T2R2 東京工業大学リサーチリポジトリ
Tokyo Tech Research Repository

論文 / 著書情報
Article / Book Information

題目(和文)	知識ベース指向処理に関する研究
Title(English)	A Study on Knowledge-base Oriented Processing
著者(和文)	横田治夫
Author(English)	Haruo Yokota
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:乙第2242号, 授与年月日:1991年7月31日, 学位の種別:論文博士, 審査員:
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:乙第2242号, Conferred date:1991/7/31, Degree Type:Thesis doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Powered by T2R2 (Tokyo Institute Research Repository)

知識ベース指向処理に関する研究

横田　治夫

目次

第１章　序論 …… 1
１．１　研究の目的 …… 1
１．２　研究の背景 …… 1
１．３　研究の概要 …… 2

第２章　演繹データベースによる知識ベース指向処理 …… 4
２．１　はじめに …… 4
２．２　述語論理と関係データベース …… 4
２．３　演繹データベースにおける SLD 演繹 …… 5
２．３．１　間送評価によるプラン生成 …… 6
２．３．２　メタ述語を利用した実現方法 …… 8
２．３．３　再帰的定義に対する不動点意味 …… 9
２．３．４　途中結果を保持することによる最適化処理 …… 12
２．３．５　不動点検出による停止性 …… 13
２．４　演繹データベースにおける単位演繹 …… 14
２．４．１　冗長な導出形生成の抑制 …… 14
２．４．２　setting 収集アルゴリズム …… 15
２．４．３　setting を使った単位演繹データベース …… 16
２．４．４　不動点処理の拡張 …… 17
２．５　まとめと考察 …… 20

第３章　項関係上の単一化検索を使った知識ベース指向処理 …… 22
３．１　はじめに …… 22
３．２　単一化検索 (RBU) モデル …… 22
３．２．１　項関係 …… 22
３．２．２　単一化検索 …… 23
３．３　単一化によるホーン節の導出 …… 24
３．４　項関係上の RBU 演算による SLD 演繹 …… 26
３．５　選択関数を用いる単位演繹 (SUD 演繹) …… 30
３．６　項関係上の RBU 演算によるSUD 演繹 …… 32
３．７　まとめと考察 …… 35

第４章　RBU に基づく知識ベース指向処理システムの実現 ………………………… 37
４．１　はじめに ……………………………………………………………………… 37
４．２　知識検索モジュールの命令体系 ……………………………………… 37
４．３　専用ハードウェアを用いた構成 ……………………………………… 40
４．３．１　全体構成 ……………………………………………………………… 40
４．３．２　単一化エンジン ……………………………………………………… 41
４．３．３　多ポートページメモリ ……………………………………………… 44
４．４　インデックスによる検索高速化 ……………………………………… 46
４．４．１　項の表現方法 ………………………………………………………… 46
４．４．２　ハッシングとトライ構造 …………………………………………… 48
４．４．３　LOSR 上の単一化 …………………………………………………… 51
４．４．４　トライ構造のたどり方 ……………………………………………… 53
４．４．５　インデックスの効果 ………………………………………………… 54
４．５　まとめと考察 ………………………………………………………………… 58

第５章　知識検索と推論処理の並列化 ……………………………………………… 59
５．１　はじめに ……………………………………………………………………… 59
５．２　並列知識ベース指向処理システム …………………………………… 59
５．２．１　並列論理型言語 GHC ……………………………………………… 59
５．２．２　GHC からの RBU 並列アクセス ………………………………… 60
５．２．３　要求駆動によるストリームインタフェース …………………… 62
５．３　並列 RBU と GHC による並列演繹 ………………………………… 64
５．３．１　並列 SLD 演繹 ……………………………………………………… 65
５．３．２　並列 SUD 演繹 ……………………………………………………… 68
５．４　並列知識ベース指向処理システムの試作 …………………………… 70
５．４．１　試作システムの構成 ………………………………………………… 70
５．４．２　複数コマンドページによる動的負荷分散 ……………………… 71
５．５　試作システム上の並列演繹の実行 …………………………………… 72
５．６　まとめと考察 ………………………………………………………………… 75

第６章　並列知識ベース指向処理システム上での意味ネットワーク探索 ……… 77
６．１　はじめに …… 77
６．２　意味ネットワークの項関係への格納 …… 78
６．３　並列トラバース …… 78
６．４　プロセッサ台数とインデックスの効果 …… 81
６．５　まとめと考察 …… 86

第７章　並列知識ベース指向処理システム上での並列状態遷移探索 …… 87
７．１　はじめに …… 87
７．２　状態遷移探索の並列性 …… 87
７．３　木型プロセス構成上での優良優先状態遷移探索 …… 89
７．４　永久プロセスによるノードの構成 …… 91
７．５　RBU を用いた状態遷移処理 …… 95
７．６　並列化と葉ノードの数 …… 99
７．７　まとめと考察 …… 100

第８章　結論 …… 101

謝辞 …… 103

参考文献 …… 105

第1章　序論

1．1　研究の目的

計算機システムを人間により身近なものとするためには，人間の常識や経験に関する知識などを利用して処理を進めることが要求される．さらに，人間が普段行っている推論処理を，計算機に蓄えた知識を使って代行することも必要となる．そのような処理に利用される知識は，システムを人間に近づけようとすればするほど量を増すことになる．大量の知識が無秩序に存在すると，それらに対する操作や管理が困難となる．このため，個々の知識をまとめて知識ベースとして管理し，要求される知識に対する高速アクセス手段を提供することが必要となってくる．そのような，知識を知識ベースとして管理して，処理を行う方法を知識ベース指向処理と呼ぶことにする．

本研究では，知識ベース指向処理を行うための，モデル，機構，構成方法，機能，高速化手段等について，検討・検証を行う．

1．2　研究の背景

知識ベースという言葉は，もともとはエキスパートシステム [Barr-Feigenbaum 81] に使われる知識（多くの場合 IF-THEN ルール）の集合を指していた．しかし，知識を利用した処理が多種にわたるようになってきたため，より一般的な名称として使われることが多くなってきた[横田-伊藤 86]．これにともなって，格納対象も IF-THEN ルールだけではなく，述語論理，フレーム，意味ネットワークといった各種の知識表現 [Nilson 80，Barr-Feigenbaum 81] になってきている．さらには，図形や音声といったマルチメディアを対象としたもの，それらを取り扱うためにオブジェクト指向の手法を取り入れたものも考えられるようになってきている[Kim et al. 88，横田-西尾 90]．

知識ベース指向で処理を進めるために，知識ベースを管理し検索する機構と，検索された知識を使って推論を進める機構とを分けて考えることにする．まず，そのような構成のシステムとしては，演繹データベース・システム[Gallaire 83]を挙げることができる．演繹データベースとは，述語論理で記述された知識の一部をデータとして関係データベースに格納し，関係データベース・システムと推論機構とで協調しながら演繹を行うものである．

演繹データベースの問題点は，実際に知識を格納しているのが関係データベースであるため，知識を表現するための構造や変数は文字列としてしか格納できないことである．このため，一般に演繹データベースでは，関係データベースに格納される知識（節）に関しては関数を含まないという制約がつく．もし関数や変数を文字列として格納した

場合には，推論を行うために検索の条件や検索結果を検索の度に文字列と変換する必要がある．さらに，構造を反映した効率的な検索も不可能である．

知識ベース指向処理の高速化のためには，処理の並列化は重要な課題である．並列処理を記述するための言語としては，ストリームベースで論理型指向の GHC [Ueda 86]，Parlog [Clark-Gregory 86]，Concurrent Prolog [Shapiro 86]，オブジェクト指向の ABCL/1 [Yonezawa et al. 88]，Concurrent Smalltalk [Yokote-Tokoro 88]，並列 Lisp の MultiLisp [Halstead 85] など数多くの処理系が提案されている．これらの言語の多くは，いかなる並列動作をしても矛盾が生じないように，並列処理単位間でグローバルなデータが共有できない．つまり，内容が更新されうる知識を静的に保持して，それらに対して複数処理単位から並列にアクセスすることが許されていない．知識ベースを永久プロセス[淵他 87]などを用いて動的に実現する方法も考えられるが，更新した内容を恒久的に保持できない上，知識の容量が大きくなった場合の高速アクセス手段が与えられていないため，並列言語だけで知識ベース指向処理を実現することには難がある．

知識ベースに格納される知識は，複雑になると同時に，ますます大容量化することが予想され，並列処理に対する期待も大きい．それら要求を満足するような環境を提供することが，計算機システムを人間に近づけるためには急務であると言える．

1．3　研究の概要

本研究では，まず知識ベース指向処理の第一歩として演繹データベースの実現を取り上げる．次に，知識ベースへの格納対象を広げるため，知識ベースの新たなモデルを提案し，その実現方法と並列化の方法について検討する．さらに，試作システム上でのいくつかの実験結果を通じて，知識ベース指向処理環境についての考察を行う．

演繹データベースには，演繹処理にデータベースに対するインタフェース機能を付加したという見方と、データベースの問い合わせ言語に演繹機能を持たせたいう見方がある．それぞれに大きな違いがあるわけではないが，ここでは演繹機能の拡張という見方で議論を進める．演繹処理の進め方には，大きく分けて２種類の方法がある．一つは，与えられたゴールから推論を始め，探索木を根から葉に向かって演繹を進める方法である．もう一方は，探索木を葉から根に向かって演繹を進める方法である．第２章では，それぞれの方法について実現方法[Kunifuji-Yokota 82, Yokota et al. 84, Yokota et al. 86]を提案し，それぞれ方法の特徴について述べる．

次に，演繹データベースが複雑な格納対象を扱えないという問題に対処するため，関係データベースの枠組みを拡張したモデルを提案する．このモデルは，関係データベースにおける関係（テーブル）に変数を含む再帰的な構造体である項を格納し，項を検索

するために関係代数演算に単一化を導入したもので単一化検索 (Retrieval by Unification: RBU) モデルと呼ぶ[Yokota-Itoh 86]．項関係上のRBU 演算の組み合わせによりホーン節演繹が可能となる．第３章では，RBU モデルの定義と RBU 演算の組み合わせによる２種類のホーン節演繹アルゴリズムを提案し，その論理的定式化を行う[横田他 90b]．

第４章では，RBU の実現方法について述べる．まず，単一化エンジンなどの専用ハードウェアを用いた場合の構成方法[Yokota-Itoh 86]を提案し，次に専用ハードウェアを想定しない場合のソフトウェアによる高速化手段[Yokota et al. 89]について検討する．単一化エンジンでは，結合処理における組み合わせを減らす工夫について述べる．また，ソフトウェアで実現する場合にはアクセスパスが重要となるため，変数を含んだ構造体である項に対するインデックスとして，一種の木構造であるトライ(Trie)とハッシングを組み合わせたものを提案する．トライにより，単一化検索時の比較処理ならびにバックトラック処理の発生回数を抑えることができる．また，更新時にはトライの部分的修正により対応できることから，更新処理におけるインデックス維持コストが少なくて済む．試作したプロトタイプを使って，それらの効果を確認する．

第５章以降では，知識ベース指向処理をさらに高速化するために，システム全体を並列化する方法を検討する．まず，推論する部分は並列記号処理言語 GHC を使い，知識ベースの管理については専用の並列検索モジュールとして RBU を並列化し，GHC から RBU に並列にアクセスして処理を進める知識ベース指向並列処理システム[横田他 90a]を提案する．次に，マルチマイクロ計算機を使った試作システムの構成について述べる．試作システムでは，RBU と GHC は共有メモリを介して接続され，GHC の論理変数を用いた要求駆動のストリーム通信でインタフェースをとる．また RBU は，検索／更新コマンド単位で並列に実行するための同時実行メカニズムを採用する．

知識ベースの指向並列処理システムの実験として，試作システム上で並列ホーン節演繹を行う．このとき，検索と推論の処理負荷バランスにより，プロセッサ台数に従って処理性能がどのように変化するかを調べる．また，第６章では，より実際の知識ベース処理に近いと思われる意味ネットワークの並列探索を試作システムの上で行う[横田他 90a]．このとき，第４章で提案したインデックスの効果についても確認する．さらに，第７章では，知識ベース指向並列処理システム上で意思決定支援を行うことを想定して，並列状態遷移探索を実現する手段として木型プロセス構成上で優良優先探索を行う方法を提案する[Yokota et al. 88]．

第2章　演繹データベースによる知識ベース指向処理

2．1　はじめに

知識ベース指向処理を実現する1つのアプローチとして，大量となる知識をデータベースに格納し，推論機構からそのデータベースにアクセスして知識を利用する方法が考えられる．特に，知識を述語論理を用いて表現し，その一部を関係データベースに格納する方法は，演繹データベース (Deductive Database) と呼ばれる[Gallaire 83].

本章では，述語論理において推論を行う場合の反駁木の2種類の手繰り方に準じて，演繹データベースとして，反駁木を根から葉に向かって演繹を進める方法と，葉から根に向かって演繹を進める方法の独自の実現法を提案する．

2．2　述語論理と関係データベース

一階述語論理（First-Order Logic) [Chang-Lee 73, Loveland 78] の節のうち，節中に正のリテラル（Literal）を高々1つしか含まない節をホーン節 (Horn Clause) と呼ぶ．充足不能なホーン節の集合に対しては，健全かつ完全な反駁方法がいくつか存在する[Loveland 78]. さらに，ホーン節集合に対する反駁そのものに手続き的な意味を与えて論理型のプログラミング言語とみなすこともできる [Kowalski 79]．このため，ホーン節が知識処理に用いられることが多い．論理型プログラミング言語である Prolog も，このホーン節の反駁に探索規則等いくつかの操作を加えたものである[Lloyd 84].

節の中で，リテラル1つからなる節を単位節 (Unit Clause) と呼び，正のリテラル1つからなる単位節を正の単位節 (Positive Unit Clause) と呼ぶ．また，節中に変数を含まない節を基底節 (Ground Clause) と呼び，変数を含まない単位節を基底単位節 (Ground Unit Clause) と呼ぶ．正の基底単位節からなる集合は，ホーン節の部分集合である．ホーン節を用いた知識処理において，知識が増えると正の基底単位節が増大することが多い．そこで，正の基底単位節をデータベースに格納して管理し，データベースの高速なアクセス手段を利用する方法として，演繹データベースは提案された．

一方，1970 年にE.F. Codd が提案した関係データモデルは，データベースをいくつかの定義域 (Domain) の直積の部分集合である関係のあつまりとして，関係に対する集合的な操作をいくつか与えている [Codd 70, Ullman 82, Date 86]．つまり，集合 $D_1, D_2,..., D_m$ を定義域としたとき，次のような R を m 属性の関係(Relation)と呼ぶ．

$$D_1 \times D_2 \times ... \times D_m \supset R.$$

このとき次のような t をタプル(Tuple)と呼ぶ．

$$t = (d_1 , d_2 ,..., d_m) \in R$$

ここで，関数を含まない正の基底単位節は，関係データベースにおけるタプルにそのまま対応させることができる．つまり，基底単位節のリテラルの述語名を関係名に対応させ，述語の各引き数を関係の各属性に対応させることができる．同一述語名で同一引き数の単位節は，１つの関係に格納されることになる．

演繹データベースでは，知識処理において大量となると予想される正の基底単位節のうち関数を含まないものを関係データベースのタプルとして格納し，それ以外のタプルを演繹機構側で保持して，関係データベースと演繹機構で協調しながら知識処理を行うものである．こうすることにより，関係データベースにおけるインデックス等の大量データに対する技術を利用することができる．当初Reiterが質問応答システムを内包プロセッサと外延プロセッサに分割したシステムを提案した[Riter 78]こともあり，演繹データベースにおいては，関係データベースに格納される関数を含まない正の基底単位節の集合を外延データベース (EDB: Extensional Database) と呼び，それ以外の節の集合を内包データベース (IDB: Intensional Database) と呼ぶ．

ここまでは，ホーン節の単位節をデータベースに格納して処理の効率化を図ろうという見方をしてきた．見方を変えると，演繹データベースは関係データベース問い合わせ言語の拡張と見ることもできる．関係代数，関係論理に代表される関係データベースの問い合わせ言語 [Ullman 82] は，直接格納されている外延的なデータのみを検索対象にしていたのに対し，演繹データベースでは直接は格納されていないがそこから導き出される知識までを対象としている．あるいは，内包データベースに格納された知識により，データベースの一貫性保持などを行うことも考えられる．

工学的立場からすると，すでに関係データベースシステムや論理型プログラミング言語処理系が存在し，さらには関係データベースマシンや推論マシンが存在する中，それらのシステムやマシンを結合することにより，さらに高機能な処理を提供することを目的としている[Murakami et al. 83].

２．３　演繹データベースにおけるSLD演繹

充足不能なホーン節集合から，反駁を求めるアルゴリズムの１つとして，SLD 演繹がある． SLD 演繹とは，Linear resolution with Selection function for Definite clauses の略[Apt-Emden 82]で，ホーン節に対する制限された入力演繹の一種である．Prologの処理系は，SLD演繹の一つの実現手段とみなすことができる[Lloyd 84]．ここでは，演繹データベースで SLD 演繹を実現する方法を述べる．

SLD演繹では一つの負節 G (つまりPrologのゴール)と確定節（負節以外のホーン節）の集合 D (Prologのプログラム)との和集合$D \cup \{G\}$から反駁を導く．ゴール G が与えられた

とき，その中から選択関数（Prolog では最左端リテラルを選択する）によって選ばれたリテラルと導出可能な入力節を確定節の集合 D から探索してきて，導出形 (Resolvent) を求め，その導出形を新たなゴールとして演繹を進める．この導出形が空になったところで，反駁が得られたことになる．

演繹データベースの場合，関係データベースと推論機構の二つのシステムが結合していることにより，システム間の通信量を減らすことが重要となる．SLD 演繹を素直に実現して，入力節を探索に行く度に，内包データベースと外延データベースの両方を検索していたのでは，効率が悪い．さらに，外延データベースを保持している関係データベースは，１つの検索要求に対して検索結果を集合として返すため，検索結果の取扱に工夫が必要となる．このため，内包データベースで評価可能な部分だけ先に評価しておいて，その後外延データベースにまとめて検索要求を出す方法がとられる．このような処理方法を間送評価 (Deferred Evaluation)，検索要求をプラン(Plan)と呼ぶ[Yokota et al. 84]．また，前述の２つのアプローチを比較する場合には，前者の入力節が必要な度に外延データベースに検索にいく方式はインタプリティブ・アプローチ，後者のまとめてプランを生成する方法はコンパイルド・アプローチと呼ばれる[Chakravarthy et al. 82]．

２．３．１　間送評価によるプラン生成

間送評価においては，与えられたゴールは外延データベースに格納されている節と導出すべきリテラルが最左端に現われるまでは，内包データベースのみを使って SLD 演繹を進める．外延データベースに格納されている節と導出すべきリテラルが最左端に現われた場合には，そのリテラルを導出形から取り除きプラン・リストの後ろに付け加える．なお，このプラン・リストの初期状態は空である．導出形が空になったところで内包データベースによるプラン生成は終了し，残ったプラン・リストが与えられたゴールに対する１つの検索プランとなる．このプランは，そのまま関係論理の問い合わせ言語とみなすこともできるし，関係代数演算に変換することも可能である[Kunifuji-Yokota 82]．

内包データベースのみによるプラン生成は，内包データベースの探索により別解があることと対応して，複数のプランを生成しうる．それぞれのプランに対する検索結果の集合和から得られるゴールに含まれる変数の代入結果が，演繹データベースによる SLD 演繹の結果となる．

【例２．１】以下の(1)から(6)のホーン節が内包データベースとして与えられているものとする．

(1)　$\forall X, \forall Y, \forall Z\ (uncle(X,Y) \vee \neg parent(X,Z) \vee \neg brother(Z,Y))$

(2) $\forall X, \forall Y\, (parent(X,Y) \vee \neg father(X,Y))$

(3) $\forall X, \forall Y\, (parent(X,Y) \vee \neg mother(X,Y))$

(4) $\forall X, \forall Y\, (father(X,Y) \vee \neg edb(father(X,Y)))$

(5) $\forall X, \forall Y\, (mother(X,Y) \vee \neg edb(mother(X,Y)))$

(6) $\forall X, \forall Y\, (brother(X,Y) \vee \neg edb(brother(X,Y)))$

ここで(4)から(6) の節は *father*， *mother*， *brother*，関する節がそれぞれ関係データベースの関係として外延データベースにあることを示している．このとき，

$\forall U\ (\neg uncle(tarou,U))$

というゴールが与えられると，図2-1のように導出形とプランが生成されていく．
最終的には，*[father(tarou,Z), brother(Z,U)]*というプランが得られる．このプランは，そのまま関係論理の問い合わせと見ることもできるし，次のような一連の関係代数演算に変換することもできる．

temp1 = selection(father, first-attribute="tarou")

temp2 = equi-join(temp1, brother, second-attribute = first-attribute)

result = projection(temp2, second-attribute)

これは，*father* という関係に対して，第１属性が *tarou* であるようなタプルを選択して，*temp1* という一時的な関係を作り，その *temp1* の第２属性と *brother* という関係の第１属性の間で等結合を行い *temp2* という関係を作る．この *temp2* の第２属性だけを取り出すように射影をかけたものが結果である．この場合には，ゴール中の *U* とういう変数への代入が *result* という関係として得られることになる．

内包データベースの中で，*parent* の定義が２種類あるため，これに対応してプランも２つ生成される．もう１つのプランは，*[mother(tarou,Z), brother(Z,U)]* と言うもので，上述したものと同様に一連の関係代数演算に変換される． ■

導出形	プラン
$\forall U\ (\neg uncle(tarou,U))$	*[]*
$\forall U, \forall Z\ (\neg parent(tarou,Z) \vee \neg brother(Z,U))$	*[]*
$\forall U, \forall Z\ (\neg father(tarou,Z) \vee \neg brother(Z,U))$	*[]*
$\forall U, \forall Z\ (\neg brother(Z,U))$	*[father(tarou,Z)]*
[]	*[father(tarou,Z), brother(Z,U)]*

