POPSによる構文解析とChart Parsing について

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

我々は、PrologのOR並列処理実行モデルに基づいた処理系POPS(Or-Parallel Optimizing Prolog System)をConcurrent Prolog 上にimplement した[Hirakawa 83] 。POPSモデルは、次の様な特徴を持っていた。

(1) PrologのAND-relationについてはserialに、OR-relation についてはParallelに実行する。
(2) multiprocessとmessage の伝達によりOR並列処理を実現している。
(3) graph reduction mechanism を導入し、同一計算の重複を避ける。

POPSは、OR並列度が高く同一計算の多い計算分野に特に有用である。この例として自然言語の構文解析が挙げられ、POPSによる構文解析は、他のPrologベースの構文解析システム([Pereira 80],[Matsumoto 83],etc) に散在する次の様な問題点を解決していた。

(1) 計算の重複　：　backtrackingは同じ計算を何度も繰り返し実行し、解析する文の長さに対して指数オーダの時間を要するという効率の悪さを持っている。
(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。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。
(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規則を自然な形で記述したい。

これらは、POPSによる構文解析の質的な議論であった。本報告では、POPSによる構文解析の量的な面に関する考察を行ない、従来のCFG parsing algorithm との関係について述べる。　２節では、POPSの計算モデルを示し、３節では、POPSによるDFG parsing の時間オーダに関する考察を行なう。４節では。、Earley's parsing algorithm, Chart parsing およびEarley DeductionとPOPSのモデルとの関係について考察する。

２．POPS

本節ではPOPSの計算モデルを示す。

2.1 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。

2.2 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。

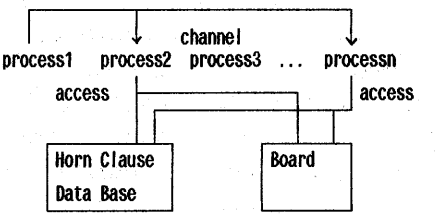


図１　POPSの構成

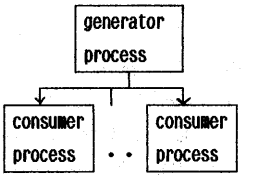


図２　プロセス／チャネル

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。

2.3 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。

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,O) 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O 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',O)

・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。Concurrent Prolog によるimplementationでは、Cycle freeのプログラムの実行の場合、全てのプロセスが消滅して終了する。

3. POPSによる構文解析

Prologの処理系であるPOPSにより構文解析を行なう場合には、DEC10 Prologに組みこまれているDCG(Definite Clause Grammar)を利用している。すなわち、DCG トランスレータによりCFGの文法をPrologのプログラムに変換し、そのプログラムを実行することにより構文解析が行なわれる。DCG トランスレータは、次に示すように文法の各カテゴリにcontinuationのための2つの変数を自動的に付加する。

```
DCG       s --> np,vp.
Prolog    s(X0,X) :- np(X0,X1),vp(X1,X).
```

各変数は入力文のある位置を示すポインタと考えられ、例えば [john,walks] という文の解析が終了した時点では次の様な値が各変数に代入されている。

```
s([john,walks],[]):-
        np([john,walks],[walks]),vp([walks],[]).
```

以降の説明では、簡単のため、listの代わりに入力文の位置を数字で示す。すなわち、上記の例は次の様に示される。

```
s(0,2):-np(0,1),vp(1,2).
```

一般に、構文解析中に現われる全てのリテラルはa(I,J)となる。

3.1 時間オーダ計算の概要

本節では、POPSによる構文解析における計算時間のオーダに関する考察を行なう。POPS自体はparallel parsingモデルとなっているためprocessor の個数等により実際の計算時間は変化する。ここでは、[Harison 78],[Aho 72] に見られる様に単一processor を仮定し、入力stringの長さに関する計算量のオーダを示す。また、parsing そのものについても、入力stringが文法で受理可能かを判定する場合と可能な全ての解(parsing tree)を求める場合では、当然、必要な計算量が異なる。POPSは後者の全解を求めるアルゴリズムであるが、POPSのprocess 動作を少し手直しすることで前者のアルゴリズムに変更が可能である。全解版をtype1 、受理版をtype2 のPOPSと呼ぶ事とする。以下では、次の 3つの場合の計算時間に関する考察を行なう。

