論理型並列プログラミング言語

---- Concurrent Prolog ----

竹内彰一

Akikazu Takeuchi

（財）新世代コンピュータ技術開発機構

1　はじめに

論理型プログラミング言語は一階述語論理に基づいており、そのプログラムの論理的性質が読取り易く、かつ数学的にも扱い易い等の理由で注目され、第５世代コンピュータプロジェクト等でもその核言語[1)]として採用されている。その逐次型の処理系で効率の良いものは DEC-10 Prolog[2),20)]を初めとしていくつか存在し、すでにその上で多くの応用プログラムが書かれている。応用分野が広がるにつれ、論理型の言語で並列プログラミングのできるものが求められるようになってきているが、これは現在この分野の研究の中心になっており、基本計算メカニズムを含めて多くの提案がなされている。本論文ではこれらの設計提案の中の１つでその表現力の強力さで注目されている Concurrent Prolog[22),28),29)] を中心に論理型の並列プログラミング言語の基本計算メカニズムおよびプログラミング・スタイルについて解説する。

2　論理型プログラミング言語

一階述語論理をプログラミング言語として解釈することの起源は1974年の IFIP で R.Kowalski [18)] が一階述語論理の部分集合である Horn 節の手続き的解釈を示した時からである。逐次型プログラミング同様、論理型の並列プログラミングにおいても基本となる Horn 節の手続的解釈について最初に説明する。

一階述語論理において節（clause）は素命題（atomic formula）の集合の対であり、次ぎのように書かれる。

```
B1,...,Bm :- A1,...,An.
```

一般に ':-' の左側を結論部、右側を条件部という。素命題は R(t1,...,tk) のような形式をしている。ただし Rは k引数の述語記号（predicate symbol）であり、tiは項（term）である。項*とは変数 X,Y,Z,...であるか f(s1,...,sj) のような表現である。ただし f

---

脚注

＊　本論文では DEC-10 Prolog と同様に大文字で始まる文字列を変数の、小文字で始まる文字列を述語記号や関数記号の表記として用いる。また、項の１つであるリストだけはカッコ“[”、“]”を用いた特別の表記を用いる。

例			
	nil	→	[]
	cons(1,2)	→	[1|2]
	cons(1,cons(2,nil))	→	[1,2]

は j引数の関数記号（function symbol ）であり、siは項である。定数は 0引数の関数記号である。Horn節とは結論部に高々１つの素命題しか持たない節である。

B1,...,Bm :- A1,...,An.　　m ≦ 1.

節形式の文（sentence in clausal form）とは節の集合{C1,...,Cn} であり、その論理的意味は連言（conjunction ）である。

C1 and ... and Cn

節 B1,...,Bm :- A1,...,An.　の論理的意味は、その中に現れている変数 X1,...,Xk　はすべて全称束縛されている（universally quantified）と見なされ、次のように表わされる。

『for all X1,...,Xk,
　　B1 or ... or Bm　　is implied by　A1 and ... and An.』

さて、Kowalskiは Horn 節で表現される論理的な含意

B　if　A1 and ... and An

を手続き宣言（procedure declaration ）と解釈し、 Bを手続き名、A1,...,An を手続き Bの本体を構成する手続き呼出し Ai の集合であると解釈した。より細かく分けるとHorn節は４種類の手続き的解釈が可能である。

① B :- A1,...,An.　　(m = 1)
　これはすでに述べたように手続き宣言と解釈される。 Bは手続き名、{A1,...,An} は手続き本体、各 Ai は手続き呼出しにそれぞれ対応する。
② B.　　(n = 0)
　これは事実のアサーションあるいはデータと解釈される。
③ :- A1,...,An.　　(m = 0)
　これは起動すべき手続きを列挙したゴール文と解釈される。
④ □　　(n,m = 0)
　空節はhalt文と解釈される。論理的にはこれは矛盾を表わしている。

一般に一階述語論理における論理的含意に関するすべての質問は節形式の文の充足不能性に関する質問で置換えられることが知られている。節形式の文の充足不能性を検証する

システム、すなわち定理証明系としてリゾリューション（resolution）を推論規則とするSL-リゾリューション・システム[17]がある。Kowalskiは Horn 節のみからなる節形式の文の充足不能性をこのシステムにおいて検証するとき、Horn節の手続き的解釈の中ではリゾリューションが手続き起動（procedure invocation）と見なされることを示した。例えば、ゴール文、

```
:- integers(0,5,L).
```

と手続き宣言

```
integers(X,U,[X | L]) :-  X ≦ U,plus(X,1,Y),integers(Y,U,L).
```

からリゾリューションにより新しいゴール文

```
:-  0 ≦ 5,plus(0,1,Y),integers(Y,5,L).
```

が導出される。これはちょうど古いゴール文により手続き integers の本体が呼出されたことに相当する。より一般的に言えば、ゴール文

```
:- A1,...,Ai-1,Ai,Ai+1,...,An.
```

と手続き

```
B :- B1,...,Bm.
```

が与えられ、かつ Bが手続き呼出し Ai と変数と項の最も一般的な置換θ（most general unifier）により同一のものに変換できたとき、リゾリューションは新しいゴール文

```
:-  (A1,...,Ai-1,B1,...,Bm,Ai+1,...,An) θ.
```

を生成する。これはまさに手続き Ai の起動と解釈することが可能である。

SL-リゾリューションに基づく定理証明系においてはリゾリューションを用いて古いゴール文を書きかえ新しいゴール文を導出し、最終的に矛盾（空節）を導くことにより節形式の文の充足不能性の検証を行なう。 Horn 節を手続き的に解釈すると、この定理証明の過程が繰返し手続きを起動しながら行なわれる計算と見なせる。すなわち、Horn節プログラムの計算においては解くべき問題が証明すべき定理として与えられ、計算は定理証明系

が生成する証明として得られる。

しかし、上述の計算は２種類の非決定性を含んでいる。１つはゴール文が複数の素命題（ゴール文に含まれる個々の素命題を今後ゴールと呼ぶ）を含むときに、どのゴールに着目してリゾリューションを行うかというゴールを解く順序に関するものである。他は、あるゴールに着目してリゾリューションを行う際に、それを行う手続き、すなわち、節が複数個あるときどの節を用いるかという手続きを用いる順序に関するものである。定理証明の立場からは、この２つは証明の完全性を保証する限りどのように処理されても、すなわちブラック・ボックスになっていてもかまわないが、プログラミングの立場からはこの２つの非決定性が処理系によりどのように処理されるか、すなわち２つの非決定性のオペレーショナル・セマンティクスが与えられていることが計算の制御上絶対必要である。

この２つの非決定性を AND-OR 木を用いて説明する。一般に Horn 節で書かれたプログラムは AND-OR 木で表現することができる[19]（図１）。AND-OR木は ANDノードとORノードからなり、 ANDノードはゴール文に対応する。 ANDノードは子ノードとして自分が持つゴールの数だけ OR ノードを持つ。ただし、 ANDノードが空節を持つときは子ノードは持たない。 OR ノードはゴールを１つ持ち、子ノードとしてそのゴールに対してリゾリューションを行える節の数だけ ANDノードを持つ。子ノードになっている各 ANDノードは親ノードの OR ノードが含むゴールをそれぞれの節により展開した結果のゴール文を含んでいる。定理証明はこの図式の中では『証明すべき定理をゴール文として含む ANDノードをルートとする AND-OR 木において、すべての OR ノードを含みかつ末端ノードが空節を含む ANDノードだけからなるような部分木を見つけること』としてとらえられる（図２）。

