TR-130

論理型プログラミング言語 Prolog による知識ベース管理システム

北上　始(富士通)、宮地泰造(三菱電機)
國藤　進，古川康一(ICOT)

July, 1985

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Vol. 26 No. 11　　情　報　処　理　　Nov. 1985

25周年記念論文

論理型プログラミング言語 Prolog による知識ベース管理システム†

北上　始††　國藤　進†††
宮地泰造††††　古川康一†††

人工知能の応用として知られているエキスパートシステムを作成するためには，知識の表現，獲得，利用といった機能を支援する知識ベース管理システムが必要である．本論文では，このシステムの構成法及び要素技術についての研究成果を報告している．本知識ベース管理システムのインプリメンテーションは，論理型プログラミング言語 Prolog で実現することを前提としている．知識ベースの構造は，著者らによって提案された次のような論理構成をとっている．知識ベースを構成している各フレーム（知識の集合体）は，主として次の五種の知識に分類されており，知識ベースの拡張性を十分に反映できるように構成されている．五種の知識としては，(1)構造化知識，(2)プログラム化知識，(3)制約型知識，(4)メソッド型知識，(5)辞書型知識がある．(1)，(2)はオブジェクト知識，(3)，(4)，(5)はメタ知識とも呼ばれている．(1)の構造化知識は，自然言語の分野で知られる概念階層関係を採用しており，(2)のプログラム化知識は，(1)で表現された知識を補うとともにそれを利用した種々のルールを表現している．(3)の制約型知識には，動的に整合性を維持するための trigger 型のメタ知識や，矛盾検出用の inconsistency 型のメタ知識があり，これらはメタ推論機能を使って容易に評価実行することができる．特に trigger 型のメタ知識は，自然言語処理の分野で知られる因果関係を表現するのに十分な機能をもっていることが述べられている．(4)のメソッド型知識は，著者らが，主に知識獲得と呼んでいる機能を提供するためのメタ知識である．ここでは，オブジェクト・レベルとメタ・レベルの二種類の知識獲得が存在することを述べている．また，知識獲得時に，制約型知識の評価実行を行うためのタイミングを制御するために，トランザクションという概念の導入が大切であることを述べている．(5)の辞書型知識は，データベースの分野のデータ辞書を拡張した知識であり，知識ベース管理の処理効率向上及び種々のインタフェース向上のために利用される．

1. はじめに

エキスパートがもつ問題解決能力をコンピュータ・システムにもたせるためには，問題解決に使用する知識を知識ベースに蓄積・管理しなければならない．そのためには，エキスパートがもつ知識をコンピュータ処理できる形式に表現し，それを系統的に獲得し，獲得された知識を容易に利用できる機能を実現するシステムを構築する必要がある．このようなシステムは，知識ベース管理システムとして知られている．

著者らは，論理型プログラミング言語 Prolog を知識ベース管理システムのインプリメンテーション用言語と仮定し，そのシステムの実現法を研究してきた．知識ベース管理の実現のためには，従来の関係データベース管理の枠を，知識表現に利用される Prolog 形式のプログラムの領域にまで拡張して，研究を進めていかなければならない．そのために，著者らは，データベース，認知心理学，形式論理学，プログラミング方法論などといった種々の分野の研究成果を有機的に統合し，新しい諸概念の研究を進めていかなければならないと考えている．

本論文では，著者らのグループによって考案された知識ベース管理の基本技術について述べ，さらに知識ベース管理システムとしてとらえた場合の各要素技術の相互関係についても述べる．なお，知識ベースの知識は，物事の客観的な事実や規則を表現したオブジェクト知識と，その知識に関する知識（使い方，意図，制約などが含まれる）を表現したメタ知識とに大別されるが，実世界の知識は，この両知識を組み合わせた表現になっている．一般に，実世界の知識は，ホーン節の枠を越えているが，著者らは，この実世界の知識のかなりの部分がホーン節形式のオブジェクト知識とホーン節形式のメタ知識に分解した知識構造で表現できると考えている．したがって，本稿で取り扱うオブジェクト知識とメタ知識は，どちらも論理型プログラ

† Knowledge Base Management System Implemented in Logic Programming Language Prolog by Hajime KITAKAMI (Fujitsu Laboratories Ltd. Kawasaki), Susumu KUNIFUJI (Institute for New Generation Computer Technology), Taizo MIYACHI (Mitsubishi Information Systems & Electoronics Development Laboratory) and Koichi FURUKAWA (Institute for New Generation Computer Technology).

†† (株)富士通研究所
††† (財)新世代コンピュータ技術開発機構
†††† 三菱電機(株)情報電子研究所

ミング言語 Prolog で記述可能なホーン節だけで表現されると制限し，プログラミング用言語のシンタックスは，DEC-10 Prolog[2] を前提としている．

2. メタ推論

知識ベースの管理という立場から知識ベースの理想的な設計目標を列挙すると，(1)効率が良く，(2)拡張性に富み，(3)種々の問題解決に対処できると同時に，(4)大容量で，(5)使いやすくなければならない，などを挙げることができる．これらの中でも，(1)～(3)の点に着目すると，Prolog の処理系自身がもつ単純な推論機能だけでは，対処しきれない．なぜなら，(1)を実現するためには，推論の効率的な制御が必要であり，(2)のためには，知識の集合体を利用目的に応じた単位で分割し，その分割単位ごとの推論を実現する必要があるからである．さらに，(3)を実現するためには，推論過程において種々の証明木が必要とされるような場合がある．以上から，本章では，このような種々の状況に対処するために必要不可欠なメタ推論について述べる．メタ推論は，"推論に関する推論"であり，demo 述語と呼ばれるメタ述語により達成できる．demo 述語の一般形式は，"demo (Frame, Goals, Control, Result)" である（付録を参照）．この demo 述語は，知識の集合体としてのフレーム Frame の中で，与えられた制御 Control に従って，与えられたゴール列 Goals を証明し，その証明プロセスから必要な情報 Result（証明木などがある）を抽出する[11]．この4引数の demo 述語は，知識ベース管理システムをインプリメントするために利用できる以外に，メタ知識を表現するためのツールとしても利用できる．以下，本論文の説明では，問題ごとに重要な機能が浮き出るようにするために，この一般形式の demo 述語を持ち出すことを避け，問題ごとに特殊化した demo 述語で解説を試みることにする．図-1に demo 述語の構造を知る上で，最も簡単なプログラム例を示

```
demo(Frame, true) :- !.
demo(Frame, (P, Q)) :- !,
        demo(Frame, P), demo(Frame, Q).
demo(Frame, P) :-
        (system(P); method_type(P)) ->
        eval (Frame, P); clause_of (Frame, (P :- Q)),
        demo(Frame, Q).
clause_of(Frame, (P :- Q)) :-
        X =.. [Frame, Clause], X,
        (Clause = P -> Q = true; Clause = (P :- Q)).
```

図-1 2引数の demo 述語の例

Fig. 1 Example of demo-predicate which has two arguments.

