TR-031

Prologによる推論機構と関係データベースの結合

横田 治夫、国藤 進、柴山 茂樹
宮崎 収兄、角田 健男、村上 国男

December, 1983

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

Institute for New Generation Computer Technology

Ｐｒｏｌｏｇによる推論機構と関係データベースの結合

横田　治夫　　国藤　進　　柴山　茂樹　　宮崎　収兄　　角田　雄男　　村上　国男
（（財）　新世代コンピュータ技術開発機構　）

［要旨］

関係データベース機構と推論機構を結合することは，知識ベース機構を実現するための研究の第一歩であると同時に，LFP(Least Fixed Point)オペレーションも扱える高機能のデータベース問合せ言語を提供することも意味している．　我々はすでに論理型プログラミング言語と関係代数の間のインタフェースを取るシステムについて提案を行ったが，そのシステムは十分に洗練されていなかった．　我々は引続きそのシステムを改良して，処理の高速化を図るとともに，以前のシステムでは不可能だったLFP オペレーション処理の自動化を可能にし，評価可能述語，あるいは cut オペレータ等も扱えるようにした．　本論文ではその改良システムについて報告する．

1．　はじめに

第五世代コンピュータ・プロジェクトでは，知識情報処理システムと呼ばれるものの構築を目標としている [1]．我々は，現在その第一ステップとして，関係データベース・マシン(RDBM)[2] とパーソナル型逐次推論(PSI) マシン [3] の開発を行っている．　Delta と名付けられたRDBMは，中期の研究開発のツールとして利用できるように実用性の高いマシンを目標としていて，関係代数をベースにしたコマンドによる質問を受け付けて，それに対するデータベース処理を高速に行うものである．　一方，PSIマシンは，FGKL(Fifth Generation Kernel Language) [1] と呼ばれるPROLOGベースの論理型プログラミング言語で書かれたプログラムを高速に実行するマシンである．

Delta はハードウェア・バックエンド・マシンであるため，ホスト・マシン（主に PSIマシン）に接続されて利用されることを想定している．　物理的には，Delta はLAN (Local Area Network)または共通バス上の共有メモリを介してホストと接続される．　前期におけるシステム構成を図１に示す．　大規模なデータベースを使ったソフトウェア開発支援システムを形成するためには，その物理的なインタフェースの上に論理的なインタフェースとして，論理型プログラミング言語と関係代数との間のインタフェースを設定する必要がある．　また，知識情報処理システム構築のための次のステップとして，知識ベース・マシン(KBM)を開発するために，接続に対する実験データを収集する事も必要である．

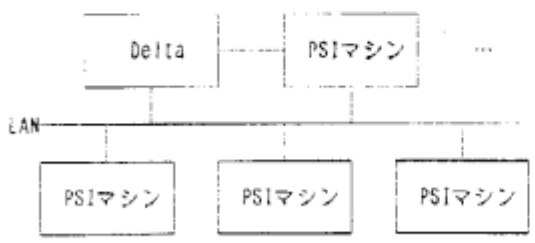

図１　システム構成

我々はすでに[5] において，PROLOGを用いてインプリメントしたPROLOGと関係代数の間のインタフェースを取るシステムについて報告を行った．　我々はその後，そのシステムに改良を加え，処理の高速化を図るとともに，各種の機能を付け加えた．本論文では，その改良したシステムについて報告する．

もともとPROLOGは，主メモリ上の小規模データベースに対し順次答の検索を行う高機能なデータベース問合せ言語と見ることもできる．　もし，システムが我々が提案するようなインタフェースを備えているとすれば，ユーザは一つのPROLOGの問合せを行うことにより，主メモリ上のデータベースだけでなく外部記憶上の大規模データベースからも，その問合せを満足するような解をすべて得ることができる．　さらに，プログラムに僅かな記述を加えることにより，ユーザはLeast Fixed Point (LFP) オペレーションと呼ばれるデータベースの問合せ処理[8,9,5] を行うこともできる．　LFP オペレーションの例としては，ある人の全ての祖先を捜す問題とか，可能な全ての航空便の接続を捜す問題などがよく知られている．　これらの問題は関係代数や関係論理だけでは扱うことが不可能であるが，その上にここで示す論理型プログラミング言語とのインタフェースを用意することにより解を得ることができる．

つまり，ここで提案するインタフェースにより，非常に高機能な大規模データベースを扱う能力が提供されることになる．　さらに，大規模なデータベースを扱う能力を持

った推論機構というのは，知識ベース機構を構築するための第一ステップと見なすこともできるのである．

PSIマシン上に置かれる本インタフェース・プログラムは，ユーザの一つのPROLOGの問合せに対して，最適化した関係代数のコマンド列を生成し，それを Deltaに送る．ユーザは，本インタフェースを用いることにより，どの述語が関係として Delta内に格納されているかを示すだけで，一般の述語だけでなく，評価可能述語やcut オペレータ等を使って，PROLOGのプログラムを，データベースとのインタフェースを考慮することなしに，書くことができる．さらにどの述語が Deltaに格納されているかという指示を，システムが自動的に示すようにすることも可能である．

2． インタフェースの基本的な考え方

論理型プログラミング言語と関係データベースとの間のインタフェースについては，すでにいくつかの提案がなされてきた[4,5,6,7]． 我々はそのうちのChakravarthyら[4] によって提案された方法に注目し，我々の目的に合うように発展させた．

