多重世界機構を用いた非単調論理

岸川徳幸　・　中川裕志
（横浜国立大学　工学部　情報工学科）

１．まえがき

否定的知識の扱いは現在のところ閉世界仮説にもとづくものが大多数である。これは証明システムの立場にたてば Nagation as Failureに外ならない。しかし次の例を考えてみよう。（1）「あなたのお父さんの電話番号は？」　と質問されたら即座に「１２３－４５６７。」（2）「Ｆ社に今年入社したＫ君の勤め先の電話番号知ってる？」「・・ええと、たしか９８７－６５４３か６５３４かだったと思うんだけど。」（3）「ミッテラン大統領の電話番号ご存じですか？」　即座に、「知りません。」

（1）はdeductive reasoning が成功する場合である。(2) はdeductive reasoning が失敗する場合である。しかるに(3) は、“フランスの電話番号は一切知らない”、という否定知識を用いた推論による。ところで、否定知識まで含めた推論を考慮すると、(2) の場合は、実はまず、例えば“Ｋ君自体を知らない”などの否定知識が無いことにより、否定知識の側から推論して失敗し、次に自分が知っている電話番号という肯定知識の側から推論しても失敗する場合である。人間の行なっている推論方法に近づこうとすれば、やはりこの三つの場合を明確に区別して扱うことを考えなければならない。

この論文で述べるシステムでは、上の三つ区分を文献1)のＵranus に導入するために証明論的なNOTでなく論理的なNOT を採用し、陽にNOT 表現可能にする。しかし、このときconsistency check をするべきことは明白であるが、すべての述語に関して行なっているのではメモリと時間の点で破綻してしまうのでユーザが特定のgoalを指定する。また、矛盾が起きた場合その解消を行なうが、原因を探索する方法として多重世界機構との関連から浅い世界から深い世界へとみてゆく方法を採っている。

２．否定と矛盾

2.1 否定の表現

人間の持つ概念は、ときどき“ひらめき”“矛盾の発見”などにより再構成されている。このような知識の動的な変化（すなわち学習あるいは知識獲得）を扱おうとする場合、従来の Negation as Failure は必ずしも十分でない。このことは文献2)でも主張されている。そこで、我々はこのような知識ベースの動的な変化を計算機上で実現するにあたって知識ベースに対するQ/A が行なわれているQ/A モードと、知識ベースの再構成が行なわれている再構成モードに分けて考える。当然ここでは、モードの変化の契機として“矛盾の発見”を考える。つまりPrologの閉世界仮説とは異なる立場をとる。そのために、肯定と否定が共に証明される“矛盾”を発見するために否定記号NOT を陽に表現する。例えば、述語(P *X)の否定は（NOT (P *X)）と表わす。

ところで、後ろ向き推論を行なうPrologのような論理型言語では、上記のように否定を明示的に表現した場合、矛盾を発見するために、まずそのgoal自

体を実行し、それが証明された後にそのgoalの否定が証明されるかどうかチェックする。しかし、これをすべての述語について行なうと多大なメモリ（空間）と時間を要し、組み合わせの爆発を起こす。これを避けるために、ここで提案するシステムは、consistency check を行なうgoalをユーザが選択的に指定できるように、次のように明示的にＣという記号をつける。

```
((head) … (C (P A))  … )
```

もし、Ｃの引数であるgoalが矛盾を生じたときは、そのgoalの証明過程の情報をもって４節で詳しく述べるDependency Directed Backtraking （ＤＤＢ）に入りその矛盾を解消する。それ以外は、通常のPrologの実行とかわりない。もちろん、Ｃは次のように

```
((head) … (C (P *))… )
((P *)  … (C (Q *))… )
```

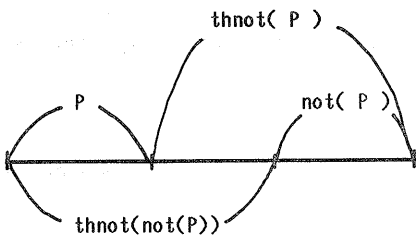
ネストした場合も実行可能である。

2.2 論理的解釈

このシステムは論理的否定を陽にあつかうのでNon-monotonic logic も表現できるのではないかと考えられる。論理的not に対して、単純に閉世界仮説を具現したPrologにおける証明不可能の意味での否定（thnot と表わす）は、

```
thnot(p)  ← p ! fail .
thnot(p)  ← .
```

