プログラミング言語　８－５
（１９８６．１０．１７）

プロダクション・システム記述言語ＰＯＰＳ２

広瀬　紳一

日本アイ・ビー・エム株式会社　東京基礎研究所

ＰＯＰＳ２は、ＯＰＳ５のパターン・マッチング機能を拡張し、選言的条件や連言の否定等の記述を可能にしたプロダクション・システム記述言語である。　本稿では、ＰＯＰＳ２の機能の概要を説明し、ＰＯＰＳ２を用いたプログラミングの例を紹介する。　また、この言語をＰｒｏｌｏｇ上に実現する際の、実行効率上の問題点について述べ、その解決法についても考察する。

Production System Description Language POPS2

Shin-ichi　HIROSE

Tokyo Research Laboratory, IBM Japan, Ltd.
5-19 Sanbancho, Chiyoda-ku, Tokyo 102, Japan

POPS2 is a production system description language which provides more powerful pattern matching facility than OPS5. The enhanced facility includes disjunctive conditions and negated conjunctive conditions in the LHS of rules.

In this paper, the facilities of POPS2 are explained with some programming examples. Efficiency problem in implementing POPS2 on Prolog is described and the reason and the solution of the problem are discussed.

１．はじめに

エキスパート・システムの開発において、最も重要な作業の一つに、『専門家の知識をどのようにコンピューター上で表現するか』の選択が挙げられる。プロダクション・システムでは、知識をプロダクション・ルールと呼ばれる if-then 型のルールの集まりで表現する。プロダクション・ルールによる知識表現は、コンピューターにあまりなじみのない、問題領域の専門家にも容易に理解できるものであり、また、システムのデバッグ、更新の際にもルール単位で編集が行えるという特長がある。そのために、試行錯誤的なプログラミングが必要とされる、エキスパート・システムの開発に広く用いられている。

プロダクション・システム記述言語の代表的なものとして OPS5[１]がある。OPS5 では、Recognize-Actサイクルと呼ばれる、

Match:　ルールの条件部と作業記憶(WM)とを照合して、適用可能なルールを見つける
Select:　見つかったルールのうちの一つを選びだす
Act:　　選ばれたルールの動作部を実行する

の三つのステップからなるサイクルの繰り返しで実行が進んでいく。一般に、このタイプのプロダクション・システムでは、Match のステップにかかる時間が大部分を占めており、この時間を短縮することで、全体の性能が大きく向上することが知られている。OPS5 で用いられている RETE アルゴリズム[2]は、このステップを高速に行うためのパターン・マッチングの方法である。RETE アルゴリズムでは、ルールをあらかじめ、ネットワークの形にコンパイルする。そして、そのネットワーク中の各々のノードに作業記憶の情報を保持させることによって、作業記憶の変化をネットワークに流してやるだけで、全体のパターン・マッチングが行えるような工夫がなされている。このことによる高速化が、OPS5 の実用性に大きく貢献しているといえる。

しかし、OPS5 の記述力には、かなり強い制限がある。例えば、

- WM要素の属性値は、アトムか数しか許されない
- 条件部で連言の否定や、WM要素パターンの選言を書くことができない
- 属性値の制限には、大小関係しか書けない
- 動作部で自由に Lisp 関数を用いることができない

等である。これらは、ルールを複数個用いたり、補助関数を定義したりすることで、解決可能ではあるが、そうした場合にも、素直な知識表現を妨げ、ルールの読みやすさや、書きやすさを著しく損なうという問題が生じる。

このような問題を解決するために、RETE アルゴリズムに変更を加え、条件部に比較的自由な条件が書け、そこで束縛された変数を動作部でそのまま使用できるというセマンティクスをもった、プロダクション・システム記述言語 POPS2 を開発した。使用言語はＶＭ／プログラミング・イン・ロジック (VM/Prolog) [４]である。本稿では、POPS2のあらましについて説明し、いくつかのプログラミング例を紹介する。また、Prolog による RETE の実現に際しての問題とその解決法についても考察する。

２．ＰＯＰＳ２の概要

２．１．ＷＭ要素

ＷＭ要素は、 Prolog の関数項で表現される。例えば、

person(yokoi, researcher, 1959)

block(block1), color(block1, blue), size(block1, big)

といった具合である。あらかじめ literalize 述語を用いて宣言しておけば、ＷＭ要素の各々の属性に名前（属性名）をつけることができる。つまり、

<- literalize(person(name, occupation, birth_year)).