しておく．図-1 の demo 述語は "demo(Frame, Goals)" 形式の demo 述語であり，フレーム Frame 単位で分割された知識の集合体ごとの推論ができるようになっている．ただし，フレーム内のホーン節形式の知識は，すべてカットシンボルと呼ばれるメタ述語を含まず，ホーン節表現として，論理和表現を含まないと仮定する．図-1 の第一番目のルールは，ゴールを証明していったときの停止条件である．第二番目のルールは，論理積の展開証明を行っている．第三番目のルールの後半部は，"P" と単一化可能な帰結部をもつホーン節をフレーム Frame の中からさがし，"Q" の証明を行っている．このホーン節をさがす部分は，clause_of 述語で実現されている．図中の ";" は，論理和記号であり，"A→B ; C" は，"if A then B else C" と同じ意味である．また，この単一化可能なホーン節が条件部をもたないとき，"Q" には，真 "true" が返され，第三番目の最後の述語 demo(Frame, true) の実行結果が，第一番目のルールにより真になる．図-1 の eval 述語は，"P" がシステム組み込み述語（system(P) で示されている），または，フレームへの専用アクセス述語（method_type(P) で示されている）のときにPを実行評価する述語である．図中の第四番目のルールは，clause_of 述語を実現するルールであるが，この中の "X" には，メタ述語 "=.." により，述語 "Frame(Clause)" が割り当てられる．

3. オブジェクト知識

オブジェクト知識は，概念の階層関係が明確に整理された構造化知識と，構造化するのが難しいために手続き的表現をとったプログラム化知識とに大別される．本章では，この二種類の知識について述べることにする．ここで，ホーン節を "$p(X_1, X_2, \cdots, X_n)$:- body." とするとき，"p" のことを説明の便宜上，概念と呼ぶことにする．

3.1 構造化知識

構造化知識は，自然言語処理の分野で知られる概念階層関係を Prolog の事実として表現した知識である．概念そのものを定義域として，その関係を表現していることから，この種の事実は，二階述語論理の知識を表現していることになる．これには，三種類の表現が知られている．一番目には，ある概念の特性（性質）を表現する property 関係，二番目には，概念間の上位・下位関係を表現する is_a 関係，三番目には，

概念間の部分・全体関係を表現する part_of 関係があり，次のように表現できる．

property(概念，特性を評価する述語)． (1)

is_a(下位概念，上位概念)． (2)

part_of(全体概念，部分概念)． (3)

property 関係は，ある概念がもつ特性がどのようなルールによって評価されるかを示す述語である．is_a 関係は，概念間の上下関係を示す最小単位であり，上位概念の特性が，下位概念に継承されるという性質をもっている．part_of 関係は，概念間の全体・部分関係を示す最小単位である．ここでは，部分概念の諸特性が全体概念に継承されるという性質をもっていると仮定する．

継承関係の例外処理は，後述するとして，これらの継承関係を調べるためには，is_a 関係をもとにして，連続的に何段もの上位・下位の概念をたどったり，part_of 関係をもとにして，連続的に何段もの全体・部分概念をたどったりする機能が必要である．これらは，おのおの super_concept や sub_concept といった述語により実現できる．また，これらを利用して，継承関係を調べる述語は，inherit 述語により実現できる．図-2 に，これら三述語の実現プログラム例を示す．この三述語は，いずれも第1引数に，知識の集合体を表現するフレーム Frame が記述されている．また，各述語とも，第2引数に，おのおのの述語に対する動作主 (agent) を記述すると，第3引数にその答が返される．さらに，三種の述語を実現するプログラムの中に，図-1 で示したメタ推論用 demo 述語を利用している．

```
super_concept(Frame, X, Y) :-
        demo (Frame, is_a(X, Y)).
super_concept(Frame, X, Y) :-
        demo(Frame, is_a(X, Z)),
        super_concept(Frame, Z, Y).
sub_concept(Frame, X, Y) :-
        demo(Frame, part_of(X, Y)).
sub_concept(Frame, X, Y) :-
        demo(Frame, part_of(X, Z)),
        sub_concept(Frame, Z, Y).
inherit(Frame, X, Y) :-
        super_concept(Frame, X, Z),
        demo (Frame, property(Z, Y)).
inherit(Frame, X, Y) :-
        sub_concept(Frame, X, Z),
        demo(Frame, property(Z, Y)),
```

図-2 概念階層関係を特徴づけるルール

Fig. 2 Rules characterizing conceptual hierarchy.

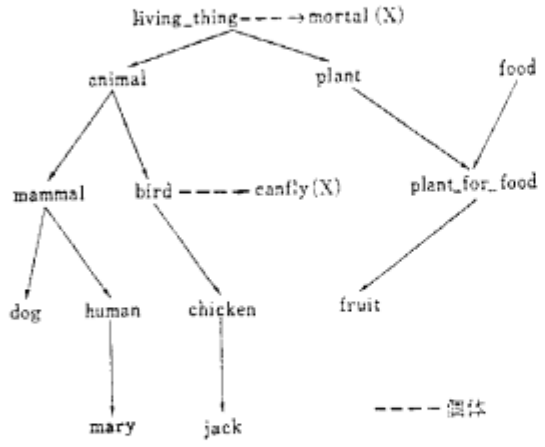

図-3 is_a, property 関係の構造
矢印 → は，is_a 関係を示し，矢印 ⇢ は，property 関係を示す．

Fig. 3 Structure of is_a and property relation. The arrow "→" means "is_a" relation and the arrow"⇢" means "property" relation.

```
/* Structuralized Knowledge */
is_a(animal, living_thing).
is_a(plant, living_thing).
property(living_thing, mortal(X)).
is_a(mammal, animal).
is_a(bird, animal).
property (bird, canfly(X)).
is_a(fruit, plant_for_food).
is_a(plant_for_food, plant).
is_a(plant_for_food, food).
is_a(dog, mammal).
is_a(human, mammal).
is_a(chicken, bird).
part_of(mammal, face).
part_of(mammal, body).
part_of(mammal, arms_2).
part_of(bird, face).
part_of(bird, body).
part_of(bird, wings_2).
is_a(mary, human).
is_a(john, human).
is_a(jack, chicken).
```

図-4 構造化知識の知識表現例

Fig. 4 Examples of knowledge representation of structuralized knowledge

```
| ? - super_concept(frame, jack, X).
X = chicken ;
X = bird ;
X = animal ;
X = living_thing
yes
| ? - inherit(frame, jack, X).
X = canfly(_291) ;
X = mortal(_441)
yes
| ? - demo(frame, canfly(jack)).
no
| ? - demo(frame, mortal(jack))
yes
```

図-5 super_concept と inherit 述語の実行例

Fig. 5 Example of executions of super_concept predicate and inherit predicate.

```
/* Programmed Knowledge */
mortal(X) :- super_concept(X, living_thing).
not_canfly(jack).
canfly(X) :- super_concept(X, bird),
     not(not_canfly(X)).
food(chickweed).
hungry(jack).
super_concept(jack, bird).
super_concept(mary, human).
super_concept(john, human).
```

図-6 プログラム化知識の知識表現例
Fig. 6 Examples of knowledge representation of programmed knowledge.