PROLOGのプログラムは，ホーン節の集まりであり，ホーン節はその形態によりfactと呼ばれるものとruleと呼ばれるものに分類できる． factとは，帰結部のみからなるホーン節で，その引数に変数を含まないものを指す． 一方，ruleとは，帰結部および条件部からなるホーン節，もしくは帰結部のみからなるホーン節で引数に変数を含むものを指す． ここで，factの大部分を関係データベース側に，ruleおよび一時的に使われるようなfactを推論機構側に格納するものとする． 今後，簡略化のために，推論機構側のデータベースをIDB(Intensional Data Base)，関係データベース側のデータベースをEDB(Extensional Data Base)と呼ぶことにする．

論理型プログラミング言語と関係データベースの間でインタフェースを取るために，まずEDB 上の関係に対応する節は常に存在するという前提のもとに，IDB 上に格納されている節のみを用いてPROLOGのゴール（ユーザの問合せ）を演繹し，それと同時にEDB に対する問合せのプランを生成するような拡張した推論機構を考える． 我々はすでに[5] において，その推論機構の基本的な考え方と，DEC-10 PROLOGで書いた実験システムについて提案を行った． そのシステムのプログラム構成を図２に，処理の流れを図３に示す． このシステムは，プラン・ジェネレータ，プラン・コンバータ， Deltaシミュレータ，およびIDB とEDB から構成されている． ここで， Deltaシミュレータは，PROLOGのfactをリレーションの一つのタプルに見立てて，関係代数レベルのコマンドを実行するシミュレータであり，あくまでも Deltaの機能を模倣するものであり， Deltaの内部動作を模倣するものではない．

プラン・ジェネレータは，論理の証明可能性を表現するためのメタ述語 demo [12,4,5]をベースにし，IDB だけを使ってPROLOGのゴールを演繹して，同時にEDB に対するプランを生成するように拡張してある． 生成されたプランは，関係データベースとしてEDB に格納されているPROLOGの述語の列である． プラン・コンバータは，そのプラン

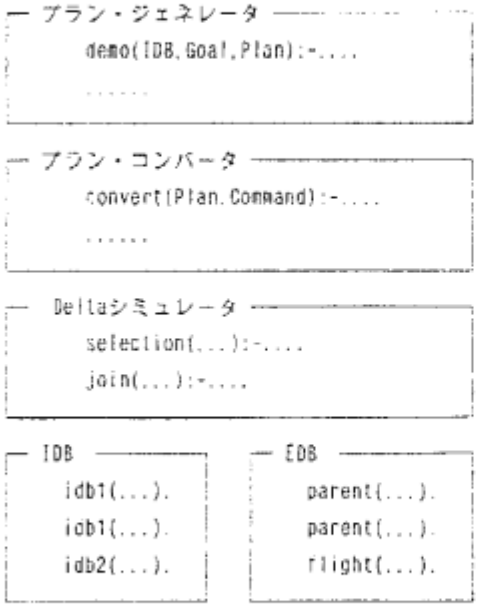

図２ プログラム構成

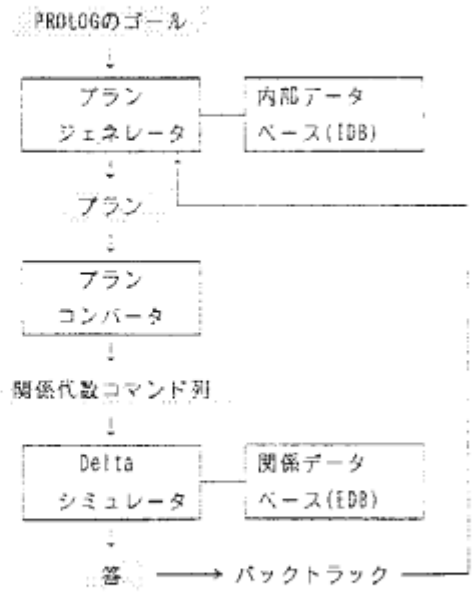

図３ 処理の流れ

を Deltaシミュレータが受付けることができる関係代数の列に変換する．　この過程は，ユーザによるバックトラックによって，すべての答が求まるまで繰返される．

例えば，親と祖先の関係を示すために，

```
idb1({ancestor(X,Y):-parent(X,Y)}).  ……(1)
idb1({ancestor(X,Y):-
      parent(X,Z),ancestor(Z,Y)}).  ……(2)
```

のような述語がユーザ・プログラムとしてIDB 中に格納されているとする．　ここで，"parent(X,Y)" という述語は "X" の親が "Y" であることを示している．　さらに，親に関するデータがEDB 上にあるとすると，

```
idb1({parent(X,Y):-edb(parent(X,Y))}).  ……(3)
```

という述語をIDB に加える．　これは，"parent"という述語がEDB に存在することを示している．

IDB に格納される節を，プログラム上で上記のように一つの述語として扱うことにより，その節がIDB 中のユーザ・プログラムとしての節であることを示すと同時に，述語名により異なる世界を構成することができる．　例えば，"idb2"という述語名を用いることにより，上記の"idb1"で示される世界とは異なる世界が定義できる（図４参照）．

ここで，"taro"の祖先を検索するために，

"?-ancestor(taro,A)." というPROLOGのゴールを，プラン・ジェネレータに送ったとすると，

```
[parent(taro,A)]  ……(4)
```

というプランが生成される．　プラン・コンバータはこのプランに対し，

```
[selection(parent,[1]=taro,temp1),*
   projection(temp1,[2],temp2),
   get(temp2)]  ……(5)
```

