:"Chart Parsing in Concurrent Prolog", ICOT Technical Report TR008,(1983)