Let's Talk Concurrent Prolog

竹 内 彰 一

ICOT Technical Memo

Let's talk Concurrent Prolog

A.Takeuchi ICOT 2nd Lab.
E.Y.Shapiro Weizmann Institute of Science

論理型言語に並列実行のoperational semantics を導入することにより、その記述力は飛躍的に高まる。 このMemoではその導入方法について述べるとともに、基本的なプログラミング・テクニックを解説する。

内容

参考文献

1. E.Y.Shapiro, " ICOT Technical Report" (to appear)
2. K.L.Clark S.Gregory , "A Relational Language for P arallel Programming " Research Report DOC81/16, Imperial College (1981)

1. 論理型言語の並列実行

論理型言語で書かれたプログラムの意味は論理的に記述されており、実行メカニズムとは独立である。しかし、計算機上でそれを解釈実行する処理系は、それぞれ論理型言語のオペレーショナル・セマンティクスを持っており、それに基づいてプログラムを実行する。

一般に論理型言語で書かれたプログラムはAND-OR Tree で表現され、逐次型インタープリタはこのtreeをleft-to-right,depth-first でtraverseする。すなわち、論理的AND で結ばれた複数のゴールの並びが与えられると、それを並べられた順に左から解き、1つのゴールに対しては、それを解くいくつかの方法を順にバックトラックを使いながら試す。

一方、並列型インタープリタはこのtreeを並列にtraverseする。この並列traverseには2種の併用可能な方法がある。1つはOR-parallel 、他はAND-parallelである。OR-parallel 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、これはAND-OR tree 上では、OR-node でプロセスがforkすることに対応する。Non-deterministic な処理をする上ではOR-parallel は重要な役割を果す。しかし、forkした各プロセスは完全に独立であるため、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-parallel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し、AND-OR tree ではAND-nodeでプロセスがfork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forkした各プロセスは独立ではなく，この共有変数を通じてinteractし合う。このHemoで述べる言語はこの共有変数に若干の制御情報をもたせ、AND-parallel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。

2. Concurrent Programming 例

例1 stream merge

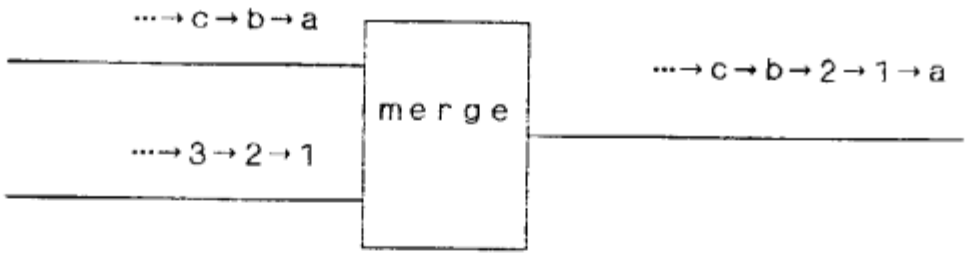
2本のdata stream をnon-deterministic にmerge するprogram を考え、streamはlist で表現されるものとしよう。（特にことわらない限り、syntaxはDEC-10 Prolog に従う。）

[a,b,c,d, …]
[1,2,3,4, …]

に対してoutput stream は

[a,1,2,b,3,c, …]

のような各input streamの順序を保存したmerged stream である。どのようにmerge するかはどうでもよく、non-deterministic に行なわれるとする。



Program

merge(X,Y,Z) : X,Yはinput stream, Z はoutput stream(merged stream)。

```
merge([S | X],Y,[S | Z]) :-  ! merge(X,Y,Z).      ・・・・ (1)
merge(X,[S | Y],[S | Z]) :-  ! merge(X,Y,Z).      ・・・・ (2)
```