で表現できる。図で表わすと



となる。したがってモダルオペレータＭ（否定が証明できないならば真）は、

```
M(p)  ← thnot(not(p)).   …… (1)
```

である。ところがもしp , not(p)のどちらも証明される場合、すなわち矛盾が生じた場合はどのようになるのか。もちろんp , not(p)が同時に証明されれば、古典論理においてはすべてが定理になるので、このシステムにおいてもそれは避けなければならない。しかし、thnot はその定義からわかるようにthnot(p)ならp のみ、thnot(not(p)) ならばnot(p)のみの述語呼び出しを行なうだけである。これでは、consistensy check などできない。したがって、そのための述語Ｃを用意する。

```
C(p)  ← p , ccheck(p) .

ccheck(p) ← not(p) , (DDB) .
ccheck(p) ← .
```

Ｃの働きは、p の述語（ pの引数がbindされたものだけについて）のconsistency check をおこなう。もし、inconsistentならば、その矛盾解消のためのＤＤＢが呼び出される。明らかにp の述語の真偽はＣをつけた場合もＣをつけない場合も全くかわりない。

さて、このＣを用いることでconsistency checkの機能を持つより強力なモダルオペレータＭが次の

ように定義できる。

M(p) ← thnot(C(not(p))). …… (2)

この真理値表は、○をSUCCESS ×FAILとして次に示すと、

p	not(p)	C(not(p))	thnot(C(not(p)))
○	○	−	−
○	×	×	○
×	○	○	×
×	×	×	○

となる。ただし、−はconsistency check にかかりＤＤＢが起動され再実行されるためこの質問を発生した時点での状態では、成功・失敗が不明である状態である。(1),(2) のMは、goalにつくという制限があるために、文献4)で述べられているMを論理記号として導入するNon-monotonic logoc 全体はシュミレートできないが、その部分系であるdefault logic 文献5)は、その推論の向きが異なる以外はよく似ており、例えば、normal default

$$\frac{p(x) : M\ q(x)}{q(x)}$$

もそのまま、

q(x) ← p(x) , M (q(x)) .

と表現できる。Mの定義の(1) では、矛盾の扱いの点で弱くそれにくらべ、(2) の定義はより強力で非単調性をあつかうSYSTEMであるdefault logic をシュミレートできるものと思える。

2.3 変数の扱い

モダルオペレータMをそのまま

M(p) ← not(p) ! p , (DDB) .
M(p) ← .

と定義することも考えられる。しかし、p が定数の場合は問題は生じないのだが、この定義では、p が一般の述語でありそこの引数にバインドされていない変数が含まれる場合もあり、その場合その変数にPrologと同様の順番で値をバインドして返すことができない。そこで、consistency check と証明論的 NOTと独立させそれを組み合わせて前節のようにする方法を採り、変数が含まれている場合にも対応できるようにした。すなわち、バインドされていない変数に、consistentな解をバインドした結果が得られる。したがって、モダルオペレータMの引数の述語に対して、変数が許されることを利用して goal

← M(p(x)) , fail .

を実行させれば、部分系 pに関して無矛盾にすることができる。

2.4 多重世界での利用法

ところで、このMが使えるPrologで実行できる述語は、概念がTREE状に表現されている世界の階層上で、副作用なしでかつWITH、WITHINを使わない述語を実行している状況である。副作用なしの述語でなければならないのは、そこでASSERTされると、その依存関係を記録し、もしその述語をサポートしているものが、RETRACT されると、ASSERTした述語も同時にRETRACT しなければならないという理由であり、WITH、WITHINを使わないのは、それ自体が概念の構成を意味しており、ＤＤＢ時の原因探索の世界のリ