以下に，super_concept と inherit 述語の実行例を示すことにしよう．図-3 に，利用された知識例を示す．図-3 に対する構造化知識の知識表現例は，図-4 に示されている．図-4 の知識は，フレーム名が“frame”と呼ばれる知識の集合体の中に蓄積されているとする．この例では，生物(living_thing))，動物(animal)，植物(plant)，哺乳類(mammal)，鳥(bird)，果物(fruit)などの間の概念階層関係を対象にしている．図-5 は，図-4 の知識に基づく実行例である．“jack は何者であるか”とか，“jack の特性はなんであるか”といった質問が，おのおの super_concept，inherit 述語で行われている．図-5 の inherit 述語による問い合わせで“jack”の特徴を調べたところ，「飛べる(canfly(X))及びいつかは死ぬ(mortal(X))という述語が特性評価のために利用できる」という結論が出た．次の問い合わせでは，demo 述語により，“jack が真に飛べるか”を調べ，その結果，例外処理を含む図-6 のプログラムにより「それが正しくない」ことが結論づけられた．その次の問い合わせでは，“jack がいつかは死ぬか”を調べ，その結果，「それが正しい」ことを結論づけられた．

3.2 プログラム化知識

前節の構造化知識は，物事を三種の関係(property，is_a，part_of)で表現しようとしていたが，このような関係だけで表現できない知識は，数多く存在する．プログラム化知識は，それを Prolog のプログラムとして表現した知識である．構造化知識とプログラム化知識の関係は，相補的であるので，上手に概念の構造化が達成されるほど，構造化知識をプログラム化知識で容易に利用することができるようになり，プログラム化知識を簡潔に表現できる．図-6 に，プログラム化知識の表現例を示す．ここでは，“生物は，いつかは死ぬ(mortal(X))”，“(大部分の)鳥は，空を飛べる(can_fly(X))”，“jack は，飢えている(hungry(jack))”，“jack は，鳥の一例である(super_concept(jack, bird))”などの知識が示されている．

4. メタ知識

メタ知識には，(1)オブジェクト知識を成長させるときに，誤った方向に成長することを防止したり整合性を維持するための制約型知識，(2)問題解決のオブジェクト知識を効率良く使用するための戦略型知識，(3)オブジェクト知識及びメタ知識の形式的な枠組みを示すための辞書型知識，(4)知識の集合体に対する問い合わせ，更新操作などを行うためのメソッド型知識などがある．知識の集合体の論理構造については，図-7 に示す通りである[13]．図の矢印(→)は，メタとオブジェクトの関係を表す記号であり，“A→B”は，オブジェクト知識(あるいはメタ知識)Bのメタ知識がAであることを示している．また，図の双方向矢印(↔)は，知識の相互依存関係を表している．したがって，プログラム化知識と構造化知識をオブジェクト知識とすると，その他の知識は，メタ知識になる．また，メソッド型知識は，辞書型知識により管理されているが，辞書型知識及びその他の知識を操作するための知識である．以後，本章では，紙数の関係で，(1)と(4)について述べ，(2)と(3)については，他の機会で述べることにする．

4.1 制約型知識

制約型知識には，次の種類がある．一つは，知識ベースの更新アクセスなどのオペレーションに対し，動的に整合性をとるために，種々の知識の追加または削除

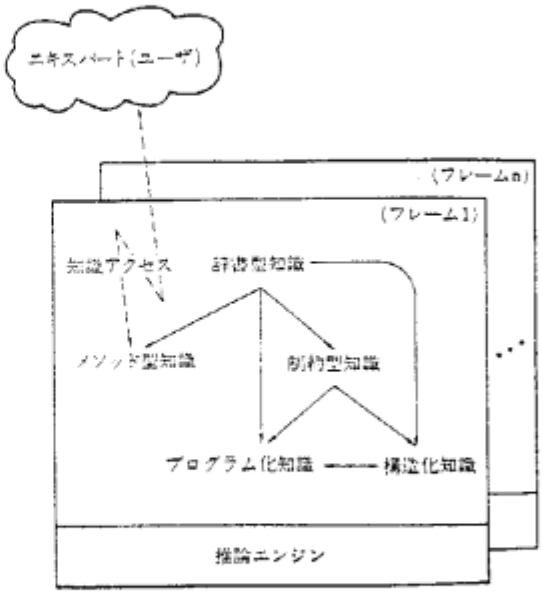

図-7 知識の集合体(フレーム)の論理構造
Fig. 7 Logical structure of knowledge set (frame).

を行う機能をもつ trigger 型のメタ知識である．他の一つは，知識ベースの更新アクセスとしてのオペレーションに対する無矛盾性を監視する inconsistency 型のメタ知識である．このメタ知識は，オブジェクト知識が矛盾する条件を記述しているので，否定型ルールと見なすことができる．両メタ知識ともに，後述するトランザクション処理[5]の最後に起動される遅延型 (delayed-type) のメタ知識と，トランザクション処理とは独立であって更新などの知識アクセスがあるごとに起動される即時型 (immediate-type) のメタ知識とに分類される．以上から，これらのメタ知識は，次のように表現される．

trigger(起動タイプ，オペレーション) :—
　起動条件，!，複数の更新オペレーション．(4)
inconsistent(起動タイプ，オペレーション) :—
　起動条件，!，矛盾判定条件． (5)

両知識とも，起動タイプとして，immediate, delayed のタイプをもつ．両知識中に現れている記号"!"は，カットシンボルと呼ばれるメタ述語であり，ここでは，"A, !, B" を "Aならば Bが成立する" と解釈している．図-8 に，制約型知識の知識表現例を示しておく．図中には，五つの制約型知識があるが，その中から二つを取り出して，説明してみよう．第1番目のメタ知識は，trigger 型のメタ知識であり，知識の集合体としてのフレーム frame に，"Agent が Information を Receiver に与える" という知識 (give(Agent, Receiver, Information)) を追加する (insert) ときに，Information が食べ物 (food) であり，Receiver が空腹 (hungry) であれば，その frame に "Receiver は，Information を食べる" という知識 (eat(Receiver, Information)) を連続的に追加する (insert) 処理を行う．次に，3番目のメタ知識を説明してみよう．この知識は，inconsistency 型のメタ知識であり，フレーム frame に "X の上位階層が Y である" という知識 (is_a(X, Y)) を追加した (insert) 後は，必ずその "Y" が "living_thing" につながっていなければ矛盾すると主張している．

以上の制約型知識の実行は，前述の demo 述語を利用することにより，容易に実施できる．trigger 型のメタ知識は，知識ベースのオペレーションに伴い，関係する trigger 型の知識だけをすべて実行すれば良い．inconsistency 型のメタ知識は，知識ベースの更新が妥当か否かを判定し，一つでも矛盾する知識が存在すれば，その更新を禁止すれば良い．この制約型知識を実行するプログラム例を図-9 に示す．図中の maintenance 述語は，trigger 型メタ知識を実行することにより，知識ベースの知識をメンテナンスする述語である．また，図中の check_inconsistency 述語は，矛盾を判定する述語である（真ならば矛盾，偽ならば無矛盾）．

ここで，前出の図-4 に基づき，"is_a(chickweed,

```
/* Constraints */
trigger(immediate, insert(frame, give(Agent, Receiver, Information))) :-
    food(Information), hungry(Receiver), !,
    insert(frame, eat(Receiver, Information)).
trigger(immediate, insert(frame, eat(Agent, Information))) :-
    not(ill(Agent)), food(Information), !, insert(frame, cheerful(Agent)).
inconsistent(delayed, insert(frame, is_a (X, Y))) :-
    not(super—concept(Y, living_thing)).
inconsistent(delayed, X) :-
    member(X, [insert(frame,_), delete(frame,_), update(frame,_)]), !,
    canfly(Y), not_canfly(Y).
inconsistent(immediate, X) :-
    member(X, [insert(frame,_), delete(frame,_), update(frame,_)]), !,
    hungry(Y), full(Y).
```

