TM-0071

Concurrent Prologインタプリタの シーケンシャル・インプリメンテーション

宮崎 敏彦

August, 1984

©ICOT, 1984

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

Concurrent Prolog インタプリタのシーケンシャル・インプリメンテーション

（１）　計算モデル

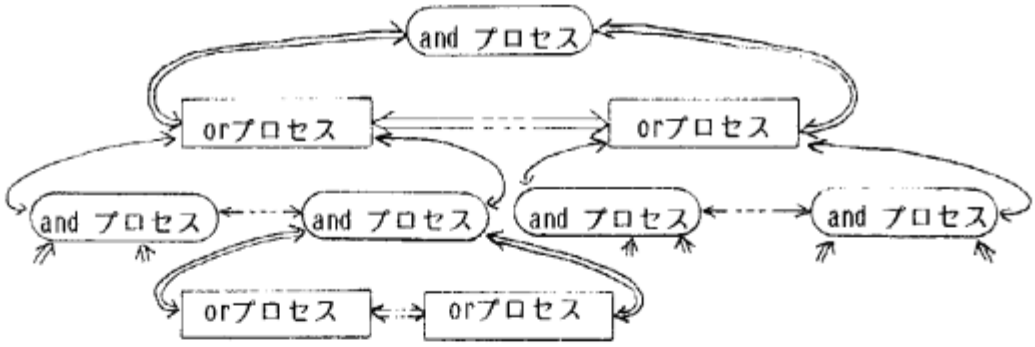
Concurrent Prolog （以下単にCPと呼ぶ）の計算過程はPrologと同様、and-or木で表現することができる。

CPにおける各goalはand nodeに対応し、あるgoalに対応する幾つかのclauseはそれぞれ or node で表される。ここではこのand nodeとor node をand プロセス、orプロセスとよぶ。

図．１は、あるgoalを解く実行過程の一場面を表したものである。 ⟷ でつながれるループをand ループと呼び ⟺ でつながったループをorループと呼ぶ。

and ループは一群のand プロセスの並びとそれらの親であるorプロセスよりなる。and ループに含まれるand プロセスがひとつでも失敗した場合はそのand ループ全体の失敗である。orループはそのorループに含まれるorプロセスのひとつでも成功すれば全体が成功し親のand プロセスが成功となる。逆に、失敗したorプロセスはループから除去される。

and ループ上にあるすべてのand プロセスが成功するとcommitされて、親のorプロセスは、そのbodyパートをand プロセスとして展開する。



図．１　計算木

orプロセス生成の最も単純なタイミングは、goalの実行時，つまりand プロセスの実行時である。しかし図．２に示す様なプログラムの様に１番目のclauseがhead unificationをおえた後すぐにcommitする場合、他のorプロセスを生成するのは明らかに無駄であろう。

```
  goal: p(X)          /*  X は未定義  */
clause: p(a) :- | body1.
        p(X) :- check(X) | body2.
```

図．２

そこで、以下のように改良する。

① まずgoalと各clauseのhead unificationと、単純なguard （算術比較等）をsequentialに行なう。（単純なguard のことをimmediate guard と呼ぶ）

② ①の後、それぞれのclauseについて

a) failであれば、そのclauseは無視する。

b) suspend したら、どのへんすうでsuspend したか記録してundoする。

c) succeed であり、

c.1) immediate でないguard が残っていれば、そのclauseを記録してundoする。

c.2) commitに到達したら、今までの記録を破棄し、当該clauseのbodyパートのgoalに対応するand プロセスを生成して元の呼び出しもとのand プロセスと置換する。

