並列処理向き論理型言語の提案

相田　仁，田中英彦，元岡　達
（東京大学　工学部）

１．はじめに

現在、ＩＣＯＴをはじめとする各所で並列推論マシンの研究が進められ、いくつかのアーキテクチャモデルが提案されている。しかし、現在までのところ、それらの並列推論マシンが仮定している言語は基本的に pure Prolog にとどまっており、知識情報処理の分野をはじめとする各種の応用分野における問題解決を記述するのに不十分である。そこで、ここでは並列推論マシンを記述するのに有用と思われるいくつかの機能を含む論理型言語について考察をおこなう。あらかじめおことわりしておくが、この言語はまだ検討段階であり、ここでの記述はそのアウトラインのみを示すものである。シンタックスについても未定であり、とりあえず DEC-10 PROLOG にできるだけ従っておくことにする。

２．プログラミング・スタイル

数年前に我々が Prolog に取り組みはじめた際に、論理プログラミングに期待していた点として次のような点があった。
・第１に、一階述語論理には時間的前後関係が仮定されていないので（時間の概念がないので）、並列処理に適しているのではないかという点。
・第２に、プログラムの仕様を論理で与えることができれば、プログラムも論理であるため、プログラムが仕様を満たしていることの検証、あるいは仕様からプログラムの自動合成をおこなうことが可能ではないかという点。
・第３には、論理型言語が一階述語論理に基礎をおいており、述語論理はさらに人間の思考過程のモデル化であるので、従来人間が機械（計算機）の立場に立ってプログラミングしていたのが、人間により近い立場でプログラミングできるようになるのではないかという点。

しかし、一般の人々はそこまで高い期待を抱いていたわけではないらしく[1]、Prologがあゆんだ道筋は基本的に Lisp の場合と変わりなかった。現在 Prolog を Lisp から隔てているものは、変数の論理性（変数どおしを先に単一化しておいてからあとで値を束縛する）と、不完全データ構造の取り扱いだけであるように見受けられる。

新しい論理型言語を考えるにあたって、まず、プログラミング言語の基本的な２つのプログラミング・スタイルについて、ふりかえってみたい。

手続き的なプログラミングは、計算機の動きかたを記述するものである。その極端はアセンブラによるプログラミングであり、プログラムにより、計算機の動作を完全に規定することができる。一般の手続き型言語の場合には、コンパイラの最適化などのために、必ずしも計算機の動作は完全に規定されないが、基本的な動作の順序は決まっている。さらに、最も手続き型言語を特徴づけるものとし

て、与えられたプログラムを何度実行してもその動作は同一であるという点が挙げられる。もちろん、割り込みや、外部のファイル環境により影響を受けるが、それらまで含めて考えれば手続き型言語の動作は常に決定的である。

もう１つのスタイルは、宣言的なプログラミングである。その典型的な例としてはハードウェア記述用の言語が挙げられる。これで記述されるものはハードウェアの構造であり、全く同じ記述が LSI のマスクパターンを作成するのに用いられたり、正しく動作するかどうかの検証に用いられたりする。すなわち、これらの言語で書かれた記述からだけでは、その記述を与えられた計算機がどのような動作を行なうのか、全くわからない。これは極端な例であり、プログラムというよりデータと呼ぶべきものかもしれないが、一般に宣言的な言語とは何らかの非決定性を有するものということができよう。

さて、並列環境でのプログラミングを考えた場合、その動作の決定性はほとんど期待出来ない。すなわち、同一のプログラムであっても、割り付けられたプロセッサの数や位置関係が異なれば、計算の前後関係は必然的に異なってくる。この非決定性に対して、両プログラミング・スタイルの対処法はおおよそ次のようなものになる。

・手続き型プログラミングの場合は、プロセッサの割り付け（数、配置）を指定できるようにし、できるかぎり決定的に動作するようにする。

・宣言的プログラミングの場合は、決定的な動作をあきらめて入出力関係などの仕様だけを記述し、あとは処理系に任せてしまう。

並列処理の環境では、ＯＳなど一部の分野をのぞいて、後者のほうが自然であり、従来からもグラフ言語・関数型などの非手続き型言語がもっぱら研究されてきている。そこで、ここでも、論理型言語を宣言的立場に立って見直しつつ、各種の機能について考えてゆきたい。