を実行しておけば、先のＷＭ要素は、

person(name = yokoi, occupation = researcher, birth_year = 1959)

と書いても良いし、属性名を指定した時には、属性の順序は任意で良いから、

person(occupation = researcher, birth_year = 1959, name = yokoi)

でも同じＷＭ要素を示すことができる。 最後のものは、 OPS5 でいえば、

(person ^occupation researcher ^birth_year 1959 ^name yokoi)

に対応する。 POPS2 では、属性値として任意の項を用いてよい。例えば、

data(f(x,g(0)), {a,b,c})

といったＷＭ要素を使うことができ、ルールの条件部の中で、任意の部分項との照合を行うことができる。

２．２．プロダクション・ルール

POPS2 のルールは、次のような形で記述される。

rule ＜ ルール名 ＞ [: ＜ 優先度 ＞] is

＜ 条件部 ＞ => ＜ 動作部 ＞ .

これは、 OPS5 の

(p ＜ルール名＞ ＜条件部＞ --> ＜動作部＞)

に対応するものである。ただし、各ルールには、優先度として整数値を持たせることができ、競合解消の時に、優先度の大きなものから選ばれていくようになっている。

２．３．条件部

条件部には、ＷＭ要素に対するパターンの、『かつ(&)』、『または(|)』、『・・でない(¬)』の自由な組合せが書ける。例えば、『赤くて大きなブロック以外のブロック』は、

block(Block) & ¬(color(Block, red) & size(Block, big))

と書ける。特に、二重否定(¬¬)の持つ性質は、なかなか興味深い。すなわち、

foo(X) & ¬¬ bar(X,Y) => ...

は、foo(X) にマッチする各々のＷＭ要素に対し、同じ第一引数を持つ、クラス bar のＷＭ要素が一つ以上あれば、一回だけ照合に成功する。そして、 Prolog からの類推ですぐわかるように、変数 Y は、ここでは束縛されない。

また、変数は、属性値の一部と照合させることもできる。例えば、ＷＭ要素

data(f(x, g(0)), {2, 0, 1})

は、パターン

data(f(x, g(N)), {2, N, M})

とマッチする。

更に、WM要素に対するパターンの他に、任意の Prolog ゴールを、擬似的なパターンとして書くことができる。そして、その述語呼出しが成功すると、それと同じ形のWM要素が存在しているのと同様に扱われる。このようなパターンは、パターンの前に、"#"か"!"をつけることによって区別される。"#"と"!"の違いは、"#"では、そのゴールの最初の解だけが使われるのに対し、"!"では、全ての解をチェックすることである。例えば、述語 member が普通に定義されているとし、list({a,b,c}) と element(b) という、ふたつのWM要素があったとすると、

list(L) & # member(E,L) & element(E)

という条件は満足されないが、(変数 E の値は a になる)

list(L) & ! member(E,L) & element(E)

list(L) & element(E) & # member(E,L)

は満たされる。(最後の条件では、先に変数 E が b で束縛される。) このようなパターンを使うことの利点は、WMの大きさを節約できることである。member のように、解が無限にあるような場合は勿論、あるクラスのWM要素の有無を Prolog のプログラムとして記述できる場合には、実際にWM要素として持っている必要がなくなる。(無論、記憶領域よりも実行時間が大切な場合には、WM要素にしておいた方が良いこともある。)

OPS5 の要素変数 (element variable) に対応する構文として、

< 要素変数名 > = < パターン >

があり、この変数には、マッチしたWM要素のタイムタグが入れられる (次節を参照)。

WM要素に対するパターンの中には、 OPS5 と同様に、値に関する条件の、『かつ』、『または』、『・・でない』を書くことができる。しかし、これらの組合せ方は、パターンの場合と同様に全く自由である。更に、任意の Prolog 述語を、属性値に対する制限として付加することができる。例えば、うるう年であるかを調べる述語、uruu(X) を、整除の余りを求める組込み述語 rem をもちいて、

uruu(X) <- rem(X,400,0).

uruu(X) <- rem(X,4,0) & ~rem(X,100,0).