③ どのclauseもcommitに到達しなかった場合、

a) 記録が空であれば、すべてのcluaseがfailしたわけであり、呼び出しもとのand プロセスがfailとなる。

b) 空でなければ、生き残りclauseに対してorプロセスを生成し、orループを作る。また、各clauseで、

b.1) suspend したものは待っている変数中のsuspend queue に登録。

b.2) guard が残っているものは、残っているguard のgoalに対応するandノードを生成し、and ループを作る。

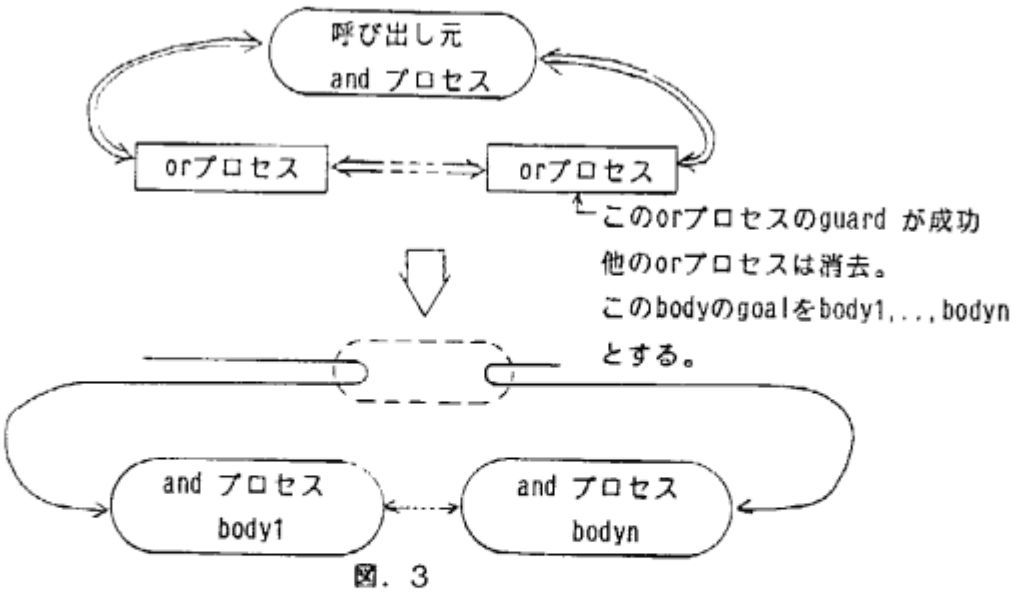


図. 3

and プロセスの生成は、上でも触れたが、以下の二種類の場合がある。

一つは、head unificationおよびimmadiate guard が成功し、immediate でないguard が残っており、他のclauseがcommitを越えていない場合である。この場合は残りのguard に対してand プロセスが生成される。

ふたつめは、guard がsucceed し、body部が空でない場合であり、body部に対するand プロセスが呼び出し元のand プロセスと置換される。図．３に例を示す。

また、上の例でbodyが空の時は単に呼び出し元のand プロセスを消去すれば良い。この際、そのand プロセスが属していたand ループが空になったら、そのループの元締であるorの成功であり、ふたたびそのorプロセスのbody goal をand プロセスとして展開する。つまりこの手続きは再帰的に適用される。

（２）多環境の実現

cpではguard の中で祖先の変数に値を書くことを許している。しかしguard の中で行った束縛情報はcommitを越えるまで公開してはならない。図．４に示す例では、どちらのclauseもheadのunification は成功する。その時 Xの値は一番目のclauseの世界では１であるが、２番目のclauseでは２となりこれらは互いに矛盾するものである。このため各clauseを並列に実行している（or並列）場合それぞれのguard は異なる束縛環境を持たなければならない。

```
  goal: p(X), … | …          /*  X は未定義  */
clause: p(1) :- guard1 |  body1.
        p(2) :- guard2 |  body2.
```

図．４

多環境を実現する為には幾つかの方法が考えられる。

① copyingによる実現

clauseごとに呼び出し側の引き数のcopyを作り、guard の仕事はすべてこのcopyを用いて行なう。そしてcommit時に呼び出し元とcopyとをunify し束縛環境を公開する。

この方式の問題点はcopyによるオーバーヘッドが大きい事と、 guardの実行には必要でないものまでcopyしてしまう場合があることである。

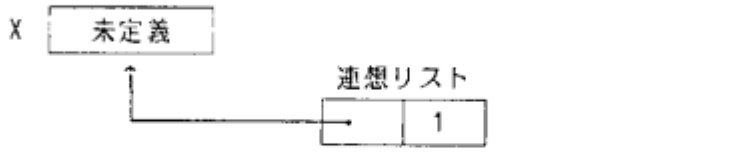
② deep binding による実現

祖先の変数に対する束縛情報は自分のguard 内で連想リストとして保存する。

commit時には保存している連想リスト中の束縛情報とをunify する。

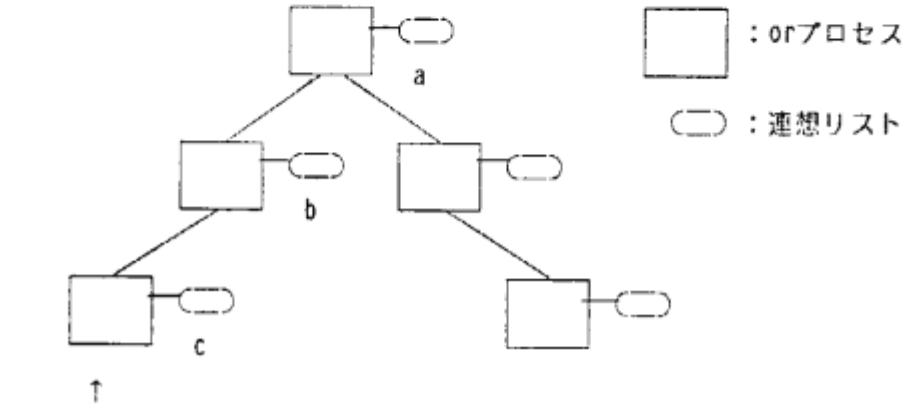
この方式の問題点は束縛環境が各guard （つまりorプロセス）に分散されている為、guard の呼び出しが深くなった場合に変数の値を得る為に祖先のguard （orプロセス）にある連想リストを探索しなければならないことである。

```
goal: p(X)… | …                    /*  X は未定義  */
clause: p(1) :- guard  |  body.
```



guard 内での束縛情報
対応する変数の値セルを指す

図．５　deep binding(1)