この AND-OR 木においては、 OR ノードの各々が手続き呼出しを表わしており、ORノード→ ANDノードの枝が手続き起動を表わしている。すなわち、上述の非決定性はそれぞれ ANDノードにおける分岐（ゴールを解く順序）、 OR ノードにおける分岐（手続きを用いる順序）として図示される。定理証明を手続き的に解釈すると、 OR ノードは手続き呼出しを表わし、空節は halt 文を表わすから、上述の文は『すべての手続きを起動し、かつそれらがすべて停止すること』を述べているに過ぎない。上の文は部分木の探索法を述べていないが、実はこれが手続きの起動の仕方等に対応し、従って論理型プログラミング言語のオペレーショナル・セマンティクスに対応している。DEC-10 Prolog のような逐次型の処理系は AND-OR 木を left-to-right, depth-first に探索して部分木をみつけだす（図３）。すなわち、 ANDノードの子ノードである各ORノードを並べられた順に左から調べ、ORノードに関してはその子ノードを左から順にバックトラック（backtrack ）を使いながら探索する。このことは Horn 節プログラムにおいて A:-B1,...,Bn.の右辺やゴール文 :- C1,...,Cm. の各手続き呼出しが左から順に起動され、かつ左隣の手続きが完全に終了してから次の手続きが起動される、また、ある手続き呼出しを実行する手続き宣言が

複数ある場合にはそれらがプログラム中に並べられた順に試されるというセマンティクスを示している。

一方、並列型処理系はこの木を並列に探索する。この並列型の探索法としては主として次の３種類の方法が提案されている。

OR並列実行

AND-OR木上のORノードで解釈プロセスが分岐する。

AND 並列実行

AND ノードでゴールの数だけ解釈プロセスが分岐する。ゴール間で共有されている変数については各プロセスがそれぞれ独自のコピーを持ち、独立に処理し、後で処理の整合性の検査を行なうこととする。

ストリーム並列実行

AND ノードでゴールの数だけ解釈プロセスが分岐する。ゴール間で共有されている変数はゴールを解く過程においても共有されており、各プロセスはこれらの共有変数を通じて動的に通信し合う事ができる。

これらを並列型プログラミング言語という観点から眺めると、OR並列実行は、１つのゴールに対する複数の解き方をそれぞれ独立にかつ並列に実行することを意味し、ゴールを満たすすべての解を探索することを意味する。非決定的な処理をする上ではOR並列は重要な役割を果す。しかし、並列に走る各プロセスが論理的に独立であり、プロセス間通信やプロセスの同期等を表現する手段がなく、これだけを用いて並列プロセスを記述するには不十分である。

一方、AND 並列やストリーム並列は論理的AND で結ばれた複数のゴールを並列に解くことを意味する。一般に各ゴールは変数を共有することがあるので、変数を共有したまま分岐した各プロセスは独立ではなく．この共有変数を通じて相互作用を及ぼし合う。しかし、AND 並列実行では各並列プロセスは論理的には共有変数を通じて関係があるがプロセス間で情報を交換する動的な手段が何もないこと等により、これらは並列事象を記述したり、あるいは、複数のプロセスを制御したりするためのプログラミング言語としては適当でない。

ストリーム並列実行はConcurrent Prolog を含めたいくつかの論理型の並列プログラミング言語の基本モデルになっているものであるが、共有変数に若干の制御情報をもたせることにより、プロセス間通信やプロセスの同期等が記述できるようになっている。すなわ

ち、並列プロセスをプログラムすることが論理の枠の中だけで行える。

3　論理型並列プログラミング言語

Concurrent Prolog は一階述語論理に基づく並列型プログラミング言語であり、イスラエルの Weizmann 研究所のE.Shapiro により設計された[22]。

Concurrent Prolog の基本となっている　論理型言語のストリーム並列実行が始めて提案されたのはM.van Emden, G.de Lucenaの1979年の論文[6]においてであり、彼等はこれを『論理のプロセス的解釈』と呼んだ。彼等はその論文の中で、１）複数のゴールの同時リゾリューションを並列計算と見なすこと、２）ゴール間で共有されている変数を並列プロセス間の通信用のチャネルとして用いること、３）論理型AND に、並列AND と逐次AND の２種類を設けること等の核となる考えをすべて提示し、さらに通信チャネル中のデータ流の方向付や通信にまつわる同期の基本的アイディアを示した。しかしながら、彼等の言語は非決定的処理の機能を持たなかった。

K.Clark, F.HcCabe, S.Gregoryらの IC-Prolog[3]はその多彩な制御プリミティブのなかにvan Emden, de Lucenaらの『論理のプロセス的解釈』に基づくストリーム並列実行のためのプリミティブを含めている。それらは、上述の３つの他に、１）入出力標記（input, output annotation）、２）ガード（guard ）であり、前者はチャネルを通じたデータの流れをシンタクティックに指定するものであり、また通信に伴う同期のためにも用いられ、後者はゴールを解く節の選択のための条件記述のために用いられた。

Clark,Gregory は IC-Prologに導入したアイディア１，２を発展させて、 Concurrent Prologの原型になった言語であるRelational Language [4]を設計した。この言語はオペレーティング・システム等の並列なシステムを記述することを目的とした野心的な言語であり、共有変数を通じたストリーム通信を中心に据えていた。Relational Language においては、１）IC-Prolog のガードの考えをHoare の CSP[16] の考えを取入れて拡張し、非決定的な節選択機構を実現し(committed choice)、２）入出力標記とストリームによりプロセス間通信・同期メカニズムを実現した。しかし、ガード中のゴールは評価時には変数を含んでいてはならないとか、通信に用いられるものはストリーム、すなわち遅延リストに限る等の制約があった。

Shapiro はこれらをベースに Concurrent Prologを設計した。Concurrent Prolog が新たに導入したプリミティブは、読出専用標記（read only annotation）とコミット（ commit ）オペレータの２つである。その設計方針は Shapiro いうところのオッカムの原則に基づいていた。すなわち、最小のプリミティブで最も強い表現力を実現することである。

4　シンタックス

以下にConcurrent　Prologの基本構成要素を説明する。基本的シンタックスは DEC-10 Prolog[2),20)]と同じである。

[プログラム]　Concurrent Prolog ではプログラムはガード付き節 (guarded clause) の並びとして表わされる。

[ガード付き節／コミット・オペレータ]　ガード付き節は次に示すように常に右辺に“｜”をもつ節である。もし“｜”を陽に含まない節があった場合は右辺の一番左に“｜”があるものと見なされる。

A　:-　G｜B.