図-8 制約型知識の知識表現例
Fig. 8 Examples of knowledge representation of constraint-type knowledge.

```
check_inconsistency(Mode, Operation) :-
        Operation =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (inconsistent(Mode, Operation) :- Goals)),
        demo(Frame, Goals).
maintenance(Mode, Operation) :-
        Operation =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (trigger(Mode, Operation) :- Goals)),
        exec_trigger(Frame, Goals), fail.
maintenance(Mode, P).
exec_trigger(Frame, true) :- !.
exec_trigger(Frame, (P, Q)) :-
        exec_trigger(Frame, P), exec-trigger(Frame, Q).
exec_trigger(Frame, P) :-
        operation_type(P)->execute(P).
exec_trigger(Frame, P) :-
        system(P)->call(P);
        clause_of(Frame, (P :- Q)), exec_trigger(Frame, Q).
```

図-9 制約型知識を実行するプログラム例
Fig. 9 Example of programs executing constraint-type knowledge.

```
| ?- transaction(insert(frame, give(mary, jack, chickweed))).
* End Transaction.
yes
| ?- demo(frame, eat(Agent, Food)).
Agent = jack,
Food = chickweed;
no
| ?- demo(frame, cheerful(Agent)).
Agent = jack;
no
```

図-10 制約型知識の実行例

Fig. 10 Example of execution of constraint-type knowledge.

```
demo(Frame, true,d(X, X)) :- !.
demo(Frame, (P, Q), d(X, Z)) :- !,
        demo(Frame, P, d(X, Y)), demo(Frame, Q, d(Y, Z)).
demo(Frame, P, d(X, Y)) :-
        (system(P); method_type(P))->eval(Frame, P), X=Y;
        clause_of(Frame, (P :- Q), Certainty),
        (Certainty==premise->X=Z;
        X=[(P :- Q); Z]), demo(Frame, Q, d(Z, Y)).
clause_of(Frame, (P :- Q), Certainty) :-
        X=..[Frame, Clause, Certainty], X,
        (Clause=P->Q=true; Clause=(P :- Q)).
```

図-11 3引数の demo 述語

Fig. 11 Example of demo-predicate which has three arguments.

vegetable)" と "is_a(vegetable, plant_for_food)" が事前に獲得されているということを前提にして，"mary gives chickweed to jack" という事象が生じたという知識が知識ベースに挿入されたとしよう．図-10 の実行例に示すように，"eat (jack, chickweed)" という知識が挿入され，それに伴い，"cheerful (jack)" という知識が連鎖的に知識ベース内のフレームに挿入されている．ここで，図-10 のトランザクション処理については，5 章を参照されたい．

4.2 矛盾知識の修正

4.1 では，二種の制約型知識の実行方式について述べてきたが，ここでは，それらの知識の中でも inconsistency 型知識に違反する知識が知識ベース（この中のオブジェクト知識に焦点をあてている）の中に存在する場合に，どのように矛盾知識を発見し，修正していくかについて述べる．

inconsistency 型知識とオブジェクト知識の間に発生する矛盾は，demo 述語により判定できることは，すでに述べた通りであるが，オブジェクト知識のどれが，矛盾知識の候補になるかを検出するためには，2 章で述べたように，矛盾判定段階で利用されたオブジェクト知識を demo 述語で記録しておく必要がある．そのための demo 述語を "demo(Frame, Goals, Result)" と表現し，インプリメントすると，図-11 のようになる．図-11 の demo 述語の第 3 引数は，証明木を取得するために，d-list 表現をとっており，ゴール demo(frame, inconsistent (Operation), d(X, Y)) の証明プロセスで，仮定型知識だけが d-list に取得される．図-11 の "Certainty" には，assumption または premise のパラメータが返される．仮定型 (assumption-type) 知識は，修正されるかも知れないと前もって定義された知識である．修正されることがない知識は，前提型 (premise-type) 知識と呼ばれている．この形式の demo 述語を解いて，もし，その結果が真になれば，矛盾が生じたことになり，d(X, Y) にその原因となるいくつかの仮定型知識が抽出される．

このように，仮定型知識の修正は，知識獲得時に，知識体系の矛盾を解消するために実施されることになるが，あるいくつかの仮定型知識が矛盾の原因になっていることがわかると，次に示す方法で，その知識を修正できる．

（1） エキスパートが事前に，その知識の別解を与えていたとき，その別解を新しい仮定型知識としてみる．

（2） その知識の否定をとる．

（3） さらに精密な知識を，このシステムが仮定する．この方法には，Shapiro のモデル推論[24),25)] で知られる精密化オペレータを適用したり[15)]，例外知識の知識表現の形態をとったりする方法がある．

（4） エキスパートがその知識の別解を直接与える．

しかしながら，demo 述語で抽出された仮定型知識が二つ以上存在するときは，上記方法で試行錯誤的に，修正される知識の組み合わせをさがし，無矛盾な知識ベースにしなければならない．この試行錯誤的な処理を効率的に実施するためには，少し工夫しなければならない．抽出された複数の知識の一つをある規則に従って取り出し，それを修正しその結果の無矛盾性判定を逐次実施することにより，すべての可能な組み合わせの知識修正を実施できる．これの基本部分をアルゴリズム化すると，次のように表現できる．

抽出された n 個の知識に 1 から順に番号をつけ，無矛盾性判定の最初からの回数を j 回目とすると，k 番目の知識を修正すべきか否かは，次の関数 $f(j, k)$ の値を求めることにより，達成される．

$$f(j, k) = (j \div 2^{k-1}) \bmod 2^k \qquad (6)$$

ここで，$f(j, k)=0$ のとき，k 番目の知識を修正し，それ以外のとき，k 番目の知識を修正しない．また，$1 \leqq k \leqq n$, $1 \leqq j \leqq 2^n-1$ が成立する．たとえば，$n=2$ とすると，

1回目は，$f(1,1)=0$, $f(1,2)=3$ となり，
2回目は，$f(2,1)=1$, $f(2,2)=0$ となり，
3回目は，$f(3,1)=0$, $f(3,2)=1$ となるので，

各回ごとの修正知識の番号は，おのおの，1番目，2番目，1番目となる．ここで，2回目までで無矛盾にならない場合は，3回目で1番目の知識が再度修正されるが，この知識は，1回目ですでに修正済みになっているので1回目の修正前の知識にもどすことになる．以上からわかるように，Prolog のバックトラック機構を直接使用せずに，修正知識を知的にもとにもどす操作が有効であり，本アルゴリズムの中で実現されている．