図 2-1　導出形とプランの生成

2．3．2　メタ述語を利用した実現方法

間送評価によるプラン生成を，論理型プログラミング言語 Prolog を用いて実現した[Kunifuji-Yokota 82, Yokota et al 84]．Prolog では，メタ述語を使うことにより Prolog 自身のインタプリタをProlog によって記述することができる[Coelho et al. 80]．インタプリタによって直接演繹の過程に手を加えることができるため，このインタプリタをプラン生成に利用することができる．当初，単一化 (Unification) 自身もインタプリタ中で行っていたが[Kunifuji-Yokota 82]，単一化及び節の選択は Prolog がもともと持つものを利用するようにした[Yokota et al. 84]．

Prolog による実現では，対象知識である内包データベースを他の Prolog の節と区別する必要がある．このため，例２．１の内包データベースを次のような形状で持つことにする．

```
idb_clauses(idb1, uncele(X,Y), [parent(X,Z),brother(Z,Y)]).
idb_clauses(idb1, parent(X,Y), [father(X,Y)]).
idb_clauses(idb1, parent(X,Y), [mother(X,Y)]).
idb_clauses(idb1, father(X,Y), [edb(father(X,Y))]).
idb_clauses(idb1, mother(X,Y), [edb(mother(X,Y))]).
idb_clauses(idb1, brother(X,Y), [edb(brother(X,Y))]).
```

ここで，*idb1* は内包データベースを区別するための識別子，第 2 引き数が正のリテラル、第 3 引き数が負のリテラルのリストである．

メタ述語によるプラン生成プログラムを図 2-2 に示す．*generate* の第 1 引き数に内包データベースの識別子を，第 2 引き数にゴールを渡すことにより，第 3 引き数に差分リストに入ったプランが得られる．さらに強制的にバックトラックを起こすことにより，別のプランを得ることができる．

例えば，

```
generate(IDB,[],d(X,X)) :- !.
generate(IDB,edb(P),d([P|X],X)) :- !.
generate(IDB,[P|R],d(X,Z)) :- !,
        generate(IDB,P,d(X,Y)), generate(IDB,R,d(Y,Z)).
generate(IDB,P,L) :-
        idb_clauses(IDB,P,Q), generate(IDB,Q,L).
```

図 2-2　メタ述語によるプラン生成プログラム

?- *generate(idb1,uncle(tarou,U),A).*

と言う問い合わせを行うと，

A = d([father(tarou,Y),brother(Y,U)|B],B) ;

A = d([mother(tarou,Y),brother(Y,U)|B],B) ;

fail

と言う結果が得られる（セミコロンは強制バックトラックを表わす）．これをプランコンバータに渡すことにより，前述したような関係代数演算のコマンド列を得ることができる．

2．3．3　再帰的定義に対する不動点意味

内包データベースは，再帰的定義による繰り返し処理を含むことがある．その場合，演繹データベースでは，その繰り返し処理の停止方法を導入する必要がある．Prolog で全解探索を行うような場合には，入力節の探索時にインスタンスが存在しないことから暗黙的に停止条件を示していた．しかし，間送評価による演繹データベースでは，プラン生成とデータベース検索とが別のシステムで実行されるため，プラン生成にのみ介在して繰り返しを停止させることは実質的でない．つまり，外延データベースに格納されているデータの内容に依存してプラン生成を制御する必要がある．

【例2．2】以下の(1)から(3)のホーン節が内包データベースとして与えられている．

(1)　$\forall X, \forall Y\ (ancestor(X,Y) \vee \neg parent(X,Y))$

(2)　$\forall X, \forall Y, \forall Z\ (ancestor(X,Y) \vee \neg parent(X,Z) \vee \neg ancestor(Z,Y))$

(3)　$\forall X, \forall Y\ (parent(X,Y) \vee \neg edb(parent(X,Y)))$

また，ゴールとして

$\forall A\ (\neg ancestor(tarou, A))$

が与えられたとする．このとき，

[parent(tarou, A)]

[parent(tarou, B), parent(B, A)]

[parent(tarou, C), parent(C, B), parent(B, A)]

というプランが次々に生成されることになる．このプラン生成は，外延データベースの内容に依存して停止されるべきである．　■

この再帰呼び出しに対するプラン生成を制御するために，プランに対して検索されるデータの集合に対して不動点意味を与える．i 番目のプランよって得られる関係を $R(i)$ とし，$R'(i)$ を $R'(i-1)$ と $R(i)$ との間の集合和演算から得られた関係とする．なお，$R'(0)$ は空集合とする．$R'(i)$ 集合和の結果であるためは i に対して単調増加となる．$R'(i)$ の内容が

R'(i-1) の内容と等しくなったとき最小不動点 (LFP: Least Fixed Poin) に達したとみなす [Aho-Ullman 79, Kunifuji-Yokota 82]．最小不動点を見てプラン生成を止めることにより，外延データベースに *parent(a, b)* と *parent(b, a)* のような循環するインスタンスが存在しても，無限ループに陥ることがなくなる．

しかし，生成されるプランが常に例２．２に示したような単純なものではないので、どの時点で最小不動点をチェックするかを推論機構側から示す必要が生じる．

【例２．３】以下の(1)から(6)のホーン節が内包データベースとして与えられている．

(1) $\forall X, \forall Y\ (ancestor(X,Y) \vee \neg parent(X,Y))$

(2) $\forall X, \forall Y, \forall Z\ (ancestor(X,Y) \vee \neg ancestor(X,Z) \vee \neg parent(Z,Y))$

(3) $\forall X, \forall Y\ (parent(X,Y) \vee \neg father(X,Y))$

(4) $\forall X, \forall Y\ (parent(X,Y) \vee \neg mother(X,Y))$

(5) $\forall X, \forall Y\ (father(X,Y) \vee \neg edb(father(X,Y)))$

(6) $\forall X, \forall Y\ (mother(X,Y) \vee \neg edb(mother(X,Y)))$

また，ゴールとして

$\forall A, \forall B\ (\neg ancestor(A,B))$

が与えられたとすると，

[father(A,B)]

[motther(A,B)]

[father(A,C), father(C,B)]

[father(A,C), mother(C,B)]

[mother(A,C), father(C,B)]

[mother(A,C), mother(C,B)]

[father(A,C), father(C,D), father(D,B)]

というプランが生成されてくる．□

この例の場合，１番目と２番目のプランの間で最小不動点をチェックすることは意味がなく，各レベルの最後のプランの後，*[motther(A,B)]* や *[mother(A,C), mother(C,B)]* というプランの後で最小不動点のチェックをすべきである．つまり，*R'(i)* の内容と *R'(i-1)* の内容が等しいかをみるような *i* は生成されるプラン１つ１つには対応せず，内包データベースに格納されている節の内容に依存する．このため，チェックのタイミングを示す特別な *check* 述語を導入する[Yokota et al. 84].

例２．３の (2) の節を内包データベースに格納する際，

idb_clauses(idb3, ancestor(X,Y), [check, ancestor(X,Z), parent(Z,Y)]).

のように再帰呼び出しの前にチェックを起動するために *check* 述語を入れる．これに対

```
generate(IDB, [], d(X,X)) :- !.
generate(IDB, check, d(X,X)) :- !,
        ( eq_ans, !, fail; true).
generate(IDB, edb(P), d([P|X],X)) :- !.
generate(IDB, [P|R], d(X,Z)) :- !,
        generate(IDB, P, d(X,Y)), generate(IDB, R, d(Y,Z)).
generate(IDB, P, L) :-
        idb_clauses(IDB, P, Q), generate(IDB, Q, L).
```

図 2-3　check 述語を使うためのプラン生成プログラム

応して，プラン生成のプログラムも図 2-3のように変更する．

ここで，*eq_ans* という述語は最小不動点を見るために行う関係間の比較演算の結果から，2 つの関係が等しい場合にフラグとして *assert* されるものである．チェックされる時点で，前のチェックの時点から結果の関係の内容が増加していなければ，この *eq_ans* が存在して，プランの生成を停止する．全体の処理の流れを示すプログラムを図 2-4 に示す．*convert* は，プランを関係代数演算列に変換するプログラム．*rdb_if* は，関係データベースとインタフェースをとるためのプログラムである．

check 述語の導入については，Prolog の処理と比較すると，無理がないものであると考えることができる．つまり，Prolog の処理においても，例えば

```
deferred_evaluation(IDB, Goal) :-
        generate(IDB, Goal, Plans),
        convert(Plans, RDB_Commands),
        rdb_if(RDB_Commands, Result_Relation),
        backtrack(Result_Relation).
deferred_evaluaiton(_, _) :-
        output( Out_Relation).

backtrack(Result_Relation) :-
        rdb_if([copy(out,out1), union(Result_Relation, out), equal( out1, out )], EQ_ans),
        (EQ_ans = no, abolish( eq_ans ); assert(eq_ans)), ! , fail.
```

図 2-4　不動点チェックを行う間送評価プログラム

ancestor(X,Y) :- parent(X,Y).

ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).

の様な Prolog のプログラムにおいて，*parent* と言う述語は暗黙的に停止条件を示している．Prolog の場合には２番目の節を，

ancesotr(X,Y) :- ancestor(Z,Y), parent(X,Z).

とするだけで意味は変わらないのに，再帰呼び出しが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明示的に示したことに対応する．

２．３．４　途中結果を保持することによる最適化処理

例えば，例２．２で生成されたプランの列

[parent(tarou, A)]

[parent(tarou, B), parent(B, A)]

[parent(tarou, C), parent(C, B), parent(B, A)]

に対する関係代数演算の列を考える．第１のプランに対する関係代数演算列は，

temp1 = selection(parent, first-attribute="tarou")

result1 = projection(temp1, second-attribute)

となる．また，第２のプランに対する関係代数演算列は，

temp2 = selection(parent, first-attribute="tarou")

temp3 = equi-join(temp2, parent, second-attribute = first-attribute)

result2 = projection(temp3, second-attribute)

となる．さらに，第３のプランに対する関係代数演算列は，

temp4 = selection(parent, first-attribute="tarou")

temp5 = equi-join(temp4, parent, second-attribute = first-attribute)

temp6 = equi-join(temp5, parent, forth-attribute = first-attribute)

result3 = projection(temp6, second-attribute)

となる．

ここで *temp2* および *temp4* は *result1* と同じ情報しか含まない．また， *temp5* も *result2* と同じ情報しか含まない．このため，各プランに対する結果を持つ関係を保持することにより最適化を図ることができる．例えば第１のプランに対して

plan_relation([parent(tarou,A)], result1(A)).

というプランと関係のペアを述語として *assert* しておき，図 2-5 に示すような *append* 述語を使ったストリングマッチャを用いてプランの１部を関係に置き換えることができる．

```
optimize(Old_plan, Final_plan) :-
        append(TPL, Rest, Old_plan),
        append(Top, Plan_Part_A, TPL),
        plan_relation( Plan_Part_B, Relation),
        eq_struct( Plan_Part_A, Plan_Part_B),
        Plan_Part_A = Plan_Part_B,
        append( Top, [Relation], TO),
        append(TO, Rest, New_Plan),
        optimixe( New_plan, Final_Plan).
optimize(Plan, Plan).
```

図 2-5　途中結果を使った最適化のためのプログラム

このプログラムにより，第2のプランは

[result1(B), parent(B,A)]

に変換される．これに対する関係代数演算列は，

temp2 = equi-join(result1, parent, first-attribute = first-attribute)

result2 = projection(temp2, second-attribute)

となる．この結果

plan_relation([parent(tarou,B), parent(B,A)], result2(A)).

という述語が *assert* され，第3のプランは

[result2(B), parent(B,A)]

に変換される．これに対する関係代数演算列は，

temp3 = equi-join(result2, parent, first-attribute = first-attribute)

result2 = projection(temp3, second-attribute)

となり，明らかに処理が軽減される．

ここで述べた最適化は，Earley が提案した導出木のパージングアルゴリズム[Earley 70]を，間送評価法に適用したものとみなすこともできる．

2．3．5　不動点検出による停止性

2．3．3において最小不動点を検出することによって間送評価の演繹データベースで再帰的定義による繰り返しを取り扱うことができることを示した．しかし，残念ながら，提案した *check* 述語では，単純な再帰的定義についてしか最小不動点を検出することができない．つまり，複雑な再帰的定義を含む知識については，*check* 述語を使った間送

評価の演繹データベースについては，その停止性を保証することはできない．

1つの演繹処理のなかで複数のチェックポイントがあるような定義の場合，例えばある節の中に単純な再帰的定義の2種類の述語が負リテラルとして含まれているような場合，プラン生成プログラムはどちらの繰り返し処理を停止すればよいのか判別できない．つまり，どのプランに対する結果の関係もすべて1つの結果を格納する関係と和集合をとってしまうため，プランと不動点の関係が1対1でなくなってしまうためである．これは，*check*述語の種類を分けても対処できない．

理論的には，幅優先に探索木を展開して無限回プランを生成すれば不動点は存在するが，そのような不動点検出方法は現実的でない．Prolog は，深さ優先に探索木を展開しながら，各再帰呼び出しごとにインスタンスのあるなしをチェックして処理を進めているために，完全ではなくなったが，ここで述べたような問題点は生じない．幅優先にした間送評価の演繹データベースは，探索規則が均等であるため完全性は保証できる．このため，1つの解を求めるだけなら停止するが，全解探索を停止させる現実的な方法が存在しない*.

2．4　演繹データベースにおける単位演繹

演繹データベースで全解を求めて停止させるために，反駁手続きを変えて見ることにする．SLD 演繹が，与えられたゴールから推論を始め，反駁木を根から葉に向かって演繹を進めるのに対し，現在ある事実から推論を始め，反駁木を葉から根に向かって演繹を進める方法を演繹データベースに適用する．

2．4．1　冗長な導出形生成の抑制

Robinson によって提案された導出原理 (Resolution Principle) [Robinson 65]は，導出形を算出する上で多くの冗長な計算を行っている．そのような不要な導出形を作らないように，各種の改良が提案されてきた．それらの改良の基本的な戦略は，節の集合を2つのグループに分類して，同一グループ内ではお互いに導出を行わないようにするものである．そうすることによって導出形の計算を大幅に抑えることができる．そのような改良を加えた演繹は意味演繹 (Semantic Resolution) と呼ばれている[Loveland 78].

SLD 演繹と単位演繹は良く知られたホーン節に対する完全な演繹手続きであるが，いずれも意味演繹の一種である．つまり，SLD 演繹では，導出形を生成するための一方の

---

* SLD 演繹の演繹データベースで全解を求めて停止させるために，演繹の途中で得られた結果をレンマとして展開を抑える SLD-AL 演繹 [Vieille 87] という別なアプローチも提案されている．

親を与えられたゴールもしくはその前のステップで生成した導出形とする．もう一方の親は，入力節と呼ばれ，確定節の中から探索される．単位演繹では，導出形を生成するための一方の親は必ず単位節でなくてはならない．

外延データベースは基底単位節の集合であるから，単位演繹は演繹データベースの演繹方法として適していることが予想される．しかし，一般の単位演繹では，与えられたゴールと独立に導出形を生成して行くため，まだ冗長な計算が多いことが問題となる．Loveland はホーン節に対する意味演繹の完全性を証明するために setting という概念を導入した[Loveland 78]．ここでは，この setting を単位演繹での冗長な導出形生成を抑えるために利用する方法[Yokota et al. 86a]について述べる．

2．4．2　setting 収集アルゴリズム

まず最初に，setting を求める方法から述べる．setting は，与えられたゴールと確定節の集合から求められる．以下に，収集アルゴリズムを示す．

【setting 収集アルゴリズム】

Step 1:　S と T を空集合とする．

Step 2:　ゴールに含まれるすべての（負）リテラルを，S と T に入れる．

Step 3:　もし，T が空集合なら S を setting として終了．
それ以外の場合は step 4 へ．

Step 4:　T から（負）リテラル L を１つ取り出す．

Step 5:　A を空集合とする．

Step 6:　すべての確定節の中から，その正リテラルが L と単一化可能で，かつ A に含まれていない節 C を選ぶ．もし，そのような節が１つもない場合には，Step 3 へ．
そうでなければ，Step 7 へ．

Step 7:　C の中のリテラルのうち，その正負を逆にしたものが S の中にない場合には，そのリテラルの正負を逆にしたものを S と T に加える．

Step 8:　C を A に加え Step 6 へ．　■

確定節とゴールに含まれるリテラル数が有限であることから，このアルゴリズムは停止する．

上のアルゴリズムを見るとわかるように，setting S は入力演繹で使われる節に現われるすべてのリテラルを含んでいることがわかる．このことから，setting に現われるリテラルにのみに注目することによって，ゴールと反駁するための最小限の正リテラルを導くことができる．すなわち，ゴールを１つだけ含む単位演繹において不要な導出を省くことができる．

単位反駁手続きの各ステップは，ある親の単位節 $C1$ および $C1$ と導出可能な負リテラルを含む節 $C2$ との間で導出を行い，新たな単位節を生成していって，ゴールにあるすべての負リテラルと導出可能な単位節をすべて求める処理と見ることができる．つまり，これは setting の要素である負リテラルと導出可能なすべての単位節を数え上げる処理とみなすことができる．このような方法を，setting 評価法と呼ぶことにする．

2．4．3 setting を使った単位演繹データベース

基底単位節を外延データベースに置き，内包データベースとゴールに対してのみ setting 収集アルゴリズムを適用することを考える．この場合，演繹に使われる外延データベース中の基底単位節と導出すべきリテラルを L としたとき，setting S は $\neg edb(L)$ をその要素として持つことになる．つまり，まず最初に，ゴールを反駁するときに使うべき基底単位節の候補を外延データベースの中から選択しておくことができる．

次に，外延データベースにある基底単位節の集合から setting の要素の負リテラルと導出可能なすべての単位節を，関係代数演算を使って求める方法を考える．setting の要素が $\neg edb(L)$ の形態をしている場合には，それは外延データベースに最初からある基底単位節である．$\neg edb(L)$ 以外の要素は，内包データベースによって示される関係代数演算によって得られる関係のインスタンスと見ることができる．以下、外延データベースに最初から存在する関係を<u>実関係</u>，内包データベースの節により生成される関係を<u>仮想関係</u>と呼ぶことにする．

例えば、内包データベースに格納されているある節 C の正リテラル Lp が示す仮想関係のインスタンスはその節 C の負リテラルが Lq 1つの場合には，リテラル Lq が示す仮想関係のインスタンスから共有する変数による射影 (projection) 演算によって得られる．また，節 C の負リテラルが複数である場合には，まずその複数のリテラル間の共有変数による等結合 (eq-join) 演算を行い，さらに正リテラル Lp に現われる変数に対する射影演算を行う．述語名と引き数の数がが同じリテラルは同じ仮想関係として，インスタンスの和集合を取る．また，ゴールの引き数があらかじめ束縛されている場合には，その束縛内容は setting に反映され，選択（selection）演算が行われることになる．

【例2．4】(1) から (3) のような節が内包データベースとして与えられ，ゴールとして (4) が与えられている場合を考える．

(1) $\forall X, \forall Y, \forall Z\ (uncle(X,Y) \vee \neg parent(X,Z) \vee \neg edb(brother(Z,Y)))$

(2) $\forall X, \forall Y\ (parent(X,Y) \vee \neg edb(father(X,Y)))$

(3) $\forall X, \forall Y\ (parent(X,Y) \vee \neg edb(mother(X,Y)))$

(4) $\forall A, \forall B\ (\neg uncle(A, B))$

このとき setting 収集アルゴリズムから，setting S は，

$$S = \{\tilde{\ }uncle(A, B), \tilde{\ }parent(X,Z), \tilde{\ }edb(brother(Z,Y)), \tilde{\ }edb(father(X,Y)), \tilde{\ }edb(mother(X,Y)) \}$$

となる．

要素 ~*edb(brother(Z,Y))*, ~*edb(father(X,Y))*, ~*edb(mother(X,Y))* は，対応する述語が外延データベース上にそれぞれ *brother*, *mother*, *father* という実関係のタプルとしてあることを示している．内包データベースの節 (2) は，*parent* という仮想関係が *father* という外延データベース上の実関係からの射影として得られることを示している．この場合には，実関係 *father* のすべての属性が仮想関係 *parent* にそのまま射影されるので，*father* に含まれるすべてのタプルをそのまま *parent* のタプルとすればよい．同様に，(3) から実関係 *mother* に含まれるタプルもそのまま *parent* のタプルとなり，結局 *parent* という仮想関係は *father* と *mother* の集合和演算によって求められることになる．次に，仮想関係 *uncle* のインスタンスは，(1) の節より，今求めた仮想関係 *parent* と実関係 *brother* との間の共有変数 Z によって示される等結合によって求められる．ここで求められた仮想関係 *uncle* のタプルがゴールに対する代入を示している．

また，ゴールが

(4') $\forall A$ (~*uncle(tarou, A)*)

であった場合には，setting S は，

$$S = \{\tilde{\ }uncle(tarou, A), \tilde{\ }parent(tarou,Z), \tilde{\ }edb(brother(Z,Y)), \tilde{\ }edb(father(tarou,Y)), \tilde{\ }edb(mother(tarou,Y)) \}$$

となり，実関係 *father* と *mother* の第 1 属性に対して *tarou* で選択をかけて和集合を取った仮想関係の *parent* （実はその第 1 属性はすべて *tarou* ）と実関係 *brother* との等結合を行うことになる． ■

このように，setting 評価法による演繹データベースは単位節の数え上げを関係データベース演算で行う．これは setting を用いた意味演繹を実現していることに他ならず，ホーン節集合に対しては健全であり完全である[Loveland 78].