この世界では環境は a,b,cの
連想リストを合わせたものになる。

図．６　deep binding(2)

③ shallow bindingによる実現

祖先の変数に対する束縛情報はそのまま祖先の変数セルに書き込み、元の値を自分のguard 内に保存しておく。そして並列に実行されるべき他の環境の異なるプロセスに移る際、値を元に戻すわけである。

この方式の問題点は、他のプロセスに移る際、context switching を必要とすることである。しかしこのオーバーヘッドはschedulingを工夫することによってある程度カバーすることができる。

ここでは上記三方式のうち、逐次処理を仮定した場合には最も効率が良いと思われるshallow binding を採用し、その実現法を詳しく述べる。

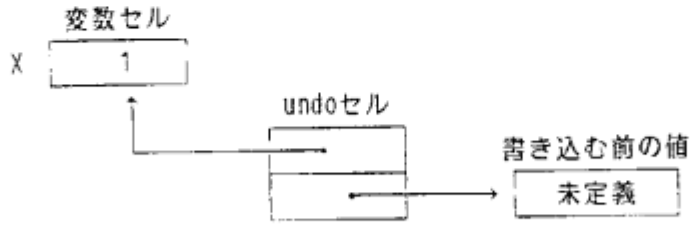
（３） shallow bindingと context switching

先に述べた様にshallow binding では、祖先の変数に対する束縛はそのまま祖先の変数

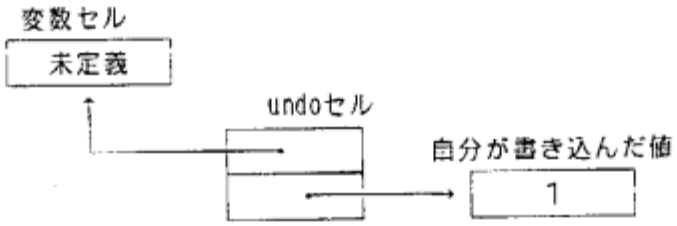
セルに書き、書かれる前の値を自分のguard 内に保存する。

```
   goal: p(X),q(X) …|…        /*  X は未定義  */
clause1: p(1) :- guard | body .
```

図．7



図．8



図．9

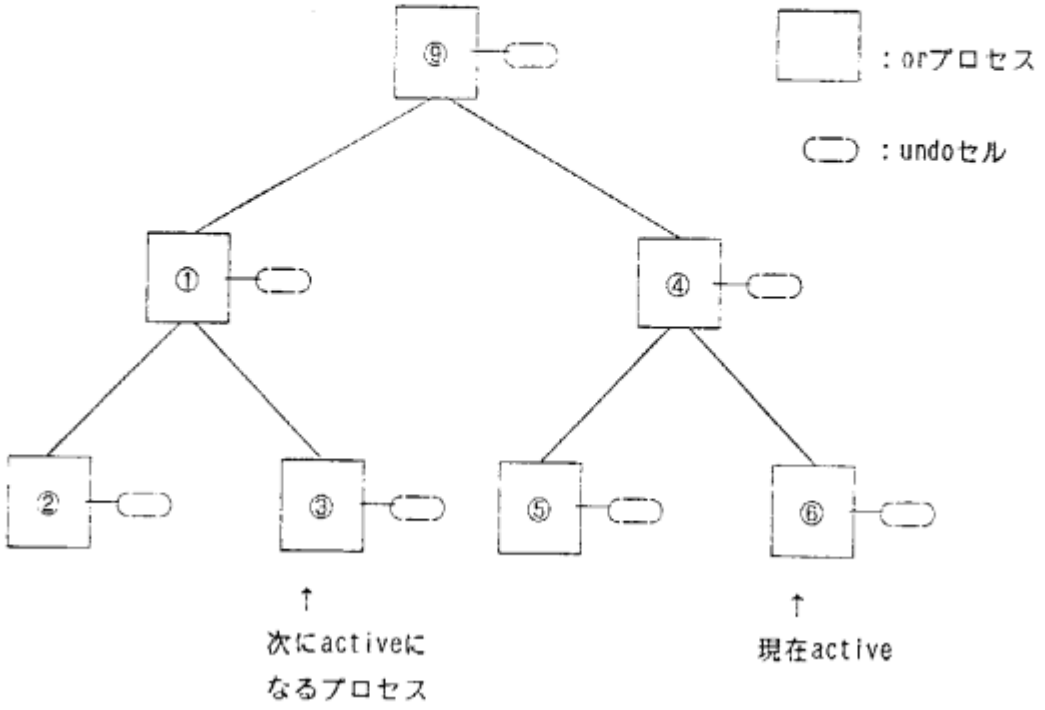
図．7の様なプログラムで goal p とclause1 とのhead unificationでは図．8の様なundoセルと呼ぶペアが作られる。しかし、clause1 はまだcommitを越えていないので、goal p にとってX は未定義でなければならない。このため、goal q を実行する際には上記のundoセルを使って図．9の様な値の入れ換えを行なわなければならない。