Aをこの節のヘッドと呼ぶ。G及びBは、それぞれ論理的にAND で結ばれた述語の並びであり、それぞれ、ガード部、本体部と呼ばれる。“｜”はコミット・オペレータと呼ばれ、論理的意味は ANDであり、逐次型 Prolog のカット・シンボルの拡張概念である。

[AND]*　論理的AND は次のように表される。

“,”

並列AND とも呼ばれ、名前から明らかなように“,”で結合しているゴールを並列に解くことを意味する。

[OR]　論理的ORは、並列ORとも呼ばれ、ゴールとユニファイ可能な節を並列に探索することを意味する。これについては次節でまた述べる。

[読出専用標記] 読出専用標記は“？”で示され、変数の個々の出現に付加することができる。

変数Xに対し　　“X？”のように書く。

脚注

＊　逐次AND というものを考えることも可能であるが Shapiroはそれをプリミティブと考えてはいないのでここでは省略する。

“？”は論理的な意味を持たず、同一節中の変数Ｘと Ｘ？は論理的に同じであるが、計算の制御上は重要な意味を持っている。すなわち、①　“？”の付いた変数（これを読出専用変数と呼ぶ）は、それがインスタンシエートされない間は、変数以外のものとユニファイしてはならないことを示す。②　また、読出専用変数は通常の変数とは常にユニファイできるが、読出専用変数とユニファイした変数はこの性質（読出専用）を自動的に引継ぐ。“？”はオペレータではないので、Ｘ？についてＸが他のプロセスにより変数以外の項にインスタンシエートされれば、この標記も自動的に消滅する。（Ｘ？についてＸがｆｏｏにインスタンシエートされればＸ？はｆｏｏとなる。）前述のように標記は変数の個々の出現について独立に付加でき、通常は複数プロセスで共有される変数に対して個々のプロセスが付加したりしなかったりする。共有変数に標記を付加したプロセスは、この変数をインスタンシエートできなくなり、他のプロセス（標記をもたない）がそれをインスタンシエートするまで（少なくともその principal functor　を定めるまで）待つことになる。（このメカニズムについては、次節で再び述べる）。

DEC-10 Prolog で書いた読出専用標記を扱うユニフィケーションの定義を以下に示す。（ただし、下では“？”は変数に付加されるDEC-10 Prolog 中の後置型のオペレータとして定義されており、上記条件①に反する場合、このunify 述語は失敗する。）

```
unify(X,Y) :- (var(X);var(Y)),!,X=Y.
unify(X?,Y) :- !,nonvar(X),unify(X,Y).
unify(X,Y?) :- !,nonvar(Y),unify(X,Y).
unify([X|Xs],[Y|Ys]) :- !,unify(X,Y),unify(Xs,Ys).
unify([],[]) :- !.
unify(X,Y) :- X =..[F |Xs],Y =..[F |Ys],unify(Xs,Ys).
```

5　基本計算メカニズム　　－－　リゾリューション　－－

ゴールが与えられたとき、それがどのようにして展開されるかについて述べる。

今、ゴールとしてＡ、また、プログラムとして次の節があるとする。

```
A1 :- G1  | B1.
A2 :- G2  | B2.
      .
      .
      .
```

An :- Gn | Bn.

各Giは空であってもよい。ゴールAに対して各節は次の3つに概念的に分類される。

① 候補節（candidate clause） Ai :- Gi | Bi.
読出専用標記の付いた変数に変数以外の項をユニファイすることなしに、A とAiがユニファイし、かつ、Giを解くことができる場合。

② 待機節（suspended clause） Aj :- Gj | Bj.
読出専用変数に変数以外の項をユニファイすることを除けば AとAjがユニファイし、Gjを解くことができる場合。

③ 失敗節（failure clause）
それ以外。

ゴールAは、それが1つ以上の候補節を持てば、その中の1つを選択し（それを Ai :- Gi | Biとすると）その節の本体部Biに展開される。このときの選択のメカニズムは、各節が並列に調べられ一番早く見つかったものが選ばれる。この方法を用いると、非決定的な節の選択が行なわれる。一度ゴールAが本体部に展開されると、他の候補節の探索は放棄される。この意味でコミットオペレータ“|”はカット・シンボルとして機能する。

ゴールAが候補節を全然持たず、かつ少なくとも1つ待機節を持つ場合は、このゴールAは少なくとも1つの候補節が見つかるか、あるいは完全に失敗するまで待機（suspension）させられる。すなわち、後に他のプロセスが読出専用変数に値を書き込み、待機の条件を解除するまでリゾリューションが延期される。

一般に複数プロセスで共有されている変数に対して、一旦それをインスタンシエートするプロセスが共有変数の値を変更するようなバックトラックを起すと、それを参照しているプロセスもバックトラックしなければならなくなり、いわゆるプロセス間にまたがった分散バックトラック（distributed backtrack ）が生じる。分散バックトラックは並列プロセスの動作が不明瞭になること、および経験上並列プロセスを記述するのに必ずしも必要でないという立場からConcurrent Prolog ではこれを排除し、かわりに失敗により項へのインスタンシエーションが変更される可能性のある共有変数についてはそれが確定するまで他のプロセスに公開しないというメカニズム（Commitmentと呼ばれる）を設けている。すなわち、この複数の節の並列探索の途中で起きるいかなる変数のインスタンシエーションも、“|”を過ぎて他の可能性が放棄されるまでは確定しないので、読出専用標記のついていない共有変数については、仮にそれが探索の途中で変数以外の項にインスタンシエートされても、“|”を過ぎるまでは他のプロセスにアクセスさせないようにしている。言替えると各節の並列な検査はそれぞれ専用の環境を作って行なわれると考えることがで

きる。

Commitmentという概念はゴールを展開する節の探索において、プロセス間にまたがる分散バックトラックを招くことなく最大限の並列性を引出すために考案された。この言葉は『ゴールの証明の成功あるいは失敗をある節（ある変数のあるインスタンシエーション）に委ねる（commitする）』というところからきている。Commitmentはコミットオペレータにより実現される。コミットオペレータによりcommitされる以前の各節選択プロセスの中の変数のインスタンシエーションは外部に対して隠しておかれるが、ある節選択プロセスがcommitに成功すると、他の節選択プロセスは放棄され、Commitされた節選択プロセスの変数のインスタンシエーションは並列に実行されている他のプロセスに公開される。もし仮に、このインスタンシエーションでそのプロセスが失敗した場合は、他の並列に走るプロセスも同時に失敗するため分散バックトラックは生じない。なぜなら、“並列に走る”ということは論理的にはAND で結合されていることに他ならないからである。

以上をまとめるとコミットオペレータの意味は次の３つからなると考えられる。

1　計算を処理中の節に委ね、他の可能性の探索を止める。
　（このことを特にcommitted choice　と呼ぶ）
2　ゴールとヘッドとのユニフィケーションおよびガード部の処理中に生じた局所的な変数のインスタンシエーションを他のプロセスに公開する。
3　ガード部の処理が終了するまでは制御を本体部へ渡さないという逐次性。

6　計算モデル[32),33)]

ここでは、Concurrent Prolog における並列計算のモデルについて述べる。

はじめにイベントという言葉を次のように定義する。

『イベントとは、ゴールがある節のヘッドとユニファイされ、かつ、その節のガード部の計算が成功して終了することである。』

第３節で述べたことを用いるとイベントが生起するための条件は、「ゴールが節のヘッドと読出専用変数のユニフィケーションに関する条件を守ってユニファイでき、かつそのガード部を解くことができること」である。ゴール　A　が節　A':- G | B1 … Bn. のヘッドとユニファイに成功し、ガード部G を解くことができたというイベントを次のように表わすことにする。

A → A'

ゴールが与えられると複数の節が並列に探索されるから、節の数に対応した複数のイベントの生起の可能性が調べられ、条件を満足した１つだけが実際に生起し、コミットオペレータにより他の可能性はすべて捨てられる。ゴール Aに対して１つもイベントが生起しない場合には、そのゴールは第３節で述べた条件に従って待機させられるかもしくは失敗する。

一般にゴールの節によるリゾリューション（イベント）はその節の本体部にあるゴールのリゾリューション（イベント）を引起すため、イベントの生起の間には因果関係が定義される。ただし、ゴールが本体部が空の節とユニファイした場合には、その本体部は空であるから、このイベントが因果関係で結合された一連のイベントの終端となる。イベント E がイベント E' を引起すとき、その因果関係を次のように表わすことにする。

$$E \Rightarrow E'$$

イベント生起の因果関係の定義は次のように述べられる。

『イベント E がイベント E' を引起すのは、

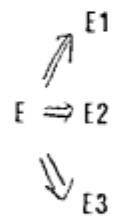
$$E = A \rightarrow A' \quad , \quad A' :- G \mid B1 \ \cdots \ Bn.$$
$$E' = B \rightarrow B'$$