2．4．4　不動点処理の拡張

間送評価で用いた不動点意味を，setting 評価用に拡張する方法を考える．つまり，setting の全ての要素に対して不動点検出を行い，全ての要素の不動点が検出できたところで処理を停止させる．こうすることによって，2．2．5 で述べた各再帰定義の不動点が 1 つのチェックポイントで検出できないという問題点を解決することができ，関数記号を含まないホーン節集合に対する全解探索の停止性を保証することができる．

setting を $S = \{ E_1,...,E_j, ..., E_n \}$，その要素 E_j に対応する i 番目の繰り返しにおける（仮想）関係を $R_j(i)$ とする．もし，$E_j (j = 1,...,n)$ が ~$edb(L)$ の形態の場合には，$R'_j(0)$ は L が示す実関係の内容を初期状態で持つ．それ以外の場合は，空とする．また全要素に対する $R_j(0)$ はすべて空とする．このときの setting 評価用の不動点検出アルゴリズムを以下に示す．

【setting 評価用不動点検出アルゴリズム】

```
beting
  initialize R_l(0),  l = 1,...,n
  i ← 1
  repeat
    begin
      t ← true
      for j = 1 to n
        begin
          enumerate R_j(i) using R_l(i-1),  l = 1,..., n
          R'_j(i) ← R_j(i) union R'_j(i-1)
          t ← (R'_j(i) = R'_j(i-1)) and t
        end
      i ← i + 1
    end
  until( t = true )
end
```

■

【例２．５】(1) から (4) のような節が内包データベースとして与えられ，ゴールとして (5) が与えられている場合を考える．

(1) $\forall X, \forall Y$ (ancestor(X,Y) ∨ ~parent(X,Y))

(2) $\forall X, \forall Y, \forall Z$ (ancestor(X,Y) ∨ ~parent(X,Z) ∨ ~ancestor(Z,Y))

(3) $\forall X, \forall Y$ (parent(X,Y) ∨ ~edb(father(X,Y)))

(4) $\forall X, \forall Y$ (parent(X,Y) ∨ ~edb(mother(X,Y)))

(5) $\forall A, \forall B$ (~ancesotr(A, B))

このとき setting 収集アルゴリズムから，setting S は，

S = { ~ancestor(A, B), ~parent(X,Z), ~edb(father(X,Y)), ~edb(mother(X,Y)) }

となる．

このときの数え上げ処理の様子を図 2-6 に示す．初期状態では，*father* と *mother* の実関

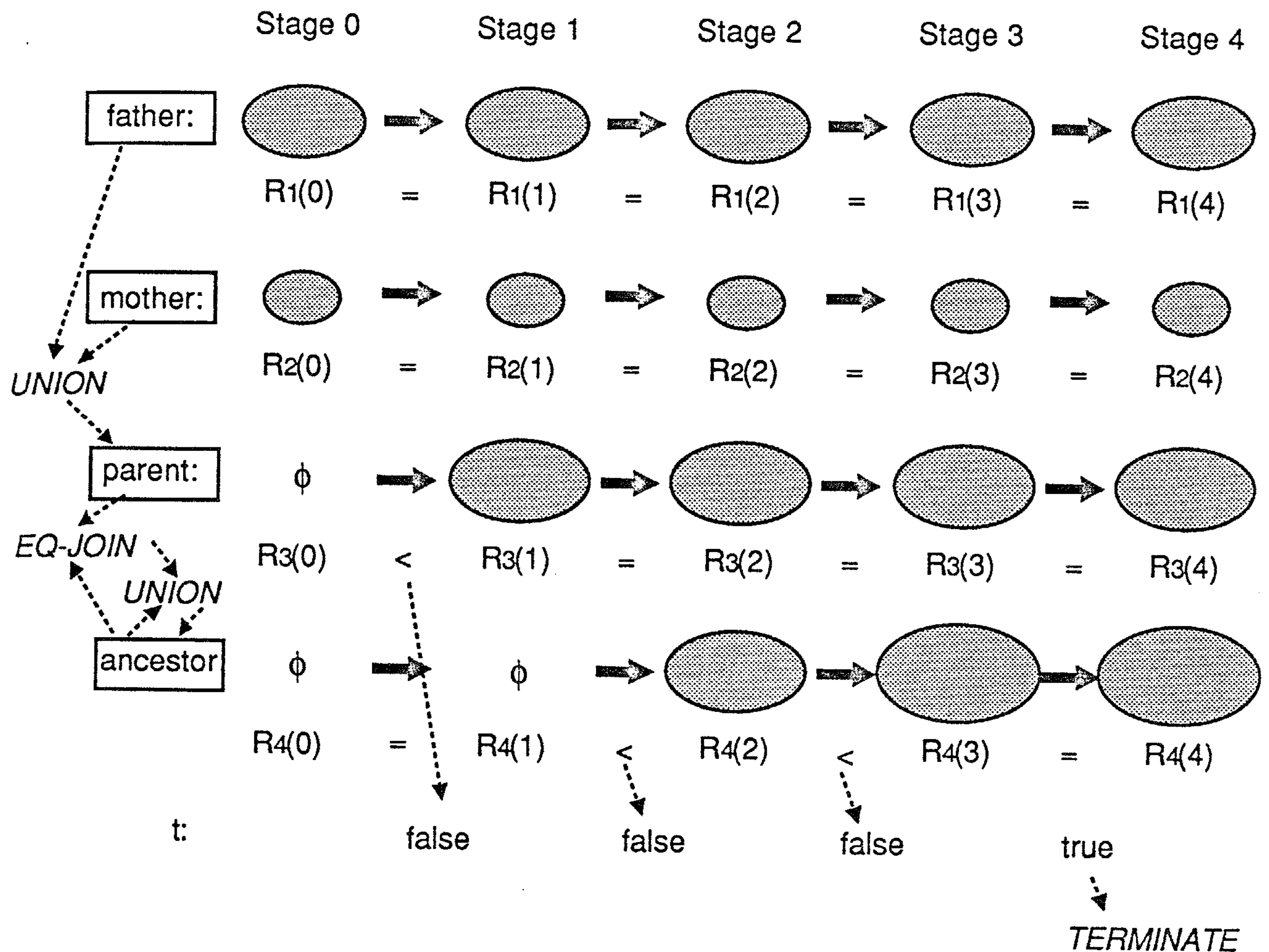

図2-6　数え上げ処理の様子

係にそれぞれタプルが入っていて，*parent* と *ancestor* の仮想関係は空になっている．それぞれの関係を $R_1(0), ..., R_4(0)$ で表わす．まず最初に，式 (3) と (4) から*parent* の仮想関係 $R_3(1)$ が初期の *father* の関係 $R_1(0)$ と *mother* の関係 $R_2(0)$ の集合和として求められる．このとき，*ancestor* の仮想関係 $R_4(1)$ は，$R_3(0)$ も $R_4(0)$ も空のため，空のままである．このステージでは，$R_3(1) = R_3(0)$ が成り立たないため *t* は *true* とならないため，次の繰り返し処理に入る．次以降のステージでは，$R_1(i), R_2(i), R_3(i)$ は変化しないが，*ancestor* の仮想関係 $R4(i)$ は，前段の *parent* の関係 $R_3(i\text{-}1)$と *ancestor* の仮想関係 $R_4(i\text{-}1)$との等結合および $R_4(i\text{-}1)$ との集合和演算によって新たなタプルがインスタンスとして付加されていく．このインスタンスの増加がなくなったところで、*t* が *true* となり繰り返し処理が終了する．このときの *ancestor* の仮想関係に格納されているタプルが，求めるゴールの変数に対する代入の集合を示している．■

実関係のタプル数は，繰り返し処理中は変化することがないので，不動点検出用の比較操作($R_j(i) = R_j(i\text{-}1)$)を省略することができる．同様に，実関係の集合和演算のみから得られるような仮想関係（例えば例２．５の *parent* の仮想関係）についても，最初の集合

和演算以後については比較操作の必要がない．このように，内包データベースを解析することにより，いくつかの関係については比較操作を省略して効率化を図ることができる．

再帰的定義が含まれる場合には，ゴールにおける変数の束縛によるしぼり込みについては，束縛されている変数の位置と再帰定義によってその有効性が変化する．例えば，例２．５において，ゴールが，

(5') $\forall A$ (~ancesotr(A, tarou))

であれば，setting S は，

S = {~ancestor(A, tarou), ~parent(X,tarou), ~edb(father(X,tarou)), ~edb(mother(X, tarou)), ~parent(X,Z), ~edb(father(X,Y)), ~edb(mother(X, Y) }

となり，ancestor に対してはしぼり込みが効くが，ゴールが，

(5'') $\forall A$ (~ancesotr(tarou, A))

の場合には，setting S は，

S = {~ancestor(tarou, A), ~parent(tarou,Y), ~edb(father(tarou,Y)), ~edb(mother(tarou,Y)), ~ancestor(Z,Y), ~parent(Z,Y), ~edb(father(Z,Y)), ~edb(mother(Z, Y) }

となり，ancestor をすべて求めることになってしまう．これは，内容的には同一の定義である，

(1') $\forall X, \forall Y$ (ancestor(X,Y) $\vee$ ~parent(X,Y))

(2') $\forall X, \forall Y, \forall Z$ (ancestor(X,Y) $\vee$ ~ancestor(X,Z) $\vee$ ~parent(Z,Y))

に定義を変更すると，状況が逆になる（第１引き数のしぼり込みの効果がなくなる）．

２．５　まとめと考察

知識ベース指向処理の１つとして，関係データベース中に知識を格納して，関係データベースと一階述語論理の推論機構を組み合わせて処理を進める演繹データベースについて２種類の処理方法を示した．

１つは，与えられたゴールから推論を始め内包データベースで導出できる部分を導出しておいて外延データベースにプランを送って演繹を進める間送評価法である．間送評価で *check* 述語を用いて最小不動点を判定することにより，簡単な再帰呼び出しを含む演繹を行うことができる．その際に，途中結果を保持しておくことにより，冗長な演算を省くことが可能である．

複雑な再帰呼び出しを含む場合には，もう１つのsetting 評価法が有効である．まず与えられたゴールから演繹に必要な関係を setting として抽出して，その関係間のデータベース演算によって演繹を進める方法である．最小不動点のチェックを，全体でなく部分で

行い，その論理積を取ることにより，どのような再帰的定義を含むような演繹に対しても，停止性を保証できる．

演繹データベースでは，内包データベースを扱う推論機構と外延データベースを扱う検索機構の間の通信頻度が全体の性能に大きく影響を与える．間送評価法では，プランを生成する度に推論機構から検索機構に検索依頼を発生する必要があった．setting 評価法では，setting 評価用不動点検出アルゴリズムを実行できる機能を検索機構側で用意さえすれば，推論機構側から検索機構側への通信は 1 回で済むことになり，効率的である．比較処理および繰り返し処理を検索機構側で用意することは，困難ではない．

第３章　項関係上の単一化検索を使った知識ベース指向処理

３．１　はじめに

知識ベース処理として，関係に項（述語）を格納し関係代数に単一化を導入することにより，知識ベース機構のみで推論を行う方法を提案する．格納される知識は構造や変数を持つことができるため，関係データベースより柔軟な内容にできる．また，知識は項関係（項を格納した関係）ごとに管理されるため，知識にモジュラリティを持たせることができ，項関係ごとに更新の排他制御を行うことにより知識の共有も可能になる．

本章では，その項関係上の単一化検索の繰り返しによるホーン節推論の理論的な定式化を行う．まず，Prologで行われているSLD演繹[Apt-Emden 82, Lloyd 84]を項関係上の単一化検索の繰り返しで実現するアルゴリズムを示し，その正当性を証明する．次に，SLD演繹が与えられたゴールから後ろ向きに推論を進めるのに対して，現在ある事実から前向きに推論を進める単位演繹[Loveland 78]を項関係上の単一化検索の繰り返しにより実現するアルゴリズムを提案する．ただし，一般の単位演繹をそのまま実現するのは困難であるため，SLD演繹と同様の選択関数を用いる単位演繹としてSUD演繹を提案し，SUD演繹を項関係上の単一化検索により実現する方法を示す．

３．２　単一化検索(RBU) モデル

３．２．１　項関係

【定義 3-1】 F_n を n 引数の関数記号(Function Symbol)の有限集合，V を次の式を満足する変数(Variable)の可算無限集合とする．

$$\forall n, F_n \cap V = \varnothing. \qquad \blacksquare$$

【定義 3-2】 次のような T を<u>項集合(Term Set)</u>と呼び，その要素 t を<u>項 (Term)</u>と呼ぶ．

i)　もし $t \in F_0$ または$t \in V$ならば$t \in T$

ii)　もし $t_1,...,t_n \in T,\ f \in F_n\ (n \geq 1)$ ならば

$$f(t_1,...,t_n) \in T. \qquad \blacksquare$$

【定義 3-3】 $T_1, T_2,..., T_m$ を項集合としたとき，次のような TR^m を m 属性の<u>項関係(Term Relation)</u>と呼ぶ（属性数が自明のときには肩文字は省略する）．

$$T_1 \times T_2 \times ... \times T_m \supset TR^m\ (m \geq 1).$$

この時次のような tt^m を<u>項タプル(Term Tuple)</u>と呼ぶ．

$$tt^m = (t_1, t_2,..., t_m) \in TR^m.$$

また，t_i を項タプル tt^m の i 番目の<u>アイテム</u>と呼び，$tt^m[i]$ で表す． $\blacksquare$

3．2．2　単一化検索

【定義 3-4】 代入(Substitution)λを$\{t_1/v_1, \ldots, t_n/v_n\}$の形式の有限集合とする．ただし，

$v_1, \ldots, v_n \in V$　$(i \neq j$のとき$v_i \neq v_j)$

$t_1, \ldots, t_n \in T$　$(t_i \neq v_i)$

とする．代入λによって項t中の変数を置き換えた$t\lambda$をtの代入例(Instance)と呼ぶ．□

【定義 3-5】 $t_1, t_2 \in T$で，$t_1\theta = t_2\theta$なる代入θが存在するとき，t_1, t_2を単一化可能(Unifiable)，代入θをt_1とt_2の単一化作用素(Unifier)と呼ぶ．□

【定義 3-6】 代入λ_aの後で代入λ_bを行うことを$\lambda_a \bullet \lambda_b$と記す．□

【定義 3-7】 単一化可能なt_1, t_2に対する単一化作用素の集合をΘとしたとき，

$$\forall \theta_k \in \Theta,\ \exists \lambda_k,\ \theta_k = \sigma \bullet \lambda_k$$

なる単一化作用素$\sigma \in \Theta$を，最汎単一化作用素(Most General Unifier)と呼ぶ．□

【定義 3-8】 項関係TR_a^mのi番目$(1 \leq i \leq m)$と項関係TR_b^nのj番目$(1 \leq j \leq n)$のアイテムが単一化可能であるような項タプルの組合せから，次のような新しい項関係TR_c^{m+n}を作る演算を単一化結合 (Unification-Join)と呼び，$uj(TR_a^m, i, TR_b^n, j, TR_c^{m+n})$と記す．

$$tt_c^{m+n} \in TR_c^{m+n}$$
$$\Leftrightarrow \exists tt_a^m \in TR_a^m, \exists tt_b^n \in TR_b^n, \exists \sigma, tt_a^m[i]\sigma = tt_b^n[j]\sigma,$$
$$tt_c^{m+n}[k] = \begin{cases} tt_a^m[k]\sigma & (1 \leq k \leq m) \\ tt_b^n[k-m]\sigma & (m+1 \leq k \leq m+n) \end{cases}$$

なおσは$tt_a^m[i]$と$tt_b^n[j]$の最汎単一化作用素である．□

【定義 3-9】 項関係TR_a^mのi番目$(1 \leq i \leq m)$のアイテムが検索条件の項tと単一化可能である項タプルから，次のような新しい項関係TR_b^mを作る演算を単一化制約(Unification-Restriction)と呼び，$ur(TR_a^m, i, t, TR_b^m)$と記す．

$$tt_b^m \in TR_b^m$$
$$\Leftrightarrow \exists tt_a^m \in TR_a^m, \exists \sigma,\ tt_a^m[i]\sigma = t\sigma,$$
$$tt_b^m[k] = tt_a^m[k]\sigma\ (1 \leq k \leq m)$$

□

【定義 3-10】 項関係TR_a^mのi番目$(1 \leq i \leq m)$のアイテムが変数である項タプルと変数でない項タプルとに分類して，次のような新しい項関係TR_b^mとTR_c^mを作る演算を変数制約(Variable-Restriction)と呼び，$vr(TR_a^m, i, TR_b^m, TR_c^m)$と記す．

$$tt_b^m \in TR_b^m$$
$$\Leftrightarrow \exists tt_a^m \in TR_a^m,\ tt_a^m[i] \in V,$$
$$tt_b^m[k] = tt_a^m[k]\ (1 \leq k \leq m)$$
$$tt_c^m \in TR_c^m$$

$\Leftrightarrow \exists\, tt_a^m \in TR_a^m,\ tt_a^m[i] \notin V,$

$tt_c^m[k] = tt_a^m[k]\ \ (1 \leq k \leq m)$ □

【定義 3-11】 項関係 TR_a^m から，$a_1,\ldots, a_n\ (1 \leq a_i \leq m)$の n個の属性を取り出して，次のような新しい項関係 TR_b^nを生成する操作を射影(Projection)と呼び，$pr(TR_a^m,[a_1,\ldots, a_n],TR_b^n)$ と記す．

$tt_b^n \in TR_b^n$

$\Leftrightarrow \exists\, tt_a^m \in TR_a^m,\ tt_b^n[i] = tt_a^m[a_i]\ \ (1 \leq i \leq n)$ □

【定義 3-12】項関係 TR_a^m と項関係TR_b^m からその集合和の項関係 TR_c^mを生成する操作を$un(TR_a^m,TR_b^m,TR_c^m)$と記す（ただし，変数名の付け替えを行う）． □

これらの操作を総称して単一化検索(RBU: Retrieval By Unification)と呼ぶ[5)]．

3．3 単一化によるホーン節の導出

ホーン節[Lloyd 84]を二進木で表現し，二進木どうしの単一化のみで導出形(Resolvent)が得られることを示す．

【定義 3-13】二進木述語列を次のように再帰的に定義する．

i) 変数 υ は二進木述語列である．

ii) Pが述語，B が二進木述語列であるとき，

$\tau(P,B)$

は二進木述語列である．

ただし，υ はいかなる述語中にも現れない変数，τ はいかなる述語中にも現れない２引数関数記号とする． □

【定義 3-14】ホーン節$C = p_0 \vee \bar{}p_1 \vee \bar{}p_2 \vee \bar{}p_3 \vee \ldots \vee \bar{}p_n$ の正リテラルと負リテラルをそれぞれ

$pos(C) = \tau(p_0,\ \upsilon)$

$neg(C) = \tau(p_1, \tau(p_2, \tau(p_3, \ldots \tau(p_n,\upsilon)\ldots)))$

という別々の二進木述語列にした組を節 C の二進木表現と呼ぶ．リテラルが空の場合には，２つの二進木述語列間の共通変数υ をその二進木述語列とする．つまり，C が正の単位節（１つの正のリテラルのみからなる節）の場合は，

$neg(C) = \upsilon$

負節（負リテラルのみからなる節）の場合は，

$pos(C) = \upsilon$

空節の場合は，

$pos(C) = neg(C) = \upsilon$

となる． □

【定理 3-1】ホーン節 C_1 の最左端の負リテラルとホーン節 C_2 の正リテラルから，C_1 と C_2 の導出形 R が導出可能な時，$neg(C_1)$と$pos(C_2)$ の間の単一化により R の二進木表現を得ることができる．

【証明】ホーン節 $C_1 = p_0 \vee \tilde{}p_1 \vee \tilde{}p_2 \vee ... \vee \tilde{}p_n$ の最左端の負リテラル $\tilde{}p_1$ を使って導出形を求めることができるようなもう一方のホーン節 C_2 は，

$$p_1\sigma = q_0\sigma \qquad 〔式1〕$$

であるような，$C_{2a} = q_0 \vee \tilde{}q_1 \vee \tilde{}q_2 \vee ... \vee \tilde{}q_m\ (1 \leq m)$ もしくは，$C_{2b} = q_0$の何方かである．

（1）C_{2a} の場合

式1から導出形は

$$R_a = (p_0 \vee \tilde{}q_1 \vee \tilde{}q_2 \vee ... \vee \tilde{}q_m \vee \tilde{}p_2 \vee ... \vee \tilde{}p_n)\sigma \qquad 〔式2〕$$

となる．C_1の二進木表現は，

$$pos(C_1) = \tau(p_0,\ \upsilon)$$

$$neg(C_1) = \tau(p_1, \tau(p_2, \tau(p_3, ... \tau(p_n, \upsilon)...)))$$

C_{2a}の二進木表現は，

$$pos(C_{2a}) = \tau(q_0,\ \upsilon')$$

$$neg(C_{2a}) = \tau(q_1, \tau(q_2, \tau(q_3, ... \tau(q_m, \upsilon')...)))$$

ここで，υ，υ' は定義3-13 から$p_0, ..., p_n, q_0, ..., q_m$中に現れない変数である．式1から，$neg(C_1)$と$pos(C_{2a})$は単一化可能であり，

$$\tau(p_1, \tau(p_2, \tau(p_3, ... \tau(p_n, \upsilon)...)))\sigma' = \tau(q_0,\ \upsilon')\sigma'$$

これより，

$$\sigma' = \{\tau(p_2, \tau(p_3, ... \tau(p_n, \upsilon)...))/\upsilon'\} \bullet \sigma \qquad 〔式3〕$$

これを $pos(C_1)$と $neg(C_{2a})$に適用すると，

$$pos(C_1)\sigma'$$
$$= \tau(p_0,\ \upsilon)\sigma'$$
$$= \tau(p_0,\ \upsilon)\sigma \qquad (\because \upsilon' \text{は} p_0 \text{に含まれないため})$$

$$neg(C_{2a})\sigma'$$
$$= \tau(q_1, \tau(q_2, \tau(q_3, ... \tau(q_m, \upsilon')...)))\sigma'$$
$$= \tau(q_1, \tau(q_2, \tau(q_3, ... \tau(q_m, \tau(p_2, \tau(p_3, ... \tau(p_n, \upsilon)...)))\sigma$$