という関係代数ベースのコマンドの列を生成し，Deltaシミュレータへ送る．　Deltaシミュレータでは，このコマンド列に従って，"parent"という名前のリレーションから一番目の属性が"taro"であるようなタプルのみを捜してきて，"temp1" という名前のリレーションを作り，次にその二番目の属性のみをもつリレーション"temp2" を作る．

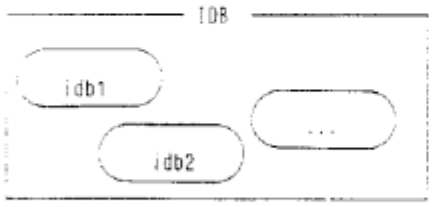

図４　多世界 idb

この"temp2" というリレーションが一連の処理の結果であるが，もしユーザがその結果に満足できない場合，あるいは全ての解を求めるような場合には，強制的にバックトラックを起して異なる結果を求めることになる．　結果に対してバックトラックが起った時に，Deltaシミュレータやプラン・コンバータは別解を持たないので，プラン・ジェネレータまで後戻りする．　プラン・ジェネレータは，このバックトラックに対して次のようなプランを別解として生成する．

```
[parent(taro,A1),parent(A1,A)]  ……(6)
```

プラン・コンバータはこれを，

```
[selection(parent,[1]=taro,temp3),
   join(temp3,parent,[2]=[1],temp4),
   projection(temp4,[4],temp5),
   get(temp5)]  ……(7)
```

という関係代数のコマンド列に変換し，Deltaシミュレータに渡す．　Deltaシミュレータはこのコマンド列に従って，"temp5" という名前の別のリレーションを作る．　もしこの結果にも満足できない場合や全ての解を求める場合には，さらにバックトラックを起して，次のようなプランを生成させる．

```
[parent(taro,A1),parent(A1,A2),parent(A2,A)]  (8)
```

この繰返しは，このようにユーザによって制御され，プランごとの解の集合が得られる．　このシステムの詳しい説明は，[5] を参照されたい．

３．　システムの改良

我々は，上記のシステムをさらに改良して，最適化したプランをより高速に生成し，さらにLFP オペレーションを機械的に取扱えるようにした．

もともとプラン・ジェネレータは，EDB 上の関係に対応する節は常に存在するという前提のもとに，IDB 上の節のみを用いて演繹を行うもので，生成されるプランは高々その一つの解に対応しているだけであり，一般のPROLOGの推

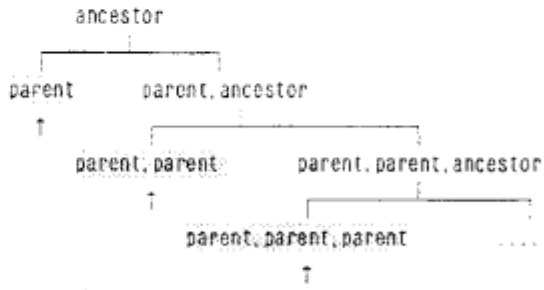

図５　探索木とプランの例

* 本論文で使う関係代数コマンドでは，［ｎ］はリレーションのｎ番目の属性を示すものとする．

論と同様に他にも解は存在し得る（図5参照）． PROLOGの推論で全ての解を求める時にバックトラックを起すように，[5] で報告したシステムでも，問合せを満足する全ての解を得るために，ユーザがその過程に介入して，強制的にバックトラックを起す必要があった． しかし，もし不用意に常にバックトラックを起すようにシステムを変更すると，停止不可能な無限再帰呼出しが起ることもあり得るし，得られた解が全解であるかどうかをユーザが調べることは極めて困難であるため，全解を求めるには以前のシステムでは不十分であることが分る．

例えば，上の例の(3) の節の中の"parent"という述語は，常にEDB 上に存在するという仮定があるので，(2) の再帰呼出しは停止条件を持たなくなる． このため，常にバックトラックを起すようにすると，この再帰呼出しは止まらなくなる． ユーザが一回一回呼出しに関与するにしても，全ての解が求まったかどうかは，常にユーザに分るわけではない． 別の表現をすると，(1),(2),(3) のプログラムで全解を求めるような処理は，LFP オペレーションと見做すこともでき，その点から言うと[5] で述べた実現方式は，完全に機械的にLFP オペレーションを取り扱ってはいないとも言える．

システムが完全に機械的にLFP オペレーションを取り扱うためには，システムは常に強制的なバックトラックを行うと同時に，空バックトラック(EBT:Empty-Backtrack) と非空バックトラック(NEBT:Non-Empty-Backtrack)とを区別する必要がある． EBT は問合せを満足する新しいタプルが一つも検索されない時のみ発生し，NEBTはそうでない時に発生するものとする． Aho ら[6] は，LFP オペレーションの問合せに対する繰返し処理として，図6に示すような処理を提案しているが，NEBTは条件が成立つ場合のwhile ループに対応し，EBT は条件が成立たない場合に対応する．

ここで，PROLOGはバックトラックが発生されると同時にバックトラックが起った部分の変数のバインディング情報が消されるため，その引数によってバックトラックの発生理由を知ることはできない． つまり，引数だけではEBTとNEBTの区別が付かない． そこで我々はバックトラックの発生理由を残すために，"assert"，"retract" というようなPROLOGの副作用を利用することにした． つまり，もしひとつの関係代数コマンド列に対して，新しいタプルが一つも検索されなかった場合には，EBT であることを示すために，"empty" という述語を"assert"してからバックトラックを起し，もし一つでも新しいタプルが見つかった場合には，NEBTであることを示すために，何も"assert"せずにバックトラックを起すことにする． 処理の流れを図7に，プログラムのメインの部分を図8に示す．