そしてふたたびclause1 のguard を実行する際にはundoセルを用いて、自分の束縛環境に戻すわけである。この様な作業をここではcontext-switching と呼ぶ。一般にundoセルは各guard ごとに必要であり、orプロセスがこれを管理する。

図．10の様なor木を考えよう。（各orプロセスはそれぞれundoセルを幾つかずつ持っているものとする。）

orプロセス⑥が現在activeなプロセスとし、次にactiveになるプロセスが③であった場合、context-switching によって保存された③の束縛環境を復元する為には、両者を結ぶ

パス上にある⑥，④，①，③のundoセルを用いれば良い。（⑨のundoセルは他のorプロセスがすべて⑨プロセスの子孫であるので使われない。）



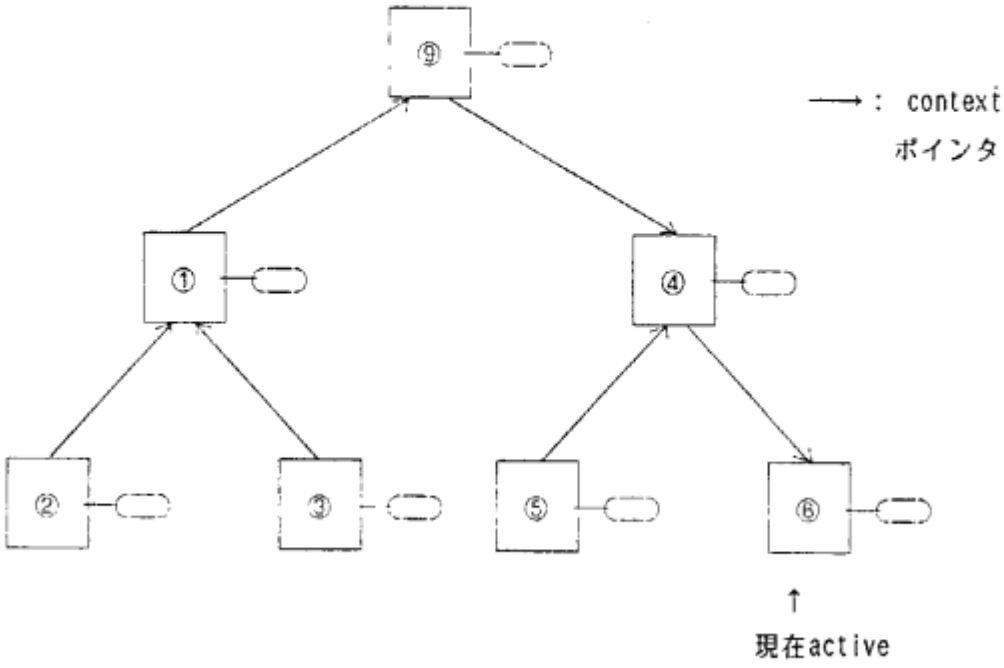
図．10

ここで注意しなければならない事は、context-switching を行なうためには現在activeなプロセスと次にactiveになるプロセスのand-or木上の関係を高速に認識できなければならない事である。そこで、ここでは各orプロセス間にcontext ポインタと呼ぶポインタを張り、どのプロセスからも常に現在activeなorプロセスに辿り着ける様にする。contextポインタとは自分の直接の祖先か直接の子孫を指すポインタであり、子孫を指す場合はそのいずれかの子孫がactiveである場合である。図．11に例を示す。

