Prologの視覚的計算モデル†

森下真一†† 沼尾雅之††

Prologの背景には宣言的な解釈と手続き的な解釈がある．宣言的な解釈に基づくPrologのプログラムは，仕様のように読みとれる点で重要だが，大規模なプログラムを宣言的に記述することは難しい．しかも近年のPrologの普及に伴い最近は比較的大規模なプログラムが作られるようになり，実際のプログラミング現場では，複雑なバックトラック，カット，副作用等の手続き的側面が，プログラムの作成・デバッグ・分析というプログラミングの過程において困難の原因となっている．本研究では，これらのPrologの手続き的側面を明確に把握できるように，その実行時の動きを視覚的に表現できる計算モデルBPM (Box and Plane Model)を提案する．BPMはゴールに対応する記号ボックスとボックス間の制御の流れを示す矢印によりバックトラックの動きを2次元的に表現する．またボックスにはその内部の実行の動きを表現する平面が対応しており，例えばカットが働いたときの制御の移行およびその有効範囲なども容易に把握できる．また実際にBPMは，Prologのプログラミング環境の一つとして開発されたデバッガPROEDIT 2においてプログラムの実行を表現するため使われている．

1. はじめに

Prologでプログラミングをする際の書き方のスタイルには大きく分けて，宣言的な解釈のできるプログラムを書く場合と，手続き的な解釈からプログラムを作成する場合とがある．宣言的な解釈ができるPrologのプログラムは仕様に近い記述ができる点で重要ではあるが，大規模なプログラムを宣言的に記述することは難しい．また，近年のPrologの普及に伴い最近はPrologで比較的大きなプログラムが作られるようになってきているが，このような実際のプログラミング現場では宣言的な解釈のできるプログラムを書くことよりも，ユーザが手続き的な解釈を念頭において大規模なプログラムを作成するケースが多い．

ここで宣言的解釈と，手続き的解釈の違いを明らかにしておきたい．宣言的解釈の背景にはプログラムをエルブラン領域の巾集合上の写像とみなし，その最小不動点をモデルとする立場がある．この立場は理論的考察をするには欠かせないものだが，そこで仮定されている計算規則は幅優先探索でありPrologの計算規則である深さ優先探索とは異なる．その差異が顕著に現れる例として幅優先探索した場合にはモデル中に存在が保証されている値でも，深さ優先探索で探す場合には節の順番や，節本体のゴールの順序に意味があるために見つけられないことがある．ここにPrologのプログラムを深さ優先探索を考慮して手続き的に解釈する必要性がある．また手続き的解釈のもとに，Prologに組み込まれたカット・or・if-then-else等の述語の意味を最小不動点モデルで説明することはできない．例えばカットを含んだプログラムは最小不動点モデル上では存在しうる値を，枝がりにより捨てる可能性がある[1]．

このようにPrologでプログラミングする際，ユーザはプログラムを手続き的に解釈することが必要不可欠である．ここでいうユーザがプログラムを手続き的に解釈するということは，具体的にはユーザが処理系の動きを理解することにあたる．この場合Prolog処理系のプログラム実行のメカニズムがユーザにとって把握しやすいものであればよいのだが，実際には必ずしも理解しやすいものではない．特にバックトラック・カット等の実行過程や，それらがルールベースの書換えと絡んだ場合の動き等は複雑で，ユーザの描いているプログラムの実行イメージと，処理系によるプログラムの実行には大きなギャップがある．そこで，そのギャップを吸収し処理系によるプログラムの実行を正確に反映し，ユーザの意図したプログラムが処理系にいかに解釈されているかを示すことのできる計算モデルを設計し，ユーザに提供することによりプログラムの作成・デバッグ・分析の作業における従来の負担を軽くすることが必要である．以上の問題意識の下，本研究ではユーザのプログラミング作業を効率化するための計算モデルとして，視覚的計算モデルBPM (Box and Plane Model)を提案する（図1を参照）．

計算の実行過程を視覚化する試みは手続き型言語において数多く提案されている．例えばALGOLの視覚的計算モデルとして提案されたcontour model[2]で

† Visual Computation Model for Prolog by SHIN-ICHI MORISHITA and MASAYUKI NUMAO (Tokyo Research Laboratory, IBM Japan, Ltd.).