プラン・ジェネレータの方では，バックトラックに対し

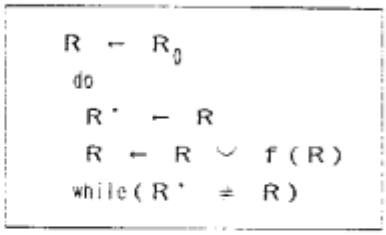

図6 LFP オペレーション

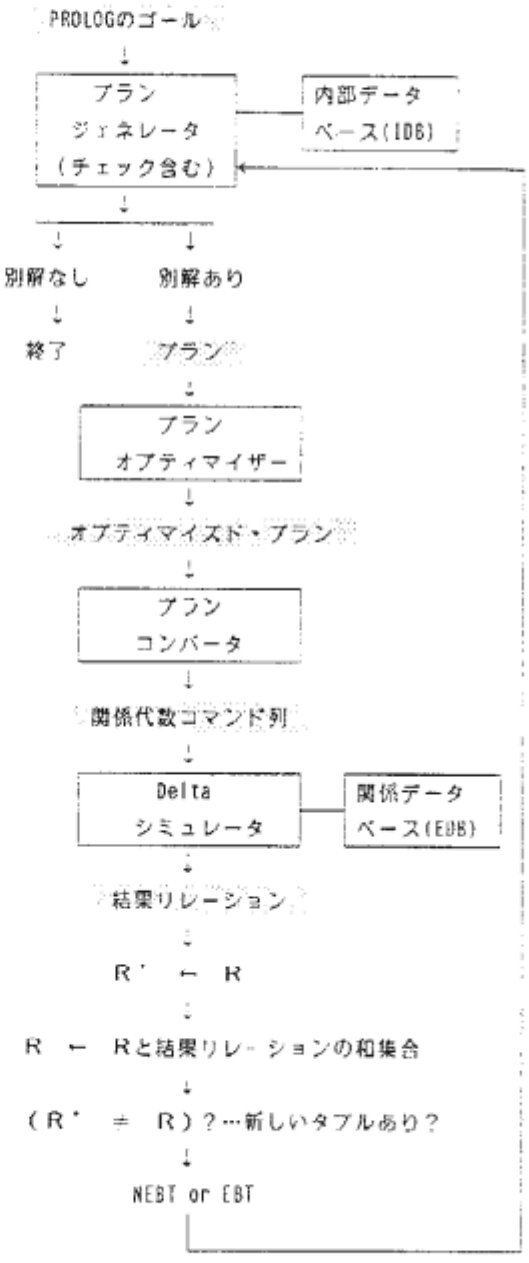

図7 改良システムの処理の流れ

て，必要に応じてこの"empty"という述語が存在するかどうかをチェックしながら，プランを生成する．　つまり，もし"empty"述語が存在しなければさらに探索木のその枝を伸ばし，存在すれば別の枝に移るようにする．　ここで，もし"empty"述語があった場合には，次の枝を伸ばせるように，"empty"述語を"retract"しておかなければならない．

この"empty"述語のチェックというのは，再帰呼出しプログラムに対する停止条件として働いている．　このため，どの時点でそのチェックを行わなければならないかはプログラム内容に依存するので，プログラムに付加情報を加えてチェックのタイミングを示すことにする．　我々はそのために"check"という述語を用意した．　ユーザは再帰呼出しのプログラムで，深さが増す時にチェックを行うように，再帰呼出しの条件部の頭にこの"check"述語を付加する．　例えば，前述の例の場合には，(2)の節を

```
idb1((ancestor(X,Y):-
        check,parent(X,Z),ancestor(Z,Y))).  ……(9)
```

のように変更する．

我々は，この"check"述語は極めて自然なものだと思っている．　というのは，もともと(2)の節の中の"parent"の述語は，PROLOGではこの再帰呼出しの停止条件として働いており，もしこれを"ancestor"と置き変えると（つまり，左再帰呼出しに変えると），論理的には正しいのだけれども，現在のPROLOGの処理系では，この再帰呼出しは止まらなくなってしまう．　つまり，"check"述語はこの停止条件を単に明確にしただけと考えることができる．　さらに，このためこのシステムでは，"check"述語を用いることにより，左再帰呼出しのプログラムも停止させることができる．　ただし，その場合には探索木の探索の仕方がdepth-first法からbreadth-first法に変るが，全ての解を求める場合には，どちらの探索方法をとっても最終的に同じ結果になるため，左再帰呼出しでも右再帰呼出しでも同じプログラムが書けることになる．　またプログラム中に複数の再帰呼出しがネストしてある場合でも，それらは同一の"check"述語を停止条件として用いればよい．　これは，"empty"という述語の存在が，一番下のレベルの再帰呼出しの深さをそれ以上伸ばさなくてもよい，ということを示しているためである．