Case1 type1 POPSによるあいまい性のない文法 (unambiguous grammar)によるparsing

Case2 type2 POPSによるあいまい性のある文法(ambiguous grammar) によるparsing

Case3 type1 POPSによりあいまい性のある文法によるparsing

ここで、あいまい性のない文法とは、S をrootとする2つの導出木T1とT2においてfr(T1)=fr(T2) ならばT1=T2が成立する文法を言う。ここでfr(T) とはT のleavesの連接 (concatenation)である。次に、parsing 時間オーダの証明の手続きを示す。

(1) type1,type2 のPOPSプロセスのアルゴリズムを示し、アルゴリズム中の各オペレーションの時間オーダを示す。

(2) 1つのプロセスの入力列に依存する部分の計算コストのオーダを示す。

(3) parsing 全体で生成されるプロセス数の入力列長に関するオーダを示し、(2) と組み合わせて全体のオーダを示す。

3.2 プロセスコスト

POPSのプロセスの動作については、既に2で述べたが、ここでは、Type1,type2 POPSの動作を形式的に示し、各オペレーションのコストおよびプロセスコストを示す。

次に各プロセスの動作を形式的に示す。

active process

```
if terminated(Clause) then
  if type1 then
     send_message(Head,OutChannel)
  else
     if is _new _message(Message,OutChannel)
        send_message(Head,OutChannel)
else begin
     board_access(Head+Channel,Result);
      if Result  ==  notyet
      begin clauses(Body _Top,NewGoals);
          create_new _processes(NewGoals,OutChannel)
      wait
      end
```

waiting process

```
if receive(Message)
  then do
    create_new _process(Clause,,Message,OutChannel);
```

active動作中のif type1 then ～elseの部分は、type1 POPSとtype2POPS の定義をマージして記述している。各オペレーションの意味は以下の通りである。

terminated: ClauseがHead<--true の形式かを判定する。

send_message : HeadをOutChannelに導出する。

is_new _message : Message が既にOutChannelに送られたかをチェックする。

board _access : 2で述べたBoard に対する参照／登録を行なう。

== : 右辺と左辺が等しいかをチェックする

clauses : データベースにより展開可能な規則のcollectionを取り出す

create_new _process : New Goals(Clauseのcollection) の各々の要素に対応するactive processを生成する。

wait: 状態をwaiting に変える

receive : message を受けた時trueとなる

create_new _process : Message を受け新たなclauseを生成し、active processを生成する。

いま、与えられた文法を固定して考えると、上記の各オペレーションの入力string長に対する計算時間は次の様になる。

constant: terminated,sendmessage,clauses, create_new _processes,receive,create_new _process

また、case1(type1 POPS,unambiguous grammar),case2 (type2 POPS,ambiguous grammar)において

O(n) : board _access,is _new _message

board accessのオーダ

文法（DCG プログラム）中の全てのリテラルは、a(I,J)の形式をしている。また、ボードに登録されているHeadの第１引数は、0<i<n(n は入力string長）のいずれかの値を取る。このため、ボードを次の様に構成する事が可能である。

```
Board = { [ 0 C11+H12,...,C1p+H1q],
          [ 1 C21+H21,...,C2q+H2q],
                    :
          [ n Cn1+Hn1,...,Cnm+Hnm]}
```

ここで、リスト[...] の個数はn+1,各リストに含まれるチャネルヘッドペアの個数は最大で文法カテゴリ数となる。ボードへの参照／登録の手続きは次の様になる。

(1) 参照／登録しようとするリテラルの第１引数に応じてリストの１つを取り出す。 = = > O(n)

(2) リストをサーチし、参照あるいは登録する=> constant

以上により、ボードへの参照／登録オペレーションはO(n)となる。

is_new _message

is_new _message はあるメッセージが既に出力チャネルに；送出されたかをチェックする。出力チャネルは、ボード中のチャネルヘッドペアのチャネルに対応し、その中のメッセージはa(i,j)の形式をしている。ここで、i は、ヘッドの第１引数と同じであるため、0<i<n を満たす特定の数である。このためチャネルに送られるmessage の総数は最大n+1 個となりis_new _message の計算時間はO(n)となる。

3.3 type1 POPSとunambiguous grammar

ここでは、type1 のPOPSによりunambiguous な文法で構文解析する場合について考察する。各オペレーションのコストは既に示したため、全パージング過程で生成されるプロセスの個数を求める。各プロセスは必ずactiveプロセスとして生成されるが、プロセスの保持しているclauseの形式に応じて２つのtypeに分けて考える。１つは、clauseのボディ部分がtrueであるt-typeプロセス（terminate)他は、ボディ部分が１つ以上の sub goal列となっているn-typeプロセス(non-terminate) である。

```
t-type    n(2,3)<-true
n-type    s(φ,X)<-np(φ,Y),vp(Y,X),
```

t-typeプロセスのコストは、プロセスジェネレーション(constant), メッセージ送信(constant)の和であるので、O(constant) である。また、n-typeプロセスについてはボードアクセスがO(n)となっている。よってprocess コストは、次の通り。

t-type	O(constant)
n-type	O(n)