これらは式2の導出形 R_a の二進木表現に他ならない．

（2）C_{2b}の場合

導出形は，

$R_b = (p_0 \vee \tilde{}p_2 \vee ... \vee \tilde{}p_n)\,\sigma$ 〔式４〕

となる．また，C_{2b}の二進木表現は，

$pos(C_{2b}) = \tau\,(q_0,\ \upsilon')$

$neg(C_{2b}) = \upsilon'$

となる．式１から，$neg(C_1)$と$pos(C_{2b})$は単一化可能であり，$pos(C_{2b})$は$pos(C_{2a})$と全く同じであるから，式３と同一の最汎単一化作用素が求まる．これを$pos(C_1)$と$neg(C_{2b})$に適用すると，

$$
\begin{aligned}
pos(C_1)\sigma' &= \tau\,(p_0,\ \upsilon)\sigma' \\
&= \tau\,(p_0,\ \upsilon)\sigma \\
neg(C_{2b})\sigma' &= \upsilon'\sigma' \\
&= \tau\ (p_2, \tau\,(p_3, ... \tau\,(p_n, \upsilon) ...)))\sigma
\end{aligned}
$$

これらは，式４の導出形R_bの二進木表現と同じである．（証明終了） □

３．４　項関係上のRBU 演算によるSLD演繹

SLD演繹とは，Linear resolution with Selection function for Definite clauses の略[Apt-Emden 82]で，制限された入力演繹の一種である．Prologの処理系は，探索規則に制約を加えたSLD演繹の一つの実現手段とみなすことができる[Lloyed 84]．SLD演繹では一つの負節 G (つまりPrologのゴール)と確定節（負節以外のホーン節）の集合 D (Prologのプログラム)との和集合$D \cup \{G\}$から反駁を導く．

【定義 3-15】負節 G_i (ただし，$G_0 = G$とする)の中から選択関数で一つリテラル $\tilde{}p_k$を選び出し，その $\tilde{}p_k$ と導出可能であるような適当な節（入力節）C_i を確定節の集合 D の中から選び G_i と C_i の間で導出を行なう．その導出の結果の代入をθ_i ，導出形を G_{i+1} とする．G_iが空節になると反駁が存在したことになる．このような，入力節の列 $C_0, C_1, ...$, 導出形の列 $G_0, G_1, ...$, および代入の列 $\theta_0, \theta_1, ...$, から得られる反駁をSLD反駁と呼ぶ． □

【定理 3-2】（SLD反駁の健全性）確定節の集合 D とある負節 G に対して，$D \cup \{G\}$ におけるSLD反駁が存在するならは，$D \cup \{G\}$は充足不能である． □

【定理 3-3】（SLD反駁の完全性）確定節の集合 D とある負節 G に対して，$D \cup \{G\}$ が充足不能であるとする．この時 $D \cup \{G\}$ におけるSLD反駁が存在する． □

定理3-2 および定理3-3 の証明については，[Lloyd 84]を参照されたい．

次に，ホーン節を二進木表現し一つのホーン節 C を一つの項タプルに対応させて，$pos(C)$ を1番目のアイテム，$neg(C)$ を2番目のアイテムとして，ホーン節の集合を項関係に格納し，その上の RBU 演算の繰り返しによりSLD演繹が行えることを示す．

【アルゴリズム 3-1】

Step 1: 確定節の集合D ($C_k \in D$) の二進木表現が格納されている2属性の項関係をTR_Dとし,

$tt_D \in TR_D$, $tt_D[1] = pos(C_k)$, $tt_D[2] = neg(C_k)$

とする．また，ゴールGも二進木表現にして2属性の項関係TR_Gに

$tt_G \in TR_G$, $tt_G[1] = pos(G) = \upsilon$, $tt_G[2] = neg(G)$

のように格納する．

Step 2: $uj(TR_G, 2, TR_D, 1, TR_{a(0)})$

$pr(TR_{a(0)}, [2,4], TR_{b(0)})$

$i = 0$

Step 3: もし，$TR_{b(i)} = \varnothing$（$TR_{b(i)}$が空集合）なら，

反駁に失敗して終了.

そうでなければ， Step 4 へ.

Step 4: $vr(TR_{b(i)}, 2, TR_{c(i)}, TR_{d(i)})$

Step 5: もし，$TR_{c(i)} \neq \varnothing$なら，反駁に成功して終了，

$TR_{c(i)}$の第一属性がゴールに対する代入を示す.

そうでなければ， Step 6 へ.

Step 6: $i = i + 1$

$uj(TR_{d(i-1)}, 2, TR_D, 1, TR_{a(i)})$

$pr(TR_{a(i)}, [1,4], TR_{b(i)})$

Step 3へ. ■

【定理 3-4】アルゴリズム1は，$D \cup \{G\}$ が充足不能である場合には，選択関数で導出形の最左端のリテラルを選ぶSLD反駁を求めて，停止する．

【証明】Step 2の$uj(TR_G, 2, TR_D, 1, TR_{a(0)})$ と，定義3-8 から，

$$\begin{aligned}&tt_{a(0)} \in TR_{a(0)}\\ &\Leftrightarrow \exists tt_G \in TR_G, \exists tt_D \in TR_D, \exists \theta_0,\ tt_G[2]\theta_0 = tt_D[1]\theta_0,\\ &\quad tt_{a(0)}[1] = tt_G[1]\theta_0,\ tt_{a(0)}[2] = tt_G[2]\theta_0,\\ &\quad tt_{a(0)}[3] = tt_D[1]\theta_0,\ tt_{a(0)}[4] = tt_D[2]\theta_0\end{aligned}$$

これより，

$$neg(G)\theta_0 = tt_G[2]\theta_0 = tt_D[1]\theta_0 = pos(C_0)\theta_0$$

この時，定理3-1と定義3-15 より，

$$neg(C_0)\,\theta_0 = neg(G_1)$$

となる．次に，$pr(TR_{a(0)}, [2,4], TR_{b(0)})$ から，

$tt_{b(0)}[1] = tt_{a(0)}[2] = tt_G[2]\theta_0 = neg(G)\ \theta_0$

$tt_{b(0)}[2] = tt_{a(0)}[4] = tt_D[2]\theta_0 = neg(C_0)\ \theta_0 = neg(G_1)$

つまり，$TR_{b(0)}$ の第１属性は最初の負節中の変数に対する代入を，$TR_{b(0)}$ の第２属性は一番目の導出形の負リテラルを表わしている．次に, Step 4の $vr(TR_{b(i)}, 2, TR_{c(i)}, TR_{d(i)})$ と，定義3-10 から，

$tt_{c(0)} \in TR_{c(0)}$

$\Leftrightarrow \exists\ tt_{b(0)} \in TR_{b(0)},$

$tt_{c(0)}[2] = tt_{b(0)}[2] = neg(G_1) \in V,$

$tt_{c(0)}[1] = tt_{b(0)}[1] = neg(G)\ \theta_0$

$tt_{d(0)} \in TR_{d(0)}$

$\Leftrightarrow \exists\ tt_{b(0)} \in TR_{b(0)},$

$tt_{d(0)}[2] = tt_{b(0)}[2] = neg(G_1) \notin V,$

$tt_{d(0)}[1] = tt_{b(0)}[1] = neg(G)\ \theta_0$

もし，$TR_{c(0)}$ が$\varnothing$ なら，Step 6の $uj(TR_{d(i-1)}, 2, TR_D, 1, TR_{a(i)})$，$pr(TR_{a(i)}, [1,4], TR_{b(i)})$ より，同様に，

$tt_{b(1)}[1] = neg(G)\ \theta_0 \bullet \theta_1$

$tt_{b(1)}[2] = neg(C_1)\ \theta_1 = neg(G_2)$

ただし，

$neg(G_1)\ \theta_1 = pos(C_1)\ \theta_1$

さらに，これを繰り返して，

$tt_{b(i)}[1] = neg(G)\ \theta_0 \bullet \ldots \bullet \theta_i$

$tt_{b(i)}[2] = neg(C_i)\ \theta_i = neg(G_{i+1})$

ただし，

$neg(G_i)\ \theta_i = pos(C_i)\ \theta_i$

となる．定義3-15 と定義3-14 から，G_i が空節，つまり $pos(G_i) = neg(G_i) = \upsilon$ のときSLD 反駁が得られたことになる．常に $pos(G_i) = \upsilon$ であるため，$neg(G_i) = tt_{b(i-1)}[2] \in V$ つまり，繰り返し中に $TR_{c(i)}$ が$\varnothing$でなくなるとき，導出形として空節が求まったことになる．定理３より $D \cup \{G\}$ が充足不能である場合，空節となる導出形が存在する．Step 2において $uj(TR_{d(i-1)}, 2, TR_D, 1, TR_{a(i)})$で，前段の導出形と導出可能な全ての入力節とから導出形を求めているため，$D \cup \{G\}$ が充足不能である場合には，アルゴリズム１はその反駁に対応する代入の列を求めて終了する．（証明終了） □

現在実現されているPrologも，導出形の最左端のリテラルを選ぶという選択関数で，SLD演繹を実現している．しかし，確定節集合 D の中から入力節C_i を探す場合に，節の

宣言された順番に従うため，完全ではない[Lloyd 84]．つまり，$D \cup \{G\}$ が充足不能であっても，反駁が求まらない場合がある．これに対して，アルゴリズム3-1では，入力節 C_i として単一化結合により可能な全ての候補が適用されるため，探索規則が均等となり，$D \cup \{G\}$ が充足不能である場合には必ずSLD反駁が得られる．

更に，アルゴリズム3-1 に変更を加えることにより，全解探索を行うアルゴリズム3-1'を得ることができる．

【アルゴリズム3-1'】

Step 1: 確定節の集合D ($C_k \in D$) の二進木表現が格納されている２属性の項関係をTR_Dとし，

$tt_D \in TR_D$, $tt_D[1] = pos(C_k)$, $tt_D[2] = neg(C_k)$

とする．

また，ゴールGも二進木表現にして２属性の項関係TR_Gに

$tt_G \in TR_G$, $tt_G[1] = pos(G) = \upsilon$, $tt_G[2] = neg(G)$

のように格納する．

Step 2: $TR_{x(0)} = \varnothing$

$uj(TR_G, 2, TR_D, 1, TR_{a(0)})$

$pr(TR_{a(0)}, [2,4], TR_{b(0)})$

$i = 0$

Step 3: $vr(TR_{b(i)}, 2, TR_{c(i)}, TR_{d(i)})$

$un(TR_{c(i)}, TR_{x(i)}, TR_{x(i+1)})$

Step 4: もし，$TR_{d(i)} = \varnothing$（$TR_{d(i)}$が空集合）なら終了，

$TR_{x(i+1)}$の第一属性が結果を示す．

そうでなければ，Step5 へ．

Step 5: $i = i + 1$

$uj(TR_{d(i-1)}, 2, TR_D, 1, TR_{a(i)})$

$pr(TR_{a(i)}, [1,4], TR_{b(i)})$

Step 3へ． ■

導出形の最左端のリテラルを選ぶ選択関数を持つ SLD 演繹において全ての場合の反駁が有限時間内に得られる場合，アルゴリズム3-1' はその全ての反駁における変数への代入を求めて停止する．反駁は入力節の選定によりいろいろ有り得るが，アルゴリズム3-1' では可能な全ての入力節を単一化結合により適用しているため，$TR_{b(i)}$ の第二属性はi番目の繰り返しにおける全ての導出形の集合となる．定義3-15 から新たな導出形は必ず

その前段の導出形から作られる．全ての場合の反駁が有限時間内に得られる場合には，$TR_{d(n)} = \varnothing$ なる n が存在することになる．この時，可能な全ての反駁によって得られる代入は，$TR_{c(i)}\ (1 \leq i \leq n)$ の第一属性に現れる．$TR_{x(o)} = \varnothing$ と $un(TR_{c(i)}, TR_{x(i)}, TR_{x(i+1)})$ より，全代入は $TR_{x(i+1)}$ の第一属性に含まれる．

3．5　選択関数を用いる単位演繹（SUD 演繹）

SLD 演繹が後ろ向きに推論を進めるのに対して前向きに推論を進める方法を考える．前向きの推論方式としては単位演繹[Loveland 78]がある．単位演繹とは，反駁において導出形を求める場合に，導出する一方の節を正の単位節に限定して演繹を進める方式である．しかし，一般の単位演繹を単一化検索を用いて実現することには難がある．そこで，SLD 演繹と同様に，正の単位節と導出を行なう対象のリテラルを選択関数を用いて選ぶ演繹方式を提案し，SUD 演繹（Unit resolution with Selection function for Definite clauses）と名付ける．本章では，SUD 演繹の定義を行い，その健全性と完全性を示す．

【定義 3-16】 $U_i\ (D \supset U_0)$ を正の単位節の集合，$H_i\ (G \cup (D - U_0) = H_0)$ を複合節（正の単位節以外のホーン節）の集合とする．複合節の集合 H_i 中の適当な節 C_i の中から選択関数により負リテラル $\tilde{}p_k$ を選び，その $\tilde{}p_k$ と正の単位節 $u_k \in U_i$ との間で導出を行う．得られた導出形を R_i，代入を θ_i とし，

i) R_i が正の単位節なら，$R_i \in U_{i+1}$

ii) R_i が複合節なら，$R_i \in H_{i+1}$

また，

$U_{i+1} \supset U_i$ かつ $H_{i+1} \supset H_i$

とする．空節の R_i が求まると反駁が存在したことになる．このような，正の単位節の集合の列 $U_0, U_1, \ldots$，複合節の集合の列 $H_0, H_1, \ldots$，および代入の列 $\theta_0, \theta_1, \ldots$，から得られる反駁を SUD 反駁と呼ぶ． ■

【定理 3-5】（SUD反駁の健全性）ホーン節の集合 $S = D \cup \{G\}$ に対して SUD 反駁が存在するならば，S は充足不能である．

【証明】 $G = \tilde{}p_1 \vee \tilde{}p_2 \vee \ldots \vee \tilde{}p_m\ (1 \leq m)$ とする．SUD 反駁が存在するならば，定義15より，R_i が空節になる i において，$p_1\theta \in U_i$，$p_2\theta \in U_i$，…，$p_m\theta \in U_i$ なる θ が存在する．S のモデルを $M(S)$ とすると，$M(S) \supset U_i$ である．よって，$G\theta$ は $M(S)$ において真とは成りえない．つまり，S は充足不能である．（証明終了） ■

次に，SUD反駁の完全性を示す．このため，Anderson ら[Anderson-Bledsoe 70] が演繹の完全性を示すために提案した超過リテラルパラメタ（ELP:Excess Literal Parameter）と充足不能最小集合（Minimally Unsatisfiable Set）を用い，帰納的に証明する．

【定義 3-17】Sを節の集合$\{C_1,...,C_n\}$とし，LCを$C_1,...,C_n$中のリテラル数の合計とする．つまり，

$$LC = \#(C_1) + ... + \#(C_n)$$

ここで，$\#(C_i)$は節C_i中のリテラル数である．Sに対する超過リテラルパラメタを

$$ELP(S) = LC - n$$

とする． □

【補題 3-1】SをC_iが空節でない基底節（変数を持たない節）集合とする．$ELP(S)$が0になるのは，全てのC_iが単位節の時でありかつその時だけである．

【証明】LCとnとが等しいことから明らか． □

【補題 3-2】Sを，次のような正の単位節pおよび節$\tilde{}p \vee C_2$を含む基底節の集合とする

$$S = \{p, \tilde{}p \vee C_2, C_3,...,C_n\}$$

この時，節$\tilde{}p \vee C_2$をその節とpとの導出形C_2で置き換えた基底節の集合，

$$S' = \{p, C_2, C_3,...,C_n\}$$

は，$ELP(S') = ELP(S) - 1$を満足する．

【証明】nが変わらずにLCが1減ることから明らか． □

【定義 3-18】節の集合$S = \{C_1,...,C_n\}$が充足不能であり，SからどのようなC_iを除いても充足可能となるようなSを，充足不能最小集合と呼ぶ． □

【補題 3-3】SおよびS'を補題2と同様の節集合とする．Sが充足不能最小集合の時，以下の何れかが成り立つ．

i) C_2が空節．この場合，$S = \{p, \tilde{}p\}$

ii) C_2は空節でなく，S'が充足不能最小集合

iii) C_2は空節でなく，$S' - \{p\}$が充足不能最小集合

【証明】もしC_2が空節だとすると，Sの最初の２つの節はpと$\tilde{}p$であり，この２つの節自身で充足不能最小集合を形成する．よって，$S = \{p, \tilde{}p\}$．次にもしC_2が空節でない場合を考える．Sは充足不能であり，S'もやはり充足不能である．S'もしくは$S' - \{p\}$が充足不能最小集合でないとすると，S'もしくは$S' - \{p\}$からある$C_i (2 \le i \le n)$を除いたものも充足不能となる．$3 \le i \le n$時充足不能だとすると，C_iはSからS'への導出には使われていないため，$S' - \{C_i\}$も充足不能となり，Sが充足不能最小集合であることに反する．C_2を除いても充足不能だとすると，$\{p, C_3,...,C_n\}$が充足不能ということになり，$S = \{p, \tilde{}p \vee C_2, C_3,...,C_n\}$が充足不能最小集合であることに反する．（証明終了） □

【補題 3-4】Sが空節を含まない基底ホーン節のみからなる充足不能最小集合の時，Sの中に正の単位節pと最左端の負リテラルが$\tilde{}p$となるような節Cが存在する．

【証明】Sの中の全ての正の単位節pに対して，S中いかなる節の最左端の負リテラルも

$\tilde{\ }p$ とならないと仮定する．この場合，最左端以外の負リテラルと正の単位節とで導出を行っても新たな単位節は生成されない．つまり，空節は決して生成されない．これはS が充足不能最小集合であるという前提に反する．（証明終了） □

【補題 3-5】基底のホーン節からなる集合S が充足不能な時，S において選択関数として最左端の負リテラルを選ぶようなSUD 反駁が存在する．

【証明】S を空節を含まない充足不能最小集合としても，反駁の存在を証明するのに一般性は失われない．$n = ELP(S)$ の帰納的証明を行う．

$n = 0$ の時：S が充足不能最小集合であることおよび補題3-1から，$S = \{p, \tilde{\ }p\}$．これより反駁は存在する．

$n > 0$ の時：補題3-4 から，S 中に正の単位節 p および$\tilde{\ }p \vee C$なる節が存在する．S' をp と$\tilde{\ }p \vee C$をその間の導出形 C で置き換えたものとする．補題3-3よりS' もしくは $S' - \{p\}$ は充足不能最小集合．補題3-2 より， $ELP(S') = ELP(S) - 1$ よって，帰納法よりSUD反駁が存在する．（証明終了） □

【定理 3-6】（SUD 反駁の完全性）ホーン節の集合S が充足不能な時，S において選択関数として最左端の負リテラルを選ぶようなSUD 反駁が存在する．

【証明】補題 3-5と持ち上げ定理(Lifting Theorem)[Loveland 78]より，直ちに示される．（証明終了） □

3．6 項関係上の RBU 演算による SUD 演繹

次に，U_i 中およびH_i 中の節をそれぞれ３属性の項関係に格納して，単一化検索の繰り返しによりSUD 演繹が行なえることを示す．ここで，３属性の項関係の第一属性は，節中の正リテラルを，第二属性は負リテラルを，第三属性は代入の内容（初期状態の負リテラルに対する代入）を保持するために用いられる．

【アルゴリズム 3-2】

Step 1: 正の単位節の集合$(u_k \in U)$の格納されている

３属性の項関係を$TR_{u(0)}$ とし，

$tt_{u(0)} \in TR_{u(0)}$, $tt_{u(0)}[1] = pos(u_k)$,

$$tt_{u(0)}[2] = tt_{u(0)}[3] = neg(u_k) = \upsilon$$

また，複合節の集合$(C_h \in H)$の格納されている

３属性の項関係を$TR_{h(0)}$ とし，

$tt_{h(0)} \in TR_{h(0)}$, $tt_{h(0)}[1] = pos(C_h)$,

$$tt_{h(0)}[2] = tt_{h(0)}[3] = neg(C_h)$$

とする．

Step 2: $i = 0$

Step 3: $uj(TR_{h(i)}, 2, TR_{u(i)}, 1, TR_{a(i)})$

$pr(TR_{a(i)}, [1,5,3], TR_{b(i)})$

$vr(TR_{b(i)}, 2, TR_{tu(i)}, TR_{th(i)})$

Step 4: $vr(TR_{tu(i)}, 1, TR_{c(i)}, TR_{c(i)})$

もし，$TR_{c(i)} \neq \emptyset$なら，反駁に成功して終了，

$TR_{c(i)}$の第三属性がゴールに対する代入を示す．

Step 5: $un(TR_{u(i)}, TR_{tu(i)}, TR_{u(i+1)})$

$un(TR_{h(i)}, TR_{th(i)}, TR_{h(i+1)})$

Step 6: もし，$TR_{u(i+1)} = TR_{u(i)}$ かつ $TR_{h(i+1)} = TR_{h(i)}$ なら，

反駁に失敗して終了．

そうでなければ，$i = i + 1$ として Step 3 へ． □

【定理 3-7】アルゴリズム3-2 は，$D \cup \{G\}$ が充足不能である場合には，導出形の最左端のリテラルを選ぶという選択関数で SUD 反駁を求めて，停止する．

【証明】繰り返し中のStep3の $uj(TR_{h(i)},2,TR_{u(i)},1,TR_{a(i)})$ と $pr(TR_{a(i)}, [1,5,3], TR_{b(i)})$ から，

$$tt_{b(i)} \in TR_{b(i)}$$

$$\Leftrightarrow \exists tt_{h(i)} \in TR_{h(i)}, \exists tt_{u(i)} \in TR_{u(i)}, \exists \theta_i,$$

$$tt_{h(i)}[2]\theta_i = tt_{u(i)}[1]\theta_i,$$

$$tt_{b(i)}[1] = tt_{h(i)}[1]\theta_i = pos(C_i)\theta_i = pos(R_i)$$

$$tt_{b(i)}[2] = tt_{u(i)}[2]\theta_i = neg(u_i)\theta_i = neg(R_i)$$

$$tt_{b(i)}[3] = tt_{h(i)}[3]\theta_i = neg(C_0)\theta_0 \bullet \dots \bullet \theta_i$$