変換された関係代数演算がモノトニック[8]なものになるようなPROLOGのプログラムである限り，このシステムは一つの問合せに対する全ての解を求めて，プランの別解がなくなった所で停止する．　なお，新しいタプルが見つかったかどうかを知るために，図6のRとR´に対応するリレーションをそれぞれ用意しておいて，Rとそれぞれのプランに対応する結果リレーションの論理和をとって，図6と同様のチェックを行ってNEBTとEBTの区別をすることにする（図7参照）．　もし，単に結果リレーションが一つもタプルを持たない場合にEBTを起すようになっていると，データベース中にそれ自体が周期的であるようなデータ，例えば"parent(taro,taro)"とか"parent(taro,hanako)"と"parent(hanako,taro)"の組のようなもの，があった場合には，結果リレーションは空にならないので，システムはたとえ"check"述語を用いても停止できない．　その場合には，データベースの無矛盾性のチェック[10]をするか，ユーザが注意して問合せを"?-ancestor(X,Y),X<>Y."のようにする必要が生じる．　前述のように論理和を使って新しいタプルの生成をチェックをすることにより，上記のようなデータがあった場合にも，無矛盾性のチェックやユーザによる特別な指定なしに，システムは停止する．

今回の改良の主題は，上述のLFPオペレーションの取扱いの他に，プラン生成の高速化とプランの最適化であり，それらについては，以下の章で述べることとする．　それ以外のプログラムのコンポーネント，つまりプラン・コンバータとDeltaシミュレータ，については特に大きな変更はない．　ただし，報告[5]では，評価可能述語はデータベース側で評価するようにすれば取り扱えることは述べたが，実際にどう実現するかについては示していなかったし，プラン・コンバータの中に，p(X,X)のような型の述語が与えられた時に生じるエラーがあったので，今回のシステムでは，プラン・コンバータを評価可能述語およびp(X,X)型の述語を取り扱えるように改良した．　その他，not述語

```
go(IDB,Goal) :- demo(IDB,Goal,_,Plan),
        optimize(Plan,OptPlan),
        convert(OptPlan,Command),
        delta(Command,Relation),
        backtrack(Relation).
go(_,_) :- listing(out).

backtrack(Relation) :-
        delta([copy(out,out1),union(Relation,out),equal(out1,out)],_),
        eq_ans(no), !, retract(empty), fail.
backtrack(_) :- eq_ans(yes), !, assert(empty), fail.
```

図8　プログラムのメイン部分

の関係代数コマンドへの展開は検討中であるが，not 述語を含まないようなプランに対しては，プログラムの中に評価可能述語が入っていても，"selection"，"join"，"projection"の３種類の関係代数コマンドで足りることが分った．　なお，プラン・コンバータは，別の見方をすると，関係論理の問合せから関係代数の問合せに変更する変換プログラムと見ることもできる．

4．　プラン・ジェネレータの改良

[5] で示したプラン・ジェネレータは，ユニフィケーションさえもプログラムで行うようになっていたため，非常に複雑で，一つのプランを生成するのにかなりの時間を要した（特に，述語の中から，構造を解析しながらユニファイ可能な変数を捜し出すのに問題がかかった）．　プラン生成を高速化するため，改良した新しいプラン・ジェネレータでは，問合せのゴールを演繹するために，PROLOGがもともと持つユニフィケーションの機能を利用することにした．

我々は[10]の中で，[11]にある"A PROLOG Interpreter in PROLOG"を基にした拡張型のdemo述語を提案した．　このdemo述語はPROLOGのもともとのユニフィケーション機能を利用しており，さらに同時にcut オペレータ(!) も扱うことができるものである．　このdemo述語をプラン・ジェネレータに用いることにより，新しいプラン・ジェネレータはシンプルで洗練されたものになり，以前のプラン・ジェネレータに比べてプログラムは約り分の１のサイズに縮小された．　図９にそのdemo述語を使った新しいプラン・ジェネレータの核の部分を示す．　以前のプラン・ジェネレータは，ゴールを先頭から評価していく時に，EDB 上にある述語の評価の時点でその述語をゴールの一番最後に置いて，ゴールがEDB 上の述語のみになるまで評価して行き，結果としてEDB 上にある述語の列を得るというものであった．　これに対し，今回のプラン・ジェネレータは，EDB 上のの関係に対応する述語を評価する時点で，その述語を出力用のリストに付け加えていくというものである．　方法は異なるが，EDB 上の述語の評価を後回しにするという原理は同じである．

図９のdemo述語の第一引数は，IDB のスコープを示している．　つまり，この引数にIDB の世界の名前を示してやることにより，その世界の中で推論を行うことができるようになっている（図４参照）．　第二引数はPROLOGの問合せを入力するためのもの，第四引数はプランを出力するためのもので，出力は差分リスト（D-LIST:Difference-LIST）"d(X,Y)"によって行われる．　D-LISTはリストの先頭と末尾の二つのポインタの組を持つことによりappendなどのリスト処理を効率よく行うためのもので，"d(X,Y)"の"X" がリストの先頭，"Y" がリストの末尾を指している．　また，第三引数は、cut オペレータを扱うための作業用変数として使われる．　cut オペレータは，単にプランの生成過程を制御するものであるが，結果的に探索スペースを制御することになっている．　前述の"check" 述語と共に用いることにより，ユーザは自分の要求する答の範囲を，規定することができる．

また，プログラム中の"idb-clause(IDB,P,Q)" という述語は，"IDB" という変数で示される世界の中で"P" とユニファイ可能な帰結部を持つ述語の内の一つとユニフィケーションを行い，その条件部を"Q" として取ってくるためのものである．