†† 日本アイ・ビー・エム(株)東京基礎研究所

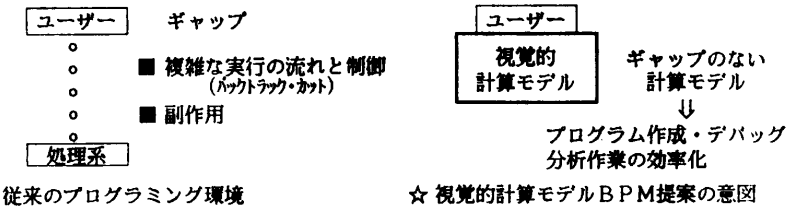


図 1 従来のプログラミング環境とBPMの意図
Fig. 1 Conventional programming environment and intention of BPM.

は，計算進行中の大域変数・局所変数の束縛状態および，手続き呼出しの仕組を視覚的に表現できるようになっている．それに対し本稿で提案する BPM では，Prolog 特有の問題点として次の項目に注目し，その意味付けを行う．

（1） Prolog プログラムの複雑なバックトラックの動き

（2） バックトラックを制御するカット

（3） 実行制御述語 (or, not…etc.)

（4） ルールベースを変更する述語

BPM は，DEC 10 Prolog のデバッガ[3] で用いられているボックスモデルを整備・拡張したものである．以下では，上記の各々の項目についてその問題点と BPM による解決のアプローチを説明する．

まず(1)の問題点は複雑なバックトラックの動きである．従来のボックスモデル[3] と呼ばれるものは，一つのゴールに注目しその実行過程をボックスに矢印を配置して表現したがバックトラックの動きを表現するにはボックス間の制御の流れを表さなくてはならない．その際大事なことは，どの程度の範囲のボックスをひとまとまりの単位として，ボックス間の制御の流れを表現するかということである．BPM では Prolog の変数束縛が節単位であるという性質に注目し，節内のゴールをひとまとまりにし，各ゴールに対応するボックスを2次元的に配置し，ボックス間の制御の流れをボックス間に矢印を結び付けることで，バックトラックを自然に表現する．

次に(2)のカットの意味であるが，従来は探索木の枝がりをする機能として説明されていた[1]．しかしカットが多用される通常のプログラムの動きを理解する場合，各々のカットの有効範囲がどこにあるかを把握したり，カットによる失敗が実行された後にプログラムの実行制御がどこに移るか知りたいときなど，従来の方法では必ずしも分かりやすい説明がなされているとは言い難い．これに対し BPM では，カットの有効範囲を明確にし，カットによる失敗が実行されたときの制御の移り先を明確に表現することができる．

次に項目(3)を解説する．or, not 等の実行制御述語は，高階述語を用いて定義することができるが，通常の処理系ではあらかじめシステム組込み述語として用意されている．ところで，このようなシステム組込み述語と高階述語とでは意味が異なったものになるという問題点がある．特にこれらの述語がカットとともに用いられた場合にこの違いはカットの有効範囲の差異となって顕著に現れる．したがって計算モデルでも，これらのシステム組込みの実行制御述語の表現ができることが必要になってくる．本稿では or 述語を例に，BPM によっていかに実行過程が表現されるかを示す．そしてカットとともに用いられた場合の問題点を考察し解決案を与える．