次に、parsing 全体における、n-typeプロセス、t-typeプロセスの個数のオーダを求める。

(1) n-typeプロセス

n-typeプロセスは次の様なclauseに対応する。

a(i,J)<-b(k,L),rests,0<i,k<n, J, are variables

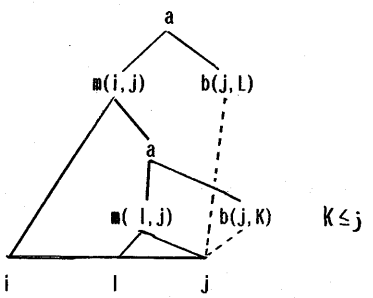
ここで、n-typeプロセスが生成された時点では、i およびk は実際のある値に決まっており、１つのt-typeプロセスはボード中の１つのチャネルヘッドペアにチャネルを介して接続している。逆に、１つのチャネルヘッドペア、b(k,L)+Channel, を考えると、それはunambiguous grammar の場合には、あるヘッドa に関して１個のn-typeプロセスと接続可能である。[*3]いっぽう、 3.2で示した様に、ボードにはO(n)のチャネルヘッドペアが存在する。よって、全parsing 過程を通して生成されるt-ty

peプロセスの個数はO(n)となる。

*3 : unambiguous grammar を用いた、type1 POPSによるparsing 過程において、ある1つの文法規則 a->m,b,n. (b は empty productionでない) から導出された a(i,K) <- b(j,L), n(L,K). をclauseとして持つプロセスが存在した場合、a(l,K) <- b(j,L),n(L,K). where l not equal i をclauseとして持つプロセスは存在しない。

[概略証明]

今、 a(i,K) <- b(j,L),n(L,K), a(l,K) <- b(j,L),n(L,K). i<l を持つプロセスが存在すると仮定する。nonterminal m に関するparse は終了しているため、 m(i,j), m(l,j) が同時に成立している。 i<lおよびunambiguous grammar の定義により、m(l,j)に対応するparse treeは、m(i,j)のparse treeのsubtree となっている。



これが成立するのは, K=j, すなわちb がempty productionの場合だけである。よって上記2つのプロセスが同時に存在することは無い。

(2) t-typeプロセス

t-typeプロセスは、a(k,J)<-trueの形式のclauseを持ち、次の2つの場合に生成される。(ここでk は固定している)

(a) clauseとしてx(i,J) ->a(k,L),rests.を持つactiveプロセスがderiveされて生成される。
(b) clauseとしてa(i,J)->x(k,L). の形式を持つwaiting プロセスがメッセージを受けて生成する。

今、文法を固定しているため、a に関するunit clause はconstant orderであり、a をbodyに含む文法数もconstant order である。a をbodyに含む規則1つに関して言えば、上記(a) の 0<i<n により高々、n 個のt-typeプロセスを生成するのみである。よって(a) によるプロセス数はO(n)となる。(b) では、a をheadとして持つ文法は Constant order であり、その各々に対して(b) のi が0 i<n であるため、やはり生成されるプロセス数はO(n)である。よってk を固定して時a(k,J)に対してプロセス数はO(n)である。今k は,0<k<nより全体としてはt-typeプロセス数は、O(n 2) となる。

以上より、type1 POPS によるunambiguous grammar のTotal の計算時間オーダは、次のようになる。

n-type processコスト×ntype process 数+
t-type process コスト×t-type process数 =
O(n)×O(n)+O(constant) ×O(n^2) =
O(n^2)

3.4 type2 POPS とambiguous grammar

ここでは、入力列が文法で受理可能か否かを判定するtype2 のPOPSについて考察する。

t-typeプロセスのコストは、プロセスジェネレーション(constant), 新しいメッセージか否かのチェック(O(n)), メッセージ送信(constant)の和であるので、O(n)である。また、n-typeプロセスについてはボードアクセスがO(n)となっている。

n-type processコスト O(n)
t-type processコスト O(n)