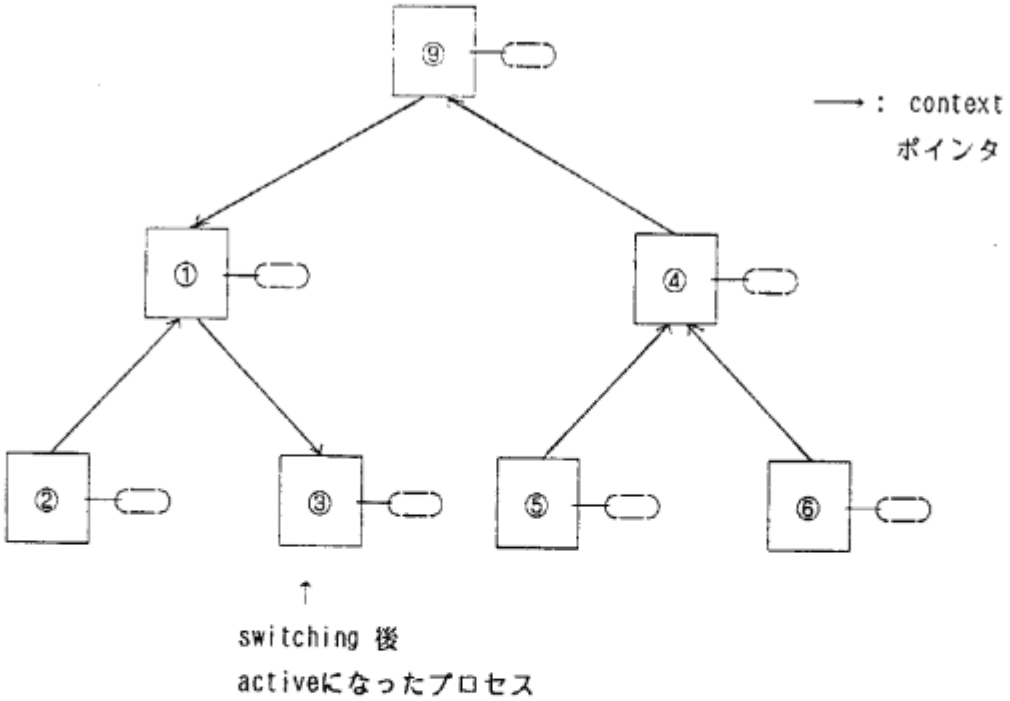
これによって、context-switching は以下の様に表わせる。

① 次にactiveとなるorプロセスからcontext ポインタを辿り、現在activeなプロセスまで行く。このとき辿ったcontext ポインタの向きを逆にする。
② 今辿ったポインタを逆に辿りながら、そのパス上にあるorプロセスのundoセルを用いて値を入れ換える。
③ 自分自身に辿り着けば自分自身のcontext ポインタをnil にする。（activeなプロセスのcontext ポインタはnil とする。）

図．11の例でorプロセス③へのcontext-switching 終了後の状態を図．12に示す。



図．11



図．12

（４） Data Objects

本インタプリタで取り扱うデータはすべてタグ部と値部のペアで表わされる。タグはその値部のデータタイプを表わすものである。以下にデータタイプの一覧を示す。

a) undefined
変数の値がまだ未定義であることを意味する。

b) atom
atomを示す。

c) integer
整数を示す。

d) list
リストを示す。プログラムを表わすコード中か、molecule中に現われる。

e) vector
ベクタを示す。プログラムを表わすコード中か、molecule中に現われる。

f) molecule
クローズ中に現われる構造体(compound term) を表現するためのmoleculeへのポインタを意味する。

g) reference
データ中のリンクを作り出すのに用いる。

h) read-only-reference
データ中のリンクを作り出すのに用いる。ただし、その先のデータが読み出し専用であることを表わす。

i) local-variable,local-reference,(global-variable,global-reference)
プログラムのコード中に現われる。local(global) 変数であることを示す。

j) read-only-local-variable,read-only-local-reference
(read-only-global-variable,read-only-global-reference)
プログラムのコード中に現われる。read-only-annotationが付いたlocal(global) 変数であることを示す。

k) undefined-with-suspension-queue
suspension-recordeへのポインタを表わす。論理的には未定義の意味。
suspension-recordeはsuspension-queueのtop とtailを指すポインタを持つ。

（５） DereferenceとHead unification

ある変数が他の変数とunify した場合、いずれかから他方に向ってreference ポインタ（あるいはread-only-reference ポインタ）が張られる。一般にunify した変数は幾つかのreference ポインタ（あるいはread-only-reference ポインタ）を通して他の変数を参

照しているか、すでに具体的な値を持っているかのいずれかである。このとき参照ポインタを辿って変数の値を得ることをdereferencc と言う。
dereference によって得られる値は未定義かterm (atom, integer, list, vector) である。
図. 13にdereference のアルゴリズムを示す。

```
F : read-only-reference-flag

dereference(VarType,VarCell)
  begin:
    if ( VarType = read-only-variable ) then F:=on else F:=off;
    return( do-deref(VarCell) );
  end;

do-deref(VarCell)
  begin:
    case( tag(VarCell) )
      ref:   do-deref( value(VarCell) );
      roref: F:=on; do-deref( value(VarCell) );
      undef: return( VarCell );
      susp:  return( VarCell );
      term:  return( VarCell );
  end;
```