と定義しておけば、『うるう年生まれの人』というパターンは、

person(birth_year = X : uruu(X))

と書くことができる。特に、一引数の述語による制限で、その引数が直前の変数と同じ場合には、

person(birth_year = X : uruu)

としても良い。

2．4．動作部

動作部は、任意の Prolog 述語呼出しで良い。特に、そこに現れる変数が条件部にもあり、照合によって値を持てば、動作部の実行時に、その値に束縛されて呼ばれる。WMを操作するための述語として、

make(X) : XをWM要素として追加する

remove(N) ： タイムタグがNであるWM要素を削除する

modify(N,M) ： タイムタグがNであるWM要素にMで示される変更を加える

等がある。例えば、 goal(active) というWM要素を goal(satisfied) に変更するためのルールは、

```
G = goal(active)  =>  modify(G, { 1 := satisfied })
```

となる。もし、クラス goal の第一要素に status という名前がつけられていれば、

```
G = goal(status = active)  =>  modify(G, { status := satisfied })
```

でも良い。また、halt() という述語が定義されており、これを実行すると、その動作部の実行が終了した後、停止する。

２．５．トップ・レベル

前節で述べたWMの操作をおこなう述語は、全て通常の述語であるので、そのままトップ・レベルでも使うことができる。その他に、 OPS5 と同じ働きを持った、load(File), run(), run(N), cs(), pm(Ruleid), wm(), wm(N) 等がある。また、ソース・ファイルには、WMの初期状態を宣言しておくことができ、述語 reset() の実行によって、いつでもWMを初期状態に戻すことができる。さらに、wm 述語では、あるクラスのWM要素だけ、または、あるパターンにマッチするものだけ、という指定もできる。デバッグ用の述語には、break(Ruleid) や trace(X) 等があり、大体 OPS5 の pbreak や watch に対応している。

競合解消はルールの優先度に基づいて行なわれ、それで一意に決まらない時には、最も最近に成立した（もしくは、最も古くから成立している）ルールを優先する。 OPS5 と比べると、あまり細かい制御はできないが、競合集合の内容を、ユーザーが簡単に見ることができ、必要に応じて競合解消を行なう述語を定義することは、比較的容易である。

３．ＰＯＰＳ２の応用例

３．１．工程設計システムへの応用

POPS2 を用いて工程設計をおこなうエキスパート・システムのプロトタイプを試作してみた。このシステムは、

ｉ）基本的な情報から加工順序に関する制限を生成する

ii）制限を満たす加工プランの候補を生成する

という二つのフェイズからなっており、現在50個のルール(平均4個、最大14個の条件、91個の動作、うち66個が作業記憶を変更するもの)を含んでいる。OPS5 でこれと同等なものを作るためには、更に、同程度の数の補助的なルールと、ユーザー定義の動作を追加する必要があろう。また、二番目のフェイズで生成される解の数をできるだけ制限するために、はじめは、最も厳しい制限を課しておいて、そのもとではプランの生成が不可能である場合には、徐々に制限を緩めていくという方法をとっている。

はじめのフェイズは、専門家の知識を用いて、加工順序に対する制限を生成するが、ここでの制限には大きく分けて、ふたつの種類がある。すなわち、絶対に満足されなければいけないものと、満たされていることが望ましいというものであり、後者が、制限を弛緩

してゆくときの対象となる。このフェイズで用いられるルールの特徴としては、作業記憶要素の属性値間の関係を扱うものが多いということが挙げられる。例えば、

```
rule dependency is
  phase(constraint_generation) &
  face(F0,Dir0) &
  face(F1,Dir1:inverse(Dir1,Dir0)) &
  face(F2,Dir2:ip(Dir2,Dir0)) &
  face(F3,Dir3:(ip(Dir3,Dir0)&ip(Dir3,Dir2)))
 => .....
```