4.3 因果関係

制約型知識のうちでも，trigger 型のメタ知識は自然言語処理の分野で知られる因果関係 (causal relation)[3] の記述に利用される．因果関係は，事象間の原因と結果の関係である．知識ベースにおいては，世界の変化を知識ベースの変化とみなすと，知識ベースに更新があるごとに，それに伴って因果関係の更新があることになる[22]．これをプログラムとして整理すると図-12 のようになる．知識ベースの変化は，更新アクセス（オペレーション）によって生じると考えることができるので，因果関係 Cause は，原因 Operation 1 と結果 Operation2 によるオペレーションの対として表現することができる．この関係は，trigger 型メタ知識の特殊ケースとして表現されていることになる．図-12 の一番目のルールは，原因から結果を知りたいとき，二番目のルールは，結果から原因を知りたいときに利用されるルールである．一番目のルール中に現れている find_causal_operation は，delete, insert, update といった知識ベースへのメソッドによる更新アクセス（オペレーション）を trigger 型メタ知識の条件節の中から見つける述語である．ところで，順序関係 (order relation ; ≦) は，通常，反射法則 (reflexive law ; $X \leqq X$)，反対称法則 (asymmetric law ; $X \leqq Y$ かつ $Y \leqq X$ ならば，$X=Y$)，推移法則 (transitive law ; $X \leqq Y$ かつ $Y \leqq Z$ ならば，$X \leqq Z$) が成立するのに対して，因果関係は上述の関係の反射法則だけが成立しない[4] といわれている．推移法則は，因果関係でも成立していることから，この概念をルール化して，因果チェインをたどるルールを示すと図-13 に示すようになる．このルールは，メソッド型知識として，エキスパートまたはユーザが使うことができる．

因果チェインをたどる述語 causal_chain の実行例を，図-14 に示そう．この図は，えさを与える (give) という事象が生じたときに，どのような事象が生ずるかを示す実行例である．この問い合わせに対する答として，えさを食べる (eat) という事象と元気になる (cheerful) という事象が連鎖的に生ずることが示されている．ここでは，因果関係の表現が trigger 型メタ知識で表現できることを示したが，4 (式) の起動条件の中に時間に関する表現がなされたときは，タイムト

```
cause(Operation1, Operation2) :-
        var(Operation2), !,
        Operation1 =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (trigger(Mode, Operation1) :- Goals)),
        find_causal_operation(Operation2, Goals).
cause (Operation1, Operation2) :-
        Operation2 =.. [Access_Operator, Frame, Object],
        clause_of(Frame, (trigger(Mode, Operation1) :- Goals)),
        setof(X, find_causal_ operation(X, Goals), Set),
        length(Set, 1), Set=[Operation2].
```

図-12 cause 述語のプログラム例
Fig. 12 Example of programs of cause-predicate.

```
/* causal chain */
causal_chain(Operation1, Operation2) :-
        cause(Operation1, Operation2).
causal_chain(Operation1, Operation3) :-
        cause(Operation1, Operation2),
        causal_chain(Operation2, Operation3).
```

図-13 causal_chain を表現するルール
Fig. 13 Rules representing the causal_chain

```
|-? - causal_chain(insert (frame, give(Agent, Receiver, Information)), X).
Receiver = _64,
Information = _91,
X = insert(frame, eat(_64, _91)),
Agent = _40 ;
Receiver = _64,
Information = _91,
X = insert(frame, cheerful(_64)),
Agent = _40
yes
```

図-14 causal_chain 述語の実行例
Fig. 14 Example of execution of causal_chain-predicate.

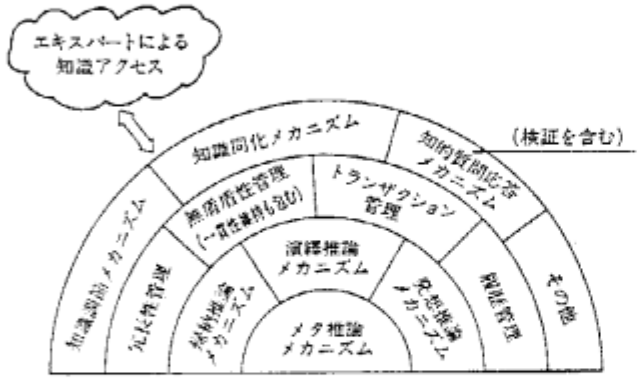

図-15　知識獲得機能のダイヤグラム
Fig. 15　Diagram of knowledge acquisition functions.

リガと呼ばれるメタ知識になる．

4.4　メソッド型知識

ここでは，知識獲得としての知識ベースアクセスに重点をおいて，そのアクセス法を提供するためのメソッド型知識について述べる．

知識獲得時に，エキスパートから獲得される知識には，構造化知識やプログラム化知識といったオブジェクト知識の他に，制約型知識のようなメタ知識があることは，すでに述べた．このときの知識獲得能力は，認知心理学で知られるピアジェの知性発達モデル[7]を参考に実現されており[11]，他に例をみない構成をしている．このモデルは，図-15 の最上位階層であり，その階層では，メソッド型知識の能力を人間の知識獲得能力としてとらえ，知識の同化（Assimilation），調節（Accommodation）をはじめとする種々の機能をエキスパートに提供している[11),15),16),21)]．知識同化は，知識ベースのフレームに暖昧な新知識（assumption-type の新知識）を無矛盾に追加することを意味する．知識調節は，知識ベースのフレームに正しい新知識（premise-type の新知識）を無矛盾に追加するために，そのフレーム内の知識を修正することを意味する．知識の同化・調節の本質的な問題は，知識ベースの無矛盾性を維持し，選択的に知識ベースの非冗長性を維持することである．さらに，知識ベースが持っている知識を，エキスパート（またはユーザ）が容易に調べられるようにするために，知的な質問応答（Intelligent Question-Answering）に基づく知識の検証（Verification）能力をもつ．

さて，知識獲得機能を支える基礎としては，図-15 に示すように，最初にメタ推論メカニズムを挙げることができる．このメカニズムは，推論エンジンによる推論過程を自由自在にあやつるメカニズムでもある．図中のメタ推論をベースとして，その次に基本的なメカニズムには，演繹，帰納，発想という推論メカニズムがある．メタ推論メカニズムは，ある制御条件下でゴール列を証明する役割を演じるのに対し，演繹推論メカニズムは，与えられたホーン節が証明可能か否かを解決する役割を演ずる．また，帰納推論メカニズムは，ファクトからルールを合成する役割を果たすメカニズムであり，このメカニズムでは，Shapiro が研究したモデル推論アルゴリズム[24),25)]を高速化などのために改善し，実現している[11),15)]．さらに，発想推論メカニズムでは，類推[8]などの高度な推論を行うメカニズムが必要であると考えている．さらにその上には，冗長性管理，無矛盾性管理，トランザクション管理，履歴管理などがある．冗長性管理は，知識ベース内の知識の冗長性を除去する問題を考えることであり[11),15)]，無矛盾性管理は，制約型知識で，知識ベースを無矛盾にする管理である．トランザクション管理については，5章を参照されたい．履歴管理は，

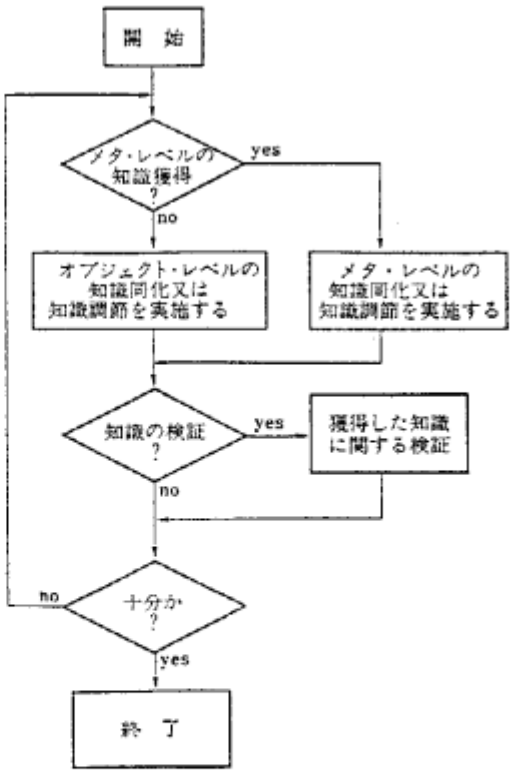

図-16　知識獲得プロセス（均衡化プロセス）
Fig. 16　Knowledge acquisition process (Equilibration process).