３．一般的推論機能

3.1 探索順序指定

論理型言語を並列処理する場合、並列に処理しうる部分をすべて並列に処理してゆけば、非常に大きな並列性を取り出し得ることが知られている。問題となるのは、むやみに並列に処理しようとすると、いわゆる数の爆発を引き起こしてしまい、制御不能に陥る可能性がある点である。実際には、チェックポイントを設けておき、枝刈りをすることになるが、その場合にも多くの計算が無駄になってしまう。そこで、標準的には並列探索を行なうとしても、プログラマが爆発の危険が解っている場合などには、逐次探索が指定出来るのが望ましい。また、複数の AND リテラルの解の探索については、現在までのところ、常に並列処理を行なうことは無矛盾性検査のオーバーヘッドの点から現実的でない。そこで、標準（ DEC-10 PROLOG に一番近いシンタックス）としては並行処理を行なうにとどめておき、強制的に並列処理を行なわせるための記法を用意する。並行リテラル間のディスパッチの方法としては、Cuncurrent Prolog [2] で用いられているような変数に付けた annotation による方法が適当であろう [3] 。

・逐次 AND

p & q

p の解を求めおわってからq の解を求める。

・並行 AND

p , q

p の解とq の解を並行して求める。通常はp の解を先に求めるが、データの束縛状況などによりシステムが適当に判断して、q の解を先に求めたり、p とq の解の探索をコルーチン的に行なったりすることもある。

・並列 AND

p // q

p の解とq の解をプロセッサ環境の許す範囲内で並列に求める。一般に、無矛盾性検査が必要となる。

・逐次 OR

p :- q;;

p :- r

前の定義節を適用した場合の解がすべて求まってから後の定義節を適用する。

・並列 OR

p :- q.

p :- r

プロセッサ環境の許す範囲内で定義節の適用を並列して解く。

・annotated 変数

X ?

並行 ANDにおいて、annotationのついた変数が未束縛である間、それを含むゴールリテラルの処理の優先度を低下させる。

3.2 探索範囲指定

探索の順序とともに、探索の範囲を指定する機能も、応用プログラムにおいて有用である。並列処理の環境では、全解探索が自然であるが、ただ１つだけ解を得れば十分である場合や、ＯＳにおける共有資源の管理のように、積極的に１つを選び出さなければならない場合もある。このような場合に、どの解が選ばれたかによってあとの処理を変えたい場合も多いので、いわゆるガード節を用いるのが適している。

また、変数の全ての値に対してある条件が成り立つときのみを取り扱いたいことがある。ここで考えている論理型言語においては、通常の Prolog と同様に、変数の変域について特に考えないので、全ての値に対してということは、変数が未束縛のまま解ける場合に相当する。この機能は単一化のモードを切り換えるこ

により実現可能であるが、指定された変数が他のリテラルと共有されていた場合の処理方法など詳細については未検討である。

・全称変数

```
X ! p
```

X の構造内に現れる変数の任意の値に対してp が成りたつときに限って成功とする。

・存在変数

```
X ^ p
```

X の構造内に現れる変数の適当な値に対してp が成りたつときに成功とする。特に指定がない場合もこの扱いとなる。

・guarded 述語

```
p :- g1 | b1
p :- g2 | b2
p :- g3 | b3
```

ガード部(g1,g2,g3)のいずれかの解が完全に求まると他の定義節の処理を打ち切る。p を頭部に持つすべての定義節にガードがなければならない。

3.3 否定的知識

本来 Horn 節は肯定的知識しか扱えないが、論理型言語の定義節を書き換え規則としての解釈する場合には、完全性は成り立たないものの、否定的知識を肯定的知識と対等に扱うことができる。否定的知識はリテラルの前に－をおいて表わすことにすると、例えば、

```
 fly(X) :- member(X, [swallow, eagle]).
-fly(X) :- member(X, [penguin, ostrich]).
```

通常のゴールに対しては、肯定的定義節と否定的定義節をともに適用する。もし、否定的定義節が、変数に関して全称的に成功した場合には、そのゴールの失敗と見なし、処理を打ち切ることができる。もとのゴールにおいて全称変数となっていた場合には任意の値に対してでよい。また、探索範囲を限定するため、肯定的定義節または否定的定義節のみを適用するよう指定もできる。

```
?- +fly(X).    肯定的定義節のみを用いて解く
?- -fly(X).    否定的        〃
```