であるときに

$$B \in \{Bi \mid 1 \leq i \leq n\}$$

の場合及びその場合に限られる。』

一般に１つのイベントは AND並列性に対応して複数のイベントを引起す。



このイベントという概念を用いるとConcurrent Prolog におけるプロセスという概念は

次のように定義される。

『ゴール　A によって引起されるプロセスとはゴール A　の関係するイベントによって直接あるいは間接に引起される一連のイベントすべてを言う。』

すなわち、ゴール　A によって引起されるプロセスとは、ゴール A　を解く過程全体を指す。従って、ゴール A　によって引起されるプロセスが終了するのはそれが含むすべてのイベントが生起し終ったときである。言替えればゴール A　が完全に解かれたときゴール A　によって引起されたプロセスは終了する。

一般に AND並列性に対応して、1つのイベントが複数のイベントを引起すことが可能であるため、1つのプロセスはその中で複数のプロセスに分岐することがある。このような並列に走る複数のプロセスは共有変数を通じて通信し合い、さらに共有変数に対する読出専用標記により同期することができる。

リゾリューションにより1つのゴールがある節の本体部に展開されるときは、コミットオペレータの逐次性より常にその節のガード部にある素命題がすべて解かれなくてはならない。従って、イベントが生起するためにはガード部の計算に成功しなければならないが、実はガード部の計算自体が1つあるいは複数のプロセスになっている。（ガード部が複数プロセスからなる場合は、コミットオペレータは実はプロセスのforkに対するjoinを表わしている。）そして、ガード部の計算がまたガード部の計算を起動するから、一般にガード部の計算は何重にもネストされることがある。

7　Concurrent Prolog によるプログラミング

7．1　ストリーム　－（プロセス間通信）－

Concurrent Prolog においてはプロセス間の通信は、そのプロセス間で論理的に共有されている変数を通じて行なう。すなわち、あるプロセスが別のプロセスへ情報（これを以後メッセージと呼ぶ）を送る場合には、それらの間で共有されている変数へメッセージをインスタンシエートすることにより伝える。論理変数に対する破壊的代入は許されていないから、1つの共有変数を用いた通信は1回しかできない。しかし、一般に1つの変数を共有するプロセスの数に制限はないため、あるプロセスが共有変数をメッセージにインスタンシエートするとその変数を共有するすべてのプロセスに同時にメッセージが伝わることになる。すなわち、メッセージの放送（broadcast ）がそのまま実現されている。プロセス間の共有変数は一般にプロセスが分岐したときに生成される。

```
p(X) :-   q(X,Y), r(Y?).
```

上の例では論理変数Y がゴール q で引起されるプロセスとゴール r で引起されるプロセスの間で共有され、それらの間の通信に用いられる。しかし、前述のように１つの論理的に共有されている変数を通じたプロセス間の通信は１回しかできないため、プロセス間で引続き何度も通信するためには新しい共有変数を動的に生成する必要がある。このための方法として一般的なのは、単に共有変数を伝えたいメッセージにインスタンシエートするだけでなく、メッセージと変数を組にしたデータ構造にインスタンシエートする方法である。

［＜メッセージ＞｜＜変数＞］

上では、メッセージをリストのcar 部、変数をcdr 部にもつデータ構造－遅延リスト－を例として示した。このデータ構造はRelational Language [4]で用いられている。この場合、メッセージと組にされる変数はメッセージ送信側から受信側へメッセージと共に送られることになり新しい論理的に共有される変数となる。従って次回のメッセージの伝送はこの変数を用いて行なうことが可能となり、常にメッセージをこのような形式で送る限り新しい共有変数が生成されるから継続的なメッセージ伝送が可能になる。この方式による通信の簡単な例として、整数を１つ生成する度に送信するプロセスと、それを受信する度に書き出すプロセスとの通信の模様を以下に示す。

例１　ストリーム

ゴール：　integers(0,N),outstream(N?).

```
integers(X,[X | N]) :-                          ( 送信 )
      Y is X+1  |  integers(Y,N).
outstream([X | N]) :-                           ( 受信 )
      wait_write(X)  |  outstream(N?).
```

wait_write は引数がインスタンシエートされるまで待機する機構を備えたConcurrent Prolog のシステム述語である。 is は右辺の算術式を評価し、結果を左辺の項とユニファイする中置型のシステム述語であり、この述語も右辺が変数を含まなくなるまで待機する機構を備えている。一般にConcurrent Prolog のシステム述語はすべてこの機構を備えていなければならない。outstream は共有変数 N がインスタンシエートされていないときは読出専用変数に対するユニフィケーションの条件より待機させられることに注意。なお、このタイプの通信はClark & Gregory の論文[4]ではストリーム通信と名付けられており、

ストリーム通信によりプロセス間で流れるデータの系列をストリームと呼んでいる。ここでもこの名称を以後採用する。

例２　merge

２本のストリームを各々のストリーム内の要素の順序を保持したまま、非決定的に１本のストリームにマージするプログラムを考える。これは次のように書ける。

ゴール：　merge([1,2,3],[4,5,6,7],Z).

```
merge([X|Xs], Ys,[X|Zs]) :- |merge(Xs?,Ys,Zs).
merge(Xs,[Y |Ys],[Y|Zs]) :- |merge(Xs,Ys?,Zs).
merge([],Ys,Ys).
merge(Xs,[],Xs).
```

merge は第１、第２引数が入力ストリームに対応しており、第３引数が出力ストリームである。２本の入力ストリーム双方にデータがある場合には、第１、第２節が競争し、先にコミットオペレータに到達した方が他を消すという形で非決定的処理を実現している。

例３　素数の生成

ストリーム処理による、エラトステネスのふるいのアルゴリズムに基づいた素数生成のプログラムは次のように書ける。

ゴール：　primes.

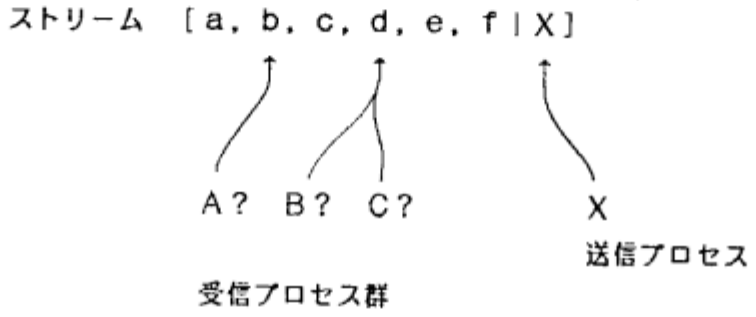
```
primes :- |integers(2,I),sift(I?,J),outstream(J?).
sift([P |I],[P |R1]) :- |filter(I?,P,R),sift(R?,R1).
filter([N|I],P,R) :- 0 is N mod P |filter(I?,P,R).
filter([N |I],P,[N |R]) :- otherwise| filter(I?,P,R).
```

integers と outstream の定義は例１にある。述語primesはユーザによって最初に起動されるべきゴールである。対応する節は３つのプロセス、integers,sift および outstream を生成する。integers は２より始まる整数の無限列をストリームとして次々と生成する。siftはその整数列をそれまでに得られた素数によってふるいにかける。outstream は sift を抜け出てきた数列（これは素数しかない）を次々とプリントする。filterの第２節中のotherwise はConcurrent Prolog のシステム述語であり、通常ガード部におかれ、