時系列の知識に対する時間的な推論を中心に扱う機能と考えているが，これについては，機会を改めて，述べることにしたい．

次に，エキスパートの手によって，いかにしてメソッド型知識を利用するのかについて説明してみよう．獲得される知識は，オブジェクト知識とメタ知識の二種類に大別されているので，図-16 に示される二種類の知識獲得プロセスが存在すると考えることができる．ピアジェの知性発達モデルで知られる均衡化は，このプロセスを通して達成される．このプロセスに関する例としては，積み木の問題，文法推論の問題としてすでに発表済みである[11),16)]が，ここでは，DCG (Definite Clause Grammar) の文法推論を例として，簡単に説明することにする．

この文法推論においては，オブジェクトレベルの知識獲得が利用されるが，その獲得プロセスにおいて文法推論に対する制約条件としては，次の形式のメタ知識が定義されているものとする．

```
inconsistent(immediate, insert(dcg_frame, __)) :-
        s(X, [ ]), !, exist_variable (X).        (7)
```

これは，文法ルールにより生成される文章には，どの文章も変数(未定義の単語)が含まれてはならないという制約条件である．このような状況で，ユーザが文法ルールに関する知識の断片の一つを知っているものとすると，ユーザは，次のような知識同化コマンドを入力することになる．

```
| ?- assimilate(dcg_frame,
        (s(X, Y) :- np(X, Z),
        vp(Z, W))).                              (8)
```

ところが，このコマンドの中で，“W”を“Y”と書かなければならないにもかかわらず，“W”と書いてしまったとすると，(7)式に違反するので，エラーメッセージが出力されることになる．そこで，(8)式の“W”を“Y”におきかえて，このコマンドを再実行すると，知識の同化が矛盾なく実行されることになる．次に，文法ルールの断片として同化された知識に基づいて，具体的な文章をいくつか推論してみると，ユーザが具体例としてもっている文章が推論されていないことがある．これは，文法ルールが不完全であることによる．ここでは，文章として出現する前置詞句の表現能力がないとして話を進めることにする．この文法ルールを半自動的に修正するためには，知識調節コマンドを次のように使用すれば良い．

```
| ?- accommodate(dcg_frame,
        s([hermia, walks, in, forest], [ ])).
                                                 (9)
```

ここでは，[hermia, walks] の文章までは，文法ルールから推論できるが，[in, forest] の前置詞句までは推論できないという前提なので，[hermia, walks, in, forest] の文章が推論できる文法ルールを作りあげるための知識調節コマンドを実行している．その結果，文献16) では，文法ルール “vp(X, Y) :- vi(X, Y)” が “vp(X, Y) :- vi(X, Z), pp(Z, Y)” に修正されることがわかっている．ここで vp(X, Y), vi(X, Z), pp(Z, Y) は，おのおの動詞句，自動詞，前置詞句である．

5. 制約型知識のトランザクション処理

ここでは，制約型知識を実行するタイミングを明確

```
transaction(X) :-
        lock(X),
        execute(X),
        save_transaction(X),
        unlock(X).
transaction(X) :-
        restore_transaction(X),
        unlock(X).
execute(X) :-
        execute1(X), !,
        maintain_constraints(delayed, X).
execute1(true) :- !.
execute1((P, Q)) :-
        execute1(P), execute1(Q).
execute1(P) :-
        method_type(P), call(P), !,
        maintain_constraints(immediate, P).
execute1(P) :-
        system(P) ->call(P);
        clause(P, Q), execute1(Q).
maintain_constraints(Mode, true) :- !.
maintain_constraints(Mode, (P, Q)) :-
        maintain_constraints (Mode, P), maintain_constraints(Mode, Q).
maintain_constraints(Mode, P), :-
        operation_type(P), !,
        \+check_inconsistency(Mode, P), maintenance(Mode, P).
maintain_constraints(Mode, P) :-
        system(P) ->true;
        clause(P, Q), maintain_constraints(Mode, Q).
```

図-17 トランザクション処理のプログラム例
Fig. 17 Example of programs of transaction processing.

```
| ?- super_concept(frame, X, plant).
X = plant_for_food;
X = fruit;
no
| ?- transaction(insert(frame, is_a(chickweed, vegetable))).
* Transaction Error.
yes
| ?- transaction((insert(frame, is_a(chickweed, vegetable)),
|                 insert(frame, is_a(vegetable, plant_for_food)))).
* End Transaction.
yes
| ?- super_concept(frame, X, plant).
X = plant_for_food;
X = fruit;
X = chickweed;
X = vegetable;
no
```

図-18 トランザクション処理の例
Fig. 18 Example of transaction processing.

にするために，トランザクション処理の概念を導入して，制約型知識とトランザクション処理の関係について述べる．知識ベースに対する更新は，4.1 節で示されたように，知識獲得プロセスの一環として，実施されるが，この更新により，知識が誤った方向に成長することを防止するための判定が制約型知識により常に行われている．この判定のタイミングには，更新があるたびに行われる場合と，ある一連の更新群が終了した後に行われる場合の二つがある．後者は，一連の更新群を定義するためにトランザクションという処理単位が必要であり，このトランザクション処理の中では，遅延型の制約型知識の扱いが重要になってくる．図-17 に，トランザクション内の一組の更新アクセス群を無矛盾に実行するプログラム例を示す．一般に，トランザクション処理は，排他制御の区切りとしても利用されるが，ここでは，重要でないので，図中の lock と unlock 述語については，無視してもらいたい．図-17 の transaction 述語を実行する第一番目のルールは，トランザクションを execute 述語で実行し，これが成功した場合に，後始末をする処理が save_transaction 述語で行われている．第二番目のルールは，トランザクション処理が失敗したときの処理を示しており，この後始末は，restore_transaction 述語で行われている．execute 述語を実現するルールは，図-17 に示してある通り，トランザクション内のゴール列を実行する execute1 述語と，これにより，遅延型 (delayed_type) の制約型知識を実行する maintain_constraints 述語から成る．maintain 述語は，遅延型でも即時型でも評価実行できる述語であり，この述語を実現する三番目のルールに制約型知識の具体的評価が行われている．ここで，check_inconsistency 述語及び maintenance 述語の実現例は，すでに，4.1 節で示した．