次に、parsing 全体における、t-typeプロセス、n-typeプロセスの個数のオーダを求める。

(1) n-typeプロセス数

n-typeプロセスは次の様なclauseに対応する。

a(i,J) <- b(k,L),rests. 0<i,k<n J,L are variables

ここで、n-typeプロセスが生成された時点では、i およびk は実際のある値に決まっており、1つのn-typeプロセスはボード中の1つのチャネルヘッドペアにチャネルを介して接続している。逆に、1つのチャネルヘッドペア、b(k,L)+Channel, を考えると、0<i<n であるため、それはあるヘッドa に関して最大 n 個のn-typeプロセスと接続可能である。よって、1つのチャネルヘッドペアに対してO(n)のn-typeプロセスが存在しうる。一方、 3.2で示した様に、ボードにはO(n)のチャネルヘッドペアが存在する。よって、全parsing 過程を通して生成されるt-typeプロセスの個数はO(n)×O(n)=O(n^2) となる。

(2) t-typeプロセス数

t-typeプロセスに関しては、 3.3 (2)と同じ議論が成立し、次のようになる。

t-typeプロセス数 O(n^2)

以上により、 parsing全体のコストは、

t-typeプロセス数 x t-type プロセスコスト +
n-typeプロセス数 x n-type プロセスコスト
= O(n^2)xO(n) + O(n^2)xO(n)
= O(n^3)

3.5 type1 POPS,ambiguous grammar

type1 POPSは、可能なparsing treeを全て出力するシステムである。通常、入力長n に対する解の総数は組み合わせ的な増加をし、それに供ないprocess 数も組み合わせ的に増加する。よって、アルゴリズムの時間オーダも組み合わせ的に増加する。次に示す文法は、入力長n に対し、n のカタラン数個の解を持つ。

S →S,S
S →a.

この様な場合には、1つの解に対する計算時間をアルゴリズムの良さの尺度に求める事が、妥当であるように考えられるが、例えば、次に示すような文法を考える。

S →E,b.　E →E,E.　G →a,G.
S →G,c.　E →a.　　G →a.

この文法は、a^n b 又は a^n c の入力stringを解析可能である。両者のparsing には、共に組み合わせ的時間オーダがかかるが、a^n b については解はn-1 カタラン数個、a^n c については1つのみ存在する。よって、一概には1つの解に対する計算時間を評価する事はできないと思える。むしろ、計算時間のパラメタに文法のサイズ、その他の要素を考慮する必要があると思われる。type1 POPSのアルゴリズムについて言えば、type2 POPSが入力列に対し$O(n^3)$ でparsing が可能である点を考慮して、効率的には問題の無いものであると言えよう。また、POPSはモデル自体が並列性を有し、並列プロセッサ上にimplement が容易であるという特性を持っている。現在、POPSは、Concurrent Prolog[Shapiro 83]上にimplement されておりConcurrent Prologの並列処理マシン上で実行する事を想定している。

4.　Earley's Algorithm,Chart Parsing and POPS

本節では、POPSによる構文解析（DCG プログラムの実行）と他の構文解析アルゴリズムによる構文解析の関係について述べる。

E→・E*E E→・E+E E→・a	E→E・*E E→E・+E E→a・	E→E+・E	E→E+E・ E→E・*E E→E・+E	E→E*・E	E→E+E・ E→E・*E E→E・+E E→E*E・
		E→・E*E E→・E+E E→・a	E→E・*E E→E・+E E→a・	E→E*・E	E→E*E・ E→E・*E E→E・+E
				E→・E*E E→・E+E E→・a	E→E・*E E→E・+E E→a・

図3　文法 E->a ｜E+E ｜E*E.　入力"a+a*a"

4.1 Earley's Algorithm とPOPS

CFG の"Good practical algorithm"の1つにEarley's algorithmがある。Earley's Algorithmは、長さn の入力stringに対して(n+1)x(n+1) のrecognition matrixを生成する。recognition matrixは、図3に示すように文法規則に1つのメタシンボル・を加えたものを要素として持ち、全てのparsing を含んでいる。この場合、全てのparsing とは、全てのparsing treeの生成（POPSの場合）を意味するのではなく、recognition matrixより、あるアルゴリズムにより全parsing treeが生成されるということを意味している。

次にbit vector版のEarley's algorithmを示す。

```
begin
t<0> = [ predict({s}) ]
       [      0       ]
       [      :       ]
       [      0       ]
  for j := 1 to n do
  begin
    scanner:
      t<j> := t<j-1> * {a<j>};
    computer:
      for k:= j-1 down to 0 do
        t<j> := t<j> U (t< k> * t<k,j>);
    predictor:
      t<j,j> := predict(U t <i,j>) where 0 < i < j
  end
end
```