この時，$tt_{h(i)}[2]\theta_i = tt_{u(i)}[1]\theta_i$ から，$neg(C_i)\theta_i = pos(u_i)\theta_i$ となる．また，定義3-14 より，R_iが単位節なら $neg(R_i)$ は変数になる．よって，$vr(TR_{b(i)}, 2, TR_{tu(i)}, TR_{th(i)})$から，$tt_{tu(i)} \in TR_{tu(i)}$ なる $tt_{tu(i)}$ は単位節，$tt_{th(i)} \in TR_{th(i)}$ なる $tt_{th(i)}$ は複合節を表現する．もし，$vr(TR_{tu(i)}, 1, TR_{c(i)}, TR_{d(i)})$ の $TR_{c(i)}$ が$\emptyset$ならば，$un(TR_{u(i)}, TR_{tu(i)}, TR_{u(i+1)})$ から $tt_{tu(i)} \in TR_{u(i+1)}$，$un(TR_{h(i)}, TR_{th(i)}, TR_{h(i+1)})$ から $tt_{th(i)} \in TR_{h(i+1)}$として繰り返される．これより，アルゴリズム3-2 は，SUD演繹を実現している．定理3-5, 3-6より$D \cup \{G\}$ が充足不能である場合，またその時だけ，導出形R_iが空節となる．これは，

$$pos(R_i) = neg(R_i) = \upsilon$$

つまり，

$$tt_{tu(i)}[1] \in V$$

に対応する．すなわち，$D \cup \{G\}$ が充足不能である場合，繰り返し中に$TR_{c(i)}$ が$\emptyset$ でなくなる．そのとき，アルゴリズム3-2 はSUD 反駁に対応する代入の列を求めて終了する．

（証明終了）

アルゴリズム3-2の繰り返しでは，新しく導出された単位節や複合節を，前の単位節の集合および複合節の集合に含めて単一化結合を行っている．このため，同一の導出形を何度も作ってしまい，明らかに無駄な処理が多くなる．そこで，新しく作られた導出形とのみ単一化結合を行うように変更したアルゴリズム3-3を提案する．

【アルゴリズム 3-3】

Step 1: 正の単位節の集合$(u_k \in U)$の格納されている

3属性の項関係を$TR_{u(0)}$とし，

$tt_{u(0)} \in TR_{u(0)}$,

$tt_{u(0)}[1] = pos(u_k),\ tt_{u(0)}[2] = tt_{u(0)}[3] = neg(u_k) = \upsilon$

また，複合節の集合$(C_h \in H)$の格納されている

3属性の項関係を$TR_{h(0)}$とし，

$tt_{h(0)} \in TR_{h(0)}$,

$tt_{h(0)}[1] = pos(C_h),\ tt_{h(0)}[2] = tt_{h(0)}[3] = neg(C_h)$

とする．

Step 2: $uj(TR_{h(0)}, 2, TR_{u(0)}, 1, TR_{a(0)})$

$pr(TR_{a(0)}, [1,5,3], TR_{b(0)})$

$vr(TR_{b(0)}, 2, TR_{tu(0)}, TR_{th(0)})$

Step 3: $i = 0$

Step 4: $vr(TR_{tu(i)}, 1, TR_{c(i)}, TR_{d(i)})$

もし，$TR_{c(i)} \neq \emptyset$なら，反駁に成功して終了，

$TR_{c(i)}$の第三属性がゴールに対する代入を示す．

Step 5: $uj(TR_{h(i)}, 2, TR_{tu(i)}, 1, TR_{ax(i)})$,

$pr(TR_{ax(i)}, [1,5,3], TR_{bx(i)})$,

$vr(TR_{bx(i)}, 2, TR_{ux(i)}, TR_{hx(i)})$,

$uj(TR_{th(i)}, 2, TR_{u(i)}, 1, TR_{ay(i)})$,

$pr(TR_{ay(i)}, [1,5,3], TR_{by(i)})$,

$vr(TR_{by(i)}, 2, TR_{uy(i)}, TR_{hy(i)})$,

$uj(TR_{th(i)}, 2, TR_{tu(i)}, 1, TR_{az(i)})$,

$pr(TR_{az(i)}, [1,5,3], TR_{bz(i)})$,

$vr(TR_{bz(i)}, 2, TR_{uz(i)}, TR_{hz(i)})$,

Step 6: $un(TR_{u(i)}, TR_{tu(i)}, TR_{u(i+1)})$,

$un(TR_{h(i)}, TR_{th(i)}, TR_{h(i+1)})$,

$un(TR_{ux(i)}, TR_{uy(i)}, TR_{uw(i)})$,

$un(TR_{uw(i)}, TR_{uz(i)}, TR_{tu(i+1)})$

$un(TR_{hx(i)}, TR_{hy(i)}, TR_{hw(i)})$,

$un(TR_{hw(i)}, TR_{hz(i)}, TR_{th(i+1)})$

Step 7: もし，$TR_{u(i+1)}=TR_{u(i)}$ かつ $TR_{h(i+1)}=TR_{h(i)}$ なら，

反駁に失敗して終了.

そうでなければ，$i=i+1$ としてStep 4へ. ■

【定理 3-7】アルゴリズム3は，アルゴリズム2と同様に $D \cup \{G\}$ が充足不能である場合には，導出形の最左端のリテラルを選ぶという選択関数で SUD 反駁を求めて，停止する.

【証明】ここでは，$uj(TR_a^m, i, TR_b^n, j, TR_c^{m+n})$ を $TR_c^{m+n} = TR_a^m \ {}_i\&_j \ TR_b^n$，$un(TR_a^m, TR_b^m, TR_c^m)$ を $TR_c^m = TR_a^m \cup TR_b^m$ で表わすことにする．単一化結合は，集合の要素間の組合せであるから，

$$
\begin{aligned}
& TR_{h(i+1)} \ {}_2\&_1 \ TR_{u(i+1)} \\
&= (TR_{h(i)} \cup TR_{th(i)}) \ {}_2\&_1 \ (TR_{u(i)} \cup TR_{tu(i)}) \\
&= (TR_{h(i)} \ {}_2\&_1 \ TR_{u(i)}) \cup (TR_{h(i)} \ {}_2\&_1 \ TR_{tu(i)}) \\
&\quad \cup (TR_{th(i)} \ {}_2\&_1 \ TR_{u(i)}) \cup (TR_{th(i)} \ {}_2\&_1 \ TR_{tu(i)})
\end{aligned}
$$

この内，繰り返しにより，

$$TR_{h(i)} \ {}_2\&_1 \ TR_{u(i)}$$

の結果は既に求められているため，

$$
\begin{aligned}
& (TR_{h(i)} \ {}_2\&_1 \ TR_{tu(i)}) \cup (TR_{th(i)} \ {}_2\&_1 \ TR_{u(i)}) \\
& \quad \cup (TR_{th(i)} \ {}_2\&_1 \ TR_{tu(i)})
\end{aligned}
$$

のみを計算すればよい．(証明終了) ■

アルゴリズム3-1’と同様に，アルゴリズム3-2，アルゴリズム3-3を全解探索用に変更することは簡単である.

3．7　まとめと考察

演繹データベースでは知識を表現するための構造や変数を文字列としてしか扱えないという問題点を解決するため，テーブルに変数を含む構造体である項を格納し単一化を使って検索を行う単一化検索モデルを提案し，そのモデル上でホーン節の演繹を行う方法を示した.

ホーン節推論の方法としては，Prolog などで使われている SLD 演繹と，単位演繹に SLD 演繹同様の選択関数を導入した SUD 演繹に対する実現アルゴリズムを示した．項関

係に格納された演繹の対象であるホーン節の集合が充足不能の場合には，提案したアルゴリズムはそれぞれ反駁を求めて停止する．これにより，RBU モデルの論理的根拠として，ホーン節を項関係に知識ベースとして格納した場合の定式化を行うことができた．

さらに同一の枠組みで，単にホーン節にとどまらない各種と演繹方法の実現も可能と思われる．

第４章　RBUに基づく知識ベース指向処理システムの実現

４．１　はじめに

本章および次章では，第３章で提案したRBUに基づいて知識ベース指向処理システムを構成する方法について考える．知識ベース指向処理システムは，RBU演算を行う知識検索モジュールとその制御を行う推論モジュールから構成されるものとする．まず本章では，知識検索モジュールの実現方法について述べる．最初に，知識検索モジュールとして実際にサポートすべき命令体系について述べ，次にその実現方法を示す．

RBUに基づく知識検索モジュールの実現方法としては，RBU演算の特徴を考慮して専用ハードウェアを用いて実現する方法と，汎用プロセッサ上にソフトウェアで実現する方法が考えられる．そこで，専用ハードウェアとして単一化エンジンおよび多ポートページメモリを利用した場合のシステムの構成方法[横田他 85a，横田他 85b，Yokota-Itoh 86，Morita et al. 86]と，汎用プロセッサ上でソフトウェアのみで実現する場合の高速化の手段として専用インデックス[Yokota et al. 89]を提案する．専用インデックスについては，試作したプロトタイプを用いた評価についても述べる．

４．２　知識検索モジュールの命令体系

知識検索モジュールを実用に供するためには，第３章で述べた検索系の命令の他に，更新系，定義系などの命令をサポートすることが必要となる．つまり，検索系はRBU演算及び関係代数的な検索処理を行うもの．更新系は，項関係の内容を変更するためのもの．定義系は，項関係の定義の変更およびインデックスの処理を行うためのものである．さらに，知識検索モジュールが推論モジュールから利用されることを想定した時，検索系はさらに２種類に分類される．すなわち，検索結果を新たな項関係として残すものと，推論モジュール側にストリームとして渡すものとを準備する必要がある．前者を項関係型，後者をストリーム型と呼ぶことにする．

分類別にコマンドのフォーマットをまとめる．なお，コマンド内の[] は省略可能なことを示し，略号の意味は以下のものとする．

RN (Relation Name):	項関係名
TRN (Target Relation Name):	操作対象の項関係名
RRN (Result Relation Name):	結果を格納する項関係名
ST (Stream):	結果を出力するためのストリーム用変数
CND (Condition):	検索条件
AN (Attribute Number):	属性番号

AL (Attribute number List):	出力対象の属性番号リスト
KAN (Key Attribute Number):	インデックス対象属性番号
AC (Attribute Count):	属性数
TID (Tuple Identifier):	タプル識別子
SI (Status Information):	実行状態情報

1) 検索系コマンド

a) 単一化結合 (Unification-Join)

項関係型： *ujr(TRN1, AN1, TRN2, AN2, [AL], RRN1, SI)*

ストリーム型： *ujs(TRN1, AN1, TRN2, AN2, [AL], ST1)*

項関係 *TRN1* の属性 *AN1* と項関係 *TRN2* の属性 *AN2* の間で単一化可能な組み合わせを選びで *[AL]* 示される属性を項関係（ストリーム）として出力する．詳しくは定義 3-8 参照．なお，*AL*が省略された場合には全属性を出力する．また，正常終了情報，エラー情報等は項関係型の場合は *SI* により，ストルーム型の場合はその内容によって示される．

b) 単一化制約 (Unification-Restriction)

項関係型： *urr(TRN, CND, [AL], RRN1 [,RRN2], SI)*

ストリーム型： *urs(TRN, CND, [AL], ST1 [,ST2])*

項関係 *TRN* から条件*CND* を満足するタプルを選び *[AL]* で示される属性を項関係（ストリーム）として出力する．詳しくは定義 3-9 参照．なお，*RRN2* もしくは *ST2* が示される場合には，条件を満足しないタプルをそちらに出力する．また，定義 3-10 の変数制約は，単一化制約の中の条件で吸収し，別命令とはしていない．

c) 射影 (Projection)

項関係型： *prr(TRN, AL, RRN1, SI)*

ストリーム型： *prs(TRN, AL, ST1)*

項関係 *TRN* から *AL* で示される属性を項関係（ストリーム）として出力する．詳しくは定義 3-11 参照．

c) 集合和 (Union)

項関係型： *unr(TRN1, TRN2, RRN1, SI)*

ストリーム型： *uns(TRN1, TRN2, ST1)*

項関係 *TRN1* と項関係 *TRN2* の集合和を項関係（ストリーム）として出力する．詳しくは定義 3-12 参照．

2) 更新系コマンド

a) 挿入／追加 (Insert)

ins(TRN, Tuple [, TID], SI)

項関係 *TRN* にタプル *Tuple* を挿入する．*TID* が指定されている場合には，その直後に挿入し，省略されている場合には，先頭に挿入する．

b) 削除 (Delete)

del(TRN, TID, SI)

項関係 *TRN* から *TID* で示されるタプルを削除する．

c) 変更 (Change)

chg(TRN, TID, AN, Term, SI)

項関係 *TRN* の *TID* で示されるタプルの属性 *AN* の内容を *Term* に変更する．

3) 定義系コマンド

a) 作成 (Create)

crt(RN, AC [,KAN], SI)

属性数 *AC* の新しい項関係 *RN*（のディクショナリ）を作成する． *KAN* が指定されている場合には，インデックス属性の設定を行う．

b) 消去 (Erase)

ers(RN, SI)

項関係 *RN* を消去する．

c) インデックス作成 (Make_index)

mki(RN, KAN, SI)

項関係 *RN* の *KAN* で示される属性にインデックスを作成する．

d) インデックス除去 (Remove_index)

rmi(RN, KAN, SI)

項関係 *RN* の *KAN* で示される属性のインデックスを除去する．

4) その他のコマンド

a) タプル数集計 (Count)

cnt(TRN, TC)

項関係 *TRN* に含まれるタプルの数を数える．エラーがあった場合には，タプル数が負となって返される．

4．3　専用ハードウェアを用いた構成

4．3．1　全体構成

RBU 演算をサポートするハードウェアとして，図4-1に示すような構成の知識ベース指向処理マシンを提案する[Yokota-Itoh 86]．図中の CP はコントロールプロセッサ，MM はコントロールプロセッサ用の主メモリ，IOP はI/O インタフェース用プロセッサ，UEi は単一化エンジン，DSi はディスクシステムを示す．ここで，ディスクシステムは，ディスクユニットとディスクコントローラからなるものとする．

二次記憶に格納された項関係に対し，高速に RBU 演算を行うために，ディスクシステムから単一化エンジンに項の集合をストリームとして流すことを前提としている．また，複数の単一化エンジンを使って，並列に検索を行うことを考える．図中で m 台の単一化エンジンと n 台のディスクシステムが結合されたのマルチポートページメモリは，複数の処理装置からページ単位で同時にアクセスしても競合が起こらないように設計された共有メモリである[Tanaka 84]．本システムでは，ディスクキャッシュとして，あるいは単一化エンジン用のバッファメモリとして多ポートページメモリを用い，ディスクと単一化エンジンとの間のパイプライニングを実現する．つまり，ディスクシステム上に置かれたデータは，ページ単位で順次多ポートページメモリ上に移され，そのデータをストリームとして単一化エンジンが読み取り，RBU 演算の結果をまた多ポートメモリ上に書き出す．そして，結果がページ単位にまとまったところでディスクシステムに結果が格納されることになる．

コントロールプロセッサは，上述したディスクシステムと単一化エンジン間のデータフローを制御し，それらのコンポーネントの並列実行環境を管理する．コントロールプロセッサから各コンポーネントに対する制御コマンドやそれらのコンポーネントからのレスポンスは，多ポートページメモリを介しても送受可能であるが，多ポートページメ

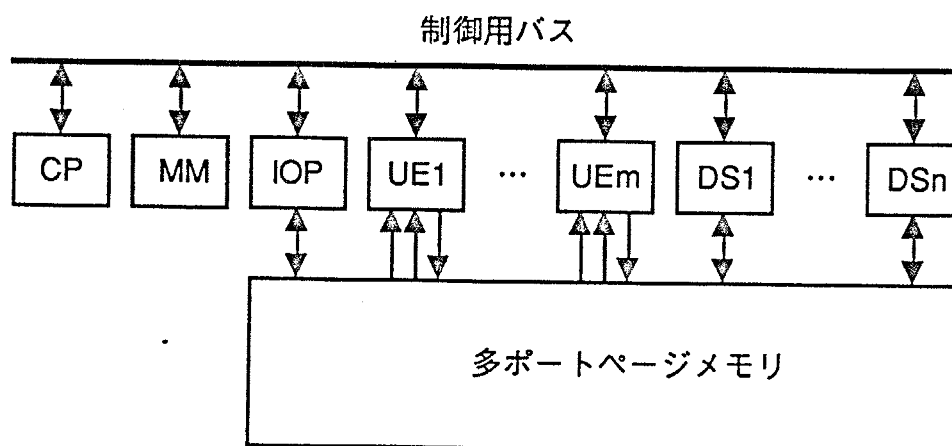

図4-1 ハードウェア構成案

モリがページ単位のアクセスを前提として設計されているため，少量の情報を高速に送受するために制御用バスを利用することにする．

4．3．2　単一化エンジン

単一化エンジンの構成を図4-2に示す．単一化エンジンは，単一化結合を前提として構成するため，マルチポートページメモリとの間で，同時に２つの入力ストリームと１つの出力ストリームを処理できるよう，３本のデータパスを持つ．単一化結合での単一化の項の組み合わせは，本来２入力ストリーム中の項の数の積になる．項の間にある順序付けを行うことにより，この組み合わせの数を減らすことができる[Morita 86]．可変長ソータとペアジェネレータはそのためのユニットである[横田他 86a].

項の間の順序付けを行うために，ジェネラリティの概念を導入する[Yokota-Itoh 86]．T_1 と T_2 を項とすると，

$$T_1 \leq T_2 \quad \Leftrightarrow \quad \exists\theta,\ T_2\theta = T_1$$

のように順序付けを行う．このジェネラリティによる順序付けは半順序になっているため，次のような規則を適用して全順序にする．ただし，関数名どうしの比較は辞書順序に従うものとし，S^n はn組の項 $s_1, s_2, \ldots s_n$ を示すものとする．

a) $f(S) < g(T) \quad \Leftrightarrow \quad f < g$

b) $f(S^n) < f(T^m) \quad \Leftrightarrow \quad n < m$

c) $f(S^n) < f(T^n) \quad \Leftrightarrow \quad (s_1, \ldots, s_{i-1}) \approx (t_1, \ldots, t_{i-1}),\ s_i < t_i\ (1 \leq i \leq n)$

d) $f(S) < X \quad \Leftrightarrow \quad X$ が変数

e) $X < Y \quad \Leftrightarrow \quad X, Y$ が変数で X が先に出現

f) $X \approx Y \quad \Leftrightarrow \quad X, Y$ が変数で対応した場所に出現

g) $f(S^n) \approx f(T^n) \quad \Leftrightarrow \quad (s_1, \ldots, s_n) \approx (t_1, \ldots, t_n)$

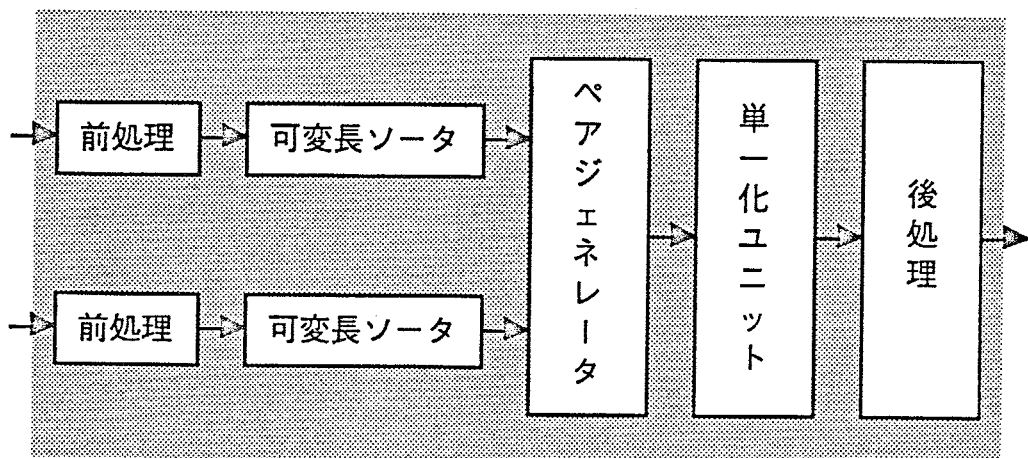

図4-2 単一化エンジンの構成

単一化ユニットは，ペアジェネレータにより生成された２つの項の組の間の最汎単一化作用素を求める．また，前処理ユニットは項関係のタプルから処理対象のアイテムを切り出し，後処理ユニットは単一化ユニットで得られたさい汎単一化作用素をもとのタプルに適用するためのユニットである．

一般的な Prolog 処理系などでの単一化アルゴリズムは，構造体共有方式にしろ構造体コピー方式にしろ，変数毎にバインドすべき相手をポインタでつなげて行くことを基本としている．その様なポインタをたどる操作は，ストリーム処理を基本とする専用ハードウェアには向いていない．もともと Robinson によって提案された単一化アルゴリズムは以下のようなものである[Robinson 65].