他の節のガード部がすべて失敗した時にのみ成功する述語である。

並列プロセス間での継続的な通信を共有変数を介して行なうことは見方を変えると並列プロセス間で再帰的構造をもつ無限構造体を共有しているとも見れる（例えば、ストリーム通信の場合では、共有する再帰的構造をもつ無限構造体は無限長のリストである）。ただし、このような無限構造体は常に部分的にインスタンシエートされた形でしか現れないので実際上は問題ない（例えば、ストリーム通信の例でいえば、遅延リスト [X | Y] の中の変数Y の中に無限の構造が隠されている）。このような見方に立つとき、通信は送信者の書込み位置を示すポインタと受信者の読込み位置を示すポインタとが独立に無限構造体の上を移動する現象として見える。具体的な例として、ストリーム通信の場合についてこれを説明する。

送信プロセスがもつポインタは、常に遅延リストのテイルを指し、受信プロセスはこれより手前（ヘッドに近い方）を指すポインタを読出専用変数の形で持つ。



送信プロセスのポインタは送信される新しいデータが書き込まれる位置を指し、個々の受信プロセスのポインタは次に読み出す（受信する）データが存在する位置を指している。各プロセスは送信・受信を行なうたびにポインタを１つずつ後（テイル方向）へずらす。受信プロセスのポインタ変数を読出専用変数にすることにより、受信プロセスの持つポインタが決して送信プロセスのポインタを越えてテイル方向へ移動することはないことが保証されている。送信プロセスのポインタと受信プロセスのポインタとの間にあるデータはまだ読み出されていないデータを表わしているが、これはちょうど受信バッファ中にあってまだ読み出されていないデータに対応する。送信プロセスのポインタと受信プロセスのポインタはいくら離れていてもかまわないので、これは無限長バッファを持つ通信を行なっていることに相当する。

7．2　オブジェクト／恒久的プロセス[23),29),30),31)]

Concurrent Prolog は種々の有用なプログラミング上の概念を表現できるが、そのなか

で最も重要なものの１つにオブジェクト指向型プログラミングがある。ただし、ここでいうオブジェクトの概念はactor 理論[12),35)] のそれに非常に近く、オブジェクト指向型プログラミングとして、計算が分散している複数のオブジェクト間でメッセージを交換し合いながら行なわれるようなプログラミング・スタイルを考えている。最初にここでのオブジェクトという概念に対するとらえ方を示し、次ぎに簡単な例を示し、Concurrent Prolog におけるオブジェクト指向型プログラミングの原則を示す。

オブジェクトの考え方

- ●オブジェクトは内部状態を持つことができ、メッセージを受取ったときにアクティブになるプロセスである。
- ●オブジェクトの内部状態は外部からはメッセージを送ることによってしか操作できない。
- ●オブジェクトは、その計算中に他のオブジェクトにメッセージを送ることができる。
- ●オブジェクトは１つの定義からいくつでも生成することが出来る。

例４　オブジェクトの例　　計数器

```
counter([clear | S],State) :- counter(S?,0).                                  (1)
counter([up  | S],State) :- NewState is State+1 | counter(S?,NewState).       (2)
counter([down  | S],State) :- NewState is State-1 | counter(S?,NewState).     (3)
counter([show(State) | S],State) :- counter(S?,State).                        (4)
counter([],State).                                                            (5)
```

Concurrent Prolog におけるオブジェクト指向プログラミングの原則

①オブジェクトはその内部状態を引数として持ち、自分自身を再帰的に呼ぶ恒久的なプロセスとして実現される。

例えば、(1) から(4) の節はすべて自分自身を再帰的に呼びだしている。また第２引数をcounter の内部状態の値を保持するために用いている。

②オブジェクトは共有変数を通じて通信し合う。

並列AND で生成された複数プロセスは共有変数により論理的に結合されており、これらの変数はオブジェクト間で通信用に持ちいられる。メッセージの伝送はこの共有変数にメッセージをインスタンシエートすることによりなされる。一度インスタンシエートされた変数は２度とメッセージ送信用に使えないので通常はストリーム通信のように変数をメッセージと次ぎの通信に使う変数とのリスト［＜メッセージ＞｜＜変数＞］にインスタンシ

エートする。３つ以上のプロセスが１つの変数を共有する場合にはメッセージは同時に複数のオブジェクトに伝送されるということができる。counter の第１引数はこのようなメッセージ・ストリームの入力に用いられている。例えば、ゴール文

```
:- user(C),counter(C?,0)
```

においてはオブジェクト user が変数　C を通じてcounter オブジェクトにメッセージを送れる。

③オブジェクトはメッセージを受取った時にアクティブになる。それ以外の時は待機状態にある。

読出専用変数を変数以外のものとインスタンシエートしようとしたプロセスを待機させるメカニズムにより、メッセージを待つオブジェクトは待機状態に入る。具体的には、入力ストリームにメッセージがない、すなわち第１引数が変数のときは、(1) から(5) の節とのユニフィケーションはすべて待機させられる。

④オブジェクトはそれが新たに生成された時にインスタンシエートされたと見なす。

ガード付き節で書かれたプログラム (1)－(5) はインスタンシエートのための型として使われる。この定義と同じ述語名を持つゴールが新たに生成されると、それはその定義の新しいインスタンスとなる。例えば、ゴール:　counter(C1?,0),counter(C2?,0)　はcounter を２つ生成したことになる。

⑤メッセージに対する返事

オブジェクトが返事を必要とするようなメッセージを受取ったときに、返事を返す方法として２通りの方法がある。１つは返事専用の別の通信用チャネルをあらかじめ用意しておく方法であり、他はより簡単な方法であってメッセージの一部に返事を返してもらうための通信用変数を含めて送るものである。例えば、counter の内部状態を尋ねるメッセージは'show(X)' というメッセージが使われる。この場合には変数X が応答用に使われる。'show(X)' を送ったオブジェクトは相手のcounter がこの変数に答をインスタンシエートするのを待っていればよい。

７．３　Concurrent Prolog におけるプロセス間通信の新しい特徴[23),30),32),33)]

従来から共有変数を通じて並列プロセス間で通信を行なう方法は提案されており、それ

らはメモリーを共有したり、グローバル変数を共有したりしてきた。これらの方式とConcurrent Prolog における共有変数による通信の相違は、共有しているものの抽象度の高さにある。従来方式のものでは共有されるものは物理的なもの（メモリーセルなど）であったのに対し、Concurrent Prolog で共有されるものは論理的な変数であってユニフィケーション等の論理演算の対象にもなりうる非常に抽象度の高いものである。この抽象度の高さゆえにConcurrent Prolog においては並列プロセス間の高度の通信を簡潔に記述することができる。

例５　merge　（例２の続き）

ゴール：　p(X),q(Y),merge(X?,Y?,Z),r(Z?)