“ ! ” はcut と同じようにalternative clauseをcut するという意味を持つ。ゴール merge([a,b,c],[1,2,3],Z)が与えられたとき、これをOR-serial で実行すると 解は常に [a,b,c,1,2,3] である。 つまりdeterministic なmerge しか行なわない。 non-deterministic なmerge を実現するためには、or-parallel を導入する必要がある。すなわち（1）と（2）は両方とも並列にゴールとのunifiabilityを調べられ(or-parallel) 、最初に “ ! ” を過ぎたものが選択されるとする。これにより2本のinput streamのnon-de

terministic なmerge が可能となる。merge は共有資源への複数同時アクセスをserialize するために用いることができる重要なプログラムである。

例２　stream

例１で導入したstreamについてさらに説明する。streamは順序を持ったdataの並びで、長さについての制限はない。単なるdataの並びであるlistとの相異は、それの長さがあらかじめ決定できないこと、すなわち時間とともにdataが末尾に追加されて単調に成長するものであるという点にある。

Concurrent Prolog ではstreamを時間とともにinstantiate されていくd-listで実現する。この様子を例で示す。

stream	d-list
empty	変数　X
aを生成	X＝［a｜X１］
bを生成	X＝［a，b｜X２］　X１＝［b｜X２］
cを生成	X＝［a，b，c｜X３］　X２＝［c｜X３］

（ただし、縦方向は時間を表わす。）

例えば自然数からなるstream　１，２，３，…を生成するプログラムは次のように書ける。

```
generate(N,[N | X]) :- K is N+1,generate(K,X) .
```

ゴールをgenarate(1,X) とすることによってX に１から始まる自然数の系列を得ることができる。ただし、このようなプログラムは決して停止しないのでsequential Prolog では有用ではない。次に示す例ではAND-parallelを導入して、streamがプロセス間通信に使われることを示す。

例３．　プロセス、プロセス間通信、同期

Concurrent Prolog では、論理的にはand であるがoperational には異なる意味を持つ２種類のand 、“，”と“//”がある。これらはそれぞれ“serial and”、“parallel and”と呼ばれ、意味も名前の通りである。　従って、

Ｐ，Ｑ　　と　　Ｐ//Ｑ

は、論理的には同じであるがoperational には異なり、“Ｐ，Ｑ”はＰを解き、その後Ｑを解くのに対し、“Ｐ//Ｑ”の場合はＰとＱを同時に並列に解くことを意味する。
Concurrent Prologでは、parallel andで結ばれたそれぞれをプロセスと呼ぶ。すなわち、Ｐ//Ｑの場合はＰを解く過程、Ｑを解く過程はそれぞれプロセスである。プロセスは複数のプロセスにさらにforkすることがあり得る。

```
generate(1,X) // write_stream(X?)          ・・・・・・・・・・・・・  (1)
```

上は２つの論理的なand で結ばれたゴールを示すと同時に、２つのparallelなプロセスを記述している。write _stream(X) はstreamの各要素を順に書き出すプログラムであり、定義は、次のようになる。

```
write _stream([X | S]) :- write(X),write _stream(S?).          ・・  (2)
```

“？”は、変数の扱い方を示す制御タグであり、？のついている変数（Ｓ？）と付いていない変数（Ｓ）は論理的には同じである。？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』ということであり、（１）で言えば、変数Ｘをinstantiate するのはgenerate　プロセスであり、write _streamプロセスはＸを読み出し専用として扱うことを意味する。この“？”のことを“read only annotation”と呼ぶ。また、？の付けられた変数のことを“read only variable”と呼ぶ。（１）は２つの並列プロセスが共有変数（read only 変数）を用いて通信できることを示している。

Concurrent Prolog では、あるプロセスがread only variableをnon-variable term とunify しない限り先へ進めない場合に、そのプロセスをread only variableが他のプロセスによってinstantiate されるまでwaitさせる。この機能によりプロセスの同期も実現できる。

例4　Pipeline　処理

Streamはdata構造ではなく任意のdata構造（=term ）の動的な流れである。streamは並列プロセス間で通信用、data受渡し用等様々に用いることができる。ここでは１本のdata stream に対し並列に走る複数プロセスの各々がpipelineの各stage に対応してstreamの各成分を到着する度に処理し、結果を次のstage に対応するプロセスへやはりstreamとして出力する例を示す。与えられたlistから同一要素をすべて除去したlistを作るプログラムcompact を考える。それは次のように定義される。（出典 Clark and Gregory）