項目(4)については，ルールベースの変化を正確に把握できるモデル化を行った．

以上，視覚的計算モデル BPM の提案の意図と，モデルの扱おうとする問題点と解決のアプローチを示した．次章では BPM の定義とその表現能力について述べる．

2. BPM (Box and Plane Model) の定義と表現能力

2.1 使用する記号

● ボックス

ヘッドボックス，ゴールボックスの2種類があり，各々点線と実線で囲まれた箱で表現される（図2を参

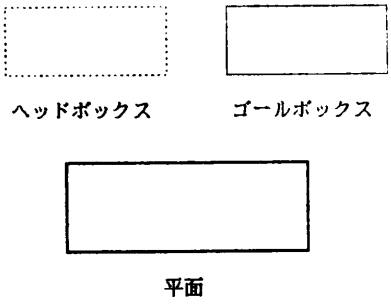


図 2 ヘッドボックス・ゴールボックス・平面の記号
Fig. 2 Symbols of head-box, goal-box and plane.

照).

・平面

平面は，内部にいくつかのボックスを含み，太線で囲まれた箱である（図2を参照).

・矢印

矢印はボックス間の制御の流れおよびゴールボックスと平面間の制御の流れを示す記号で，以下の種類がある.

→ : call/exit, ← : redo/fail,

⇒ : in_call/out_exit, ⇐ : in_redo/out_fail

2.2 記号の意味

まず，Prologのプログラムの表記について，次の記号の約束をする.

a, b, c, ……, g, g1, g2, …	述語名
g1(T1) & g2(T2) & …	節の本体
$X, Y, \cdots$	変数
<- g(T)	ゴール g(T) の実行
T, S, …	項
$\theta, \theta_1, \theta_2, \cdots$	代入，ユニファイヤ
T/X	XへTを代入
g(T), g1(T1), g2(T2), …	ゴール
$\theta_1 \circ \theta_2$	代入の合成

g(T) <- g1(T1) & g2(T2) &・節

2.2.1 ゴールの実行の表現

対象とするゴールを g(T) とする．g(T) の実行過程は，g(T) に対応して生成されるゴールボックスへ，call, exit, redo, fail の矢印を配置して表現する（図3を参照)．各々の矢印の意味は，

call　g(T) に対応するゴールボックスの生成と実行の要求.

exit　g(T) の実行結果 g(T)θ を返す.
ユニファイヤ θ を返すと考えてもよい.

redo　g(T) を再実行することを要求する.

fail　g(T) の実行が失敗したことを示す.

これらの矢印を，配置されたゴールボックスに『属する』と呼ぶことにする.

2.2.2 ゴールボックス内部の表現

ゴールボックス（対象のゴールを g(T) とする）の内部での実行過程を，ボックスと矢印を2次元的に配置して表現するものが，平面である．平面はヘッドボックスとそれに続くゴールボックスから構成される．そしてこれらのボックス間の制御の流れを表現するため矢印がボックスのまわりに配置される.

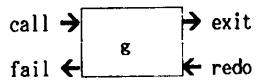


図3 ゴールボックスの制御の流れ
Fig. 3 Control flows of goal-box.

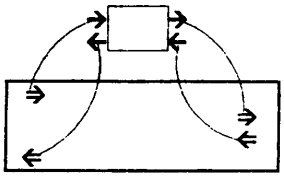


図4 ゴールボックスと平面の矢印の関係
Fig. 4 Relationship between goal-box's arrow and plane's arrow.

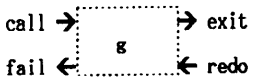


図5 ヘッドボックスの制御の流れ
Fig. 5 Control flows of head-box.

ゴールボックスに属する call(→), exit(→), redo(←), fail(←) の各矢印は，平面では各々，in-call(⇒), out-exit(⇒), in-redo(⇐), out-fail(⇐) によって対応づける．in-call, in-redo は，平面への入口を，out-exit, out-fail は，平面からの出口を示す（図4を参照).

ヘッドボックスではゴール g(T) とユニファイ可能な節をルール・ベースから取り出す過程が表現される．ヘッドボックスにもゴールボックスと同様に call, exit, redo, fail の4種類の矢印があるが（図5を参照)，これらの意味は以下のようになる.

call　g(T) に対応するヘッドボックスを生成し，g(T) にユニファイ可能な節を要求.

exit　g(T) にユニファイ可能な節.
g(T0) <- g1(T1) & …
および，T と T0 のユニファイヤ θ を返す.

redo　g(T) にユニファイ可能な節を新たに要求.

fail　g(T) にユニファイ可能な節が尽きたことを示す.