ストにのらないためである。

3．多重世界におけるDependency Directed Backtracking

3.1 Dependency Directed Backtrackingの動作原理

矛盾を発見した後は、次に矛盾の解消につとめなければならない。その矛盾解消プロセスＤＤＢが起動されると、矛盾の原因を探索するが、現在までのＴＭＳなどの非単調論理が実現されたシステムでは、その原因の選択順序について強力なものはない。そこで、多重世界機構との関連から意味論を持ち込む。人間は何か間違いを見つけると、まず最も疑わしいものとして最近の知識または特殊な知識をあげる。それは、その知識が根拠となっているものが少なく、知識体系の大きな変更がいらないためであろう。これを多重世界の言葉でいえば、世界の階層の浅い（TREEの葉に近い）ほど疑わしいものだ、となる。したがって、矛盾原因は、浅い世界から順に調べていくのが妥当である。

ＤＤＢプロセスのためには、p がなぜ成立するのか、not(p)がなぜ成立するのかについて知らなければならない。そこでＣオペレータの中でのnot(p)およびp の証明過程を記録し、その中から適当な原因を選ぶようにしなければならない。この矛盾解消には、次の四つの情報

（１）　成功したgoal
（２）　clauseが定義されている世界
（３）　clauseの定義自身
（４）　あるclauseの消去後のbacktrack
　　のための情報

が必要であり、(2) は、これをキーにして矛盾原因の選択順序を決め、(3) は定義の削除に用い、(4) は定義の削除後のバックトラック点を示す。この(1)-(4) の情報が、次のようなリスト構造をしている。

（（①　（②　．　③）　④）　．　...）

この証明情報を作るために、Ｃプロセスの中における実行は、goalを引数とした次の述語PTRACEを呼ぶ。

```
((PTRACE *P)
 (PTRACE-IN *P)
 *P
 (使われた定義を取り出し*DEFにバインド)
 (PTRACE-OUT&BACK *P *DEF))
((PTRACE *P) (PTRACE-FAIL))

((PTRACE-OUT&BACK *P *DEF)
 (PTRACE-OUT *P *DEF))
((PTRACE-OUT&BACK *P *DEF)
 (PTRACE-BACK *P))
```

PTRACE-IN ― PTRACE-FAILはLISP呼び出しで、その間は、LISP変数BACK、NEXTBACKで必要な情報を受け渡す。PTRACE-IN でバックトラック情報(4) をBACKに設定し述語自体は必ず成功し、そののち実述語*Pを呼ぶ。実述語*Pが成功するとPTRACE-OUTに入り、バックトラックの時のためにBACKをNEXTBACKに退避させ、必要な情報(1) - (4) を証明情報に入れ、BACKのトップを取り払い、述語は成功する。この直後バックトラックをおこすとPTRACE-BACK に入り、BACK、NEXTBACK、証明情報を設定しなおし、PTRACE-BACK はFAILして*Pの再実行へいく。*Pのすべての解が出つくすとPTRACE-FAIL でBACKをNIL にし、述語自体FAILする。

こうして作られる証明情報が矛盾が発生したときに使われるが、そのときの対処の方法として一番簡単なある述語をRETRACT するときを考える。そのためのその定義自体を、情報(2) を用いて(3),(4) の情報を証明情報から取り出す述語GETDCFは、

```
((GETDCF NIL *W *CF) ! (FAIL))
((GETDCF ((*GOAL (*W . *CF) *BACK)
          . *OTHERS)
         *W
         (*CF . *BACK)))
((GETDCF (* . *OTHERS) *W *CF)
 (GETDCF *THERS *W *CF))
```

と定義され、GETDCFの第二引数に世界のリストを深さの浅い順に与えるために、

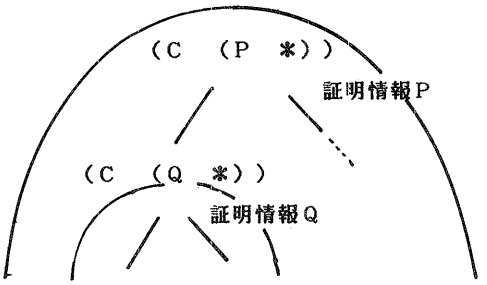
```
((GETINFORMATION *SUC *WL (*CF . *BACK))
 (MEMBER-OF *W *WL)
 (GETDCF *SUC *W (*CF . *BACK)))
```

が用意されている。この述語を実行して*CF にでる述語定義をもちいRETRACT を実行する。その後、*BACK にバインドしている情報をつかい適当なところから実行を再開するが、そのときもう一度定義されている述語を取り出し再設定をする。それは、使われている定義自体RETRACT の実行される対象になるためである。