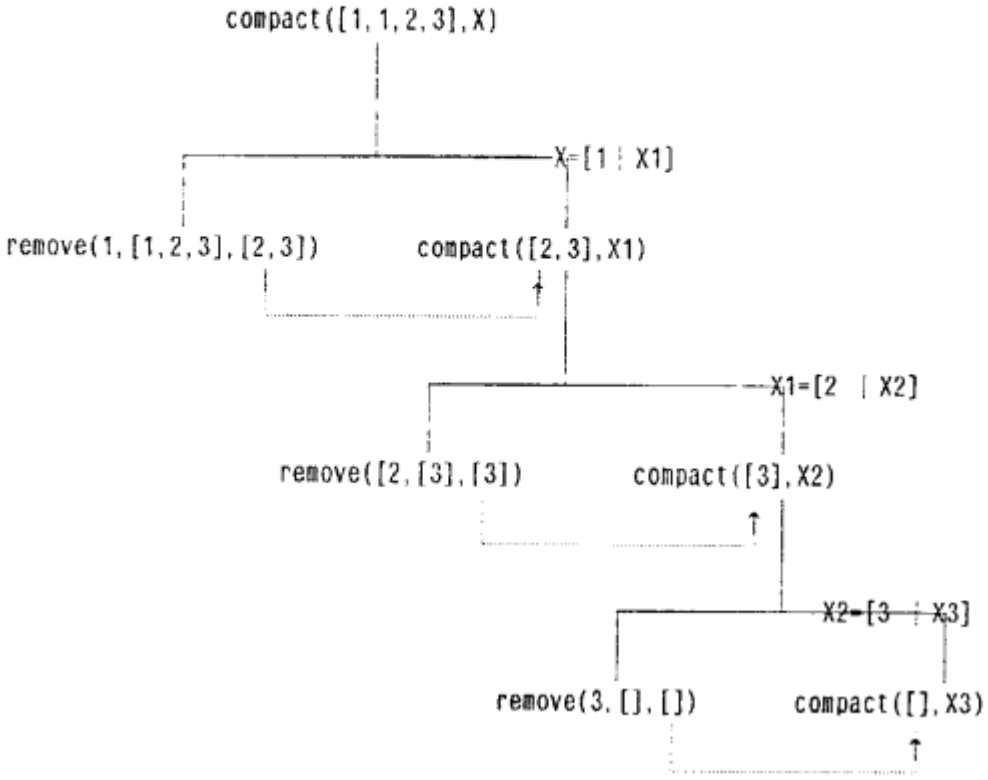
```
compact([],[]).
compact([X | S],[X | Y]) :-
        remove(X,S?,Z)//compact(Z?,Y).                    ・・・・　(*)

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

compact は第１引数に同一要素を除去すべきdata stream を取り、結果をstreamとして第２引数に返す。

動作：input streamのterminal symbol []　が来れば終了。
それ以外のときは、dataが１個到着するのを待ち、dataが来たらそれをoutput streamに返し、その要素と同じものをそれ以後のstreamから除くプロセス（remove）とその結果をさらにcompact にするプロセスとにforkする。

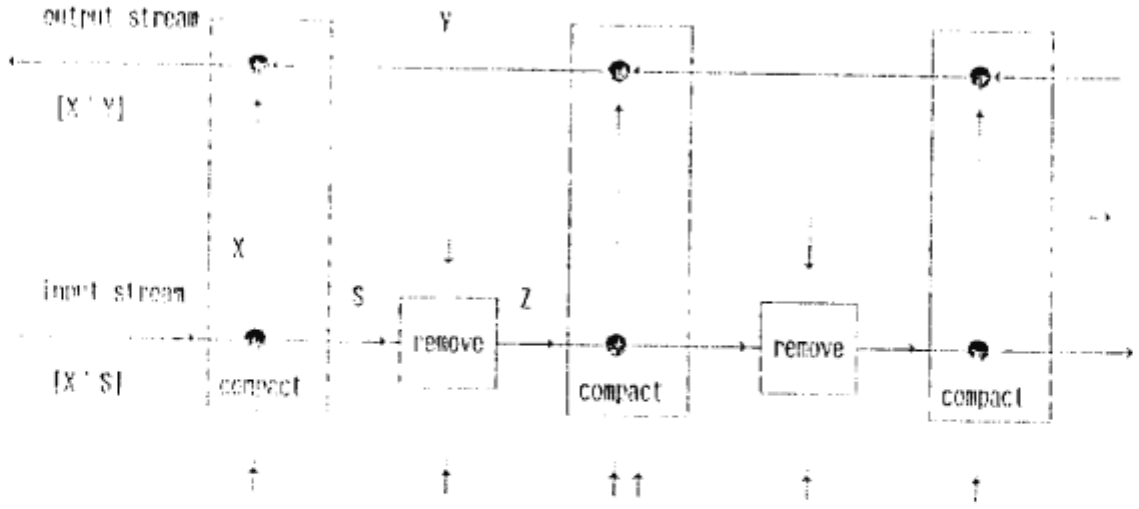
removeは第１引数にstreamの先頭の要素、第２引数に第１引数以後のstreamを取り、第２引数のstreamより第１引数の要素と同じものすべてを除いたものを第３引数に返す。removeはinput streamに対して、filterのような役割を果す。重要な点は、removeはinput streamの要素が１つ決まるとただちにそれを処理し、結果をoutput stream に出す（除去する場合には何もしない）ことにあり、ちょうどpipelineの１つのstage のように機能することである。

ゴールcompact([1,1,2,3],X)を解く際に生成されるプロセスを図で表わすと次のようになる。



（点線はdataの流れを示す）

Dataの流れを中心にしてclause(-) を図に書くと、



compact([X | S],[X | Y]) :- remove(X,S,Z)||compact(Z?,Y).

compact([X | S],[X | Y]) :- remove(X,S,Z)||compact(Z?,Y).

(ただし、実線はstream data 流を表わし、点線はdata要素の流れを表わす。)

図中ではclause(-) が2個登録して示されており、clauseとの対応をblock 図の下に示した。また、線の上あるいは下に付したX や[X | S]等は対応するclause中の変数を表わしている。compact が行なうdata操作については対応するblock 内にdata流の形で示した。removeについては図をわかりやすくするため内部の処理の様子は省いた。図から明らかなように、全体がstreamを非同期式のpipelineのように処理している。

3. Concurrent Prolog 概略

Concurrent Prolog はClark 及びGregory のRelational Language を基に拡張された論理型言語でShapiro により設計された。この節ではその言語仕様について述べる。（詳細についてはICOT Technical Report を参照）

Concurrent Prolog はsequential Prologaを完全にsubsetとして含み，同時にまた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。また、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。

3．1 Syntax