は、四つのfaceの組、(F0,F1,F2,F3)で、F1がF0と逆方向(inverse)を向いており、更に、F0,F2,F3が、互いに直交している(ip)ようなものをみつけるルールである。OPS5 のような大小比較だけが許されているようなシステムでは、このような関係を、対象となりうる全ての組について、あらかじめルールを用いて（更に、そのチェックをするようなユーザー定義の動作を書いて）、作業記憶に蓄えておく必要がある。しかし、実際に利用されるのはそのうちのごくわずかであり、あまり効率的な方法とはいえない。POPS2 では、属性値に対するチェックとして、任意の述語が書けるので、パターンマッチング時に必要なものについてだけおこなえばよく、計算時間および記憶領域の節約になる。

加工順序に対する制限は、基本的にはある加工工程をおこなうためには、他の工程のうちのあるものが既に完了していなければならない、という形をしている。二番目のフェイズでは、短かいサブ・プランに、次に実行可能な工程を加えてゆく、という方法で、次々にもとのものより、ひとつだけ長いサブ・プランを生成してゆく。そして、全ての工程を含んだプランが得られれば、それを制限をみたす加工プランの候補として出力する。ここでは、属性値間の関係のほかに、属性値として、構造を持ったもの（リスト等）を扱う必要がある。次に示すのは、プラン生成を行うためのルールを簡略化したものである。

```
rule plan_generation is
  plan(Pl) &
  feature(Fe:¬member(Fe,Pl)) &
  support(Fe,Sl:subset(Sl,Pl))
 =>
  make(plan(Fe.Pl))
```

ここで、変数Pl,Slの値はリストであり、そのsemantic restrictionとして、Prologの述語memberやsubsetが使われている。ＯＰＳ５のような属性値として構造を持ったものをとれないシステムでは、かわりに、リストや集合に名前をつけ、その要素との間の包含関係や前後関係を、作業記憶に保持しておく必要があり、ルールの読みやすさや変更の容易さの点で問題がある。

３．２．WM要素中での変数の使用

POPS2 では、いかなる関数項もWM要素として利用可能であり、例えば、

```
less_than_or_equal(0, *)
```

のようなものも許される。ここでは、このようなWM要素の利用法の二つの例を紹介する。

ひとつめは、複雑な条件の照合の中間結果としての利用である。例えば、クラス p のWM要素の三つ組、

```
D1=p(A1,B1,C1), D2=p(A2,B2,C2), D3=p(A3,B3,C3)
```

で、三つの条件

```
A1 > A2 or A1 > A3
B2 > B1 or B2 > B3
C3 > C1 or C3 > C2
```

を満たすものをみつけたいとする。 OPS5 では、選言の構文（<<...>>）では、定数しか使えないので、これらを展開して $2^3=8$ 個のルールにしてやる必要がある。そこで、次のようなルールを考えてみる。

```
D1=p(A1,*,*) &
( D2=p(A2: (A2 < A1),*,*) | D3=p(A3: (A3 < A1),*,*))
=> make( m1(D1,D2,D3) )
```

これは、はじめの条件に対応するもので、D2 か D3 のうちのどちらかが変数であるようなWM要素を生成する。例えば、m1(10,*,12) は、タイムタグ10と12のWM要素で、はじめの条件（の A1 > A3 の方）が満たされていることをあらわす。従って、残りの二つについても同様なルールをつくり（それぞれ、m2, m3 を生成するとする）、これらを、

```
m1(D1,D2,D3) & m2(D1,D2,D3) & m3(D1,D2,D3) => .....
```

でまとめてやれば、すべての条件を満たす D1,D2,D3 をみつけることができる。

もうひとつは、ルールの意味をWM要素によって制御しようというものである。WM要素 p(X,Y,Z) のうちで、X と Y が等しいもの、X と Z が等しいもの、などを動的に条件を変えて選びたいとする。このような場合には、

```
p(X, Y, Z) & q(X, Y, Z)
```