従来の Prolog で用いられていたような操作的否定（閉世界仮説にもとづく否定）はゴールの成功と失敗を反転することで従来どおり定義される。これらを組み合わせて用いることにより、いわゆる Default Reasoning[4] の機能が実現できる。

```
not p          p が失敗すれば成功
\+ p           +p        〃
\- p           -p        〃
```

```
ex.    fly(X) :- bird(X),\-fly(X).
```

3.4 高階・メタ機能

言語の記述能力の指標として、その言語自身の自明でない処理系がその言語によって記述できるかどうかが、とりあげられることがある。このような場合をはじめとして、2階以上の記述や、メタな判断を取り扱いたい場合は、各種の応用プログラムにおいてしばしばあらわれる。これらの機能として、用意しておくと役に立ちそうなものは数え上げればきりがないが、最少限、次のようなものが必要であろう。

・解の寄せ集め

```
setof(X, P, S)
```

P を満たすX 全体をまとめたものをS とする。重複の除去や、S に値が束縛される時期に関しては、ここでは議論しない。

・変数性検査

```
var(X)
```

X が未束縛である場合に限って成功となる。逐次AND 以外でゴールリテラルが結合されている場合における正しいセマンティックスの与え方については、議論の余地があろう。

3.5 多重世界

単にいろいろな仮定の下で問題を解いてみる場合のように、外界との交渉や副作用がない範囲内ならば、多数の定義節集合の上にたつゴールを並列に処理可能である。このような場合、どのゴールがどの世界（定義節集合）で解かれるべきかが明確になっていなければならない。ここでは、ゴールは常に、それを解くべき定義節集合へのポインタを暗黙のうちにひきずっているものとする。そこで、多重世界としては、現在の世界に適当な定義節を付け加えたり取り去ったりして新しい世界を作ったのち、ゴールと解くべき世界とを対にして与えてそれを解かせる方式を考えている。この方式では AND で結合されたゴール間でしか世界を共有できないが、並列処理の環境における通信などを考えた場合、妥当な方式であろう。

・現在の世界の参照

```
current_world(World)
```

現在このゴールを解いている世界（定義節集合）が World と単一化される。

・新しい世界の設定

```
create_world(Def, World [,Oldworld])
```

```
    assert_world(Def, World [,Oldworld])
    asserta_world(Def, World [,Oldworld])
    retract_world(Def, World [,Oldworld])
    abolish_world(Pred, World [,Oldworld])
```

いずれも

```
    Def:       定義節または定義節のリスト
    World:     新しい世界
    Oldworld:  もとの世界
    Pred:      述語名
```

Oldworld は省略すれば現在の世界が仮定される。Def を与える際に、変数を quote できる(DEC-10 PROLOG の numbervers の逆変換)ことが望ましい。

・指定した世界での問題の解決

```
    solve(Goal, World)
```

Goal を World の上で解く。

4. 時間遷移世界

入出力のように副作用として外界との交渉を持つ場合や、共有資源の排他制御などを記述したい場合には、時間の概念がぜひとも必要になってくる。並列処理を記述する際に最も難しい点は、このような時間の概念をきちんとおさえることであり、従来データフローや関数型プログラミングにおいても、副作用を取り扱うことは難問とされてきた。これらの言語においてとられてきた解決法は、何らかの意味で時間を表わすものを、他の入力と同等な1つのデータとしてプログラムに与え、それに同期して計算を進めてゆくという方式であった。副作用と計算が同時に進行するので、いわばリアルタイム処理方式といえる。論理型言語に並行プロセスの同期メカニズムを組み入れようとした試みである Concurrent Prolog や PARLOG [5] も基本的にこの方式にもとづいている。

しかし、各種の副作用や同期関係をすべてデータ依存関係によりあらわそうとすると、同期のために導入された、本来の処理とは直接関係のないデータがあちこちを飛びかうことになり、プログラムの理解しやすさ、実行の効率などの点において好ましくない。そこで、ハードウェアの同期回路において一般にクロック入力を通常の入力とは別扱いにして考えることが多いように、論理型言語においても時間関係は別個に取り扱ったほうがよい。

時間関係を取り扱うための論理としては時相論理と呼ばれるいくつかの体系があり、仕様を記述する目的などに実際に使われはじめている。そのなかで、 Prolog が Horn 節をそのままプログラミング言語として用いたように、時相論理をそのままプログラミング言語として用いようとする試みとして、 Moszkowski らによる Tempuraがある[6] 。Tempura は Interval Temporal Logic により各部の動作を記述しておき、それにより与えられた仕様を実現することが可能かどうかを検証するという立場でプログラミングするものである。本来、ハードウェアの仕様を記述し、設計がそれを満たしているかの検証に用いる目的の言語なので