［プログラム］ Concurrent Prolog ではプログラムはclauseの並びとして表わされる。clauseにはguarded clauseとnon-guarded clauseの２種類がある。

［Guarded Clause］ Guarded clauseは次に示すように常に右辺に“｜”をもつclauseである。

A :- G | B.

G及びBは、論理的にand で結ばれたpredicate の並びであり、それぞれ、Guards、Goals と呼ばれる。“｜”はcut symbolの拡張概念である。

［non-Guarded Clause］ “｜”を含まないclause。

［And］ 論理的and は次の２種類のoperational には異なった意味をもつand で表わされる。

sequential and	“，”
parallel and	“//”

名前から明らかなように、“，”は逐次的に実行されるべきand を示し、“//”は並列に実行されるべきand を意味する。

［Or］ 論理的orには２種類のoperational には異なった意味を持つparallel or とsequential or がある。これらはゴールにprefix operator “＝”の形で付加され、ゴー

ルを解く際のモードを指示する。

sequential or モード　　“P”
parallel or モード　　“=P”

ただし、ここでPは1つのゴールを意味する。sequntial orモードではゴールPとunify可能なclauseが逐次的に調べられ、parallel or モードではそれらは並列に調べられる。これについては後でまた述べる。

[Read Only Annotation]　Read only annotation　“？”　は変数に付加することのできる制御情報であり、“X？”のように書く。“？”の付いた変数は変数以外のものとunify してはならないことを示す。Annotation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annotationを付加したプロセスは、この変数をinstantiate できなくなり、他のプロセス（annotationをもたない）がそれをinstantiate するまで待つことになる。（これについては、3．2で再び述べる）。

Read only annotationの付いた変数とunify した変数はこの性質を自動的に引継ぐ。またannotationはoperatorではない。従って、同一clauseにXとX？が出現すれば、これらはunification に対する制御情報を除けば論理的には同一のものである。また、X？についてXがnon-variable term にinstantiate されれば、このannotationも自動的に消滅する。

3．2　Reduction

ゴールが与えられたとき、それがどのようにしてサブゴールに展開されるかについて述べる。上述のように各ゴールは“=”operatorがあるかないかによりparallel or モードで実行されるかserail or モードで実行されるかの指示がsyntactic になされている。

[parallel or　モード]

Concurrent Prolog ではparallel or で解かれるべきゴールのプログラム（ゴールと同じpredicate 名を持つclauseの集合）は常にguarded clauseのみからなってなくてはならない。もし“|”を節に含まないclause、すなわちnon-guarded clauseがあった場合は右辺の一番左に“|”があるものとして処理される。

今、ゴール＝Ａがあるとする。また、プログラムとして次のclauseがあるとする。