以下に，トランザクション処理の例を，図-18 をもとに説明してみる．図中の最初の例は，is_a(chickweed, vegetable) という is_a 関係の知識挿入例であるが，図-8 の三番目のメタ知識に違反したので，この処理が不成功に終っている．ところが，次の例では，is_a(chickweed, vegetable) 及び，is_a(vegetable, plant_for_food) を同じトランザクション処理内で知識を挿入しているので，この処理が成功している．

6. おわりに

本論文では，Prolog による知識ベース管理の基本技術として，次のような結論を得た．

（1） 構造化知識としての概念階層関係を簡単な形で表現し，この関係に関する諸性質を，super_concept, sub_concept, inherit のルールとして整理した．

（2） 著者らは，demo 述語によるメタ推論を利用して，制約型知識の実行方式を示してきた．とりわけ，trigger 型のメタ知識は，自然言語処理で良く使われる因果関係の表現を包含していることを示した．

（3） メタ・レベルの知識調節を実現するための基礎技術として，demo 述語を利用した矛盾知識の抽出法と，その効率的な修正手順を明確にした．

（4） 制約型知識の実行に種々の可能性を設けるために，その実行のタイミングを表す概念 (immediate-type と delayedtype) を取り入れ，トランザクションという処理単位で統合した．

demo 述語を利用したプログラミング技法は，メタプログラミング技法と呼ばれているが，この技法は，知識ベースのエディタやデバッガの開発，論理プログラムの自動合成，Prolog と関係データベースとのインタフェースの実現[28]などにおいてもその有用性が確認されつつある．また，この技法によるコンパイル化技術も開発されつつある[26]．しかしながら，demo 述

語そのものに対する理論的裏付けについては，現在のところ若干の結果が得られているにすぎない[39]．今後の課題としたい．さらに，曖昧な知識を確信度（Certainty Factor）として表現した場合や，時相論理及び並列処理を導入した場合の知識ベースの管理方法などは，今後の課題である．また，発想推論の問題も今後の大きな課題であることを付け加えておきたい．

参考文献

1) Allen, F.: Recognizing Intention from Natural Language Utterances, in M. Brady et al., (ed.) Computational Model of Discourse, MIT Press (1983).
2) Bowen, D. L.: DECsystem-10 PROLOG USER's Manual, DAI, University of Edinburgh (1981).
3) Bowen, K. A. and Kowalski, R. A.: Amalgamating Language and Meta-language in Logic Programming, TR 4/81, Syracuse University (1981).
4) Bunge, M.: Causality-the place of the Causal Principle in Modern Science, The President & Fellows of Harvard College (1959).（日本語訳では岩波書店，黒崎宏訳，「因果性」(1972) をみよ）．
5) Chamberlin, D. D. et al.: SEQUEL 2: A Unified Approach to Definition, Manipulation, and Control, IBM J. RES. DEVELOP. (1976).
6) Furukawa, K., Takeuchi, A., Kunifuji, S., Yasukawa, H., Ohki, M. and Ueda, K.: Mandala: A Logic Based Knowledge Programming System, Proc. of FGCS '84, Tokyo, Japan (1984).
7) 波多野誼余夫：認知心理学講座，第4巻，東京大学出版会 (1982).
8) Haraguchi, M.: An Analogy as a Partial Identity," Proc. of the Logic Programming Conference '84, 11-2, ICOT, Mar. (1984).
9) 市川亀久彌：「創造性の科学」，日本放送出版協会 (1970).
10) 北上 始，麻生盛敏，國藤 進，宮地泰造，古川康一：知識同化機構の一実現法，情報処理学会知識工学と人工知能研究会資料 30-2 (1983).
11) Kitakami, H., Kunifuji, S., Miyachi, T. and Furukawa, K.: A Methodology for Implementation of a Knowledge Acquisition System, Proc. of the 1984 International Symposium on Logic Programming, Atlantic City, U. S. A., Feb. 6-9 (1984), also available from ICOT as ICOT Technical Report TR-037 (1982).
12) 北上 始，國藤 進，宮地泰造，古川康一：Kaiser/KIM Architecture, ICOT Technical Report TM-0063 (1984).
13) 北上 始，國藤 進，宮地泰造，古川康一：大規模な知識ベース管理システムをめざして，情報処理学会知識工学と人工知能研究会資料 (1984).
14) 北上 始，宮地泰造，古川康一，國藤 進：知識獲得システムの分散処理にむけて(3)，情報処理学会第29回全国大会論文集 (1984).
15) 北上 始，國藤 進，宮地泰造，古川康一：知識の同化・調節・均衡化などのユーザ・インターフェースを備えた知識獲得システム，特集 ロジックプログラミング，日経エレクトロニクス，11.5 号 (1984).
16) Kitakami, H., Kunifuji, S., Miyachi, T., Furukawa, K., Takeuchi, A., Miyazaki, T., Ishii, S., Takewaki, T. and Ohki, M.: Demonstration of the KAISER System at the ICOT Open House in FGCS '84, ICOT TR-0081 (1984).
17) Kunifuji, S. and Yokota, H.: PROLOG and Relational Data Bases for Fifth Generation Computer Systems, Proc. of CERT Workshop on "Logical Bases for Databases" (1982).
18) 國藤 進，麻生盛敏，竹内彰一，坂井 公，宮地泰造，北上 始，横田始夫，安川秀樹，古川康一：Prolog による対象知識とメタ知識の融合とその応用，情報処理学会知識工学と人工知能研究会資料 30-1 (1983).
19) 國藤 進，北上 始，宮地泰造，古川康一：知識工学の基礎と応用—第4回 Prolog における知識ベースの管理，計測と制御，Vol. 24, No. 6 (1985).
20) Minsky, M.: Framework for Representing Knowledge, in Winston (ed.): The Psychology of Computer Vision, McGraw-Hill (1975).
21) Miyachi, T., Kunifuji, S., Kitakami, H., and Furukawa, K.: A Knowledge Assimilation Method for Logic Databases, New Generation Computing, 2-4, 385/404 (1984).
22) 宮地泰造，國藤 進，古川康一，北上 始：Constrait に基づく論理データベース管理について，情報処理学会知識工学と人工知能研究会資料，9月 (1984).
23) Ong, J., Fogg, D. and Stonebraker, M.: Implementation of Data Abstraction in the Reltaional Database System INGRES, ACM SIGMOD RECORD (1984).
24) Shapiro, E.Y.: Inductive Inference of Theories From Facts, Yale University Research Report 192 (1981).
25) Shapiro, E. Y.: Algorithmic Program Debugging, An ACM Distinguished Dissertation 1982, The MIT Press (1982).
26) 竹内彰一，近藤浩康，大木 優，古川康一：部分計算のメタプログラミングへの応用，情報処理学会ソフトウェア基礎論研究会資料 (1985).

27) 田中 譲他：推論システムとデータベースシステムとの部分評価機構による結合，第1回計測自動制御学会知識工学シンポジウム資料 (1983).
28) Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S. and Murakami, K.: An Enhanced Inference Mechanism for Generating Relational Algebra Queries, Proc. of Third ACM SIGACT-SIGMOD Symposium on Principles of Database System, Waterloo, Canada, April 2-4 (1982).
29) Weyhrauch, W.: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, 13, 13/170 (1980).

