1D-2

並列処理向き拡張論理プログラミング言語

相田 仁，田中英彦，元岡 達
（東京大学工学部）

1．はじめに

知識を計算機内部に表現し，それに基いて推論を行なわせるための核言語としてPROLOGが着目されている．しかし，現在用いられているPROLOGでは，否定の概念がとり扱いにくいなど，論理式をhorn節に限定したことに由来する問題点，及び，項や式の配列順によっては実行が停止しなかったり，効率が大きく影響を受けるなど，実行を記述順に限定したことに由来する問題点を有している．これら2種類の問題点のうち後者については，実行を並列に行なうことにより解決可能であると考えられ，すでに我々の研究室で処理系を試作し，報告した[1]．ここでは前者に関して，言語の拡張を提案し，並列処理と組み合わせることにより，知識表現により適した言語とすることが可能であることを示す．

2．PROLOGにおける否定の扱い

PROLOGは，本来，horn節のうち正のリテラルを含むものを知識として蓄えておき，負のリテラルのみから成るものに対して，それをgoalとして証明するものであった．従って，「...であるとき...である」という肯定的な知識を表現することは可能であるが，「...であるときは...でない」というように否定的内容を含む知識は，本来のPROLOGではとり扱うことができない．そこで，現在のPROLOG処理系ではcut symbolを用いることによりnagation as failureとして否定をとり扱っている．積木の世界を例にとろう．

```
On(a,b).
On(b,c).
Above(X,Y):-On(X,Y).
Above(X,Y):-On(X,Z),Above(Z,Y).
Below(X,Y):-Above(Y,X).
Below'(X,Y):-!,fail.
Below'(X,Y):-Above(X,Y),!,fail
Below'(X,Y).
```

ここで，Aboveの引数の入れ換えにより定義したBelowでは，

```
?-Below(b,a).   success
?-Below(b,X).   success,X=a
```

となるのに対し，Aboveのnegation as failureにより定義したBelow'では

```
?-Below'(b,a).  success
?-Below'(b,X).  failure
```

となり，これは直感に反する．また，

```
?-On(X,c),Below'(a,X). success,X=b
?-Below'(a,X),On(X,c). failure
?-Below(a,X),On(X,c).  success,X=b
```

に見られるように，Belowの定義を知らないとリテラルの配列順を決定できないことになり，package等の導入にも障害となる．

3．否定定義式の導入

PROLOGで定義式を正のリテラルを含むhorn節に限定した理由を振り返ってみると，resolutionの相手が限定され，決定的手続きで解を求めることが可能となるためであった．しかし，論理式から離れて，単に問題解決の手続きとしてPROLOGをながめた場合，定義式の左辺はこれをデータベースから取り出す際のkeyとしての役割を果たしているだけであり，そのリテラルが正であるか負であるかは重要ではない．

そこで，従来，PROLOGの定義式に設けられていたリテラルの正負に関する制限をとりはらい，任意の節を定義式として許すようにする．

```
Above'(X,Y):-On(X,Y).          ①
Above'(X,Y):-Above(X,Z),On(Z,Y). ②
¬Above'(X,X).                  ③
¬Above'(X,Y):-Below(X,Y).      ④
```

例えば④の定義式は，「Below(X,Y)であるときAbove(X,Y)は成り立たない」という知識を表現するものである．なお，この式は，論理的には

Below(X,Y):-Above'(X,Y). ⑤

と書き換えることができるが，④式を書いた場合，特に指定のない限り，goal内のAboveというリテラルをresolveするためだけに用いて，Belowをresolveする目的には用いないものとしておく．

このように，否定に関する知識をexplicitに定義することにより，正のリテラルと負のリテラルを全く同様に扱うことが可能となる．

?-Above'(a,X),On(X,c). success,X=b⑥

さらに，並列実行の下では否定に関する知識を用いてプログラムの停止性を高めることが可能となる．例えば，

?-Above'(c,a). ⑦

は①，②の定義の下では無限ループに陥って停止しない．これに対して③，④の定義を付け加え，かつ⑦のgoalが与えられた場合，

?-¬Above'(c,a). ⑧

を同時にgoalとして設定し，並列に実行させると，こちらはsuccessするので，この結果に基いて⑦の実行を強制終了させることができる．

一般に，あるgoal Aが設定されると¬Aもgoalとして設定して並列に実行を行なう．このとき，一方がある条件の下でsuccessするならば，その条件の下で，他方は決してsuccessしないはずなので，対応する枝を刈り込むことができる．このように2つのgoalから得られた情報を互いに利用しあうことにより，片方のみの実行では停止しないような枝を刈り込んで，プログラムの停止性を強化することが可能である．

4．non-monotonic logic

否定に関する定義を許すと，あるgoal Aについて次の3つの状態が存在することになる．

(1)Aが証明される．

(2)¬Aが（explicitに）証明される．

(3)Aも¬Aも証明できない．

従来のPROLOGでは(2)の状態は存在しなかったので(3)の状態をもって(2)の状態と見做していたわけである．これに対し，(3)の状態の取り扱いをプログラムで指定できるようにすることにより，defaultやinheritanceの概念を取り入れることができる．

今MAという記号で¬Aは証明不能であること，すなわちAが上記(1)または(3)の状態であることを表わすことにする．これにより「通常の鳥は飛ぶ」というdefaultは次のように表わされる．

Fly(X):-Bird(X),MFly(X).

Bird(robin).

Bird(penguin).

¬Fly(penguin).

今，「robinは飛べるか」というgoalが与えられたとする．

?-Fly(robin).

すると，式からまずBird(robin)がcheckされ，次いで，MFly(X)より¬Fly(robin)がsubgoalとして設定される．しかしこれに関するexplicitな記述はないのでこれはfailし，その結果，MFly(robin)はsuccessとなる．よってFly(robin)もsucceedする．一方

?-Fly(penguin).

については，¬Fly(penguin)に関するexplicitな定義があるのでfailureとなる．

5．拡張言語と論理との対応

PROLOGにおいては論理式をhorn節に限定していたため，探索の順序の関係で停止しないことはあっても，successとfailureが入れ代わることはなかった．しかし2節で述べたような言語の拡張を行なうと証明手続きがもはや言語に対して完全ではなくなり，論理的には導出可能であるにもかかわらずfailする可能性がある．ただし，誤った証明が行なわれることはないので，導出不可能なものがsuccessとなることはない．

これに対して，3節で述べたようなsubgoalのfailureに基いて後の推論を進める機能を付け加えると，誤った推論がそのまま進められる可能性があり，推論機構の能力との関係において検討する必要がある．

参考文献

1.相田，田中，元岡：並列ＰＲＯＬＯＧシステム“Ｐａｒａｌｏｇ”の性能測定，第２４回情報処理全国大会　５Ｄ－５，１９８２