【単一化アルゴリズム】

Step1: $k=0$, $W_k=W$, $\sigma_k=\varepsilon$

　　ここで W は単一化の対象となる項の集合

Step2: W_k が１つの項のみからなる場合には単一化終了（σ_kが最汎単一化作用素）

　　それ以外の場合には W_k の中の食い違い集合 D_k を求める．

Step3: D_k の中の要素 v_k と t_k について，

　　　　v_k が t_k の中に含まれない変数の場合にはStep4 へ

　　　　それ以外の場合には単一化不可能

Step4: $\sigma_{k+1}=\sigma_k \bullet \{t_k/v_k\}$, $W_{k+1}=W_k \bullet \{t_k/v_k\}$とする

Step5: $k=k+1$ としてStep2 へ

このアルゴリズムの繰り返しの部分(Step2→Step4)をハードウェア化し（単一化エレメントと呼ぶ），それを直列に接続して（図4-3），複数の項の組み合わせを流すことにより，単一化処理をパイプラインで行うことができる．図の左側から項のペアと最汎単一化作用素の初期値（空）を流すことにより，最終的に単一化が行われた項とその時の最汎単一化作用素が得られる．図の制御情報は単一化が不可能だった場合や単一化が終了

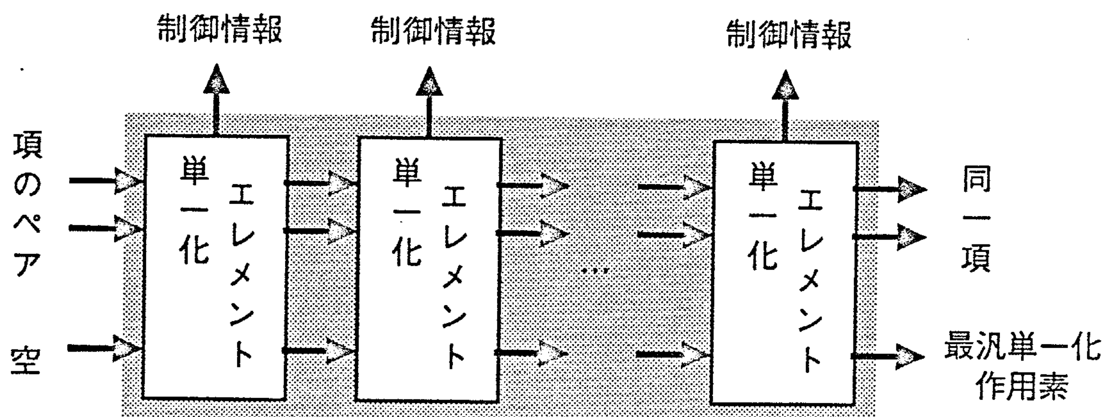

図4-3　単一化アルゴリズムのハードウェア化

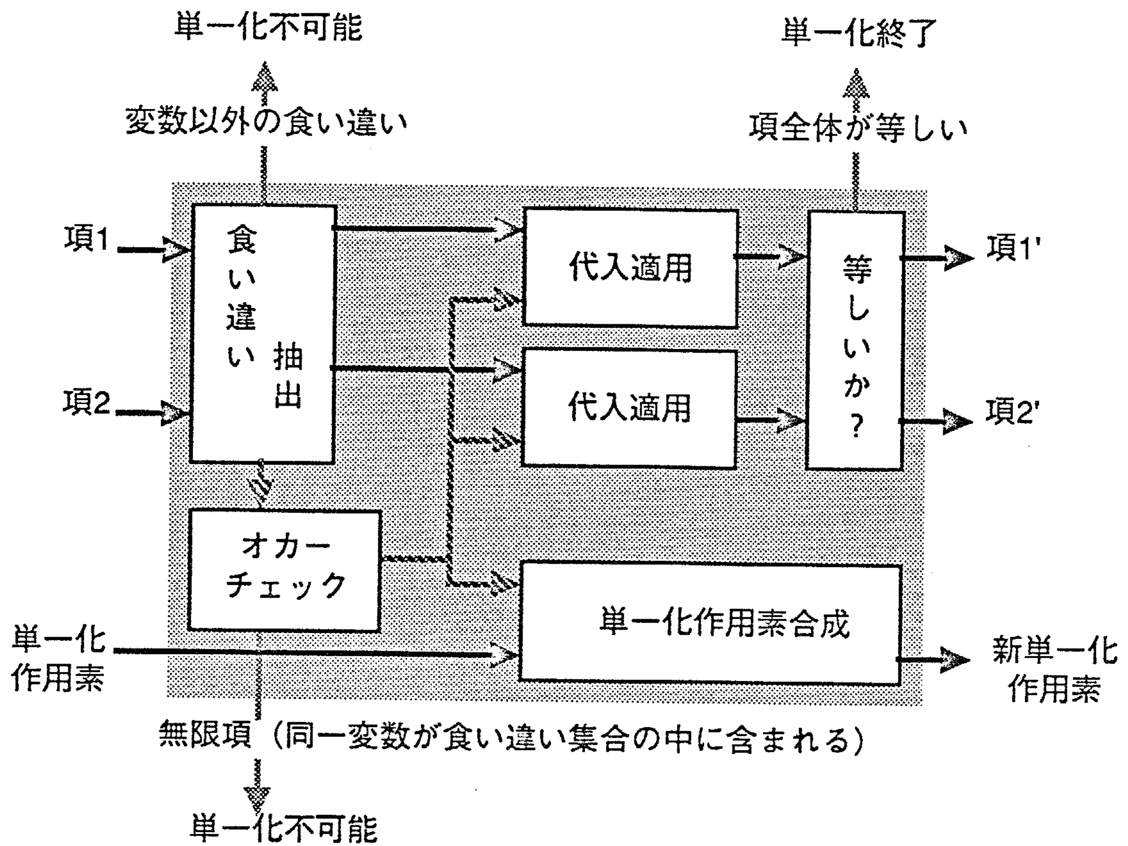

図4-4　単一化エレメントの内部構成

した場合の情報が流れる．この各エレメントの内部構成を示したものが図4-4である．これは，前出の単一化アルゴリズムをそのままハードウェア化した形になっている．Stepに対応したブロック構成で，単一化エレメント内もパイプライン化されている．

実際に単一化ユニットを構成する場合，項の中の変数の数に制限がないと，無限この単一化エレメントをつなげる必要が出て来る．これは実質的に不可能なため，図 4-5 のように切り替えスイッチを使用したり，図 4-6 のようにスイッチング・ネットワークなどを使って仮想的に無限長の接続を実現する．なお，図 4-5 のように切り替えスイッチを用いた場合には，単一化が終了した後もループ内の単一化エレメントを通過することにな

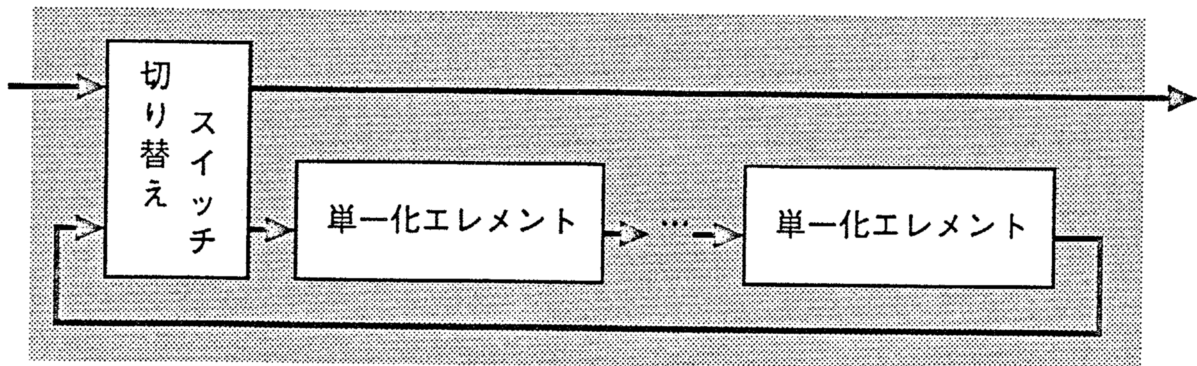

図4-5　単一化ユニットの構成法(a)

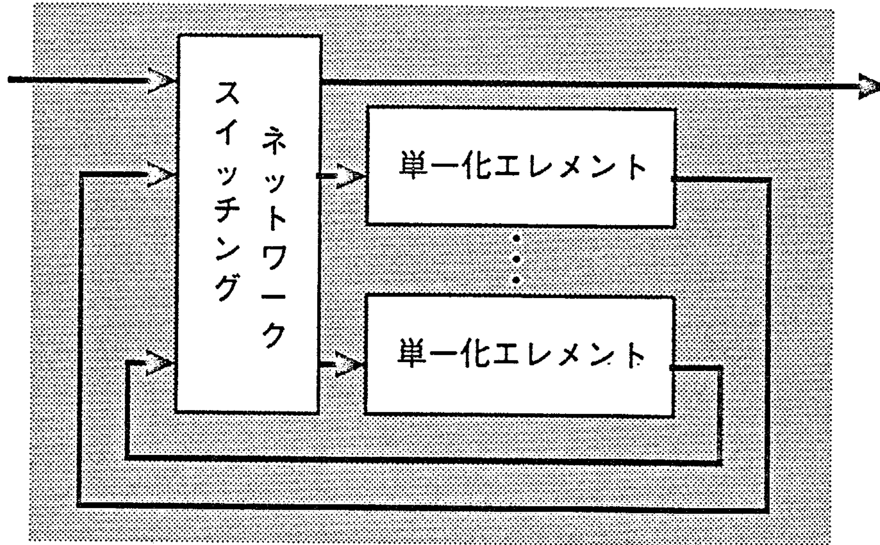

図4-6　単一化ユニットの構成法(b)

る．単一化エレメントの構成が図 4-4 に示したようになっている場合には，食い違いがなくなるだけであり，空の食い違いを代入として適用しても，単に通過するだけであるから，特別な仕組はいらないことになる．

4．3．3　多ポートページメモリ

Tanaka によって提案された [Tanaka 84] 多ポートページメモリは図4-7 に示すように，複数の I/O ポートとメモリバンクとそれらを結ぶスイッチングネットワーク及びその結合を制御するコントローラからなる．各ポートとメモリバンクとの間は，システム全体で同期しながらある周期で１つのポートがすべてのメモリバンクと接続され，システム全体としてこの接続関係が常に成立するように（たとえばある周期で接続関係を回転させ

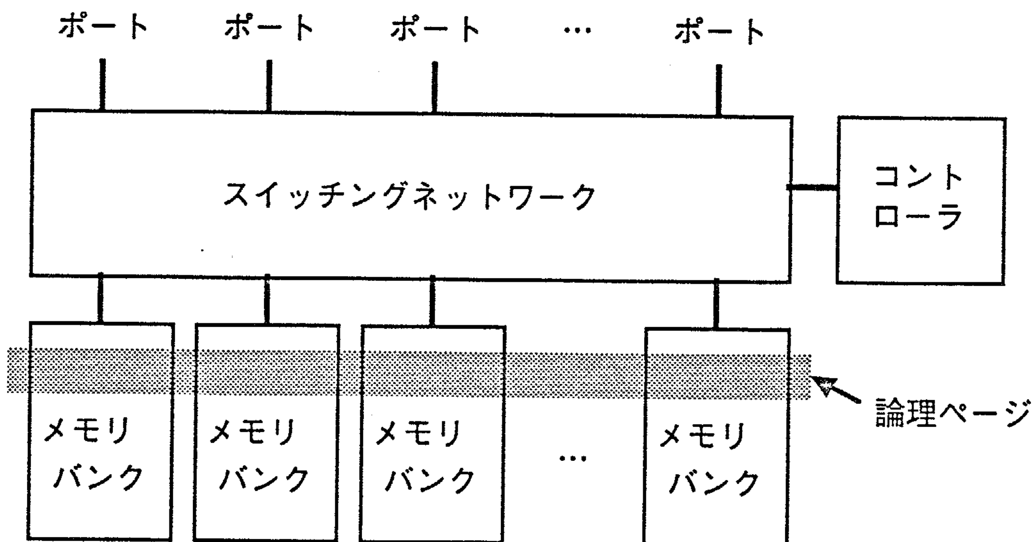

図4-7　多ポートページメモリの構成

るように）する．1つの論理ページをすべてのメモリバンクにまたがって置き，メモリバンクに置かれたデータへのアクセスの単位をページとすることにより，各ポートから競合を起こすことなく同時にアクセスが可能となる．すなわち，論理ページをメモリバンク分に分割して，時分割で各ポートからアクセスすることにより，ページ単位で見ると競合なしで同時にアクセスできることになる．

このような多ポートページメモリを実際に実現する場合には，各ポート及びメモリバンクのバス幅が，全体（特にスイッチングネットワーク）を構成するコストに大きく影響してくる．データバスとアドレスバスを別々に構成すると，それだけコストが増すため，データバスを使ってアドレスを転送する方法を提案する．ただし，頻繁にアドレスを転送していたのでは効率が悪いため，アドレス転送を減らす工夫をする[Yokota-Itoh 86].

基本的には，メモリバンク側でローカルにアドレス計算を行う．1回ページアドレスを設定したら，ページ内のアドレス計算はメモリバンク側とポートで同期を取って行う．また，メモリバンクを切り替える度にデータバスを使ってページアドレスを送ることも重いため，メモリバンク間にアドレス転送用のパスを設定し，スイッチングネットワークの切り替えに同期してメモリバンク間でページアドレスを転送しあうようにする．これは，隣接するメモリバンクどうしを接続し，スイッチングネットワークの接続関係をある方向に回転させることにより実現できる．なお，最左端のメモリバンクと最右端のメモリバンクもこのアドレス転送用パスで接続する．

さらに，ラッチを使ってアドレスを保持しておくことにより，データの転送とアドレスと転送及び設定を同時に行うことができる．すなわち，メモリバンクでそのメモリバンクに割り当てられたデータをネットワークを介してポートに転送する間ページアドレスを保持し，データを転送している間に隣のメモリバンクにそのページアドレスを転送して，スイッチングネットワークの切り替えと同時に隣から転送されたページアドレスに切り替えることにより，メモリバンク間のアドレスの転送時間を見えなくすることができる．

ここで提案する RBU 演算専用ハードウェアのように，ディスクに格納されたデータ（知識）を一時に大量に操作するような処理におけるメモリシステムとして，多ポートページメモリは非常に有効である．特に，複数の検索用プロセッサ（単一化エンジン）によって同一データ（項関係）を同時にアクセスするような場合には，ページ単位で競合なく同時にアクセスできることは大きな特長であろう．

4．4　インデックスによる検索高速化

演繹処理やプロダクション・システムなどの知識ベースを利用する処理においては，検索時間が全体の処理効率に大きく影響してくる．項に対する高速検索の手段として，4．3で示した専用ハードウェアを使った方法とは別に，ここではソフトウェアのみの実現を前提として，インデクスによる高速化[Yokota et al. 89] について検討する．

Prolog システムでは，コンパイルコードのインデクスとしてハッシングを使っているものがある．また，単一化を使った結合演算のためにハッシュ・ベクタを使った方法の提案もされている[Ohmori-Tanaka 87]．ハッシングは，ハッシュ・キーの対象となる内容が重ならないような場合に，大量の項目を検索するためのインデクスとして非常に強力である．しかし，ハッシュ・キーの内容が重なった場合には，ハッシュ・エントリの競合が生じインデクスの効果が薄らぐ．RBU では比較的似た構造の項に対して単一化可能なものを探すため，適当なハッシュ・キーを選ぶことが難しい．また，単一化を使った検索では，変数の束縛 (binding) をしなおすためのバックトラックが必要であり，単純なハッシングだけでは次に束縛すべき内容をすぐに見つけることができない．さらに，インデクスを設けることにより，更新処理においてインデクスを維持するための処理が必要になる．このインデクス維持のオーバヘッドも考慮する必要がある．

以下では，RBU 演算を高速化するために，ハッシングとトライ (Trie) 構造と呼ばれる一種の木構造を使ったインデクスを提案し，そのインデクスを使って試作したプロトタイプの評価を行う．

4．4．1　項の表現方法

項を格納，検索するためには，なんらかの方法で項を表現する必要がある．項を木構造，単一化をその木の上のパターンマッチングと見なすことができる．つまり，変数に木中の対応する位置にある部分木が代入される．図 4-8 に項 p(f(a,b),h(X)) に対応する木

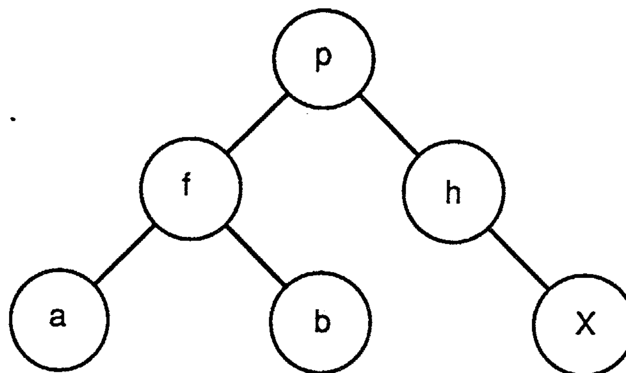

図 4-8　項 p(f(a,b),h(X)) の木構造

構造を示す．この項が，p(Y,Z) と単一化されると，変数 Y に f を根とする部分木が，変数 Z に h を根とする部分木が代入される．

木構造の表現方法として，レベル順線形化表現 (LOSR: Level Order Sequential Representation) とファミリィ順線形化表現 (FOSR: Family Order Sequential Representation) がある[Knuth 73a]．図4-8 の木構造に対する LOSR は [p-2, f-2, h-1, a-0, b-0, X] であり，FOSR は [p-2, f-2, a-0, b-0, h-1, X]である．それぞれの表現の要素は，元の木構造を再現するために，その関数記号と引数の数（つまり，そのノードにつながる枝の数）の組で表記する．

一般の Prolog のシステムは，FOSR を使っている．これは，FOSR では関数記号の引数がその関数記号に再帰的に続くため，繰り返し起こる Prolog の複雑な代入に向いているためである．一方，項の構造的な違いを判定するのために要素を先頭から比較していく場合には，FOSR は LOSR に比べて多くの比較を必要とする．例えば図 4-8 の p の木が f と h の 2 つの部分木からなることは，LOSR だと f の部分木の大きさに関係なく 3 要素の比較で判定可能であるのに対し，FOSR では f の部分木全部を比較する必要がある．RBU が，単一化を何回も繰り返すことよりも，似た構造の多くの項の中から必要な項を早く探すことを目的としているため，RBU のプロトタイプでは LOSR を用いることにする．

項の長さが可変であるため，我々は LOSR の要素を格納するために図 4-9 のようなセルを使うことにする．1 番目と 2 番目のフィールドは，対応する要素の内容を表すためのものである．その要素が関数記号の場合には，第 1 フィールドに関数記号表 (Atom Table) の対応するエントリへのポインタが，第 2 フィールドにその引数の数が入れられる．ここで，関数記号も可変長の文字ストリングであり，同一の関数記号が項関係中に何回も現われるため，直接セルに格納せずに別の関数記号表を用意している．要素が変数の場合には，第 1 フィールドには null ポインタを入れ，第 2 フィールドにその変数の番号を格納する．変数の記号は，単一化の間に付け替えられるため重要ではなく，変数のスコープに従って項タプル中で順番に番号付けを行う．この時，同一の変数には同一の番号を振る．3 番目のフィールドは後で述べる トライ構造を構成するためのもので，4 番目のフィールドには次のセルへのポインタを格納する．最後のセルで次のセルがない場合には，null ポインタを入れておく．

Atom table entry or null	Arity or variable number	Alternate cell	Next cell

図 4-9　セル構造

表4-1　項関係の例

第1属性	第2属性
p(X,g(Y))	r(X,Y)
q(f(a,X),g(X))	r(f(a,X),X)
p(X,g(b))	r(h(a,b),f(a))
q(f(X,Y),g(c))	s(X,g(Y,c))
p(f(a,b),h(X))	s(a,g(b,c))
p(f(a,X),h(X))	s(a,X)

図4-10は，表4-1に示した項関係の第1属性に対するLOSRのセル構造である．ここでは，見易さのため関数記号表へのポインタは省略して関数記号と引数の数の組で示し，変数は V と変数番号をしめす添字で表現している．また，第3フィールドは省略して，nullポインタは斜線で示している．

我々は，このセルを項を格納するのとインデクスを構成するために利用する．このため，図4-10に示したようなセル構造を単純セルリスト(Flat Cell List)と呼ぶことにする．

4．4．2　ハッシングとトライ構造

ハッシングは，高速検索のためによく用いられるインデクス手法の一つである[Knuth 73a]．しかし，知識ベース処理のための項関係は多くの似た構造が格納されることが想定される[Yokota et al. 88]ため，ハッシングだけではその効果が期待できない．またRBUではバックトラックを使って，単一化可能な項をすべて検索することを目的としているため，単一化可能な項をインデクスのなるべく近い位置に置くことが望ましい．そこで，

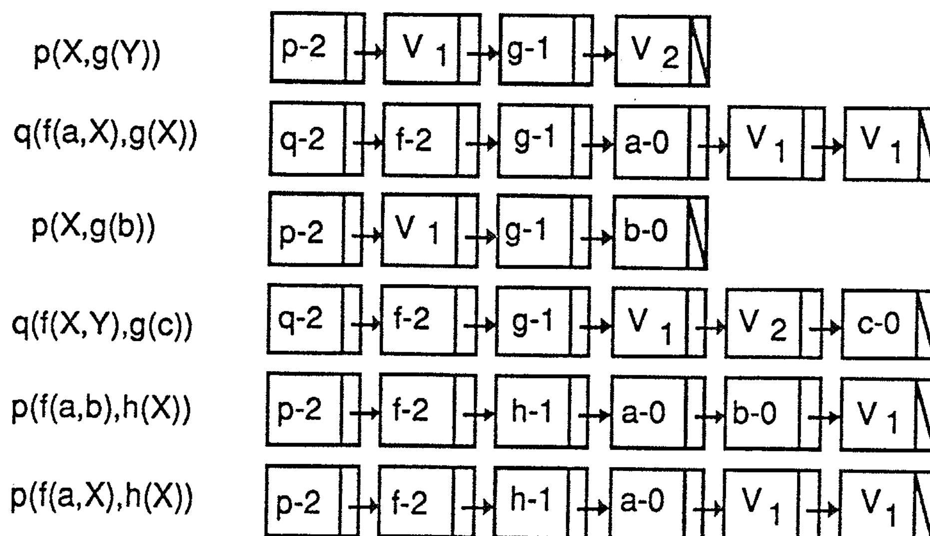

図4-10　LOSRに対するセル構造

我々はハッシングとトライ構造を組み合わせたインデクスを提案する．

トライ(Trie)構造とは，頭から見て同じ内容の要素をまとめた一種の木構造である[knuth 73a]．トライ構造は，一般には数値や文字の列を格納して，記憶容量を節約するのためにに使われるが，ここでは項のLOSRの要素に適用してインデクスの一部として利用する．