ここで注意してほしいのは，評価可能述語（例えば"="，"<"，"<>"など）は，このプラン・ジェネレータの部分で評価されるのではなく，プラン・コンバータで"selection" とか"join"の条件に変えられて，Deltaシミュレータで評価されるものであるため，このプラン・ジェネレータのプログラム上には，特にそれを取扱うような部分はない点である．　その代りに，IDB 中のプログラムとして，例えば"=" に対しては，

```
idb(X=Y:-edb(X=Y)).  ……(10)
```

のような述語が必要となる．

このプラン・ジェネレータを，DEC-2060上のDEC-10 PRO

```
demo(IDB,true,_,d(X,X)) :- !.
demo(IDB,!,_,d(X,X)).
demo(IDB,!,cut,c(X,X)).
demo(IDB,check,_,d(X,X)) :-
        (empty, !, retract(empty), fail; true).
demo(IDB,edb(P),_,d([P|X],X)) :- !.
demo(IDB,not(P),V,d([not(L)|X],X)) :- !, demo(IDB,P,V,L).
demo(IDB,(P;Q),V,L) :- !, (demo(IDB,P,V,L); demo(IDB,Q,V,L)).
demo(IDB,(P,Q),V,d(X,Y)) :- !, demo(IDB,P,C,d(X,Z)),
        (C==cut, V=cut, !; demo(IDB,Q,V,d(Z,Y))).
demo(IDB,P,V,L) :- idb_clause(IDB,P,Q), demo(IDB,Q,V,L),
        (V==cut, !, fail; true).
```

図９　新しいプラン・ジェネレータの核

LOG コンパイル・バージョンで実行した結果，前述の例に対してCPU タイムで 5msから 110msで結果が得られた．これは以前のプラン・ジェネレータに比べると約20から50倍の速さである．　さらに，プラン・ジェネレータのプログラム中のループの数が減っているため，探索木の深さが深くなるほど，この差は大きくなる．

5．　プラン・オプティマイザーの導入

今までのシステムでは，最適化の処理をなにもしていなかったため，冗長な関係代数コマンドや一時的なリレーションが生成されていた．　たとえば，(7) のコマンド列の中の"temp3" は，(5) のコマンド列の中の"temp2" と同じ情報しか持たない．　つまり，実際は(7) のコマンド列の中の"selection(parent,[1]=taro,temp3)"というコマンドは冗長であり，"temp2" のリレーションを残しておけば，不要であることが分る．

冗長なコマンドやリレーションの生成を避けるために，プラン・ジェネレータで生成したプランとそれに対応するリレーションを組にして記憶しておき，最適化するのに利用することにする．　つまり，例えば(4) のプランと(5) のコマンド列実行の結果できるリレーションに対応する組を，

plan([parent(taro,X)],temp2(X)). ……… (11)

という述語の形にして覚えておくようにする．　新しく生成されたプランは，図１０で示すようなPROLOGの"append" 述語を基に作られたストリング・マッチャを使って，冗長な部分を，記憶してあるプランとリレーションの組のプランの部分から捜し，もし見付かった場合には，そのプランと組になっているリレーションによってその部分を置き換えるようにする．　探す対象の長さは自由であり，ストリングの先頭や末尾だけでなくストリング中のどこにあっても構わないし，ひとつのストリングの中に複数あっても構わない．　なお，図１０のプログラムの中の"eq-struct" という述語は，述語どうしの比較の時，単に"=" だけだと不要なインスタンシエーションまで行われてしまうため，これを避けるために，構造が等しいかどうかのみをチェックするためのものである．

```
optimize(TN,OUT1) :-
        plan(PL1,Rel),
        append(TPL,Rest,IN),
        append(Top,PL2,TPL),
        eq_struct(PL1,PL2),
        PL1 = PL2,
        append(Top,[Rel],TO),
        append(TO,Rest,OUT2),
        optimize(OUT2,OUT1).
optimize(X,X).
```

図１０　プラン・オプティマイザーの核

このプラン・オプティマイザはプラン・ジェネレータとプラン・コンバータの間に置かれる（図７参照）．　例えば，このオプティマイザを使うと(6) のプランは，

[temp2(X1),parent(X1,X)] ……… (12)

のように最適化される．　また，(7) の関係代数のコマンド列は

[join(temp2,parent,[1]=[1],temp3),
　projection(temp3,[3],temp4)] ……… (13)

というコマンド列に置き変えられる．　この最適化は，基本的でかなり簡単なものではあるが，これによりPROLOGと関係代数の間のインタフェースの効率はずいぶん改善される．　特に前述の例のように，再帰呼出しなどの場合は効果が大きい．

6．　おわりに

本論文では，以前提案した論理型プログラミング言語と関係代数の間のインタフェースを，実用に供することもできるように，機能を拡張し，処理の高速化を図ったので，それについて報告した．　特に，論理型プログラミング言語でLFP オペレーションを完全に機械的に扱えるようにしたことは，大規模なデータベースに対するユニバーサルなデータベース問合せ言語のサポートとして，意味が深いと思われる．　また，プランの生成を効率化したり，プランを最適化したりして処理を高速化することは，実際にこのインタフェースをPSI マシンと Deltaの間で使うためには必要なことである．　さらに，評価可能述語やcut オペレータが使えるということも，このインタフェースの実用化には必要な項目の一つである．　ただ，今回行った最適化は，まだまだ簡単なものだけで，データベース中の統計情報を考慮した最適化等，処理の効率化の研究を今後とも続けていく予定である．

また，このインタフェースは，与えられたハードウェアおよびソフトウェア環境での知識情報処理システムの研究の第一ステップの一つとしてすぐれた候補であると思われ，今後このインタフェースを基に知識ベースの研究を進めていくつもりである．　しかし，我々はこれが知識ベース構築のための唯一の候補だと思っているわけではなく，他にもいろいろな候補はあると考えている．　例えば，[13]で提案したような，知識ベースの中でユニフィケーションを行うというのもその内の一つである．　我々は，各種の知識ベース機構の研究を通して，第五世代のコンピュータ・システムにとって有用な知識ベースマシンの構築を目指している．

［参考文献］

[1] ICOT (ed.): Outline of Research and Development for Fifth Generation Computer Systems, May 1982.