図. 13 dereference

ここで F (read-only-flag) がonのときは参照ポインタのchain の中にread-only-reference が出現したことを表わす。

head unificationの処理の概要を表. 1に示す。
ここで、 A or B は基本的にはどちらでも良いことを表わす。(local 変数同士のunification はゴミ集めを考えた場合注意する必要あり。)
SUSPEND はこのunification がsuspend することを意味する。この時suspend の原因となった変数が undefならば、それをsuspというデータタイプに変え、suspend recordを作る。suspend したgoalはそのsuspend recordが指すsuspend queue に登録される。変数がsuspであれば、すでにあるsuspend queue の最後にそのgoalをつなげる。

表．1 unification

	undef	susp	undef^{ro}	susp^{ro}	term
undef	C:=REF(P) or P:=REF(C)	C:=REF(P) or P:=REF(C)	C:=ROREF(P)	C:=ROREF(P)	C←P
susp	C:=REF(P) or P:=REF(C)	C:=REF(P) or P:=REF(C)	C:=ROREF(P)	C:=ROREF(P)	C←P
undef^{ro}	P:=RORFF(C)	P:=ROREF(C)	SUSPEND	SUSPEND	SUSPEND
susp^{ro}	P:=ROREF(C)	P:=ROREF(C)	SUSPEND	SUSPEND	SUSPEND
term	P←C	P←C	SUSPEND	SUSPEND	unify(P,C)

図．14の例では、

```
  goal: p(X?)              /*  X は未定義  */
clause: p(1) :- guard | body.
```

図．14

goal p はsuspend し、変数X の持つsuspend queue に登録される。登録されたプロセスはその変数に値が入ることのよってactivateされる。

変数とのunification を以下でもう少し詳しく述べる。

変数とのunification は、shallow binding では図．15の手順で表わすことができる。

read only 変数と通常の変数とのunification は、通常の変数の変数セルから read only変数の変数セルへ、読み出し専用を意味するread-only-reference ポインタを張る。このように、読み出し専用とはその変数セルへの参照ポインタの属性である。

```
X  : 変数セル
X↑: 変数セルの番地

 if  instantiated(X) then
      Xとunify;
 elseif  X は自分のlocal 環境内の変数セル  then
      X↑:=unify する相手;
 else  〔祖先の変数セルのはず〕
     undoセルをallocate;
      X↑と Xの値をundoセルに登録;
     undoセルをundoフレームに登録;
 endif;
```

図. 15　変数とのunification

（６） Commitment

commit操作は、commitしようとしているORプロセスがおこなう。操作の手順は以下のとおり。

①commitしようとしているORプロセスの兄弟のORプロセスを殺す（これをabort と呼ぶ）。

②guard 内での束縛情報を公開する（これをexportと呼ぶ）。

③commitしたORプロセスに対応するclauseのbody部をAND プロセスとして展開し、各プロセスをactive queueに登録する。

もしbody部が空なら、祖父のORプロセスがcommit。

exportは当該ORプロセスが管理しているundoセルすべてに対して、図. 16を行なう。

```
X↑: undoセル中の変数X の変数セルへのポインタ
V  : undoセル中のセーブした値。（変数X の現在のglobalな値）

if   X↑が指す変数セルが親のAND プロセスに属す  then
      現在のX の値とV をunify ;
else  〔もっと祖先の変数セル〕
    祖父であるORプロセスの管理するundoセルとして登録;
endif ;
```

図. 16　export

例を示そう。

```
goal: p(f(A))            /* Aは未定義*/

clause1: p(X):-g(X),… | ….
clause2: q(f(1)):---- | ….
```

図．17

図．17の場合clause2 におけるcommitでは変数A の値が 1であるという情報をclause1 のguard 内に対してのみ公開でき、clause1 のcommitで始めてgoalの世界に対して公開される。つまりclause2 のcommit操作ではA に関するundoセルを持つclaus1に対応するＯＲプロセスに登録する。

commitにおけるunify の仕事はhead unificationと以下の点で異なる。

①suspend queue を持つ変数セルに値が入ったならば、そのエントリをすべてactive queueに登録する。

②commitのunify でsuspend した場合は、当該ORプロセスを対応する変数セルのsuspend queue に登録する。

(7) local 環境の管理

OR並列における各guard 内の束縛環境をlocal 環境と呼ぶ。

図．18の例では、clause1 がcommitした時、A が 1であるという束縛環境はclause1 のguard 内にのみ公開され、goalの世界には公開されない。つまり変数A に対応するundoセルは、

```
goal: p(f(A))      /* Aは未定義 */

clause1: p(X):-q(X),… | ….
clause2: q(f(1)):---- | ….
```

図．18

clause1 に対応するORプロセスに登録される。これに対して、図．19の例の場合clause4 がすでにcommitを越えていれば、clause3 のcommitにおいて、A に対応するundoセルはexport時に消えてしまう。なんとなれば、変数A はgoal1 のlocal 環境に属しているからである。

```
goal1: p(X),q(X), …¦….          /* Xは未定義 */

clause3: p(f(1)):-…¦….
clause4: q(f(A)):-…¦….
```