```
A1 :- G1  | B1.
A2 :- G2  | B2.
      .
      .
      .
An :- Gn  | Bn.
```

Giは空であってもよい。ゴールＡに対してＡ１，…，Ａｎは次の３つに分類される。

① candidate.　　Ai :- Gi | Bi.
Read only annotationの付いた変数にnon variable term をunify することなしに、ＡとＡｉがunify し、かつ、Ｇｉが解くことができる場合。

② suspended.　　Aj :- Gj | Bj.
ＡとＡｊがread only variableにnon variable term をunify することを除けばunify し、Ｇｊを解くことができる場合。

③ fail　　Ak :- Gk | Bk.
それ以外。

ゴールＡは、それが１つ以上のcandidate clauseを持てば、その中の１つを選択し、（それを　Ai :- Gi | Biとすると）　Biにreduceされる。このときの選択のメカニズムはparallel　モードでは、各clauseは並列に調べられ、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、non deterministic な処理が可能となる。一度ゴールＡがreduceされると、他のalternative clauseのチェックは消去される。この意味で“｜”symbolはcut symbolとして機能する。

ゴールＡがcandidate を全然持たず、かつ少なくとも１つsuspended clauseを持つ場合は、このゴールＡは少なくとも１つのcandidate が見つかるか、あるいは完全にfailするまでsuspend される。

このreduction の途中で起きるいかなる変数のbinding も、“｜”を過ぎて他の可能性が除去されるまでは確定しない。従って、read only annotationのついていない共有変数については、仮にそれがreduction の途中でnon variable term にinstantiate 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。

[serial or モード]

通常のゴールAはDEC-10 Prolog と同じようにserial or モードで展開される。ゴールAに対するプログラムはparallel or と異なりguarded clause, non-guarded clause の両方を含んでいてよい。従ってclauseは“|”を含まないときはnon-guarded clauseとして処理される。“|”の意味はDEC-10 Prolog でのcut symbolと完全に同じである。ただし、この場合においてもread only 変数に対するunification の制限についてはparallel or モードと同じである。

このモードでの実行はclauseが必ずしも“|”を含まないためdeep backtrackが可能であり、変数のbinding についてもbacktrack により何度も変更されることがありうる。一般に複数プロセスで共有されている変数に対してそれをinstantiate するプロセスが共有変数の値を変更するようなbacktrack を起すとそれを参照しているプロセスもbacktrackしなければならなくなり、いわゆるdistributed backtrack が生じる。Distributed backtracking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parallel or モードの場合と同様にbacktrack によりbinding 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。

Commitmentという概念はdeep backtracking を行なうようなプロセスを含む並列プロセスの処理においてdistributed backtracking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『プロセスがある変数のあるinstantiation 以外にはsuccessの可能性が全くないときに、そのプロセスのsuccess あるいはfailをそのinstantiationに委ねる（commitする）』というところからきている。Commitされた変数のinstantiation 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instantiation でもプロセスがfailした場合は他の並列に走るプロセスも同時にfailするためdistributed backtracking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。処理系としては共有変数については常に他のinstantiation の可能性を調べておき、それがなくなった時点ですぐに他のプロセスに変数のinstantiation を公開するという機構が必要になる。