[2] Murakami,K.,Kakuta,T.,Miyazaki,N.,Shibayama,S., and Yokota,H.: A Relational Data Base Machine: First Step To Knowledge Base Machine,Proc. of 10th Annual International Symposium on Computer Architecture,Stockholm/SWEDEN,June 1983, pp. 423-426.

[3] Nishikawa,N.,Yokota,H.,Yamamoto,A.,Taki,K.,and Uchida,S.: The Personal Sequential Inference Machine (PSI): Its Design and Machine Architecture,Proc. of Logic Programming Workshop, Algrave/PORTUGAL,June 1983, pp. 53-73.

[4] Chakravarthy,U.S.,Minker,J., and Tran,D.: Interfacing Predicate Logic Languages and Relational Database,Proc. of the First Int. Logic Programming Conf.,Faculte des Sciences de Luminy Marseille,France,Sept. 1982, pp. 91-98.

[5] Kunifuji,S.,and Yokota,H.: PROLOG and Relational Data Base for Fifth Generation Computer System, Proc. of ONERA-CERT Workshop on "Logical Base for Data Bases" edited by Gallaire,H.,Minker,J., and Nicolas,J.M.,ONERA-CERT,Toulouse,Dec. 1982.

[6] 田中，堀内，田川：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，計測自動制御学会，第一回知識工学シンポジウム，東京，昭和58年 3月

[7] Gallaire,H.: Logic Data Base vs Deductive Data Base,Proc. of Logic Programming Workshop, Algrave/PORTUGAL,June 1983, pp. 606-622.

[8] Aho,A.V. and Ullman,J.D.: Universality of Data Retrieval Languages,ACM/SIGPLAN Conf. on Principles of Programming Language, San Antonio, Jan. 1979, pp. 110-117.

[9] Naqvi,S.A. and Henschen,L.J.: Synthesizing Least Fixed Point Queries into Non-recursive Iterative Programs, Proc. of 8th IJCAI,Karlsruhe/WEST GERMANY, Aug. 1983, pp. 25-28.

[10]Kernel Language Design Group of ICOT Research Center: Conceptual Specification of the Fifth Generation Kernel Language Version 1(KL1), Internal Document of ICOT Research Center.

[11]Coelho,H., Cotta,J.C., and Pereira,L.M.: HOW TO SOLVE IT WITH PROLOG, 2nd edition,Laborato'rio National de Engenharia Civil, LISBOA, 1980.

[12]Bowen,K.A.,and Kowalski,R.A.: Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science Syracuse University, June 1981.