これらの矢印を，配置されたヘッドボックスに『属する』と呼ぶことにする.

次にゴールボックスに対応する平面の表現を次のプログラムを例に説明する.

```
descendant(X, Y) <- offspring(X, Y).
descendant(X, Z) <-
    offspring(X, Y) & descendant(Y, Z).
offspring(abraham, ishmael).
offspring(abraham, issac).
```

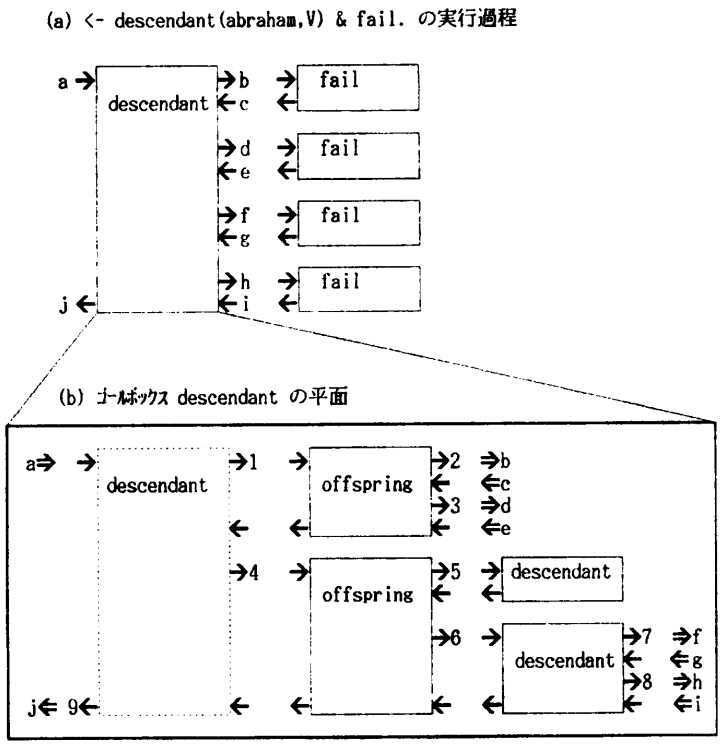


図 6 BPM による実行過程の表現の例
Fig. 6 Example of representation of execution process in BPM.

offspring(issac, esau).

offspring(issac, jacob).

図 6 (a)は，ゴール <- descendant(abraham, V) & fail. の実行過程を BPM で表現したものである．ゴールボックス descendant は a→で生成され初めの実行結果

→b descendant(abraham, V)θ_b, θ_b= {ishmael/V}

を返す．続いて ←c, ←e, ←g の再実行の要求があり各々の実行結果として，

→d descendant(abraham, V)θ_d, θ_d= {issac/V}

→f descendant(abraham, V)θ_f, θ_f= {esau/V}

→h descendant(abraham, V)θ_h, θ_h= {jacob/V}

が返されたのが分かる．最後の再実行の要求 ←i に対しては解の探索に失敗して j←で終わる．

図 6 (b)は，図 6 (a)のゴールボックス descendant に対応する平面である．ゴールボックスの矢印 a→～j←は，各々平面の矢印 a⇒～j⇐ へ置きかえられる．また節を取り出す過程は，ヘッドボックス descendant に属する矢印で表現されている．→1 で descendant の初めての節が返され，→4 で 2 番目の節が返されたことが分かる．

ところで平面中では exit 矢印に返されるユニファイヤを一つ一つ追うことで，ユニフィケーションの過程を調べることができる．→1 で descendant の初めの節

descendant(X, Y) <-
offspring(X, Y).