アルゴリズムの詳細は文献[Harrison 78] 参照することとし、ここでは、ポイントのみを述べる。次は、アルゴリズム中の基本オペレーションである。

predict　nonterminal symbolの集合に対し、文法規則による展開を行なう。（Topdown 予測）
ex.　S->A,B. A->a. A->D,E.において
predict({S}) = {S-> ・A B, A->・a, A->・D E}

scanner　Topdown 予測されているterminal symbol と入力シンボルとの照合を行なう。
ex. 前例 A->・a の予測に対しa が入力stringにあった場合に、A->a・をrecognition matrixに追加する。

computer　scanner で認識されたnonterminal symbolに対し、そのnonterminal symbolを予測しているrecognition matrix中の要素より新たな要素を生成し、追加する。この計算は可能なかぎりくり返し適用される。（Bottom Up の本

の成長に対応）

ex. A->a・およびS-> ・A,C よりS->A・C を生成

Earley's algorithmとPOPSを比較すると、次に示す対応関係がある。

Earley	POPS
predict	active processにおける子プロセスの生成（規則の展開）
scanner	active processにおけるterminated clause の処理
computer	子プロセスからのmessage 送出およびwaiting process における新プロセスの生成

Earley's algorithmにおけるrecognition matrixの要素は、そのままPOPSのプロセスに対応させることが可能である。このようにEarley's algorithmとPOPSによる構文解析は、parsingの動作としては等価であることが示される。基本的な違いは、Earley's algorithmがrecognition matrixを、POPSは全parsing 結果を出力するように動作する点にある。このため、入力stringの認識のみを行なうtype2 POPSがambiguous な文法に関してO(n^3) でparsing することと、Earley's algorithmがO(n^3) でrecognition matrixを作成することは等価なことと見ることができる。

4.2 Earley DeductionとPOPS

Earley Deduction proof procedure schema は、Earley's algorithmに基づいている。Earley Deductionは、program とstate とよばれる2つのdefinite clause の集合に対して適用される。program は入力stringに対応するinput clauses の集合であり、deduction の過程で固定されている。state はderived clauseの集合であり、deduction の仮定で追加されてゆく。各derived clauseは、selected negative literal を持つ。推論規則は、新たなderived clauseを追加することにより、current state を新たなstate にmap する働きをし、instantiation とreduction の2種類がある。instantiation は、Earley's algorithmのpredict オペレーションに対応し、あるclauseのselectedリテラルとnon-unitclauseの正リテラルとを（もし可能なら）unification した結果をcurrent state に追加する。reduction は、current state 中のclauseのselectedリテラルに対し、program およびcurrent state 中のunitclauseをunify し、derivationを行なった結果をcurrent state に追加する操作である。但し、deriveされた結果が既にcurrent state にあるclauseにsubsume される場合は追加を行なわない。reduction は、Earleyのscanner,compute オペレーションに対応する。

Earley DeductionとPOPSは、次のように対応づけられる。

Earley Deduction	POPS
instantiation	active processにおけるclauses,create-new-processオペレーション
reduction	message の送出、waiting processによる新active processの生成
subsume check	Board への参照／登録

このことから、POPSは、次のような特徴を持つEarley Deduction処理系と見做すことができる。

(1) multi process,message passing を計算の基礎とし、並列性をexplicitに表現している。

(2) channel の接続の概念により、Earley Deductionのreduction 時のstate のサーチの手間の最適化が行なわれている。

(3) POPSのプロセスは、Earley Deductionのstate の要素に対応するが、POPSではプロセス自体は順次消滅し、Board のみが残る。

(4) Earley DeductionではclauseのsubsumptionTが必要であるが、POPSではselectedリテラルのequivalence をcheck するだけである。

(5) Earley Deduction では、selectedリテラルはsubgoal の任意の1つであるが、POPSでは先頭のsubgoal と固定している。(AND=serialなinterpretation)

以上のように、基本的にPOPSとEarley Deductionは同等と見做せるが、POPSのモデルにchannel の概念が入っていることにより、Earley Deduction では行なえない処理が可能となる。これについては5節で簡単に触れる。

4.3 Chart parsingとPOPS

Chart parsing は、Chart と呼ばれるデータ構造をbookkeeping に使用することを特徴とした、CFG のparsing algorithm