このmerge プログラムは２本のストリームを１本にマージする。最初の２つの節はそのためのものである。後２つの節はそれぞれ２本の入力ストリームの中の１本のストリームが尽きた（[ ] はストリームが尽きたことを表す）場合を記述しており、尽きていない方の入力ストリームを表わす変数と出力ストリームを表わす変数をユニファイし、merge プロセス自体は停止している。このユニフィケーションにより、以後残った入力ストリームのデータは何の中継もなしに（merge プロセスはもはや存在してない）直接出力ストリームへ流れる。重要な点はこのようなデータの流れの変更を入力ストリームの送信者にも出力ストリームの受信者にも全然知らせることなく２つの変数のユニフィケーションで行えるということである。

例６　switch

ゴール：　p(X),switch(X?,Y,Z),q(Y?),r(Z?)

```
switch([on | X],[],X).
switch([A  | X],[A  | Y],Z) :- switch(X?,Y,Z).
```

このswitchプログラムは'on'というデータが来るまでは入力ストリームを第２引数にある通信用変数に転送し続け、'on'というデータが来た後は第２引数に対応する通信チャネルを閉じ、入力ストリームを第３引数にある通信用変数に転送する。merge の例と同様にデータの流れの変更はストリームの送受信者双方に見えないように直接的に行なわれ、その変更は switch プロセスが停止した後も残る。

以上２例は、ストリーム通信で使われる共有変数どうしのユニフィケーションによる動的な通信ネットワークの再構成の簡単な例であった。次ぎの例は２つの変数をユニファイ

することにより新しく共有変数ができ、離れたオブジェクト間に直接通信できるパスが新たに形成される例である。

例７　queue manager

ゴール文：　user1(X),user2(Y),merge(X?,Y?,Z),qm(Z?,Q,Q).

```
qm([enqueue(X) | S],Head,[X | Tail] ) :- qm(S?,Head,Tail).
qm([dequeue(Y) | S],[Y  | Head],Tail) :- qm(S?,Head,Tail).
```

qm(queue manager) プロセスは内部状態としてキューを持ち、'enqueue(X)'および'dequeue(Y)'メッセージを受取る。キューはd-listとして実現されており、qmの第２、第３引数がそれぞれキューのヘッドとテイルへのポインタに対応している。ゴール文にあるように起動時にはqmの第２，第３引数は等しくしておかなければならない（これは空のキューに相当する）。'enqueue(X)'メッセージを受取ったときはキューのテイルをX でインスタンシエートし、'dequeue(Y)'メッセージを受取ったときは Yをキューのヘッドの要素にインスタンシエートする。qmプロセスは本質的にはenqueue されたX とdequeue された Yとをそれぞれ到着順に並べ、対応のとれるもの同士をユニファイしているだけである。従って対応のとれる Xと Yとがともに変数だった場合には XとY がユニファイされる結果、X をenqueue したプロセスとY をdequeue したプロセスとの間に新しい共有変数ができ、将来X をenqueue したプロセスにおいてX が何かにインスタンシエートされたとき、その値は互いに相手を知らなくても、それをdequeue したプロセスにqmを介さずに渡る。

8　他の言語との比較

Parlog[5]は Clark,GregoryによるRelational Language に継ぐ論理型の並列プログラミング言語であり、基本的特徴をRelational Language から引継いだ他に、Concurrent Prolog からとりいれた特徴、さらにはParlogで新たに導入された特徴などがある。Relational Language は読出専用標記の原型となった入出力標記を持っていたが、基本的に変数に対する入出力の仕様を与えるモード宣言とコンパイル時の徹底したモード解析により最適なコードを生成しようという思想に基づいて設計された。しかし、それがために、通信に使われる共有変数は論理変数としての完全な性質を失っていた。例えば、通信のスタイルをストリーム通信に限っていたり、ストリーム通信により送られるメッセージは変数を含んでいてはならないというような制約があった。メッセージ中の変数は、メッセージに対する返事をその中の変数にインスタンシエートさせることにより送信プロセスに応答を伝えるときなどに非常に有用な手段であり、Concurrent Prolog では通信は純粋に論理変数を用いて行なわれていたので当然可能であった。ParlogはRelational Language のモード

宣言とコンパイル時の徹底したモード解析の思想を入出力標記同様引継いでいるが、上述のメッセージ中の変数による応答を可能にするようにストリーム中の変数による逆方向通信の考えを実現するための特別のモード宣言を追加している。

Parlogのモード宣言と Concurrent Prolog の読出専用標記は共に変数に対する入出力仕様を与えるものであるが、両者の違いは、モードは一度宣言しておけば、あらゆる呼出しに適用されるのに対し、読出専用標記の方は呼出しのたびごとにプログラマの責任で設定する必要があるという点にある。これは Parlog がコンパイラにより最適なコードを生成することを目的としたこと、一方、 Concurrent Prolog はプログラマに強力な表現力を持ったプリミティブを与えることを目的にしたことと関係があるように思える。

Parlog はこのほかにも新しい重要な特徴を持っている。最大の特徴は、committed choiceを基本計算メカニズムとすることにより失われた、あるゴールを満足するすべての解を求めるという全解探索の機能を、集合表現をインターフェースとして導入したことである。これによりデータベースのような条件を満たすものをすべて求めるといった応用も容易に記述できるようになった。また、さらに本質的に決定的な関係に対しては関数表記を許し、記述上の便宜を計っている。

9 おわりに

本論文では論理型の並列プログラミング言語の基本計算メカニズム及びその特徴的なプログラミング・スタイルについて、 Concurrent Prolog を中心に解説した。

ここでは、まとめにかえて、Concurrent Prolog をベースにした研究状況について述べることとする。

Shapiro はWeizmann研究所において、Concurrent Prolog の表現力を調べるために種々の並列アルゴリズムの記述をConcurrent Prolog で行なっている[11),21)] 。この過程でいろいろ新しいプログラミング手法が開拓されているようである。また、Concurrent Prolog でオペレーティング・システムを記述することも試みている[24)] 。しかし、Concurrent Prolog などの言語でシステム・プログラミングをすることについては、通信ネットワークをマージを用いてはるオーバヘッドによる効率の悪さの指摘[10)] がある。これに対する解は今のところネットワークの通信コストのバランスを動的に保つ方法が提案[25)] されているだけであり、より高速なインプリメンテーションを含め、今後の検討が必要なところである。Shapiro はまた最近 Bagel という並列アーキテクチャを提案し[26)] その上でConcurrent Prolog にプロセッサ標記というものを導入した言語を使ってシストリック・アルゴリズム等の研究をしている。

ICOTでも Concurrent Prologの強力な表現力を証明するようないくつかのプログラムの開発が行なわれている。それらは、有限長バッファ付きのストリーム通信[9),32),33),34)]や Cocnurrent Prolog による Or 並列プロログ処理系の記述[14),15)] などである。ICOTでは現在 Concurrent Prolog をベースに並列推論マシン上の言語、核言語第１版の設計を行なっているが、それは Concurrent Prolog の並列性を基礎としており、Parlogの集合表現などを取入れている。また、問題解決・推論や知識表現のような、より高レベルの応用に使うためにConcurrent Prolog をベースにしたMandala という言語を設計中[7),8)]である。Concurrent Prolog を用いた本格的な応用は現在の処理系が遅いため、まだあまりないが、ハードウェアの仕様記述・検証の試み[27)] や並列構文解析の記述[13)] 等に用いられ始めている。

論理型並列プログラミング言語は歴史の浅い言語である。文献[22)] にある DEC-10 Prologでかかれた Concurrent Prologの処理系（学習用には十分使える）を手近なProlog処理系に移植して、実際にプログラミングしてみることがこの新しい言語を理解する最良の方法であると思う。最後に日頃御指導いただく ICOT 第２研究室室長古川康一氏にここで厚く感謝する。

[1] T.Chikayama: ESP - Extended Self-contained PROLOG - as Preliminary Kernel Language of Fifth Generation Computers, Journal of New Generation Computing, Vol.1, No.1 (1983), pp. 11-24.

[2] W.Clocksin & C.Mellish: Programming in Prolog, Springer-Verlag 1981.
及びその邦訳本：Prologプログラミング、中村訳、日本コンピュータ協会.

[3] K.Clark et al.: IC-Prolog language features, in Logic Programming, K.Clark & S.Tarnlund (ed.), Academic Press, 1982, pp.253-266.