とユニファイヤ

θ_1= {abraham/X, Y/V}

が返されると，次にユニファイした後の節の本体中の offspring(abraham, Y) に対応するゴールボックスは →1 の右側に配置され，最初の実行結果

offspring(abraham, ishmael)

とユニファイヤ

θ_2= {ishmael/Y}

が →2 に返される．そして ⇒b へと抜け，⇒b には実行結果

descendant(abraham, V)θ_b

が返される．このユニファイヤ

θ_b= {ishmael/V}

は，

$\theta_1 \circ \theta_2$= {abraham/X, ishmael/Y, ishmael/V}

を変数 V へ制限したものである．

次に ⇐c の再実行の要求に対しては，ゴールボックス offspring から次の実行結果

offspring(abraham, issac)

とユニファイヤ

θ_3= {issac/Y}

が→3 に返され ⇒d へ抜け，⇒d には実行結果

descendant(abraham, V)θ_d

が返される．このユニファイヤ

θ_d= {issac/V}

は，

$\theta_1 \circ \theta_3$= {abraham/X, issac/Y, issac/V}

を変数 V へ制限したものである．

続いて ⇐e の再実行の要求に対しては，制御はヘッドボックス descendant まで戻り，→4 で 2 番目の節

descendant(X, Z) <-
offspring(X, Y) & descendant(Y, Z).

と，ユニファイヤ

θ_4= {abraham/X, Z/V}

が返される．そしてユニファイした後の節の本体中のサブゴール offspring(abraham, Y) に対応するゴー

ルボックス offspring から，実行結果

offspring(abraham, $Y)\theta_5$

とユニファイヤ

θ_5= {ishmael/Y}

が返される．次に θ_4 と θ_5 により束縛されたサブゴール descendant(abraham, ishmael) に対応するゴールボックスが呼び出され，失敗したことが分かる．そしてまた後戻りして最終的に ⇒f, ⇒h へと抜ける．ここで ⇒f, ⇒h のユニファイヤ

θ_f= {esau/V}

θ_h= {jacob/V}

は各々，→4, →6, →7, →8 のユニファイヤ

θ_4= {abraham/X, Z/V}

θ_6= {issac/Y}

θ_7= {esau/Z}

θ_8= {jacob/Z}

により合成された

$\theta_4\circ\theta_6\circ\theta_7$= {abraham/$X$, esau/$V$, issac/$Y$, esau/$Z$}

と，

$\theta_4\circ\theta_6\circ\theta_8$= {abraham/$X$, jacob/$V$, issac/$Y$, jacob/$Z$}

の，変数 V への制限である．

2.2.3 BPM によるカットの意味

平面から out-fail 矢印(⇐)を出口として抜け出る場合には2通りある．一つはヘッドボックスがユニファイ可能な節を取り出すのに失敗して fail で抜け，out-fail 矢印で平面から抜け出る場合である．例えば図6(b)では，ヘッドボックス descendant(abraham, V) はユニファイ可能な節を取り出すのに失敗し (9←)，j⇐ から平面を抜け出ている．他の一つは，カットのゴールボックスが fail して平面から抜ける場合である．

すなわち，カットのゴールボックスの意味を，call (→)に対しては exit(→) を返すが，redo(←) に対しては fail(←) を返し，続いて out-fail(⇐) で平面から抜け出ると規定する（図 7 を参照）．また以後カットの有効範囲（スコープ）は平面であるという言い方をする．例としてカットを使って not を以下のように定義した場合の，not の実行過程を図8に挙げておく．

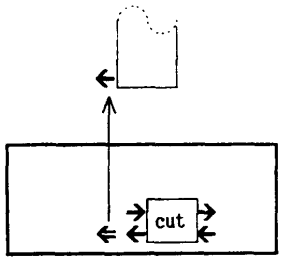


図 7 カットの意味
Fig. 7 Semantics of cut.

