知識工学と人工知能　36－2
（１９８４．９．１９）

Prolog/KR からUranusへ
－　多重世界機構の拡張　－

中島秀之　戸村哲　諏訪基
（電子技術総合研究所）

1.はじめに

Prologを知識表現用に拡張した言語がProlog/KR［中島1982］であるが、我々はこれをLisp Machine上に移植するとともにUranusとして更に機能拡張を行なった。それにはwindow system の利用等も含まれるが、ここでは多重世界機構の拡張に話を限る。なお、Uranus(Uranus, Universal Representation-Aimed Novel Uranus System)の名はギリシャ神話に登場する世界の支配者である神から取った。これは多重世界機構により、実世界の記述が可能になるとの望みからである。

Prologには知識表現にとって本質的な機能のうちいくつかが欠けている。Prolog/KR は多重世界機構を持っており、これによってPrologと知識表現のギャップを埋めることができる。多重世界機構を用いることにより以下のようなことが可能になる。

(1) ひとつの概念に関する主張（assertion)を一箇所に集めることができる。この場合、世界の名前が記述される概念の名前に対応し、フレーム理論によるものと似た定式化がなされる。概念の階層構造内での属性の継承の制御は十分強力である［Nakashima 1984；中島1984］。

(2) 状態の遷移を時系列としてとらえる場合に、各々の世界を、ある特定の時刻における世界の断面と考えることができる。

ここでは、これらの多重世界機構の果たすべき役割と、世界の関係を操作するためのプリミティブに関して考察する。

2.多重世界機構

Uranusは動的に結合される多くの互いに独立な世界よりなる。各々の世界で定義された述語は外側からは見えない。ある世界に入るにはwithというプリミティブを用いて、

```
(with　世界名　.　述語呼出し)
```

のようにする。述語呼出しの部分には普通の述語の呼出しの他、定義など任意のものが任意個書ける。世界の名前は任意のものを使ってよく、その世界が既に存在すれば、それが使われ、存在しない場合には新たに作られる。

例えば、Japaneseの世界でfirst-numberという述語を実行するには

```
(with Japanese (first-number *x))
```

とする。おそらく*x= いち　となるであろう。同様に、

```
(with English (first-number *x))
```

を実行すると*x= one となり、

```
(with French (first-number *x))
```

なら*x= unという具合である。このように同じ述語でも、世界ごとに定義が変わりうる点に注意されたい。

述語呼出しの部分には普通の述語の呼出しの他、定義など任意のものが任意個書ける。従って、先程のfirst-numberを定義するにもwithを用いることになる。

```
(with Japanese (assert (first-number  いち))
               (assert (second-number に))
                    …)
(with English  (assert (first-number one))
               (assert (second-number two))
                    …)
```

世界は何重にもネストして用いることができ、その場合内側の世界からは外側の世界の定義が（原則として）全部見える。この世界間のネスティングは定義時ではなく実行時に決まる。例えば世界Aではpという述語が、世界Bではqという述語が定義されているとすると

```
(with A (with B (and (p *x) (q *x))))
```

においてはp、qはそれぞれ世界A、Bの定義を参照することになる。

同じ述語がいくつかの世界で定義されている場合には、内側のものから順に試みられる。例えば述語pが、世界A、B、Cでそれぞれ

```
(with A (assert (p a)))
(with B (assert (p b)))
(with C (assert (p c1))
        (assert (p c2)))
```

のように定義されているとしよう。これを

```
(with A (with B (with C (p *x))))
```

のように呼び出すと

```
*x=c1, c2, b, a
```

の順で解が得られるし、

```
(with C (with B (with A (P *x))))
```

のように呼び出すと、解は

```
*x=a, b, c1, c2
```

の順になる。

外側の世界の定義を隠してしまうにはassertではなくdefineを用いる。assertが、これまでの定義に追加する働きを持っているのに対し、defineはこれまでの定義を消去してしまう働きがある（ただし、外側の世界を変えることはしない）。先程の例で世界Bにおけ

るpをdefineを用いて

```
(with A (assert (p a)))
(with B (define p ((b))))
(with C (assert (p c1))
        (assert (p c2)))
```

のようにしておくと、

```
(with C (with B (with A (P *x))))
```

の呼び出しの解は

```
*x=a, b
```

となり、Cの定義が見えなくなる。

初期のProlog/KR ＊ではassertやdefineの実行時に外側の定義をコピーして、上記の機能を実現していた。しかし、これは完全な動的ネスティングとは言えない。Uranusでは実行時に内側から外側へと定義を走査する。この際、defineによる定義を区別するためにnil を最後にマーカとして置いてある。バックトラック時に別の節を探すときにこのnilに遭遇すると外側の世界の探索を打ち切る。

3.概念の階層と属性の継承

多重世界の機能を用いると概念の階層構造と、それらの間の属性の継承が表現できる。一つの世界が一つの概念に対応すると考え、その概念の持つ属性はその世界の中で定義される述語とすることによって知識のフレーム[Minsky 1975] 的な表現が可能になる。例えばペンギンに関する知識は

```
(with PENGUIN
    (assert (AKO bird))
    (assert (INHABITANCY Antarctica))
    (assert (COLOR black))
    (assert (COLOR white))
      ... )
```