[4] K.Clark & S.Gregory: A Relational Language for Parallel Programming, Research Report DOC81/6, Imperial College, 1981.

[5] K.Clark & S.Gregory: PARLOG: A Parallel Logic Programming Language, Research Report DOC83/5, Imperial College, 1983.

[6] M.H.van Emden & G.J.de Lucena: Predicate Logic as a Language for Parallel Programming, in Logic Programming, K.Clark & S.Tarnlund (ed.), Academic Press, 1982, pp.189-198.

[7] 古川 et al. : Concurrent Prolog 上の知識情報処理用プログラミング言語／システム Mandala （曼陀羅）について、TM-0028, ICOT, 1983.

[8] K.Furukawa et al. : Mandala: A Concurrent Prolog BasedKnowledge Programming Language/System,情報処理学会、知識工学と人工知能研究会資料32-1 (1983).

[9] K.Furukawa & A.Takeuchi: Bounded Buffer Stream Communication in Concurrent Prolog to appear in Journal of New Generation Computing.

[10] D.Gelertner: A Note on Systems Programming in Concurrent Prolog, Proc. of 1984 International Symposium on Logic Programming, Atlantic City (1984).

[11] L.Hellerstein & E.Shapiro: Implementing Parallel Algorithms in Concurrent Prolog: The MAXFLOW Experience, Proc. of 1984 International Symposium on Logic Programming, Atlantic City (1984).

[12] C.Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence 8, (1977), pp.323-364.

[13] H.Hirakawa: Chart Parsing in Concurrent Prolog, TR-008, ICOT, 1983.

[14] H.Hirakawa et al. : Or-Parallel Optimizing Prolog System: POPS - Its Design and Implementation in Concurrent Prolog, TR-020, ICOT, 1983.

[15] H.Hirakawa et al.:Eager and Lazy Enumerations in Concurrent Prolog, TM-0036, ICOT, 1984.

[16] C.A.R.Hoare: Communicating Sequential Processes, Comm. ACM, Vol.21, No.8 (1978), pp.666-677.

[17] R.Kowalski & D.Kuehner: Linear resolution with selection function, Artificial Intelligence 2,(1971), pp.227-260.

[18] R.Kowalski: Predicate Logic as Programming Language, Proc. of IFIP'74 (1974), pp.569-574.

[19] R.Kowalski: Logic for Problem Solving, North Holland, 1979.

[20] L.M.Pereira et al.: User's Guide to DECsystem-10 PROLOG, University of Edinburgh, 1978.

[21] A.Shafrir & E.Shapiro: Distributed Programming in Concurrent Prolog, IR CS83-12, Dept. of Applied Mathematics, The Weizmann Institute of Science (1983).

[22] E.Shapiro: A Subset of Concurrent Prolog and its Interpreter, TR-003, ICOT, 1983.

[23] E.Shapiro & A.Takeuchi: Object Oriented Programming in Concurrent Prolog, Journal of New Generation Computing, Vol.1, No.1 (1983), pp.25-48.

[24] E.Shapiro: Systems Programming in Concurrent Prolog, Proc. of 11th Annual ACM SIGACT SIGPLAN Symposium on Principles of Programming Languages, Salt Lake City (1984).

[25] E.Shapiro: Fair, Biased and Self-Balancing Merge Operators: Their Specifications and Implementation in Concurrent Prolog, Proc. of 1984 International Symposium on Logic Programming, Atlantic City (1984).

[26] E.Shapiro: The Bagel: A Systolic Concurrent Prolog Machine, to appear in ICOT TR.

[27] N.Suzuki: Experience with Specification and Verification of Complex Computer Using Concurrent Prolog, in Logic Programming and its Applications, D.H.D.Warren & M. van Caneghan (ed.), Lawrence Erlbaum Press.

[28] 竹内：新世代プログラミング③：新しいプログラミング・スタイルと言語（１）－ Concurrent Prolog, ｂｉｔ, Vol.15, No.10 (1983), pp.19-24.

[29] 竹内：新世代プログラミング⑤：新しいプログラミング・スタイルと言語（３）－ Concurrent Prolog によるオブジェクト指向プログラミング, ｂｉｔ, Vol.15, No.12 (1983), pp.62-70.

[30] 竹内 & Shapiro: 論理型言語によるオブジェクト指向プログラミングについて、情報処理学会第２６回全国大会予稿集 (1983) pp33-34.

[31] 竹内 et al. : Concurrent Prolog によるオブジェクト指向プログラミング、Proc. of Logic Programming Conference'83, Tokyo (1983).

[32] 竹内 & Shapiro：論理型言語によるコンカレント・プログラミングについて、情報処理学会ソフトウェア基礎論研究会4-4 (1983).

[33] A.Takeuchi & K.Furukawa: Interprocess Communication in Concurrent Prolog , Proc. of Logic Programming Workshop'83, Albufeira (Portugal) (1983), pp.171-185.

[34] 竹内 &古川： Concurrent Prolog におけるプロセス間通信の実現、情報処理学会第２７回全国大会予稿集 (1983), pp.363-364.

[35] 米澤：Actor 理論について，情報処理、Vol.20, No.7 (1979), pp.580-589.

図1. AND-OR木 の例

□ は ANDノード, ○ は ORノード を表わす.

C1, C2, C3 は それぞれ 次のような節 である.

C1　　$p_1 :- s, t.$

C2　　$p_2 :- u.$

C3　　$p_3.$

図2. 定理証明の AND-OR木上での表現

太線で囲まれた証明木をみつけることが 定理証明の目的である.

図3　逐次型探索

逐次型処理系は 木を left-to-right, depth-first に探索する.

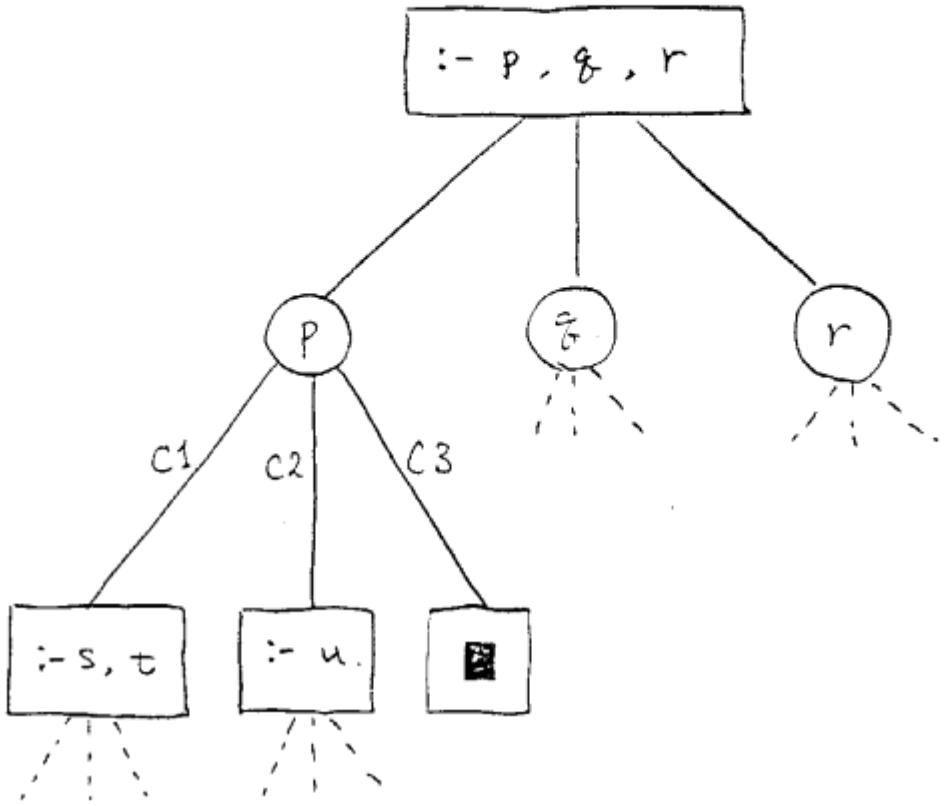


図 1.