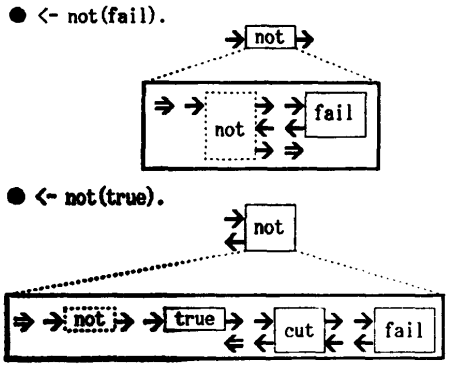


図 8 カットの実行の例
Fig. 8 Example of execution of cut.

not(P) 〈- P & cut & fail.

not(P).

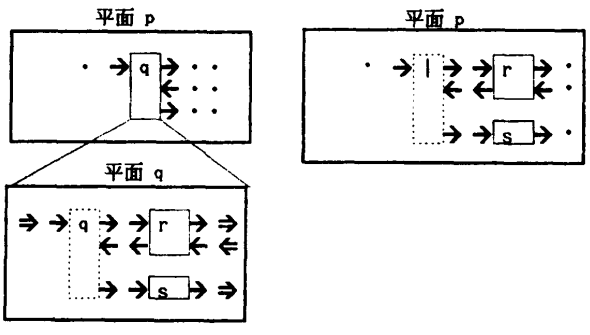
2.2.4 or 述語の意味

Prolog では or の関係を表現するのに，同一述語名同一な引数の数をヘッドにもつホーン節をつくることで解決できる．しかし or を表現するのに，いちいち新しい述語名そしてホーン節をつくるのでは手間がかかる．そこで通常の処理系では or を表現する述語(or 述語) が書けるようになっている．

```
p 〈- … & q & …          p 〈- … & (r|s) & …
q 〈- r.
q 〈- s.
```

(a) 述語qによるプログラム　(b) or 述語 | によるプログラム

図 9 or 述語のプログラムの例
Fig. 9 Example of program using or-predicate.



(a) 図9(a)の実行例　(b) 図9(b)の実行例

図 10 図9のプログラムの実行例
Fig. 10 Examples of executions of programs in Fig. 9.

例えば or 述語を | としたとき，**図 9**（a）では | を用いず述語 q で or を表現しているのに対し，図 9(b)は述語 q のかわりに | を使用した場合のプログラムである．

BPM では | を含むプログラムの実行過程を表現する際，| を用いないプログラムの実行過程の図から，自然に表現し直せるように配慮している．例えば**図 10**（a），(b)は各々，プログラム図 9（a），(b)の実行過程の一部を表現したものである．この際，図 10(b)の表現は，図 10(a)の平面 q を平面 p のなかのゴールボックス q へ射影することにより得ると考える．またプログラム図 10(a) の述語 q を図 9(b)では | へ書き換えたことに対応して図 10(a) のヘッドボックス q は図 10 (b)では，擬ヘッドボックス | に置き換える．

この擬ヘッドボックス | は新たな概念だがそのセマンティクスはヘッドボックスの意味に類似する．厳密には次のようになる．まず論理式 A|B において or 述語 | の前の論理式Aを第一論理式，Bを第二論理式と呼ぶことにする．そして擬ヘッドボックス | に属する矢印の意味を次のように定める．

call	擬ヘッドボックスを生成し，第一論理式を要求する
exit	論理式を返す
redo	次の論理式を要求する
fail	第一論理式，第二論理式が既に返され，擬ヘッドボックスの終了を意味する

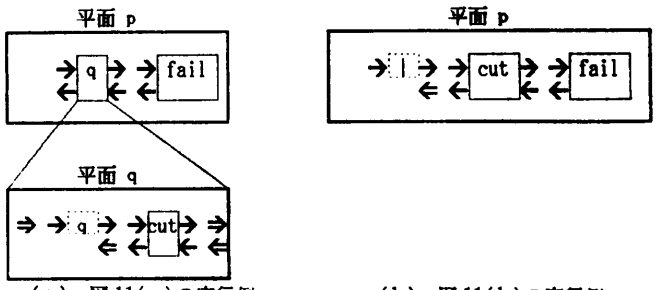
以上のように BPM では or 述語の意味を平面の射影で説明する．しかしカットが第一論理式ないし第二論理式に現れるときは，平面の射影操作を基本にカットのスコープを考慮して意味を与える必要がある．