4. CPSI　　Concurrent Prolog Subset Interpreter

この節ではShapiro がPrologで作成した Concurrent Prolog のSubset（以下CPS と呼ぶ）のInterpreter について、前節で述べたものとの相異を中心に、構文について説明する。

① CPS はPrologの上に作成されているためsequential Prologを一応subsetとして含んでいる。しかし、これは以下に述べるように非常に制限された形であって本来のConcurrent Prolog のように融合されたものではない。

- CPS ではclauseはすべてguarded clauseとみなされる。従って“|”を陽に含まないclauseについては右辺の一番左に“|”があるものとして処理される。（この意味でdeep backtracking はsupport されない）
- sequential Prolog とのつなぎはCPS のプログラムからsequential Prologで書かれたプログラムを呼出す形で提供されている。これには３種類の方法がある。１つはsystem predicate 'call' を用いる方法。２番目はsystemという名のpredicate であらかじめそのプログラム名をassertしておく方法。３番目はそのプログラムをcompile しておく方法である。これらによってsequential Prologで書かれたプログラムをCPS で書かれたプログラムの中から呼出すことができる。実際には制御がProlog インタープリタに渡される。このため、CPS よりPrologで書かれたプログラムを呼出すと、そのプログラムが完全に終了するまでCPS のインタープリタに制御が返らないことに注意。例えばappendというプログラムをCPS の中から呼出すにはguarded clauseの右辺において call(append([a,b],[c,d],X)) とするか、あらかじめ system(append(_, _, _))というassertion を行なっておくか、あるいはcompile しておけば良い。これらのプログラムはProlog インタープリタで処理されるので当然deep backtracking が可能であるがcommitmentのメカニズムは提供されていない。効率上はPrologで書ける部分はPrologで書いてcompile しておくのが良い。

② Guarded Clause
基本的なsyntaxはDEC-10 Prolog と同じである。ただし“,”はすべてparallel andを意味しsequential andはない。cut に相当する“!”はvertical bar で表わす（両脇を“.”でくくる必要はない。“P. !. Q”ではなく“P | Q”）。

③ Read Only Annotation

Xを変数とするとX？と書かれる。

④ System Predicate

DEC-10 Prolog のSystem Predicateがすべて使える。これらはCPSIより Prolog interpreter に渡されてそこで処理される。

⑤ OR-serial

Clauseの選択はPrologと同じ順序で行なわれ、parallel or はsupport されない。ただし、前述のようにclauseはすべてguarded clauseとみなされる。

⑤ 注意

Read only annotation“？”は論理的にはそれを付加された変数がinstantiate されたときに消えるものであるが、現在のimplementationではこれをPrologの postfix operatorとして実現しているため変数がinstantiate されても“？”はそのまま残る。例えば、X？についてXがfoo にinstantiate されたときX？はfoo ？になる。また、X？がYとunify し、このYがY？として使われたりする場合、Xがfoo にinstantiate されるとY？は(foo？)？になる。
CPSIが使うunification はこの余分な“？”をあたかも存在しないかのように透明なものとして扱うように拡張されているが、Prologのunification やsystem predicate は“？”を単純にoperatorとして扱ってしまう。このため、③で述べた場合のようにCPSIの中からProlog interpreterを呼び出すときに渡すdataが“？”を含んでいるとこの余分な“？”のためにfailすることがある。例として次のmemberというプログラムを考えよう。