(昭和60年7月1日受付)

```
demo(FRL, true, [Res, Cond, Cut, Count], [true, d(X, X)]) :- !.
demo(FRL, Goals, [Res, Cond, Cut, 0], [overflow, d(X, X)]) :- !.
demo(FRL, !, [Res, Cond, Cut, Count], [Result, d(X, Y)]).
demo(FRL, !, [Res, Cond, cut, Count], [Result, d(X, Y)]).
demo(FRL, (P ; Q), [Res, Cond, Cut, Count], [Result, d(X, Y)]) :- !,
     (demo(FRL, P, [Res, Cond, Cut, Count], [Result, d(X, Y)]);
     demo(FRL, Q, [Res, Cond, Cut, Count], [Result, d(X, Y)])).
demo(FRL, (P, Q), [Res, Cond, Cut, Count], [Result, d(X, Z)]) :- !,
     demo(FRL, P, [Res, Cond, Value, Count], [Result1, d(X, Y)]),
     (Value == cut, Cut = cut,  Result = Result1, Z = Y, !;
     Result1 = true -> demo(FRL, Q, [Res, Cond, Cut, Count], [Result, d(Y, Z)]);
     Result = Result1).
demo(FRL, P, [Res, Cond, Cut, Count], [Result, d(X, Y)]) :-
     system (P) -> P, Result = true, X = Y;
     Count1 is Count - 1,
     clause_of(FRL, ID, (P :- Q), Certainty, [Res, Cond]),
     (Certainty == premise  -> X = Z ; X = [(P : - Q) | Z]),
     demo(FRL, Q, [Res, Cond, Cut, Count1], [Result, d(Z, Y)]),
     (Cut == cut, !, fail ; true).
clause_of((FR, FRL), ID, (Head :- Goals), Certainty, [Res, Cond]) :- !,
     (clause_of(FR, ID, (Head :- Goals), Certainty, [Res, Cond]);
     clause_of(FRL, ID, (Head :- Goals), Certainty, [Res, Cond])).
clause_of(FR, ID, (Head :- Goals), Certainty, [Res, Cond]) :-
     next_clause(FR, ID, Clause, Certainty, Cond),
     (Clause = (Head :- Goals);
     Clause = Head, Goals = true).
clause_of(FR, [ ], (Head : -true), Certainty, [Res, Cond]) :-
     Res\ == [ ], unify(Res, Head).
next_clause(FR, ID, Clause, Certainty, Cond) :-
     next_clause1(FR, ID, Clause, Certainty, ID1),
     Clause\ == [ ], check_condition(ID, Cond).
next_clause1(FR, [P1, P2, P3], Clause, Certainty, [P1, P2, P3]) :-
     not(var(P1)), X = .. [FR, P1, P2, P3, Clause, Certainty], X.
next_clause1(FR, ID, Clause, Certainty, [P1, P2, P3]) :-
     var(P1), !, dictionary(FR, [P1, _, _], _),
     X = .. [FR, P1, P2, P3, _, _], X, !,
     next_clause1(FR, ID, Clause, Certainty, [P1, P2, P3]).
next_clause1(FR, ID, Clause, Certainty, [P1, P2, P3]) :-
     X = .. [FR, P2, Q2, Q3, _, _], X, !,
     next_clause1(FR, ID, Clause, Certainty, [P2, Q2, Q3]).
check_condition(ID, [ ]).
check_condition(ID, [ID | Y]) :- !, fail.
check_condition(ID, [X | Y]) :- check_condition(ID, Y).
unify((C, CL), P) :- !,
     (unify(C, P) ; unify(CL, P)).
unify(P, P).
```

図-19 4引数 demo 述語のプログラム例

Fig. 19 Example of programs of demo-predicate which has four arguments

付 録

一段的な demo 述語（4引数）のプログラム図-19 に，4引数の demo 述語のプログラム例を示す．この述語の第1引数には，現在関係している知識の集合体のいくつかがリストとして記述される．第2引数には，証明すべきゴール列が記述され，第3引数には，[Res, Cond, Cut, Count] が記述される．Res は，現在関係している知識の集合体に，一時的に追加される知識であり，demo 述語の証明過程で，知識の集合体の一部として利用される．Cond は，知識の集合体に存在する知識の中で，証明に利用したくない知識の識別子であり，一般に，その識別子のリストとなる．Cut は，カットシンボルによる制御を行うための作業用変数である．Count は，深さ方向に推論を進めていったときに許される最大推論ステップ数である．これにより，ループによるメモリのパンクを防止できる．第4引数には，推論の結果として，二つの情報が返される．一つ目は，推論が成功したか否かが "true/overflow" で返される．二つ目は，推論過程で利用された仮定型知識だけから成る証明木（前提型知識が除かれている）が返される．

図-19 の demo 述語を利用した1つのプログラムを図-20 に示しておく．図-20 のプログラムは，演繹推論メカニズムを提供する deduce 述語である．この述語は，知識の同化などに利用されている[11]．deduce 述語は，demo 述語とは違って，ホーン節（P :- Q）の証明問題を解く能力がある．図中の select_variable_list 述語は，証明すべきホーン節 Clause がもつ変数（これは全称限定されている）を選択する述語である．skolem 述語は，そこで選択された変数に，二度と現れることのない定数を割り付ける述語である．

```
deduce(FRL, Clause, Cond) :-
  verify((select_variable_list(Clause, Variable_list),
    skolem(Variable_list), (Clause = (P :- Q) -> >demo(FRL, P, [Q, Cond, Cut,
    50], [true, Ptree]) ; demo(FRL, Clause, [[ ], Cond, Cut, 50], [true, Ptree]))).

verify(P) :- \+(\+(P)).

select_variable_list(Clause, Variable_list) :- !,
      select_variable(Clause, [ ],Variable_list).
select_variable((P :- Q), Vs, Vf) :- !,
      select_variable(P, Vs, Vi), select_variable(Q, Vi, Vf).
select_variable ((P ; Q), Vs, Vf) :- !,
      select_variable(P, Vs, Vi), select_variable(Q, Vi, Vf).
select_variable((P ; Q), Vs, Vf) : -!,
      select_variable(P, Vs, Vi), select_variable(Q, Vi, Vf).
select_variable(P, Vs, Vf) :- !,
      select_variable1(P, Vs, Vf).

select_variable1(P, Vs, Vs) :-
      atomic (P), !.
select_variable1(P, Vs, Vf) :-
      var(P), !, unique_variable1(P, Vs, Vf).
select_variable1(P, Vs, Vf) :-
      P=..[Predicate_name | Arguments],
      select_variable2(Arguments, Vs, Vf).

select_variable2([ ], Vs, Vs) :- !.
select_variable2([A1 | A2], Vs, Vf) :-
      select_variable1(A1, Vs, Vi),
      select_variable2(A2, Vi, Vf).

unique_variable(X, [ ], [X]).
unique_variable(X, [Y | Z], [Y | Z]) :-
      X==Y, !.
unique_variable(X, [Y | Z], [Y | W]) :- !,
      unique_variable(X, Z, W).

skolem(X) :- !, skolem1(X, 0).
skolem1([X | Y], Count) :-
      name('S', [L1]), Count1 is Count+1, name(Count1, L2),
      name(X, [L1 | L2]), !, skolem1(Y, Count1).
skolem1([ ], Count).
```

図-20 deduce 述語のプログラム例

Fig. 20 Example of programs of deduce-predicate.