例として**図 11**（a），(b)のプログラムをあげる(このプログラムは単なる例で実用上の意味はない)．この二つのプログラムはカットのスコープが異なるため同一の動きをしない．このことは，各々の実行過程を表した**図 12**（a），(b)を見ることでより明らかになる．すなわち図 12（a)のカットのスコープが平面

```
p <- … & q & fail.        p <- … & (cut|s) & fail.
q <- cut.
q <- s.
```

(a) 述語 q によるプログラム　(b) or 述語によるプログラム

図 11 カットと or の関係を含むプログラムの例
Fig. 11 Examples of programs including cut and or-relation.



(a) 図 11(a)の実行例　(b) 図 11(b)の実行例

図 12 図 11 のプログラムの実行例
Fig. 12 Examples of executions of programs in Fig. 11.

q にあるのに対し図 12（b)のカットのスコープは平面 p にある点が異なる．

ここで与えたのは実は，C-prolog および VMPROLOG (VM/プログラミング・イン・ロジック) においてシステムに組み込まれている or 述語の意味である．しかしながら，or 述語を

```
or(P, Q) <- P, or(P, Q) <- Q
```

のように高階述語で定義する処理系の動きを表現する場合には，図 11（a)のプログラムのように新たな述語 q を使う場合とカットのスコープが同一となる意味を与えなくてはならない．すなわち，スコープは一つ下のレベルの平面へ移り，実行過程の動きは述語図 12（b)と類似のものになる．

2.2.5 ルール・ベースの状態と BPM

BPM では，ルール・ベースの状態を図形中に陽には表現せず，各矢印にその時点でのルール・ベースの状態が対応していると考える．ここではルール・ベースにルール R を登録する述語を assert (R), R を削除する述語を retract (R) とする．**図 13** はゴール assert (R) & retract (R) を実行したときの BPM の表現である．

ルール R は，1→ ではルール・ベースへ登録されておらず，ゴールボックス assert で登録され，→2 ではルール・ベースに存在しており，3→ でもルール・ベースの状態は変化せず，ゴールボックス retract で削除されるため，→4 では存在しない．

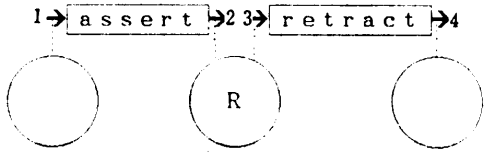


図 13 ルール・ベースの変化
Fig. 13 Change of rulebase.

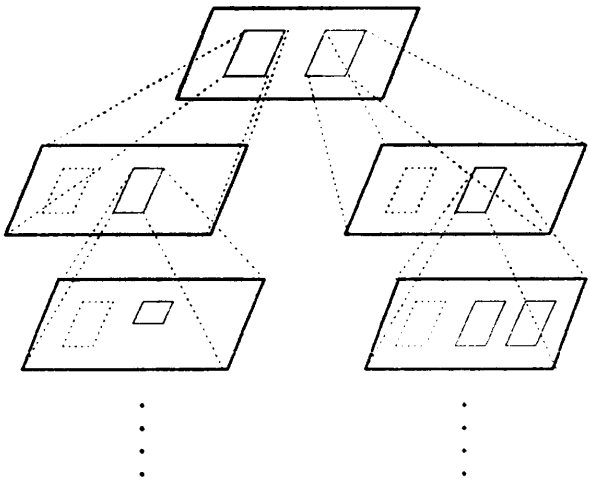


図 14 計算の全体像
Fig. 14 Whole image of a computation.