```
member(X,[X | Y]).
member(X,[Y | R]) :- member(X,R).
```

ゴールmember(a?,[a,b,c])を与えたとき、上のプログラムをCPSIで実行すればtrueとなるがPrologで実行すると余分な“？”のためfailする。またsystem predicate write の場合、write(a?) は常にa？を書き出す。
以上のようにCPSIとPrologの世界との間では“？”の扱いについてギャップがあるため注意が必要である。CPSIではこの余分な“？”を除去するためのpredicate waitを用意している。
waitは第１引数として？を付加されているかもしれないnon variable term を取り、第２引数に？のとれたtermを返す。

```
wait(((foo)?)?)?,foo).
```

上の例ではterm foo に付いた３つの余分な？を取除いている。waitはCPSIの中で使われる場合、『第１引数が変数のときにそれがinstantiate されるまで待ちinstantiate されたら第２引数にその値（“？”があればそれを除いた値）が返される。』という別の（名前からすれば本来の）機能を持っている。

以下に第２節で示したプログラム例をCPS で書き出したものを掲げる。

例１　stream merge

```
merge([S | X],Y,[S | Z]) :- | merge(X,Y?,Z).
merge(X,[S | Y],[S | Z]) :- | merge(X?,Y,Z).
```

第２節で示したプログラムは？を用いていない。従ってstreamにdataが来ないときにwaitする機能はない。

例２　stream

```
generate(N,[N | X]) :- K is N+1 | generate(K,X).
```

例３

（１）のゴールを解くには次のようにする。

```
solve((generate(1,X),write_stream(X?))).
```

（２）は例えば次のように書くこともできる。

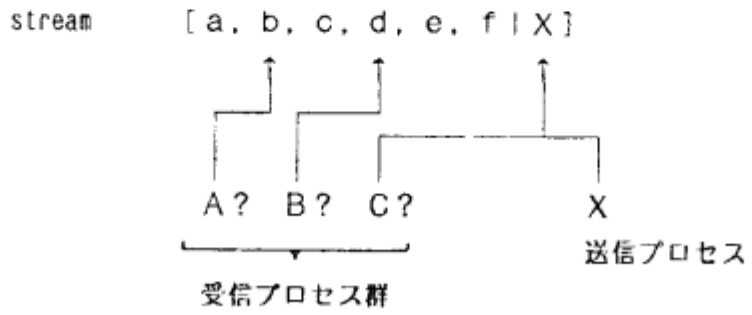
```
write _stream([X | S]) :-call((write(X),nl)) |
                        write _stream(S?).
```

例4　pipeline 処理

```
compact([],[]).
compact([X | S],[X | Y]) :- remove(X,S?,Z) ,write_stream(Z?,Y).

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

5. Stream Oriented Communication

Stream についてはすでに例を用いて説明した。ここではその概念をまとめておく。Streamは、d-listで実現される。Streamによるcommunication は、複数プロセスがこのd-listを共有すること、すなわち、d-listの中へのpointer(d-listの中をpoint する変数)を持つことで行なわれる。この場合、送信プロセスがもつpointer は、常にd-listのtailを指し、受信プロセスはこれより手前（headに近い方）を指すread only variableを持つ。



送信プロセスのポインターは送信される新しいdataの書き込まれる位置を指し、受信プロセスのポインターは次に読み出す（受信する）dataの位置を指している。各プロセスは送信・受信を行なうたびにpointer を１つ後（tail方向）へずらす。受信プロセスの持つpointer は決して送信プロセスのpointer を越えてtail方向へ移動することはない。これは受信プロセスのpointer 変数をread only variableにすることによって保証される。送信プロセスのポインターと受信プロセスのポインターとの間にあるdataはまだ読み出されていないdataを表わしている。これは丁度受信buffer中にあってまだ読み出されていないdataに対応する。送信プロセスのポインターと受信プロセスのポインターはいくら離れていてもかまわない。従ってこれはunbounded buffer を持つ通信を行なっていることに相当する。