という条件を記述し、クラス q のWM要素を変えることによって、目的を達成できる。例えば、WM要素 q(A,A,B) があるときには、上の条件は、p(1,1,2)、p(3,3,3) などによって満足されるが、p(1,2,3) や p(1,2,1) では満足されない。そして、q(A,B,A) に対しては、p(1,2,1)、p(3,6,3) などが選ばれることになる。

４．高速化の試み

４．１．セル領域の導入

RETE アルゴリズムの特徴として、部分マッチングやルールのインスタンスのために、大量のheap領域を動的に使う、ということが挙げられる。VM/Prologでは、heapとしてユーザーが自由に使えるのは、ルール・ベースだけであるから、これらの情報は、axiomとして組込み述語addaxとdelaxを用いて管理する必要がある。我々も当初、この方法でインプ

リメントを行っていたが、処理系の評価のために、いくつかの例題を実行しているうちに、重大な問題に気づいた。例えば、ルール

```
test(N:N>0)  =>  N1 := N - 1  &  make(test(N1))
```

を、初期のWM要素として、test(200)を与えて走らせた場合と、test(3200)の場合では、単に、同様のことを200回繰り返すか3200回繰り返すかの違いしか無いはずであるが、それらの実行時間の比は、約23倍にもなっていた（表１）。このことを調べているうちに、VM/Prologでは、実行が進むにつれて、delax述語の実行時間が徐々に増大していく、という結論を得た。この原因は、次のように考えられる。VM/Prologでは、ルール・エリアの使用されていない部分を、ポインターでつないで管理している。そして、削除されたルールが占めていた領域は、単にポインターで連結されるわけではなく、隣接する領域が未使用であるかどうかもチェックして、しかるべき位置に挿入される。このこと自体は、メモリー管理上、できるだけ大きな空領域を持っておく、という意味で、一般的かつ有効な方法であるが、今回のように、ルール・ベース中に何らかの状態を保存しておきたい（これもまた、よくあることである）場合、特に、それらの情報がひんぱんに変化するような時には、十分であるとは言えない。つまり、何度もaddaxやdelaxの実行をおこなった結果、空領域の連鎖が非常に長くなってしまい、delax述語の実行で新たにできた空領域を、正しい場所に挿入するための手間が、大きくなってしまうのである。（逆にaddax述語の場合には、空領域の連鎖の先頭から、十分な大きさのものを見つけてくるだけであるから、よほど大きなaxiomでないかぎりあまり問題はない。）従って、POPS2 のトップ・レベルのrun述語の実行につれて、ルール・エリアのfragmentationが進行してゆき、はじめに述べたような、症状が出てくるわけである。

この問題を避けるためには、addax,delax述語をできるだけ使用しないようにすればよい。POPS2 でこのようにして管理している情報は、次の５種類に分類できる。

1) WM要素
2) RETE ネット・ワークのノード
3) 部分マッチングやコンフリクト・セット
4) WMの大きさや、実行したステップ数を保持するカウンター
5) literalizationや競合解消戦略等、その他の制御情報

このうち、実行中に頻繁に変更されるのは、１、３、４である。また、１は、一般の項になるが、３と４の情報は、整数か整数のリストである。（３は、WM要素のリストで表現した方が良いが、記憶領域の節減のため、タイム・タグのリストで表現している。）今回は、とりあえず、扱いの簡単な３と４に限って、Lisp風のセルを用いて、これらの情報を管理するようにしてみた。このことによる改善を表１に示す。Loop200とLoop3200は、先に述べたものである。改良後では、期待どおり、実行時間の比がほとんど16になっている。また、MandB は、有名な『猿とバナナ』のルールに、若干の変更を加えたもの、Proplan は、前節で述べた工程設計システムの実行例である。例は少ないが、大きな（実行時間のかかる）ものほど、改善の割合が良いことがわかる。しかし、今回の改良では、１のWM要素を扱わなかったため、まだ問題が残っている。例えば、Proplan の実行時間のうち、約４分の１が、タイム・タグからWM要素本体を得るのに使われていることがわかってい

ﾌﾟﾛｸﾞﾗﾑ	導入前	導入後
Loop200	516	347
Loop3200	11892	5556
MandB	201	142
Proplan	53714	27988