のように表現できる。ここでAKO, INHABITANCY, COLOR がスロット、bird, Antarktica, black およびwhite がその値と考えられる。

属性（プログラムを含む）の継承のルートは実行時に定まるので、一般にはその環境を作るために

＊　Version-C （Utilisp 上で動いているもの。Prolog/KR の最初の版）およびVersion-F （Franzlisp 上で動いているもの。Version-C とほとんど同じ言語仕様）。Maclisp 上のVersion-S はUranusに近い。

```
(with OBJECT
    (with ANIMAL
        ...
        (with BIRD
            (with PENGUIN
                (NUMBER-OF-WINGS *n)))...))
```

のようにwithを何重にも重ねる必要が生じる。これは煩わしいので、Prolog/KR では実行時に展開してくれる述語を

```
(assert (AKO *sub *super)
    (assert (WITH-WORLD *sub *p)
        (WITH-WORLD *super (with *sub *p))))
(assert (WITH-WORLD *x *p) *p)
```

のように定義していた。内側のassertはAKO が実行されたときにダイナミックにWITH-WORLD を定義するためのものである。また、２番目のWITH-WORLDの定義は階層の一番上ではトップレベルの環境を用いることを指示している。こうしておくと、

```
(AKO PENGUIN BIRD)
(AKO BIRD ...)
  ...
(AKO ANIMAL OBJECT)
```

の実行により必要なWITH-WORLDが定義され、

```
(WITH-WORLD PENGUIN  (NUMBER-OF-WINGS *n)))
```

の実行時に、先に出てきたようなwithのネスティングに展開される。

しかし、実行時に毎回ネスティングを作り出すのは効率の面でよくない。そもそも概念の階層構造は静的なものであるから、完全に動的なネスティングにマップする必要はない。そこで、Uranusではwithinという述語を新たに導入した。withinは第１引数として世界のネスティングを表わすリストを取る。リストの先頭が一番内側の世界で、それから順に外側へとネストする。第２引数以降はwith同様、任意の述語呼び出しが書ける。例えば

```
(within (A B C) (p *x))
```

は

```
(with C
    (with B
        (with A (p *x))))
```

とほぼ等価である。「ほぼ」と言ったのは、withinではその外側の世界を無視する点が異なっているからである。従って

```
(with X (within (A B C) ...))
```

と

(within (A B C) ---)

とは等価で、外側の世界Xは無視される。この特別な用法として

(within () ---)

でその時点での外側のネスティングをすべて締め出すことができる。

さて、このwithinを用いると

(with PENGUIN (assert (supers PENGUIN BIRD ... ANIMAL)))

としておけば、

(with PENGUIN (within [supers] (NUMBER-OF-WINGS *n)))

として一度に実行できる。ここで[supers]はUranusにおける実行可能パターンで、

(supers *)

を実行した* の部分がここに書いてあるのと等価である。一般にWITH-WORLDは

```
(assert (WITH-WORLD *w *p)
    (with *world (within [super] *p))))
```

となる。この改善によるスピードアップは、ネストを作る手間とその中での仕事の量の比によるので一概に言えないが、50%アップになったとの報告がある。

4.仮想世界としての多重世界

例えば積木AをBからCに移動させるという変化は

```
(retract (on A B))
(assert (clear B))
(retract (clear C))
(assert (on A C))
```

という動作の列で表現できる。したがって積木を動かす述語moveは

```
(assert (move *x *from *to)
    (retract (on *x *from))
    (assert (clear *from))
    (retract (clear *to))
    (assert (on *x *to)))
```

のように定義できる。

しかしながらこの解法ではバックトラックの際に問題が残る。assertやretract はバックトラックの際に元に戻らないからである。例えば、先ほどの積木Aの移動で、Cの上にすでに何か別の積木が乗っていたとすると(clear C) がないので

(retract (clear C))

が失敗する。しかしながら、この場合にもそれ以前の二つのassertとretract は元には戻らない。すべてが旨くいくことをテストしてから実行に入れば、このように途中で失敗することは防げるが、一般にはテストと実行はほとんど同じ動作の繰り返しで二度手間にな

るので、できれば一回ですませたい。

多重世界機構が状態の変化を表わすのに使える。様相論理の場合のように、公理の変化は世界の変化と考えてよい。積木の移動と同時に別の世界に移ったと考えるのである。assertやretract は、この新しい世界で行なわれ、元の世界は不変のまま残る。

積木ＡをＢからＣに動かして到達する世界をmoveA-B2C と名付けることにすると

```
(with moveA-B2C
    (retract (on A B))
    (assert (clear B))
    (retract (clear C))
    (assert (on A C)))
```