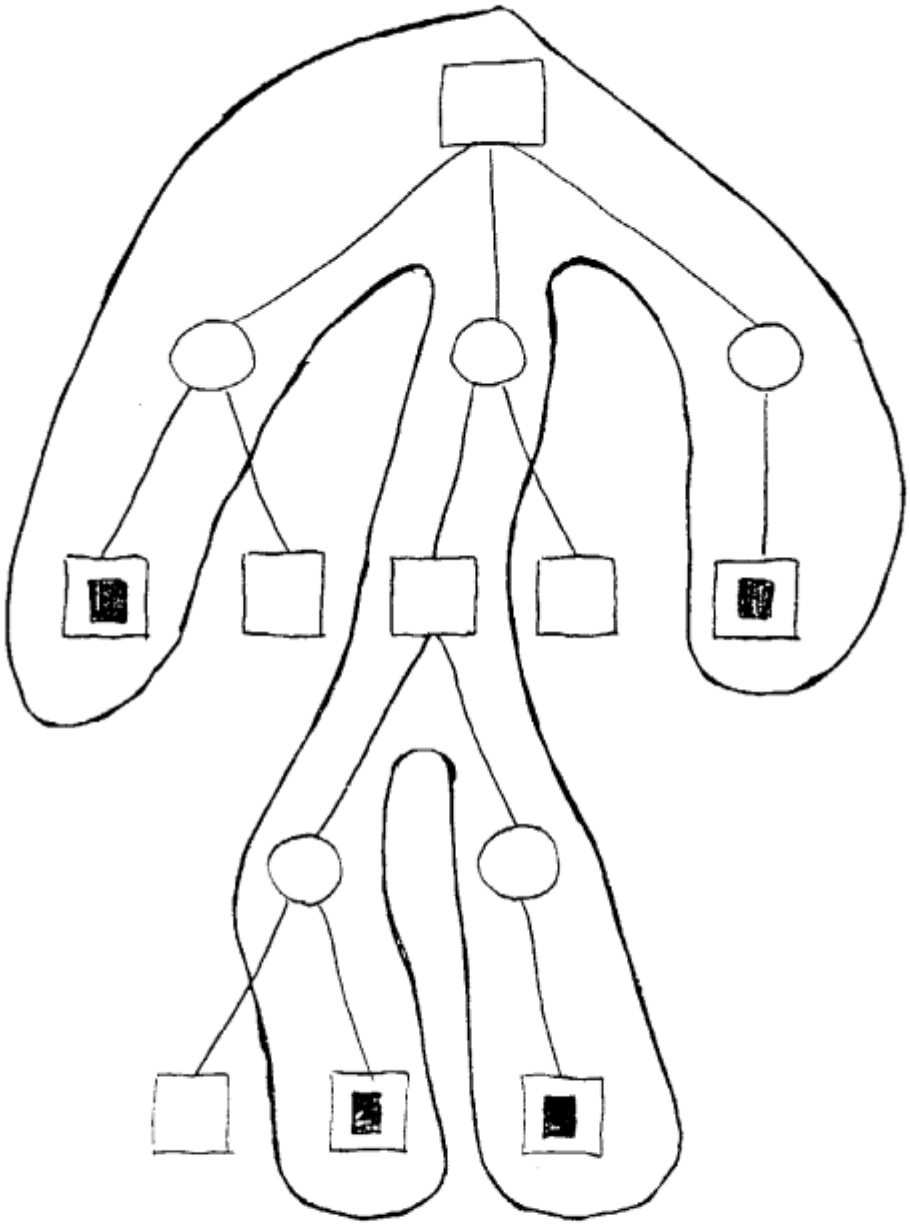


図 2.

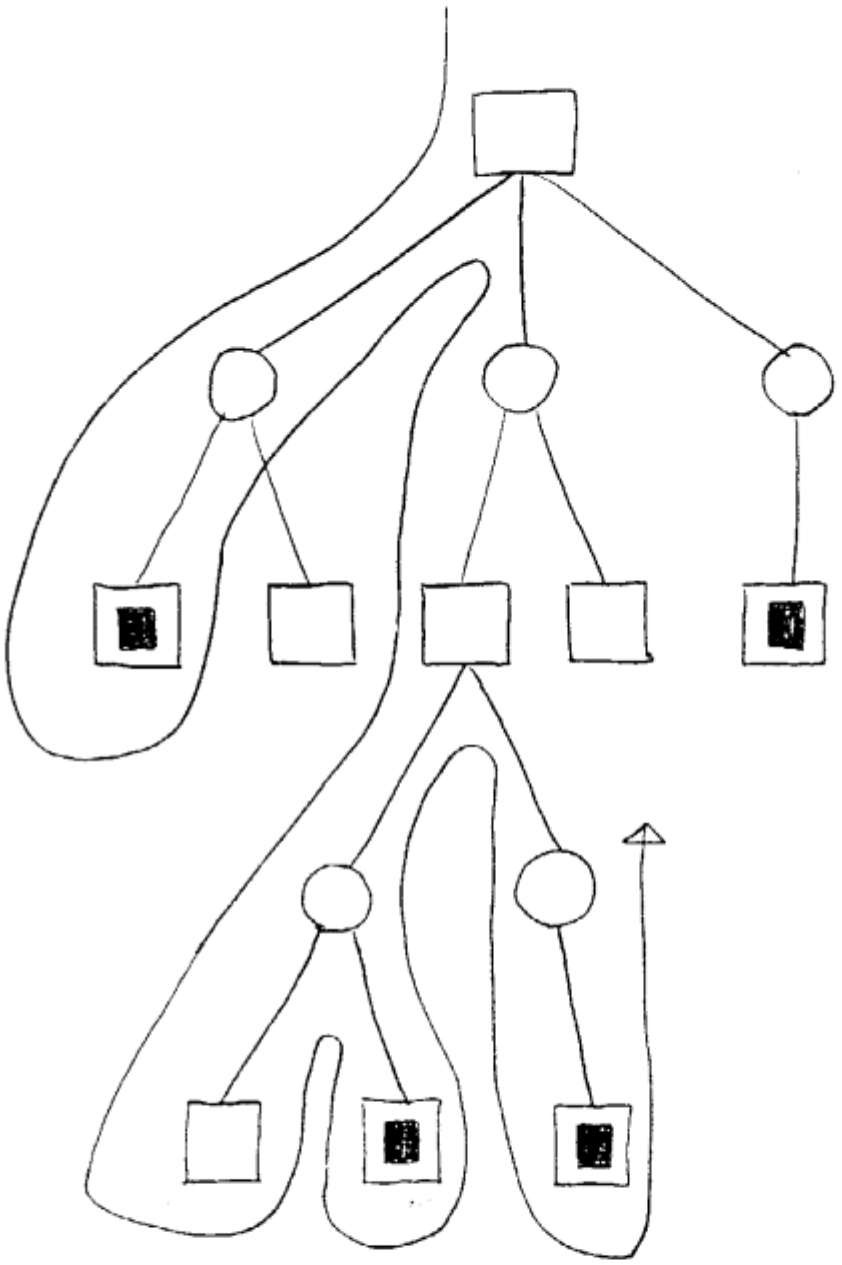


図 3