[7]、プログラムに書かれていないことは一切何もいえないなど厳格すぎる面があり、通常の応用分野に用いるには適当でない。また、その実行は組み合わせ的な計算を要する。ここでは、Tempura のアイデアをもとにして、従来の Prolog のデータベース機能の裏付けを行なった時相論理型言語を提案する。

・時間遷移述語

```
:- time_variant(p).
```

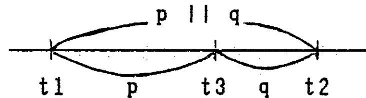
時間とともに真偽値を変える可能性のある述語はあらかじめ宣言しておく。これらの述語を定義する際に、節の右辺は空でなければならない。また、Tempura の場合と異なり、時間遷移述語の定義は変更が加えられるまでは、そのまま保たれる。定義の変更は、DEC-10 PROLOG に従って assert, retract で行なう。

・時刻オペレータ

p || q　etc.

外界でのある時刻を意味し、その時刻より前ではp が成立し、後ではq が成立することを表わす。すなわち、

$$(p \,||\, q)[t1<t<t2] \equiv \exists t3 \ (p[t1<t<t3] \wedge q[t3<t<t2])$$



また、遷移述語の定義の変更はこの時刻に一括して行なわれるものとする。一般に、外界で副作用としてあらわれるべきこのような時間順序と、計算機内で問題を解決する順序とは一致する必要はない。身近な例では、旅行の計画をたてるときには、通常、目的地に到着すべき時刻から順にさかのぼって、乗るべき列車の時刻や、どこで食事をとるかを決めてゆくことが多い。そこで、計算機内部でどの順に問題を解決するべきかの指定（アドバイス）として、次の３通りを考えている。

p && q	解の探索も p→q の順に行なう
p || q	解の探索順序は規定しない
p !! q	解の探索は q→p の順に行なう

また、論理型言語の処理においては、通常複数の選択肢が生じるが、副作用を取り扱うためには、選択肢を残したままでは都合が悪い。そこで、時間遷移述語を含むゴールの実行は３つのフェーズにわけて成される。

・第１のフェーズは論理的な解を求めるフェーズであり、旅行の例では時刻表を見ながら計画を練ることに相当する。

・第２のフェーズは解を１つに絞るフェーズであり、通常、ガード節がこの機能を果たす。旅行の例ではいくつか立てたプランのうちから１つを選び出すことがこれに相当する。

・第３のフェーズは副作用を外界に示すフェーズであり、実際に列車に乗ったり

食事をしたりすることがこれに相当する。

これら３つのフェーズは必ずしもこの順になされる必要はない。問題を解いている途中で解が１つに絞られたならば、そこまでに確定した副作用を外界に示してしまってかまわない。

実際に解を探索してゆくには、与えられたゴールを次のような公理にもとづいて変形してゆく。

(1) $p \,||\, (q \,||\, r) \equiv (p \,||\, q) \,||\, r$

(2) $p \wedge (q \,||\, r) \equiv (p \wedge q) \,||\, (p \wedge r)$

(3) $p \,||\, (q \vee r) \Leftarrow (p \,||\, q) \vee (p \,||\, r) \vee (p \,||\, q \,||\, r) \vee (p \,||\, r \,||\, q) \cdots$

(4) $(p \,||\, q) \wedge (r \,||\, s) \equiv [\,(p \wedge r) \,||\, (p \wedge s) \,||\, (q \wedge s)\,] \vee [\,(p \wedge r) \,||\, (q \wedge r) \,||\, (q \wedge s)\,]$

このうち(3)は同値変形ではないので、どれだけ展開するかにより解を求める能力が影響される。また、次のような無矛盾性の規則を利用して時間遷移述語の更新関係を整理する。

- ある時刻においてp が assert された場合には、その直後においてp は成り立つ。
- ある時刻においてp が retract された場合には、その直後においてp は成り立たない。
- ある時刻において時間遷移述語p が成り立つためには、その時刻より前からp が成立しているか、またはその直前においてp が assert されなければならない。
- ある時刻において時間遷移述語p が成り立たないためには、その時刻より前からp が成り立たないか、またはその直前においてp が retract されなければならない。