図. 19

一般には変数セル領域はそのclauseがcommitすると親の変数セル領域に属すようになる。

上の二つの例で、変数A に対するundoセルの取り扱いを区別するためには、その変数セルが所属するlocal 環境を識別できなければならない。この為には各local 環境にA という変数セルがあるか否かを捜せば良いのであるが、それでは非常に無駄が多いことは明らかであろう。そこでここでは各local 環境にそれぞれ一意に決まる番号を付け識別することにする。そしてcommitの際には、そのlocal 環境の番号は親のlocal 環境の番号への参照ポインタに変える。(local 環境番号にアクセスする為に、本インタプリタではundefined というデータタイプを未定義であり、かつその値部がlocal 環境番号を指すポインタである、と解釈する。local 環境番号は変数セルのallocation時に初期設定される。)

図. 20において、①、②、③、④、⑤の変数セル領域は何回かのcommitによって同じlocal 環境に属している。

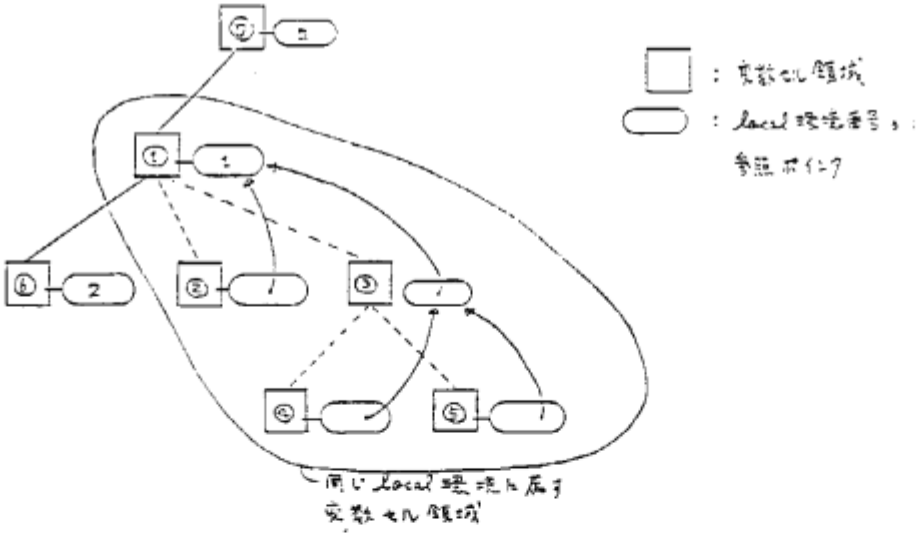


図. 20 local 環境の管理

しかしこのままだとlocal 環境番号を得るためには毎回最悪n 回の参照ポインタを辿らなければならない。このようなオーバーヘッドを避けるために、一度辿った参照パスはその最後の参照パスで付け変えるようにする。

head unificationやexportの際の、“ある変数が、当該and プロセスに属すか?”という判断は、実は“local 環境番号が同じか?”というチェックをすれば良いことになる。

このように、このlocal 環境番号を用いることによって、head unificationやexportの際にundoセルを作る（あるいは登録する）か否かが決定されるのである。

（７） Main Areas

①コード領域

プログラムコードがある領域。プログラム（clause）の内部形式は次節で述べる。

②変数セル領域

論理変数の変数セルがある領域。変数の属性に従ってlocal frame 領域とglobal frame領域がある。各frame はclauseが呼ばれたとき（head unificationの前）に確保される。

a) local frame

local 変数の変数セル領域。固定的に取られる。

deallocateのタイミングは、fail時，commit時及び兄弟の他のorプロセスがcommitしたことによって自分が死ぬ時である。

b) global frame

global変数の変数セル領域。clauseの実行にともなって動的に伸びる。

deallocateのタイミングは、fail時及び兄弟の他のorプロセスがcommitしたことによって自分が死ぬ時である。

③ undo セル領域

先に述べたshallow binding を行なう為のundoセルを保持する領域。guard の実行にともなって動的に伸びる。context switching のときに参照，書き変えが行なわれる。deallocateのタイミングは、fail時，兄弟の他のorプロセスがcommitしたことによって自分が死ぬ時及びcommit時（ただし親のundo frameに登録されないものだけ）である。

④プロセス管理テーブル領域

and プロセスとorプロセスを管理するためのテーブル領域。

allocateのタイミングはプロセス生成時。deallocateのタイミングは、fail時，commit時及び兄弟の他のorプロセスがcommitしたことによって自分が死ぬ時である。