さて、Cが次のようにネストした場合、

```
((head) … (C (P *))… )
((P *)  … (C (Q *))… )
```

どう変わるのだろうか。上記の場合を図にしてみると、



となる。(C (P *)) の証明情報Pは、上の木のうち証明情報Q部を除いた部分である。すなわち、述語Pが矛盾を起こしたときその原因探索を述語Q部の証明過程まではおよぼさないという意味である。これは、Prologはdepth first で実行されるから、先に深い方である述語Qの矛盾が解消されその真偽は一意に決定されてしまい、そこより浅い述語Pでは再び探す必要がないことに対応している。

3.2 Dependency Directed Backtrackingを使った実例

矛盾原因を、注目している世界からStandardな世界（最も深い世界）へと探していく例を次にあげる。

```
manmal世界
    ((FLY *) (BIRD *) (HAS-STRONG-WING *))
    ((NOT (FLY *)) (BIRD *) (ABNORMAL *))
bird世界
    ((ABNORMAL *) (PENGUIN *))
    ((BIRD JILL))
penguin 世界
    ((PENGUIN JILL))
    ((HAS-STRONG-WING JILL))
```

のときユーザからの質問

```
(WITHIN (PENGUIN BIRD MANMAL)
        (C (FLY JILL)))
```

を行なうと、(FLY JILL),(NOT (FLY JILL)) が共に証明でき矛盾が発生するので、ＤＤＢに入る。ペンギンJILLは飛ばないので(FLY JILL)のgoalの証明過程のなかにあやしげなものがある。世界の階層が

```
              IS-A          IS-A
MANMAL  ←——  BIRD  ←——  PENGUIN
```

の順になっているので、ＤＤＢから次のように順にきいてくる。

```
((HAS-STRONG-WING JILL)) Y or N:
((BIRD JILL)) Y or N:
((FLY *) (BIRD *) (HAS-STRONG-WING *))
                    Y or N:
```

ここで注目すべきは、ルールもその原因として挙げられていることである。矛盾原因がわかったあとの定義の処理は、ＤＤＢという述語を変更することにより、ユーザが自由に扱え、これだけでなく世界の構造を大きくかえる述語も作成し実行させることが可能である。いまそれを世界を二分割するように作られていたとすると、一番目の((HAS-STRONG-WING JILL))にY と答えると((HAS-STRONG-WING JILL))が存在しない世界 PENGUIN' が作られ、そして、世界の構造が、

```
MANMAL  ←  BIRD  ←  PENGUIN
                 ←  PENGUIN '
```

となる。このような矛盾解消法は、単に述語を IN と OUTに分割する文献3)のＴＭＳより強力であると考えられる。

４．おわりに

今回作成したシステムは、二つの利用の仕方があるように思われる。一つは、モダルオペレータＭを簡単に表現できかつconsistency check 機能をもっていることより、非単調論理の実験システムとして、そしてもう一つは、矛盾発生時の人間の考えに即した原因の効率的探索とその解消のための融通性をもつ概念学習のツールとしてである。

第一の点については、これからはPrologのnot との関連もみながらもっと広い非単調論理をを実現するシステムへと発展させてみたい。

また第二の点については、現在までの研究で知識ベースの動的変化を表現するのためのツールはそろってきたように思われる。ここで述べたシステムの、ある述語の成り立つ世界と、成り立たない世界に二分する能力だけでもＴＭＳなどよりは知識表現、学習に関してはかなり強力な手段を提供しており、これをどのように使いこなすかが今後の課題である。

【参考文献】

１）中島、戸村、諏訪
「Prolog/KR からＵranus へ」
第36回知識工学と人工知能研究会資料(1984)

２）国藤、北上、宮地、古川
「論理型言語Prologによる知識ベースの管理」
Proc. of the Logic Programming Conference
(1985)

３）J.Doyle "A Truth Maintenance System",
Artificial Intelligence 12 (1979)

４）D.McDermott and J.Doyle
"Non-monotonic Logic I"
Artificial Intelligence 13 (1980)

５）R.Reiter "A logic for default reasoning"
Artificial Intelligence 13 (1980)