[13]横田，角田，宮崎，栗山，村上：知識ベースマシン構築のための一考察，情報処理学会第27回全国大会，2K-5，昭和58年10月

APPENDIX

This appendix shows an execution example of interface between PROLOG and Relational Data Base.

```
| ?- go(idb1,ancestor(taro,A)).

plan :  [father(taro,_40)]
opt_plan :  [father(taro,_40)]
command :  [selection(father,[1]=taro,temp1),projection(temp1,[2],temp2)]
memo :  plan([father(taro,_40)],temp2(_40))

plan :  [mother(taro,_40)]
opt_plan :  [mother(taro,_40)]
command :  [selection(mother,[1]=taro,temp3),projection(temp3,[2],temp4)]
memo :  plan([mother(taro,_40)],temp4(_40))

plan :  [father(taro,_210),father(_210,_40)]
opt_plan :  [temp2(_210),father(_210,_40)]
command :  [join(temp2,father,[1]=[1],temp5),projection(temp5,[3],temp6)]
memo :  plan([father(taro,_210),father(_210,_40)],temp6(_40))

plan :  [father(taro,_210),mother(_210,_40)]
opt_plan :  [temp2(_210),mother(_210,_40)]
command :  [join(temp2,mother,[1]=[1],temp7),projection(temp7,[3],temp8)]
memo :  plan([father(taro,_210),mother(_210,_40)],temp8(_40))

plan :  [father(taro,_210),father(_210,_294),father(_294,_40)]
opt_plan :  [temp6(_294),father(_294,_40)]
command :  [join(temp6,father,[1]=[1],temp9),projection(temp9,[3],temp10)]
memo :  plan([father(taro,_210),father(_210,_294),father(_294,_40)],temp10(_40))

plan :  [father(taro,_210),father(_210,_294),mother(_294,_40)]
opt_plan :  [temp6(_294),mother(_294,_40)]
command :  [join(temp6,mother,[1]=[1],temp11),projection(temp11,[3],temp12)]
memo :  plan([father(taro,_210),father(_210,_294),mother(_294,_40)],temp12(_40))

plan :  [father(taro,_210),mother(_210,_294),father(_294,_40)]
opt_plan :  [temp8(_294),father(_294,_40)]
command :  [join(temp8,father,[1]=[1],temp13),projection(temp13,[3],temp14)]
memo :  plan([father(taro,_210),mother(_210,_294),father(_294,_40)],temp14(_40))

plan :  [father(taro,_210),mother(_210,_294),mother(_294,_40)]
opt_plan :  [temp8(_294),mother(_294,_40)]
command :  [join(temp8,mother,[1]=[1],temp15),projection(temp15,[3],temp16)]
memo :  plan([father(taro,_210),mother(_210,_294),mother(_294,_40)],temp16(_40))

plan :  [mother(taro,_210),father(_210,_40)]
opt_plan :  [temp4(_210),father(_210,_40)]
command :  [join(temp4,father,[1]=[1],temp17),projection(temp17,[3],temp18)]
memo :  plan([mother(taro,_210),father(_210,_40)],temp18(_40))

plan :  [mother(taro,_210),mother(_210,_40)]
opt_plan :  [temp4(_210),mother(_210,_40)]
command :  [join(temp4,mother,[1]=[1],temp19),projection(temp19,[3],temp20)]
memo :  plan([mother(taro,_210),mother(_210,_40)],temp20(_40))

plan :  [mother(taro,_210),father(_210,_294),father(_294,_40)]
opt_plan :  [temp18(_294),father(_294,_40)]
command :  [join(temp18,father,[1]=[1],temp21),projection(temp21,[3],temp22)]
memo :  plan([mother(taro,_210),father(_210,_294),father(_294,_40)],temp22(_40))

plan :  [mother(taro,_210),father(_210,_294),mother(_294,_40)]
opt_plan :  [temp18(_294),mother(_294,_40)]
command :  [join(temp18,mother,[1]=[1],temp23),projection(temp23,[3],temp24)]
memo :  plan([mother(taro,_210),father(_210,_294),mother(_294,_40)],temp24(_40))

plan :  [mother(taro,_210),mother(_210,_294),father(_294,_40)]
opt_plan :  [temp20(_294),father(_294,_40)]
command :  [join(temp20,father,[1]=[1],temp25),projection(temp25,[3],temp26)]
memo :  plan([mother(taro,_210),mother(_210,_294),father(_294,_40)],temp26(_40))
```

```
plan :  [mother(taro,_210),mother(_210,_294),mother(_294,_40)]
opt_plan :  [temp20(_294),mother(_294,_40)]
command :  [join(temp20,mother,[1]=[1],temp27),projection(temp27,[3],temp28)]
memo :  plan([mother(taro,_210),mother(_210,_294),mother(_294,_40)],temp28(_40))

plan :  [mother(taro,_210),mother(_210,_294),father(_294,_378),father(_378,_40)]
opt_plan :  [temp26(_378),father(_378,_40)]
command :  [join(temp26,father,[1]=[1],temp29),projection(temp29,[3],temp30)]
memo :  plan([mother(taro,_210),mother(_210,_294),father(_294,_378),father(_378,_40)

plan :  [mother(taro,_210),mother(_210,_294),father(_294,_378),mother(_378,_40)]
opt_plan :  [temp26(_378),mother(_378,_40)]
command :  [join(temp26,mother,[1]=[1],temp31),projection(temp31,[3],temp32)]
memo :  plan([mother(taro,_210),mother(_210,_294),father(_294,_378),mother(_378,_40)

plan :  [mother(taro,_210),mother(_210,_294),mother(_294,_378),father(_378,_40)]
opt_plan :  [temp28(_378),father(_378,_40)]
command :  [join(temp28,father,[1]=[1],temp33),projection(temp33,[3],temp34)]
memo :  plan([mother(taro,_210),mother(_210,_294),mother(_294,_378),father(_378,_40)

plan :  [mother(taro,_210),mother(_210,_294),mother(_294,_378),mother(_378,_40)]
opt_plan :  [temp28(_378),mother(_378,_40)]
command :  [join(temp28,mother,[1]=[1],temp35),projection(temp35,[3],temp36)]
memo :  plan([mother(taro,_210),mother(_210,_294),mother(_294,_378),mother(_378,_40)

out(ichiro).
out(hanako).
out(yasuo).
out(keiko).
out(shigeki).
out(etsuko).
out(mayumi).

A = _40

yes
| ?- listing([idb1,father,mother,temp1,temp2]).

idb1((ancestor(_1,_2):-parent(_1,_2))).
idb1((ancestor(_1,_2):-check','parent(_1,_3)','ancestor(_3,_2))).
idb1((parent(_1,_2):-father(_1,_2))).
idb1((parent(_1,_2):-mother(_1,_2))).
idb1((father(_1,_2):-edb(father(_1,_2)))).
idb1((mother(_1,_2):-edb(mother(_1,_2)))).
idb1((_1=_2:-edb(_1=_2))).
idb1((_1>_2:-edb(_1>_2))).
idb1((_1>=_2:-edb(_1>=_2))).
idb1((_1=<_2:-edb(_1=<_2))).
idb1((_1<_2:-edb(_1<_2))).
idb1((_1<>_2:-edb(_1<>_2))).

father(taro,ichiro).
father(ichiro,yasuo).
father(hanako,shigeki).

mother(taro,hanako).
mother(ichiro,keiko).
mother(hanako,etsuko).
mother(etsuko,mayumi).

temp1(taro,ichiro).

temp2(ichiro).

yes
| ?-
```