[ms]

表１．セル領域の導入による実行時間の短縮

る。これは、Lispのように、heap領域内のデータが複数の axiom で共有できれば、全く不要になるものである。

４．２．部分計算による高速化

Prolog での部分計算は、一般に、メタ・インタープリターの技法を用いた処理系の高速化に有効である。POPS2 のネットワーク・インタープリターは、まさにこの技法を用いたものであり、部分計算による高速化が期待できる。そこで、早速、簡単な部分計算のプログラムを用いて、実行速度がどの程度向上するかを調べてみた。結果としては、部分計算後のプログラムは、もとのものにくらべて、２～３倍程度速くなることがわかった。ただし、現在の部分計算プログラムでは、前節での改良にかかわらず、同程度の実行速度を持つプログラムが得られるようである（例えば、MandBでは、ともに70ms位になる）。また、Proplanでは、部分計算前と後で実行時間に、ほとんど差がないという結果も得ている。これは、おそらく、部分計算によって一つの述語呼出しが展開され、複数の述語呼び出しになるため、各々の節が大きくなり、そのためのオーバーヘッドが無視できない程になっているためと思われる。また、現在は、ほとんど可能な限り、部分計算をしてしまうようにしてあるため、部分計算後のプログラムが、巨大になってしまうという問題もある。従って、今後は、部分計算むきの処理系の構成や、どの部分を部分計算すれば効率的であるか、等を検討していく必要があろう。とはいうものの、現状でも、比較的ルールは少なく、多量のWM要素を扱うようなアプリケーションにおいては、かなり、有効な高速化の手法であるといえよう。

５．おわりに

OPS5 のパターン・マッチング等の機能を拡張して作られたプロダクション・システム記述言語 POPS2 を紹介した。

POPS2 で拡張された機能には、大きく分けて

－ RETE アルゴリズム自体の拡張（ノードのタイプの追加）[3]によるもの

－ インプリメンテーション言語を Prolog にしたことによるもの

の二種類がある。後者の貢献度が高いものとしては、直接実行される動作部や、 Prolog 述語を用いた擬似WM要素のパターン（この場合は、ノードも追加されている）の機能などがある。しかし、大部分の機能は、ノードのタイプの追加によっており、他の言語によるインプリメンテーションにも応用可能である。これは、RETE アルゴリズムが、ネット

ワークというハイ・レベルなモデルを持っているためであり、今回のような拡張を行うにあたっては、非常なメリットであった。

今後の方向としては、システムの高速化が最も重要な点であろう。また、機能の拡張も必要に応じて行っていきたい。例えば、p(X)というパターンにマッチするWM要素の全体のうちで、『Xの値がN番目に大きいもの』という条件などは、必要性が高いように思われる。しかし、基本となる言語の機能にも増して重要なのが、プログラム開発環境である。現在の POPS2 には、実行中のルール名や、WMの変化をトレースする程度のデバッグ用機能しかないが、今後は、マルチウインドウ等を用いた、より直観的に実行の様子が把握できるような環境が必要となろう。もうひとつ大事なことは、ルールベース・システムでは、一般にルール数の増加につれて、全体を管理するのが極めて困難になるということである。これに関しては、例えば、ルールをいくつかの機能別のルール群に分割して管理するといったようなことが必要となろう。この際に『・・用のルール・セット』というプロトタイプを用意しておき、実際のプログラミングにおいては、それを適当に修正して使用する、ということができれば、プログラミング時の負担はかなり軽減されることが期待できる。

参考文献

[1] Forgy, C.L.: "OPS5 User's Manual", 1981.

[2] Forgy, C.L.: "RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem", Artificial Intelligence, 19, 1982.

[3] 広瀬："強力なパターン・マッチング機能を持ったプロダクション・システム記述言語ＰＯＰＳ２"，日本ソフトウェア科学会 知識プログラミングシンポジウム資料、KP-85-8, 1985.

[4] "VM/Programming in Logic Program Description/Operations Manual", SH20-6541, IBM Corporation, 1985.