図．21に両管理テーブルを示す。

（８） Clause の内部形式

各clauseはプロセジャ単位に保存され、ほぼ元のclause形式をそのまま残している。

図．22にcaluseの内部形式を示す。

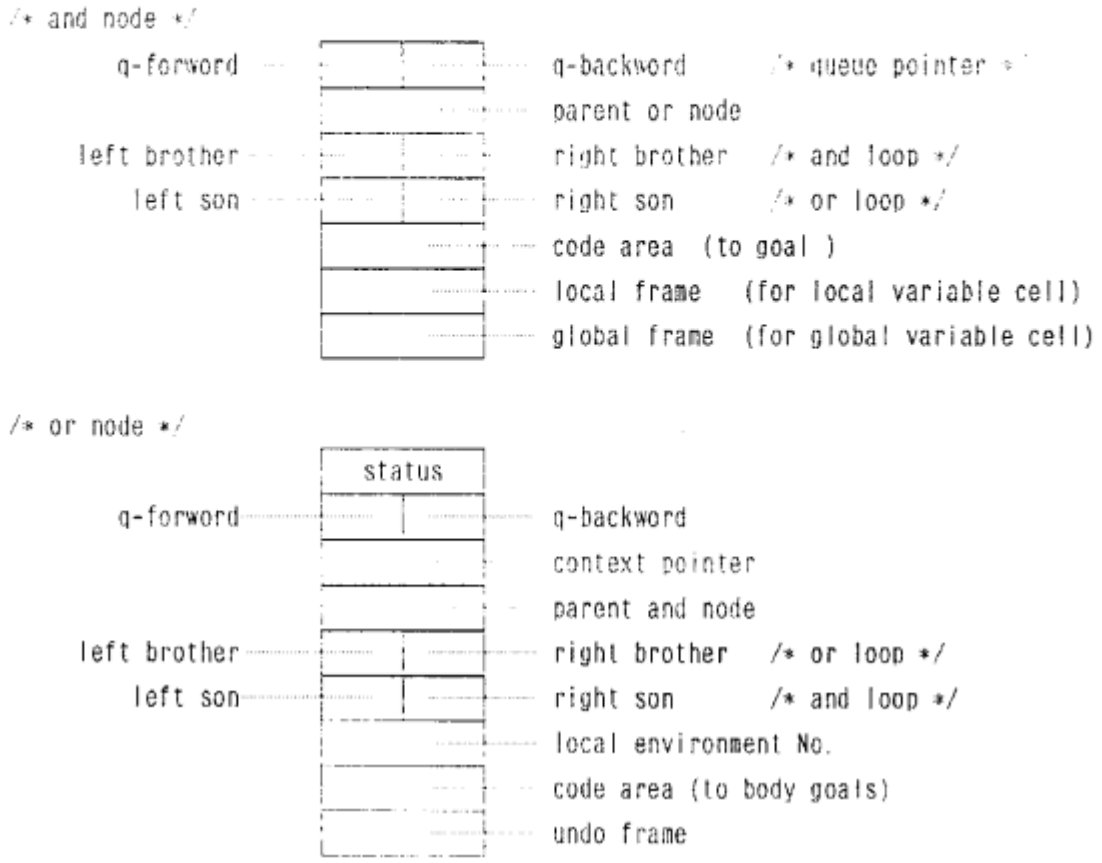


図. 21 プロセス管理テーブル

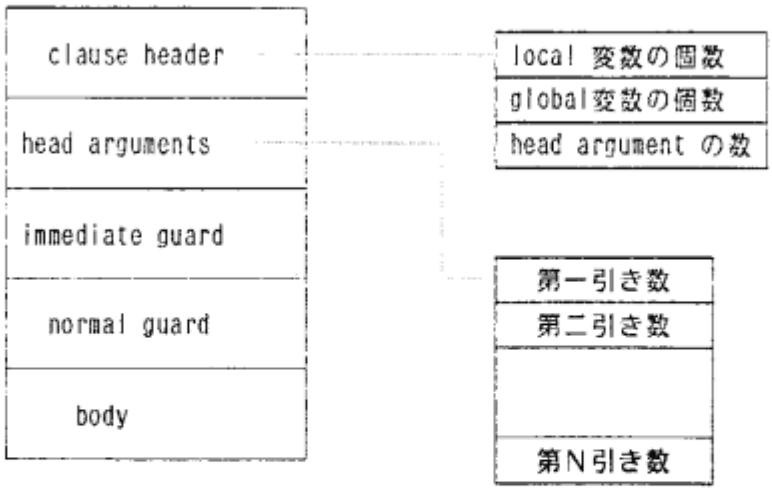


図. 22 Clauseの内部形式

参考文献

(1) E.Y.Shapiro . A Subset of Concurrent Prolog and Its Interpreter
ICOT Technical Report TR-003. (1983)
(2) Warren.D.H.D: Implementing Prolog - compiling predicate logic program
D.A.I Research Report no 39-40(May 1977)
(3) 西川　他：逐次型パーソナル推論マシンΨの設計思想とそのアーキテクチャ
ICOT Technical Report TR-014. (1983)
(4) 古川　他：核言語第一版概念仕様書
ICOT Technical Report TR-054. (1984)