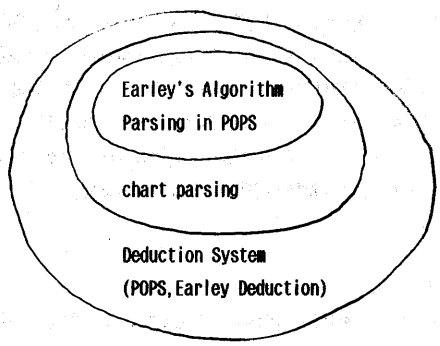


図4

のための一般的なワク組みである。parserの状態は、有向グラフであるchart として表現される。chart 中のノードは入力のpositionを表現し、1つのedgeは、文法規則の1つの適用に対応している。ここでは、詳細は[Kay 80],[Winograd 83],[Hirakawa 83]を参照することとする。[peraira 83] によれば、Earley's Parsing Algorithm は、Chart parsing の1つの例と見る事が可能である。また、同文献によれば、parsing をdeduction (ex. Earley Deduction)と見る立場から次の事が言える。

(1) chart parsingは、chart のノードを特殊なargumentとしたproof procedure である。
(2) proof procedure は、拡張されたCFG (gap 処理、dependency処理) のワク組みを提供する。

すでに述べた様にPOPSとEarley Deductionは同等であり、上記のPereira の議論は、そのままChart parsing とPOPSについて言える事である。最後に本節をまとめると図4に示すような包含関係となる。

5. ディスカッション

POPSは、message の伝達路としてchannel を使用しており、常にmessage のsenderに送出可能としている。これは、channel が無限長のバッファ(unbounded buffer) となっていると言い換える事ができる。POPSのchannel の性質を有限長バッファ(bounded buffer)に置き換え、プロセスの動作に多少の変更を加えるとシステムの動作をコントロールする事が可能となる。すなわち無限長バッファはeager な実行を、有限長バッファはlazyな実行をする。Earley DeductionとPOPSの動作は基本的には等価であるが、POPSモデルは上記の様な柔軟性を含む点が特徴の1つと言える。eager およびlazyな実行については。別紙において報告する予定である。

3節では、POPSによる構文解析の時間計算に焦点をあてて考察を行なったが、Prologの処理系としての効率という点では十分なものではない。例えば、Board への参照／登録は計算のsharing およびoccur-check のために行なわれるが、同一計算が全くなく、無限loopに陥るような事がないプログラムに対してはBoard は無意味であり、処理効率を下げるだけである。通常のProlog programmingでは、同一計算が多数現われる事は少なく、無限loopの管理はプログラマが行なう。このため、Board-less POPS の方が望ましい場合が多い。Board-less POPS については、上記のeager, lazyの議論と共にConcurent Prologにおけるenumeration の取扱いの問題と位置づけて研究していく予定である。

6. おわりに

本報告では、POPSによる構文解析に焦点をあて、CFG parsing を行なった場合の時間のオーダを示し、他のアルゴリズム等との比較を行なった。その結果、POPSによる構文解析は、入力stringの長さに関してEarley's parsing algorithmと同じ時間オーダで解析することが判明した。

［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。また、アルゴリズムの解析に助言をいただいた近藤研究員、時間計算等に関して討論いただいた、柴山研究員、坂井研究員、富士通国際研究所の横森研究員に感謝いたします。また、本論文の作成に協力して下さった藤田嬢ならびに中川嬢に感謝いたします。

［参考文献］

[Pereira 80] Pereira,F and Warren,D.H.:"Definite Clause Grammar for Language Analysis",A.I. 13,(1980)

[Pereira 83] Pereira,F and Warren,D.H.:"Parsing as Deduction", SRI T echnical Note 295,1983

[Matsumoto 83] Matsumoto,Y. et. al.:"BUP : A Bottom Up Parser embedded in Prolog", New Generation Computing vol2

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

[Shapiro 83] Shapiro,E,Y:"A Subset of Concurrent Prolog and Its Interpreter",ICOT Tec. Rep. TR-003,1983

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

[Hirakawa 83] Hirakawa,H R.Onai and K.Furukawa:"Implementing POPS in Concurrent Prolog",ICOT Technical Report TR020,1983

[Winograd 83] Winograd,T:"Language as a Cognitive Process",Add ison-Wesley Pub.,1983

[Harrison 78] M.A.Harrison:"Introduction to Formal Language Theory", Addison Wesley pub.,1978

[Aho 72] A.V.Aho and J.D.Ullman:"The Theory of Parsing,Translation and Compiling",Prentice-Hall Inc.,1972