送信プロセスが複数個ある場合にはmerge を用いるのが普通である。また、あるプロセスが１つのstreamを送信・受信双方に使うことも可能である。しかし、このような場合にはread only annotationは使えない（受信専用を意味するから）ため、プログラマは充分注意してプログラムを作る必要がある。

付録　実験システムの使い方

この節ではShapiro によりDEC2060 上にPrologでimplement されたシステムCPSIの使い方について解説する。

1982年12月16日現在　CPSIに関連するfileはすべてUS2:<CPS> に入っている。最初にこれらのfileの説明を行なう。

[Information about files]

㊀ Important files

- CPSI　インタープリタ
- CPS　CPSIで書かれたいくつかのプログラム例を含む
- DSUTIL　CPSIが使うユーティリティfile
- SYSTEM.DEF　前上
- CPSI.LOG　サンプルlog file

㊁ Sample program written in Concurrent Prolog

- AIRLN　航空機座席予約システム
- CC　finding connected component in a graph
- RMS　Reduction Machine のシミュレータ（window なし）
- RMSW　Reduction Machine のシミュレータ（window あり）
- DS　Disk arm scheduler
- WM　Multi-window User Fork
- WINLIB　WMのユーティリティ　(written in Prolog)
- WMLIB2　WMのユーティリティ　(written in Prolog)

[How to run CPS]

1. load the interpreter

```
?- [cpsi].              % CPSIをsourceでload
```

あるいは

```
?- [comp].              % CPSIをloadしてcompile する。若干のエラ
                        % ー・メッセージが出るが気にする必要はな
                        % い。
```

2. load the CPS if necessary

```
?- [cps].
```

3. load your file

```
?- [xxx].               % xxx はユーザのfileの名前
```

4. start to solve the program

```
?- solve(<goals>).      % 例えば、solve(p(X)) や
                        % solve((p(X),q(X)))など。
                        % solve は1引数であることに注意。
```

[Scheduling]

複数プロセスの実行のschedulingのモードとして２種類が用意されている。

depth _first　：suspend するまで１つのプロセスを連続して処理しsuspend したら別のプロセスの処理に移る。
breadth _first：各プロセスを１reduction ずつ順々に行なう。

モードの変更は次のように行なう。

set(smode,breadth _first).

default ではdepth _first になっている。

[トレース]

1. To set trace

実行をトレースするにあたっていろんなスイッチが設定ができる。

☒ スイッチ

reduction(_)	reduction をトレース
suspension(_)	ゴールのsuspensionをトレース
system(_)	system predicateへのcallをトレース
solve(_)	
solved(_)	
call(_)	procedure callをトレース
try _clause(_)	clauseの選択をトレース
unify(_)	unification をトレース

スイッチのセットは次のようにして行なう。

```
?- set(traceset,{list of switches}).
        %  例  set(traceset,[call( _),reduction(_)]).
```

2. To set tracing mode off

```
?- set(traceset,[]).        %  トレース・スイッチの設定をclear する
```

あるいは

```
?- set(trace,off).          %  トレース・スイッチの設定を保存する
```

3. To set tracing mode on with current trace switches

```
?- set(trace,on).
```

4. Default スイッチ設定

reduction(_) 及び suspension(_)

[Demonstration System]

現在windowを用いた２つのdemonstration system、Multi-window User Fork 及び Reduction Machine Simulator がある。それぞれは以下のコマンドにより起動される。

1. Reduction Machine Simulator with Multiwindow

```
@prolog                 %  run prolog
?- ['rmsw.dem'].        %  load files
?- go.                                  %  start
```

2. Multi-window User Fork

```
@prolog                 %  start prolog
?- ['wm.dem'].          %  load files
?- go.                                  %  start
```