LOSRの先頭の要素の等しい項を1つのトライ構造とし，頭から等しい部分は要素を共有する．このとき，前述したセルの第3フィールドを分岐に利用する．図4-5に示した項の集合をトライ構造に変換すると図4-11に示すp-2とq-2を根とする2つのトライ構造になる．例えば，p(X,g(Y))とp(X,g(b))のLOSRの先頭の3要素が等しいため3つのセルを共有し，4番目のセルに分岐を持たせる．こうすることにより，似通った構造の項は，トライ構造上で近くに置かれることになる．

単一化制約のコストは，検索条件の項の要素と検索対象の項の要素の間の比較の回数に比例する．トライ構造上で，単一化を行うことにより比較の回数を大幅に減らすことができる．図4-10と図4-11の項の集合に対して，p(f(a,b),h(c))という検索条件で単一化制約を実行する場合を想定する．比較されるセルは，次に示すトレースのようになる．

【単純セルリストの場合（図4-10）】

p-2, V1 (= f-2) , g-1 <fail>

q-2 <fail>

p-2, V1 (= f-2) , g-1 <fail>

q-2 <fail>

p-2, f-2, h-1, a-0, b-0, V1 (= c-0) <success>

p-2, f-2, h-1, a-0, V1 (=b-0), V1 <fail>

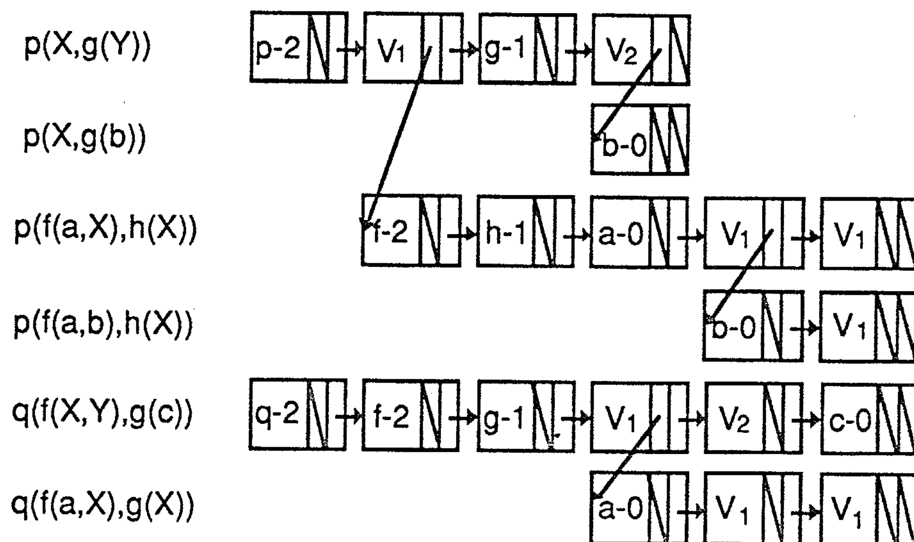

図4-11　項に対するトライ構造

【トライ構造の場合（図 4-11）】

p-2, V1 (=f-2) , g-1 <fail>

f-2, h-1, a-0, V1 (=b-0), V1 <fail>

b-0, V1 (=c-0) <success>

q-2 <fail>

このトレースから分かるように，単純セルリストだと最初の要素の比較が項の数分（6回）行われるのに，トライ構造を使うと先頭の要素の種類分（2回）で済む．当然この違いは同じ要素を先頭に持つ項の比率によってさらに大きくなる．同じことは2番目以降の要素についても言える．1つの項の平均要素数を M, 項の数を N とすると，単純セルリストを使った場合の比較の回数は最悪で $M \times N$ になる．これは，検索対象のすべての項が，最後の要素を除いてすべて等しい場合と考えることができる．この場合，トライ構造を用いることにより $M+N$ に減らすことができる．

トライ構造は単に要素間の比較の回数を減らすだけでなく，バックトラッキングの回数も減らしていることに注意しなければならない．上のトレースにおいて，行の最後の <fail>とか <success> がバックトラックに対応する．この例では，6回から4回に減っている．トライ構造がうまく均衡が取れている場合には，バックトラックの数が大幅に減少する．比較の回数もこの効果によって $M+N$ からほとんど M 近くまで減らされる．上の例では，全体の比較の回数はトライ構造を使った場合と使わない場合で，２０から１１に変化している．

我々は，似た構造の項と異なった構造の項のいずれの形態にも効果のあるインデクスを実現するために，トライ構造とハッシングを組み合わせて使うことにした．LOSR の先頭の要素をハッシュ・キーとして，その要素を持つトライ構造へのポインタをハッシュ・テーブルに格納する（図 4-12）．ハッシュ関数としては，関数記号表のエントリのビット列をその表の大きさに対応させて適当な長さに畳み込んで排他的論理和を取ったものを使うことにした．こうすることにより，似た名前の関数記号が多い場合でも効率的にちらすことができる．

このようにハッシングと組み合わせたトライ構造は，見方を変えると，ハッシングの競合解消を効率的に行う機構と見ることもできる．トライ構造の葉に相当する部分のセルの次のセルを指すためのポインタは，対応する項をキー属性に持つ項タプルへのポインタとなる．基本的には，ハッシングは1つの検索条件に対して1回実行されるだけである．検索条件に単一化可能な項は，少なくとも先頭の要素が条件と同じはずであるから，そのエントリに接続されたトライ構造の中に含まれるはずである．つまり，ハッシングは探索空間を狭めるのに用いられ，その探索空間上で効率的に単一化を行うために

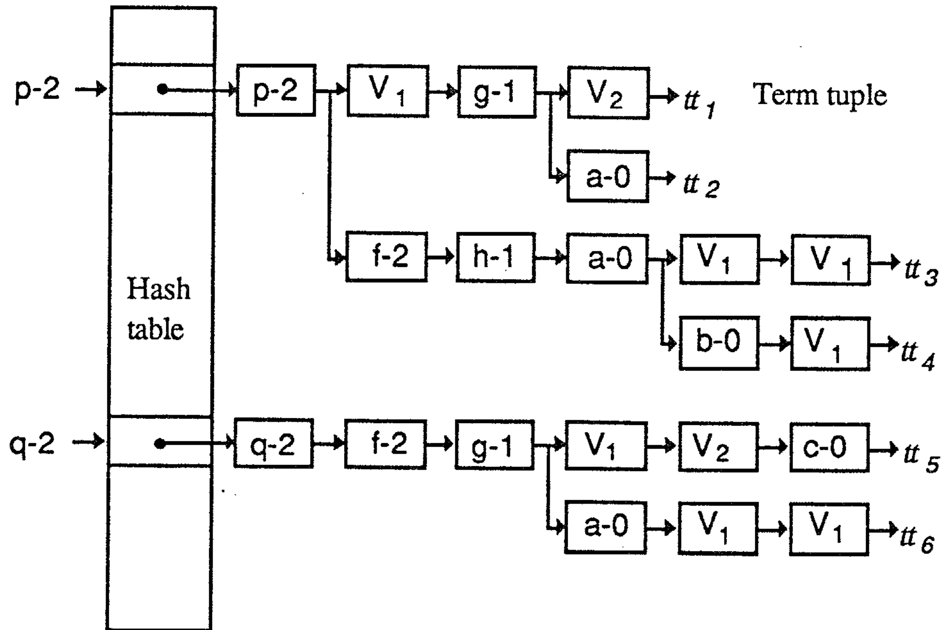

図 4-12　ハッシングとトライ構造を組み合わせたインデックス

トライ構造を使うわけである.

4．4．3　LOSR上の単一化

Prolog システム等で使われている単一化のアルゴリズムは，項が FOSR で表現されていることを前提としているため，RBU システムのためには LOSR 用の単一化アルゴリズムを用意する必要がある.

2 つの項 p(X,g(Y)) と p(f(a,b),g(c)) は，変数 X に f(a,b) が代入可能であり，変数 Y に c が代入可能であるため，単一化可能である. ここで，それぞれの項の LOSR は，[p-2, V1, g-1, V2] と [p-2, f-2, g-1, a-0, b-0, c-0] となる. LOSR どうしで単一化を行うためには，1 番目の変数 V1 (= X) には，単に対応する位置にある f-2 だけでなく，その引数である [a-0, b-0] も代入される必要がある. また，2 番目の変数 V2 (=Y) には，そのままでは対応する位置にない要素 c-0 を代入しなければならない. これらのためには，[p-2, V1, g-1, V'1, V"1, V2] という仮想的な LOSR を作る必要がある. 我々は，FIFO(First-In-First-Out list) を使ってこの変数拡張を行うことにする. つまり，単一化の時に，拡張すべき変数かそれとも拡張すべきでない要素かを示すためのフラグを，その要素の引数の分だけ FIFO に入れていくようにする. 以下にその FIFO を用いた単一化アルゴリズムを示す.

【LOSR 上の単一化アルゴリズム】

Step1:L_A と L_B を単一化の対象の項の LOSR とする. 初期設定として，非拡張フラグ n を 2 つの FIFO W_A と W_B の先頭に入れる.

Step2: <u>if</u> W_A と W_B の両方が空 <u>then</u> 単一化成功

<u>else</u> W_A と W_Bの先頭からフラグA と B を取る.

Step3: <u>case</u> A も B も n の場合

L_A と L_B の先頭から要素 E_A と E_B を取る

<u>if</u> $E_A = F_A - K_A$ かつ $E_B = F_B - K_B$ <u>then</u>

<u>if</u> $F_A = F_B$ かつ $K_A = K_B$ <u>then</u>

K 個のn を W_A と W_Bに入れ, Step2 へ

<u>else</u> 単一化失敗

<u>else if</u> $E_{A/B} = F\text{-}K$ かつ$E_{B/A}$が変数V_i <u>then</u>

変数 V_i に $F\text{-}K$ を代入し, $W_{A/B}$ にK 個の V_i を変数拡張フラグとして入れ,

$W_{B/A}$ にK 個の n を入れ, Step2 へ

<u>else if</u> E_A と E_B 両方とも変数 <u>then</u>

変数どうしで代入して, Step2 へ

<u>case</u> A/B が n かつ B/A が V_i の場合

$L_{A/B}$ の先頭から要素 $E_{A/B}$ を取る

<u>if</u> $E_{A/B} = F\text{-}K$ <u>then</u> 変数 V_i に $F\text{-}K$ を代入し,

$W_{A/B}$ にK 個の n を入れ, $W_{B/A}$ に K 個の V_i を入れ, Step2 へ

<u>else if</u> $E_{A/B}$ が $F\text{-}K$ を代入された変数 V_j <u>then</u>

変数 V_jに変数 V_i を代入し, $W_{A/B}$ に K 個の V_j を入れ,

$W_{B/A}$ にK 個のV_i を入れ, Step2 へ

<u>else if</u> $E_{A/B}$ が代入されていない変数 V_j <u>then</u>

V_j に V_i を代入し, Step2 へ

<u>case</u> A が V_i かつ B が V_j の場合

変数どうしで代入して, Step2 へ

例として[p-2,V_1, g-1, V_2] と [p-2, f-2, g-1, a-0,b-0, c-0] の間の単一化の過程を図 4-13 に示す. 初期状態では, 2 本のFIFO の先頭は非拡張フラグ nが入っている. LOSR の先頭の要素がいずれも p-2 であるため FIFO に非拡張フラグ n を 2 個ずつ追加する. 次に, 変数 V_1 と f-2 が対象となり, 2 個の変数拡張フラグ V_1 が V_1 側の FIFOに入れられる. また, V_1 の最初の要素として f-2 が入れられる. g-1 どうしの比較では, 非拡張フラグを 1 個ずつ FIFO に加える. 変数拡張フラグが FIFO の先頭に来た場合には, その変数つまりV_1 に非拡張フラグに対応する要素 a-0 と b-0 を入れる. V_2 と c-0 が対象の場合には引き数が 0 個であるため, そのまま変数に入れるだけである. FIFO が空になると終了となる. この時, 両 LOSR も最後まで行っていない場合には, 単一化が失敗したことになる.

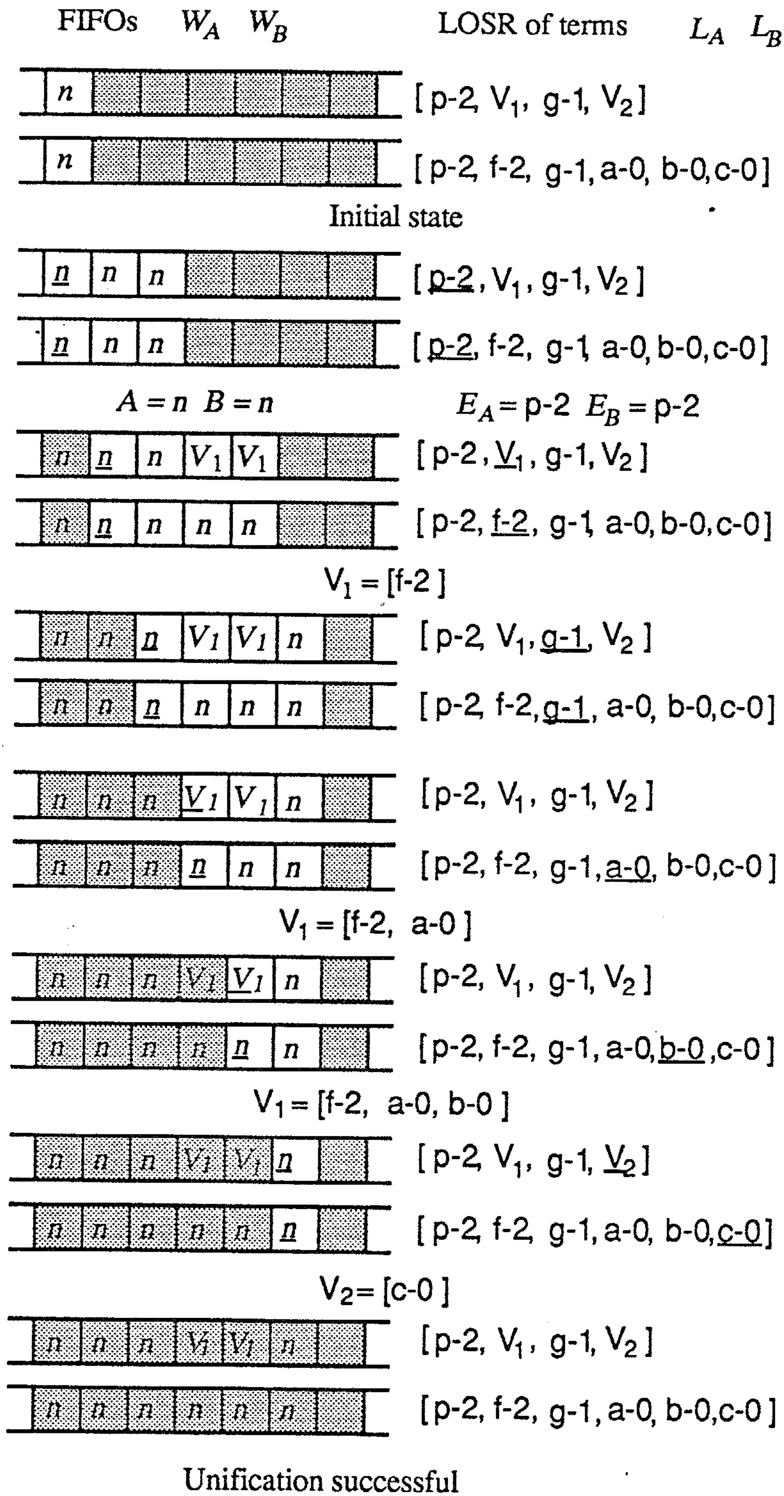

図 4-13　LOSR 上の単一化過程

4．4．4　トライ構造のたどり方

項タプルを検索するために，前述した単一化のアルゴリズムをトライ構造をたどることに利用することを考える．トライ構造の各ノードを，アルゴリズム中の L_A の要素とし，検索条件の項の LOSR を L_B とする．トライ構造をたどる場合，分岐から単一化をや

り直すことから，L_A と L_B の途中から単一化が始められるようなバックトラッキングのための機構を用意する．

各変数はレベルを持つこととし，分岐を通る度にその代入のレベルを１つずつ増やすようにする．変数代入の環境は，そのバックトラック・ポイントでの変数代入のレベルより高い代入内容を開放することにより元の状態に戻すことができる．また，図 4-13 の FIFO の中で，網かけのしていない部分が実際に作業領域として利用している部分であるが，これは先頭と最後を示すポインタによって表現することができる．最後を示すポインタが過ぎた後でも FIFO に格納された内容を残しておくことにより，そのポインタを戻すだけでFIFO の内容をバックトラックのポイントまで戻すことができる．結局，分岐のある要素 E_A が比較される場合，次のようなバックトラックのための情報をスタックに積む必要がある．

・E_A の分岐先の要素のポインタ

・その時の検索条件の要素 E_B のポインタ

・変数代入のレベル

・W_Aと W_B の先頭と最後を示すポインタ

バックトラックが起こると，そのスタックの先頭の情報を取り出して，FIFO と変数代入の状況を分岐以前の状態まで戻してやり，スタックに積まれていた分岐先のポインタと検索条件のポインタで示される要素の比較から単一化をしなおす．一方，トライ構造の葉の部分で，単一化が成功した場合には，その葉に対応する項タプルへのポインタとその時の変数代入の状況を結果として保存しておき，バックトラックを起こす．

このように，インデクスをたどっただけで，変数の代入内容まで求められるのは，この方法のもう一つの長所である．

4．4．5　インデックスの効果

C で書いた RBU のプロトタイプを使って，いくつかのタイプの項関係に対して，検索時間と挿入時間を測定し，ここで述べたインデクスの評価を行った．また，必ずしも機能は等しくはないが，比較対象として Quintus-Prolog のインタプリタについても同様の測定を行った．

次に挙げるような特徴を持つ４つのタイプの項関係を用意した．それぞれのタイプに対応するインデクスの形態を図 4-14 に示す．

タイプ *A* ：すべての項が，最後の要素を除いてすべて等しい関数記号を持つ．インデクスは，たった１つのハッシュ・エントリに分岐が葉の部分にしかないような片寄った形の１つのトライ構造が接続される．

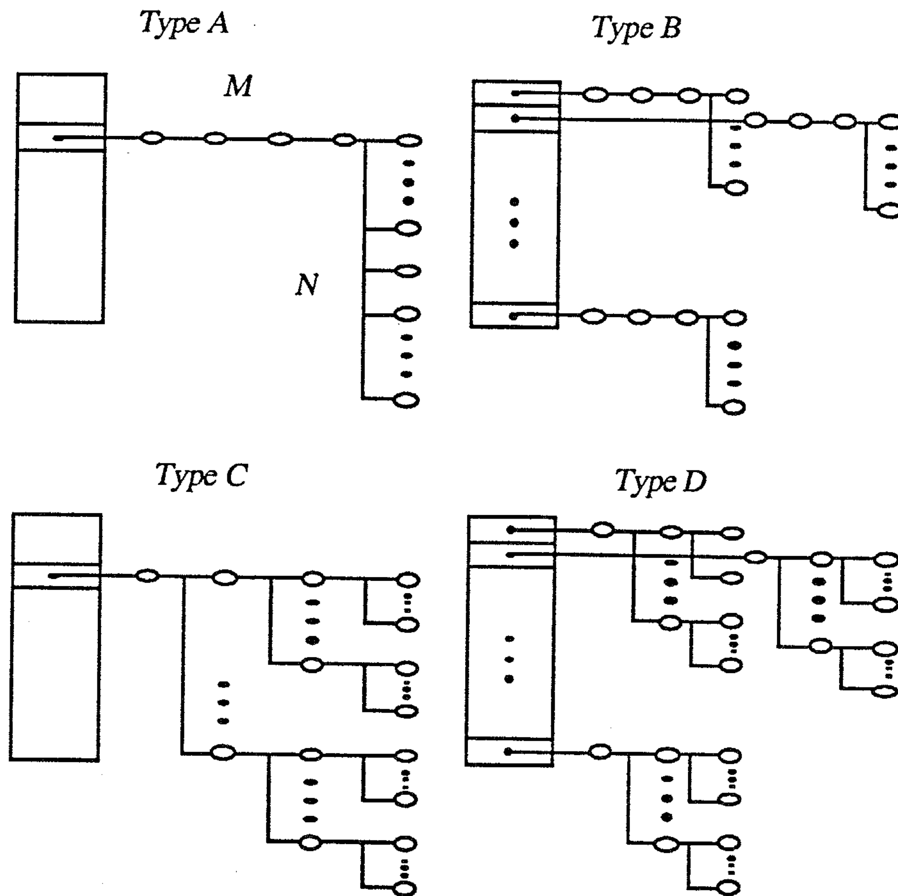

図 4-14　各タイプに対するインデックスの形態

タイプ *B* ：先頭の要素が１６の関数記号のうちの１つを持ち，それ以外の要素は最後の要素を除いて等しい関数記号を持つ．インデクスは，タイプ *A* と同様な片寄った形の１６個のトライ構造がハッシュ・テーブルに接続される．この時，それぞれのトライ構造の大きさはほぼ同じとする．

タイプ *C* ：すべての項の先頭は同一の要素を持つが，それ以外の要素は再帰的に８つの関数記号のうちのどれかを持つ．インデクスは，１つのハッシュ・エントリに均整の取れた１つのトライ構造が接続される．トライ構造の各分岐は８つに分れる．

タイプ *D* ：先頭の要素が１６の関数記号のうち１つを持ち，それ以外の要素は再帰的に８つの関数記号のうちのどれかを持つ．インデクスは，タイプ *C* と同様に均整の取れた１６個のほぼ同サイズのトライ構造がハッシュ・テーブルに接続される．

これらのタイプに対して，50，250，500，750，1000個の項タプルを持つ項関係を用意した．インデクスがある場合とない場合の RBU プロトタイプの単一化制約に要する時間，および同様の節集合に対する Prolog の節検索の時間の比較結果をタイプ*A*，*B*，*C*，*D* についてそれぞれ図4-15，4-16，4-17，4-18 に示した．