このような変形を続けてゆくと、最後に時間遷移述語の更新の時間的系列のみが残る。これが、求める副作用の実体である。

簡単な例を示そう。次にあげるのは、カウンタの例である。

```
:- time_variant(counter/1).
init :- assert(counter(0)).
up   :- retract(counter(N)), M := N + 1, assert(counter(M)).
down :- retract(counetr(N)), M := N - 1, assert(counter(M)).
```

カウンタを初期化したのち、２度カウント・アップした場合のカウンタの値は次のようにして求められる。

```
?- init || up || up || counter(X).
→  assert(counter(0)) || up || up || counter(X)
→  assert(counter(0)) ||
     retract(counter(0)), M := 0 + 1, assert(counter(M)) ||
     up || counter(X)
→  assert(counter(0)) ||
     retract(counter(0)), assert(counter(1)) ||
     up || counter(X)
            ...
→  assert(counter(0)) ||
     retract(counter(0)), assert(counter(1)) ||
     retract(counter(1)), assert(counter(2)) ||
     counter(2)   [ X = 2 ]
```

これに対して、

```
?- init || up, up || counter(X).
```

の場合には、

```
→  assert(counter(0)) || up, up || counter(X)
→  assert(counter(0)) ||
     retract(counter(0)), M := 0 + 1, assert(counter(M)),
     retract(counter(0)), X := 0 + 1, assert(counter(X)) ||
     counter(X)
→  assert(counter(0)) ||
     retract(counter(0)), assert(counter(1)),
     retract(counter(0)), assert(counter(1)) ||
     counter(1)   [ X = 1 ]
```

となり、結果が異なる。これは副作用と時間との関係を適切に示しているものと考えられ、好ましい性質であるが、

```
?- init || up, down || counter(X).
```

の場合には、[X = 1] と [X = -1] の２つの解をもつことになり、応用分野によっては好ましくない。

以上のように、ここでは、DEC-10 PROLOG 流の assert, retract により時間遷移世界を構築する場合にについて検討したが、最後に示した例のように、不自然な場合もあるので、大域変数として扱うほうがよいかもしれない。その場合は萩谷氏の研究[8] が大いに参考となろう。

５．プリ・プロセッシング

現在の Prolog プログラムをながめると、固定回数の繰り返しに対して、記述を簡単にするため、変数を用いたループにしている場合がかなりある。このような場合に、適当な記法により、実行前に展開する機能があれば、実行時の無駄がかなり減らせる。プログラム変換の研究が近年進んでいるので、論理型言語の美しさを損なうことなく、高度なプリ・プロセッシングがおこなえるものと期待している。

６．おわりに

論理型言語を宣言的立場に立って見直しつつ、
(1)解の探索が爆発するのを防ぐため、逐次探索を行なうか並列探索を行なうかの指定が可能、
(2)否定的知識や複数の定義節世界にもとづいた推論が可能、
(3)時相論理にもとづき、外界に対する副作用と内部の計算とを明確に分離、
などの特徴をもつ、新しい論理型言語について検討を行なった。

＜参考文献＞

[1] McDermott,D.: The Prolog Phenomenon, ACM SIGART Newsl., No.72, July, 1980.

[2] Shapiro,E.Y.: A Subset of Concurrent Prolog and Its Interpreter, Technical Report TR-003, ICOT, Feb. 1983.

[3] 相田　仁，後藤厚宏，田中英彦，元岡　達：論理型プログラムの書き換えモデルにおける read-only annotation の活用について，情報処理学会第24回全国大会 4H-5，1984 年３月

[4] Bobrow,D.G. and Hayes,P.J. ed.: Special Issue on Non-Monotonic Logic, Artificial Intelligence, Vol.13, No.1-2, North-Holland, Amsterdam, Apr. 1980.

[5] Clark,K.L. and Gregory,S.: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, Imperial College, London, May 1983.

[6] Moszkowski,B. and Manna,Z.: Temporal Logic as a Programming Language, Proc. of Parallel Computing 83, West Berlin, Sept. 1983.

[7] Moszkowski,B.: A Temporal Logic for Multi-Level Reasoning about Hardware, Report No. STAN-CS-82-952, Dept. of Computer Science, Stanford University, Dec. 1982.

[8] 萩谷昌己：Theory of Modal Logic Programming，ソフトウェア基礎論研究会資料 9-4，1984年６月．