が成立する。変化は新しい世界のみに起こるので、動作が成功した場合にのみ新しい世界に移ることにしておけば、バックトラックして古い世界に戻ったときには、そこでは元のままの状態が保存されている。いく通りかの動作が可能な選択点では、したがって、

```
(or (with branch1 <action 1> <further actions>)
    (with branch2 <action 2> <further actions>)
    (with branch3 <action 3> <further actions>)
        ...  )
```

のようにしておけばよい。branch1 で <action 1> を試みて失敗してbranch2 に来ても、こちらの世界では<action 1>の効果は残っていない。

一般に

```
(with nil <action 1>
  (with nil <action 2>
    (with nil <action 3>
            ...
      )))
```

で動作の列を表わすことができる。ここでnil というのは名前のない特別の世界で、毎回新しいものが自動的に作られる。

問題は、assertやretract がどの世界に対してなされるべきか、である。仮想世界のネストを作っている理由は、元の世界を変更しないためであるから、変化は一番内側の世界でおこらなければならない。assertの場合は外側の世界との和になるので、一番内側の世界にassertしておけばよい。ところが、retract の場合は、一番内側の世界にはretract すべきものがもともと存在しない。

Uranusでは、assert、define、definition、およびretract の述語の定義を操作するメタ述語は一番内側の世界に対して働くものとしている。従って、一番内側の世界に定義がない場合には、外側にそれがあっても見に行かず、retract が失敗する。definitionを取ってくるときやretract する場合に、それらが効果を持つためには、その述語が定義さ

れている世界でそれらを実行する必要がある。そこでUranusではwhere という述語を用意している。

(where p (q))

は世界のネストを内側から辿り、p が定義されている世界で(q) を実行する。そこで

(where p (retract (p a)))

を実行すればpの定義されている世界で(p a) をretract することが可能になる。

ところが、この問題に関しては先にも述べたように、一番内側の世界でretract してくれなくては意味がない。そこで、新たにretract1という述語を導入した。これは外側の定義をすべて一番内側の世界にコピーした後にretract を起動する。また、バックトラックの際に外側の世界を見に行かないようにdefineの場合同様nil のマーカを最後に付加しておく。retract1により上記の問題は一応解決されるが、一番内側の世界だけが単独で用いられた場合にはセマンティクスがおかしくなる。retract1は、あくまで仮想世界として

(with nil ···)

の中でのみ使うべきである。

この問題の根本的解決策としては負のassertion を考える必要がある。何もない世界からretract すると、それは負のassertion となり、外側の世界のassertion を打ち消すのである。様相論理的視点から言えば、こちらの方が「論理」的であろう。

5.今後の問題点

現在未解決の問題点として二つ挙げておく。

1：assertの定義を世界により変更することもありうる。その場合に、例えばAの世界のassertの定義を使ってBの世界にassertしたい場合など、Bの世界にassertの定義がなければ

(with A (with B (assert ···)))

でよい。しかしBにもassertの定義がある場合は、そちらのassertが動いてしまう。assertやretract の場合だけなら、assert-into-world やretract-in-worldのような陽に世界の名前を用いる述語を作ることも可能であるが、一般にはAの世界の定義を取り出し、それをBの世界で実行するメカニズムが必要である。現状ではdefinitionを用いれば、一応可能であるがリカーシブ・コールには対処できない。これはLispのfunarg問題と似ている。クロージャのような概念が必要なのかもしれない。

他の言語のモジュール機能に置いては、関数の名前のスペースだけを指示するので、この問題は起きない。

2：世界の集合を扱えないか。現状では、世界はネストしても、原則的には個々にバラバラの世界である。例えば世界AとBが同時に存在する場合にのみ成立するような事柄はどのように記述すればよいのであろうか。

```
(with A (assert (fallible *x) (god *x)))
(with B (assert (god Uranus)))
```

から

```
(assert (fallible Uranus))
```

が導けるが、これらはどの世界に属する知識なのか。強いて記述するなら

```
(within (A B) (assert (fallible Uranus)))
```

とでもなろうが、今のwithinは、この気分をうまく表現していない。

6.まとめ

Uranusにおける多重世界機構の拡張とその問題点を示した。この世界というメタの概念をオブジェクトとして取り扱うことにより、言語の表現力が飛躍的に増すものと考えている。しかしながら、まだ言語仕様として固定された段階には至っていない。今後の、利用と再検討が必要である。その意味で本論文は中間報告と考えていただきたい。

謝辞

本研究の機会を与えていただいた柏木寛　電子計算機部長に感謝します。

参考文献

中島秀之：“Prolog/KR の概要”記号処理研究会18-5, pp. 69-74 (1982)

中島秀之：“Prolog/KR における知識表現”情報処理学会第28回全国大会論文集, pp.1139-1140 (1984)

Hideyuki Nakashima: “Knowledge Representation in Prolog/KR ” Proc. of The 1984 International Symposium on Logic Programming, pp.126-130 (1984)

Marvin Minsky: “A Framewprk for Representing Knowledge” The Psychology of Computer Vision, Winston P.(ed.), McGraw-Hill (1975)