2.2.6 計算の全体像

BPM では計算の全体像を，平面が多層に重なり合った木で表現する（図 14）．

木のトップレベルの平面は，図1のようにヘッドボックスをもたない平面である．平面中の各ゴールボックスに対する平面はその下に描かれている．木の終端は，ヘッドボックスしかない平面である．ヘッドボックスしかない平面は，対応するゴールボックスの述語 g(T) とユニファイ可能な節がない場合およびユニファイ可能な節がファクトしかない場合につくられる．

2.2.7 ループするプログラムの表現

ループするプログラムのタイプには大きく分けて次の2種類がある．

・再帰的ループ

・バックトラックによるループ

以下各々について BPM での表現を例にとって示す．

再帰的ループの例として，プログラム

loop ⟨- loop.

を考え，ゴール ⟨- loop. を実行した場合の実行過程の表現は図 15 のようになる．図から分かるように，

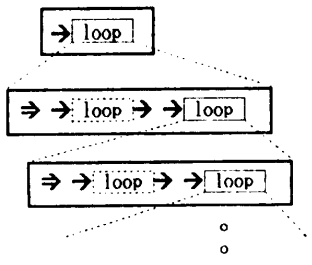


図 15 再帰的ループの例
Fig. 15 Example of recursion loop.

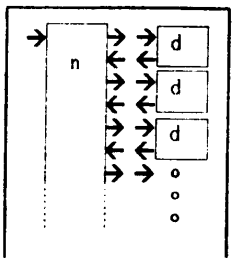


図 16 バックトラックによるループの例
Fig. 16 Example of backtrack loop.

ゴールボックス loop からは exit 矢印で抜けることなく無限に平面を生成してゆく様子が分かる．

また，バックトラックによるループの例として，プログラム

n(0).

n(s(*X*)) ⟨- n(*X*).

d(s(*X*)) ⟨- d(*X*).

を考え，ゴール ⟨- n(*X*) & d(*X*). を実行した場合の実行過程の表現は図 16 のようになる．この場合は無限に広がる平面が生成されてゆくことが分かる．

3. むすび

本稿では Prolog の手続き的側面を明確に把握するため，視覚的計算モデル BPM を提案した．BPM は，プログラム作成・デバッグ・分析という一連のプログラミング作業の効率化を目指し導入されたモデルであり，我々は過去 BPM に基づいた Prolog のデバッガ PROEDIT 2 を開発し，実際にプログラム開発環境として使用している[4),5)]．

謝辞 本研究中は，福永光一知識ベース・システム担当，および広瀬紳一氏に多大なご指導，ご助言を頂きました．ここに感謝いたします．

参考文献

1) Lloyd, J. W.: *Foundation of Logic Programming*, Springer (1984).
2) Johnston, J. B.: The Contour Model of Block Structured Process, Proc. of Symp. on Data Structures & Prog. Langs., *SIGPLAN Notices*, Vol. 6, No. 2, pp. 55-81 (1971).
3) Byrd, L.: Prolog Debugging Facilities.
4) Numao, M. and Fujisaki, T.: Visual Debugger for Prolog, *Proc. of the IEEE Second Conference on Artificial Intelligence Application*, pp. 422-427 (1985).
5) 森下真一，沼尾雅之：Prolog の視覚的計算モデル BPM とそれに基づくデバッガー PROEDIT2,

Proc. of Logic Programming Conf. in Tokyo, pp. 177-184 (1986).

(昭和61年9月11日受付)
(昭和62年1月14日採録)

森下 真一（正会員）

昭和35年生．昭和58年東京大学理学部情報科学科卒業．昭和60年同大学院理学系研究科修士課程修了．同年日本アイ・ビー・エム(株)入社，東京基礎研究所に勤務．論理プログラミング全般に興味をもつ．

沼尾 雅之（正会員）

昭和33年生．昭和56年東京大学工学部電気工学科卒業．昭和58年同大学院工学系研究科修士課程修了．同年日本アイ・ビー・エム(株)に入社．以来同社東京基礎研究所知識ベース・グループにおいて，論理型言語およびその開発支援環境の研究に従事．電子情報通信学会，ソフトウェア学会各会員．