図4-15は，４．４．２で述べた，比較の数が最も多くなる場合（タイプ *A*）におけるト

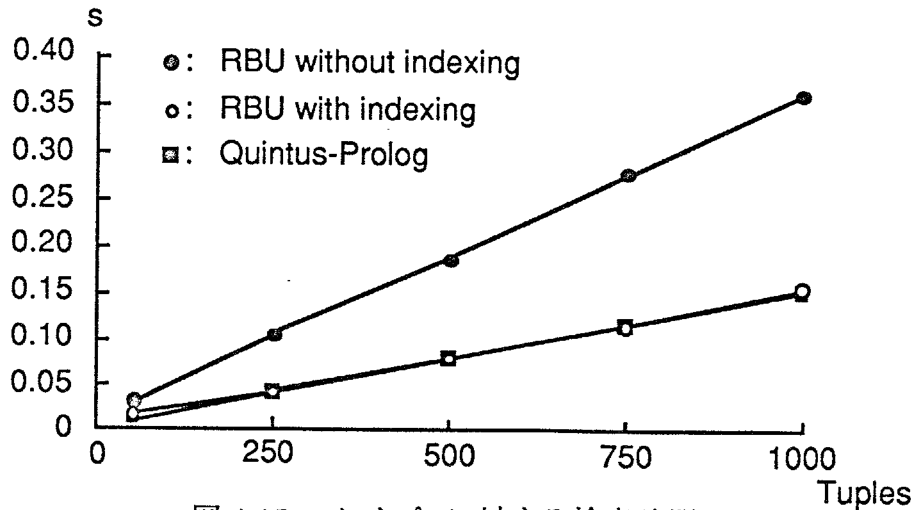

図 4-15　タイプAに対する検索時間

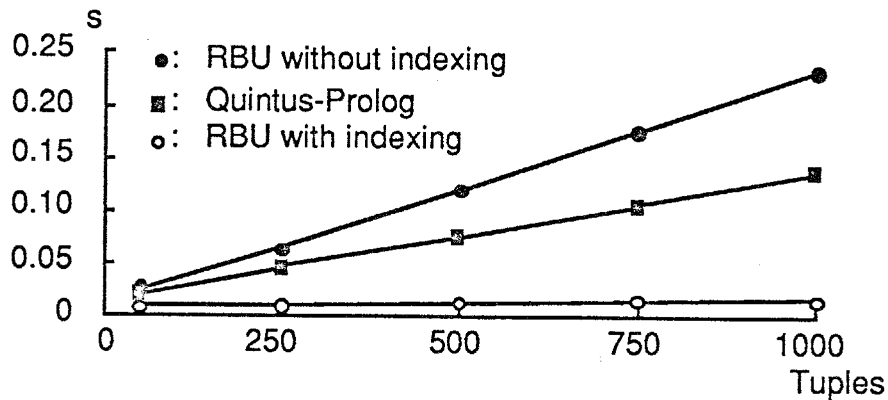

図 4-16　タイプBに対する検索時間

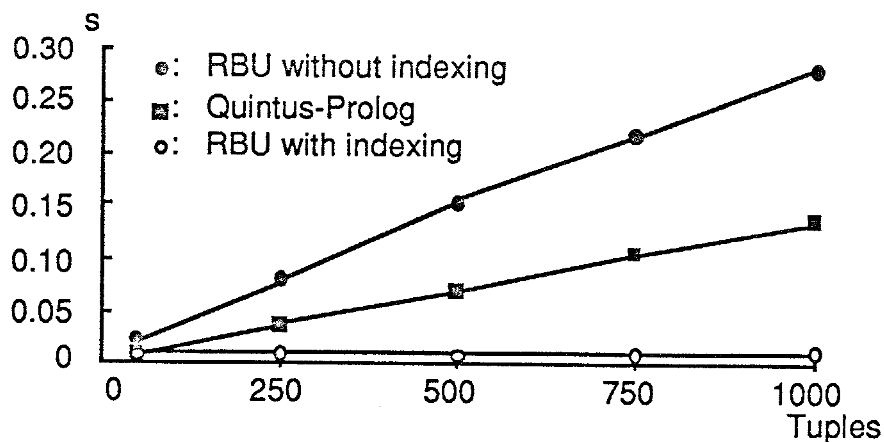

図 4-17　タイプCに対する検索時間

ライ構造の効果を示したものである．この場合には，インデクスを張ってもエントリが一つであることからハッシングの効果はない．トライ構造の葉の部分で，タプル数分 (N) のバックトラックが起こるため，インデクスの有無にかかわらず，タプル数に比例して検索時間がかかる．インデクスがある場合とない場合の差は，比較回数が$N \times M$ から $N + M$になったことによる．1つの項当たりの要素数が増えることによりこの差は広がる．

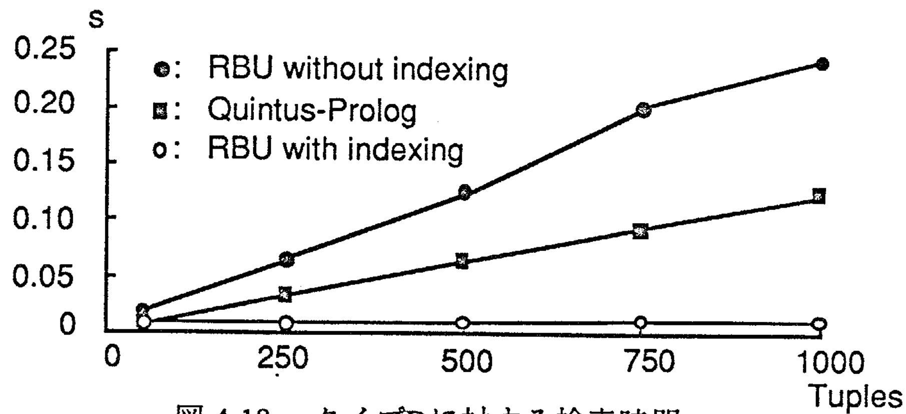

図 4-18　タイプDに対する検索時間

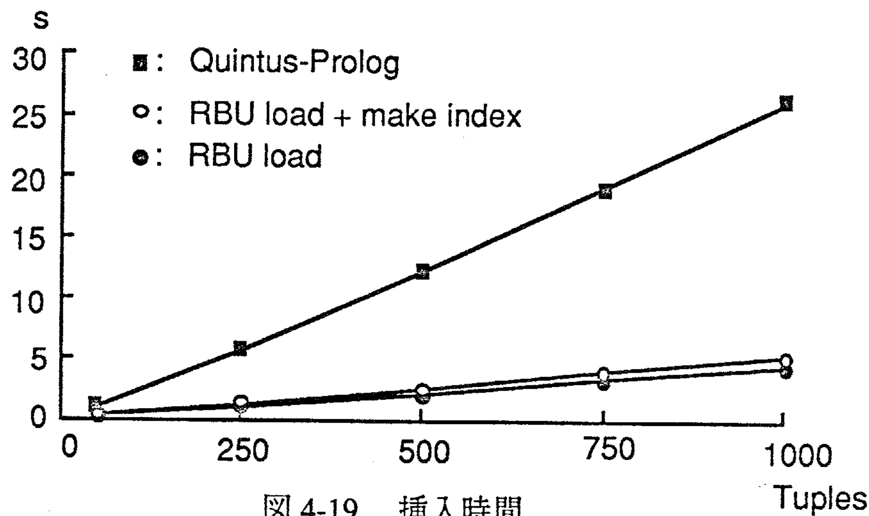

図 4-19　挿入時間

図4-16 は，タイプ B の項関係に対する同様の測定結果である．これは，ハッシングによる絞り込みの効果を示しており，タプルの数が増すことに対する検索時間の増加がほとんどないことが分かる．

タイプ C の項関係を使った測定結果を示したのが図4-17 である．この場合インデクスはトライ構造の効果だけとなるが，分岐によりバックトラックの数が大幅に削減されるため，検索時間がタプル数 (N) にまるで依存していないことが分かる．ハッシュ・テーブルの大きさにもよるが，この場合タイプ B より速くなっている．また，タイプ C にハッシングの効果を加えたのが図4-18であるが，タイプ C のみで十分に速いため，効果が余り現われていない．

この他，更新処理におけるインデクス維持のオーバヘッドを評価するための測定も行なった．本インデクスはセル構造により実現され，構造が比較的単純であるため，挿入，削除操作を高速に行なうことができる．図4-19に Prologとの挿入速度の比較と，インデクス作成にかかる時間を示した．インデクス作成が挿入の時間に対してわずかの割合でできることが分かる．

4．5　まとめと考察

RBU に基づく知識ベース指向処理システム構築のため，知識検索モジュールの構成法について述べた．まず，知識検索モジュールとして必要な命令体系を明らかにするとともに，その命令を実行するための専用ハードウェアを用いた構成，及び専用ハードウェアを仮定せずにソフトウェアで高速化をはかる方法を述べた．

専用ハードウェアを用いる方法としては，複数の単一化エンジンと多ポートページメモリを組み合わせて使う方法を提案した．単一化エンジンの中では，項に順序づけを行うことにより検索の効率化をはかるとともに，パイプライン的に単一化を行う方法を示した．多ポートページメモリでは，実現コストを下げるため，データバスを使ってアドレスを転送し，メモリバンク間でデータ転送と同時にアドレスを転送しあう方法を示した．複数の要素プロセッサと多ポートページメモリを用いた RBU による知識検索モジュールの実験ハードウェアとしては，Mu-X が試作されている[Shibayama et al. 87，伊藤他 89]．

次に，専用ハードウェアを用いないでシステムを高速化する手段として，トライ構造とハッシングを使ったインデクスを提案し，RBU 処理における要素間の比較処理とバックトラック処理が削減されることを示した．このトライ構造中で項を LOSR で表現することにより，速い段階で単一化可能な項を絞り込むことができる．LOSR 用に FIFO を使った単一化アルゴリズムを提案すると共に，トライ構造をたどるためにスタックに積むべき内容を明らかにした．さらに，トライ構造とハッシングを組み合わせることにより，いろいろなタイプの項関係に対して効果的なインデクスを供給することができることを示した．

試作した RBU のプロトタイプシステムの単一化制約に要する時間と挿入に要する時間を各種項関係に対して測定した．明らかに，トライ構造とハッシングを使ったインデクスは，検索の高速化に非常に効果があり，更新におけるインデクス維持のオーバヘッドはわずかであることが分かった．一般に使われている Prolog システムと比較して十分速い検索，更新が可能である．

第５章　知識検索と推論処理の並列化

５．１　はじめに

システム全体の高速化の要求から，処理を並列化することが重要となってきている．このため，知識検索モジュールの検索処理と，推論モジュール内の推論処理をそれぞれ並列化することを考える．

知識検索モジュールでは，知識ベース（項関係の集まり）を共有しつつ RBU コマンド単位で複数の並列プロセスから同時に検索し，推論モジュールでは検索コマンドの発行や検索結果の解析などを並列に行う．そこで，並列言語によって推論モジュールを記述する．並列言語処理系としては，ストリームベースで論理型指向の GHC [Ueda 86]，Parlog [Clark-Gregory 86]，Concurrent Prolog [Shapiro 86]，オブジェクト指向の ABCL/1 [Yonezawa et al. 88]，Concurrent Smalltalk [Yokote-Tokoro 88]，並列 Lisp の MultiLisp [Halstead 85] など数多くの処理系が提案されている．本アプローチのように２つのモジュールを結合して並列システムを構成する場合には，ストリームベースで処理を記述する方法が適している．さらに，RBU で扱う知識の構造が述語論理における項そのものであるため，論理型指向の言語と相性がよい．このため，並列推論モジュールを並列論理型言語 GHC で実現し，GHC から並列に RBU にアクセスすることにより並列知識ベース指向処理を実現することにする[横田他 90a].

本章では，GHC と RBU による並列知識ベース指向処理システムの構成方法と，そのシステム上での並列 SLD 演繹と並列 SUD 演繹の実現方法，およびマルチマイクロ計算機を使ったシステムの試作と，その試作システム上での並列 SLD 演繹と並列 SUD 演繹（第３章参照）の実験の結果について述べる．

５．２　並列知識ベース指向処理システム

５．２．１　並列論理型言語 GHC

並列論理型言語 Guarded Horn Clauses (GHC) [Ueda 86，淵他 87] は，述語単位でプロセスを生成し，その間で変数のバインディングによりストリームを生成して処理を進める言語である．シンタックス的には，

$$H \text{:-} G_1, ..., G_n \mid B_1, ..., B_m. \qquad n, m \geq 0$$

というガード付ホーン節の集合の形をしている．ここで，"$\mid$" はコミット演算子，"H" はヘッド部，"$G_1, ..., G_n$" はガード部，"$B_1, ..., B_m$" はボディ部と呼ばれる．なお，ガード部に組込述語しか許さない GHC を Flat GHC (FGHC) と呼ぶ．実現上の問題 [淵他 87] から，以降では FGHC を対象とする（特に断わらない限り，GHC は FGHC を指す）．

GHCの実行は，ゴールに対してヘッド部が単一化可能であり，ガード部がすべて成り立つとき（コミット可能なとき）に，ボディ部の述語を新たなゴールとして，述語単位で並列に同様の処理をすることにより進められる．コミットできるかできないか不確定の場合，つまりゴール中に束縛されていない変数を含み，その変数が束縛されるまでコミットの可能性が決定できない場合には，その述語単位で実行が中断（サスペンド）される．これらの機構により，小粒度に並列処理を記述することができ，同期処理などを明示的に書く必要がなく自然に並列性が抽出できることになる．

さらに，引き数としてリスト構造を使い，リストの末尾部分を未束縛変数としておくことにより，述語で実現された並列プロセス間のストリーム通信を記述することができる．つまり，転送側ではリストの末尾の未束縛変数に cdr 部が変数である新たなリストセルをメッセージとして次から次へとバインドし，受信側では変数に新たなメッセージがバインドされるまでサスペンドしていて，変数に何かバインドされたところでその内要の解析を開始する．このように，変数束縛を基にしたサスペンド機構によって，述語間で自然に同期を取った並列処理が記述可能となる．

このように GHC は並列記号処理を記述するのには強力な言語であるが，言語の意味論上の制約，つまりバックトラック不可能な don't care nondeterminism という性質[淵他 87]から，知識ベースと検索するような探索問題が苦手である．GHC で全解探索問題を解くためのコンパイル技術[Ueda 87]やレイヤードストリームと呼ばれる構造データを使った方法[Okumura-Matsumoto 87] が開発されているが，いずれも知識ベース指向処理が目指すような複数から共有でき，会話的に内要の更新ができるようなデータ構造を実現することが困難である．

より知識ベース指向処理に近い方法として，再帰呼び出しによる永久プロセス[淵他 87]の内部状態として知識ベースを管理する方法が考えられる．しかし，この場合には，内部のデータに対して高速アクセス手段を提供することが困難であり，さらにGHC の持つ単一化機能を検索に使えないため，GHC 自身で単一化処理を記述する必要が生じる[北上他 88]．これは，GHC の変数が並列プロセス間で共有されるため，変数の束縛のし直しが不可能なためである．しかし，GHC 自身で単一化処理を記述することは速度的に問題がある．特に，知識ベースの容量が増大した場合に，速度的に満足するものを実現することが難しい．このため，並列知識ベース指向処理を実現するためには，GHC を RBU のような知識検索モジュールと接続することが必要となる．

5．2．2　GHC からのRBU 並列アクセス

並列知識ベース指向処理システムでは，検索の指示や，検索された知識に対する操

作，およびユーザインタフェースの部分を GHC プロセスが担当し， RBU プロセスが知識ベースの管理を担当する．コマンドの生成や検索結果の解析において，GHC 内で使われるデータ構造が RBU と同様の項そのものであるため，GHC プログラムからの知識ベース利用が容易となる．知識ベースはシステム全体で共有され，GHC で記述されたプログラムの中から専用の組込述語を使ってその知識ベースを検索／更新するためのコマンドを RBU に渡す．RBUの処理自身も高速化のため並列化され，コマンド単位に並列に実行される（図 5-1）．各プロセスの間はストリーム通信で結ばれる．検索結果の転送は，検索処理と並列に，すべての検索結果がそろうのを待つことなく，結果が得られた時点で行うようにし，全体の並列性を高める．

知識検索モジュールにおける検索／更新の並列処理のための排他制御は，項関係単位で行う．検索処理の場合には，共有モードとしてその項関係に対する更新コマンド以外の複数アクセスを可能とし，更新処理の場合には排他モードとしてその項関係に対するすべてのアクセスを禁止する．共有モードから排他モードに移る場合は，すべての検索処理が終了するまで待たされることになる．これらの排他制御のための項関係単位の情報は，ディクショナリに1つの項目として格納される．また，排他機構はセマフォによ

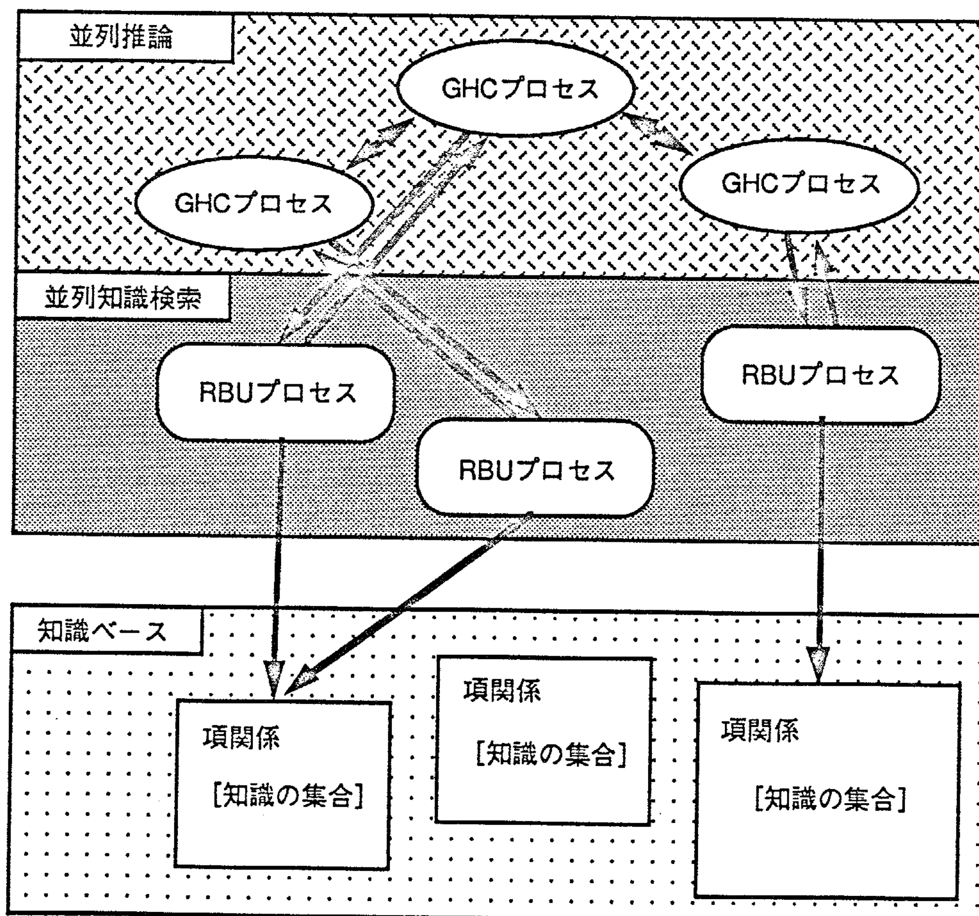

図5-1　GHCからのRBUアクセス

って実現される．

5．2．3　要求駆動によるストリームインタフェース

並列推論モジュールから並列知識検索モジュールへの検索要求指示は，前述したリスト状につながれた未束縛変数を用いたストリーム通信によって行う．このとき，複数の検索要求に対して並列に検索結果を返すために，コマンドの中に未束縛変数を入れて未完成メッセージ[淵他 87]として渡す方法を取る．検索結果の受け渡しは，要求駆動で行われることが望ましい．このため，未完成メッセージ中の未束縛変数に対するリストセルのバインドを検索結果の転送要求として用いることにする．つまり，並列推論モジュールは，検索結果が必要となった時点で，コマンド中に入れて渡した未定義変数に新たな未束縛変数をリストセルに入れて，転送要求として並列検索モジュールに渡す．並列検索モジュールは，この要求に従って検索結果を1つリストセル中の未束縛変数にバインドさせる．

例えば，表 5-1，5-2 に示すような項関係 tr1，tr2 を GHC から検索する場合を考える．表中の *\$(1)* や *\$(2)* は RBU における変数を表わしている．GHC では，大文字で始まる文字列を変数として取り扱う．このような変数の表記法は，知識検索モジュールと推論モジュールとで変数を区別するために用いている．前述したように GHC の変数は一度バインドされると修正できないため，そのまま検索の単一化に利用できない．また，この機構上で推論を行う場合，項関係に格納された知識はオブジェクトの知識であり，それを操作する GHC のプログラムはメタのレベルということになる．このオブジェクトとメタのレベルでの変数の区別のために *\$* を利用すると考えてもよい[北上他 88]．Prolog などでメタ推論を行う場合には，2章で行ったようにオブジェクトとメタを融合すること[Bowen-Kowalski 82]が可能であるが，GHC の場合は変数を同期機構に用いているため，そのままではオブジェクトとメタを融合することができない．しかし，*\$* による表記法

表5-1　項関係の例 (tr1)

第1属性	第2属性
p(\$(1),g(\$(2)))	r(\$(1),\$(2))
q(f(a,\$(1)),g(\$(1)))	r(f(a,\$(1)),\$(1))
p(\$(1),g(b))	r(h(a,b),f(a))
q(f(\$(1),\$(2)),g(c))	s(\$(1),g(\$(2),c))
p(f(a,b),h(\$(1)))	s(a,g(b,c))
p(f(a,\$(1)),h(\$(1)))	s(a,\$(1))

表5-2　項関係の例 (tr2)

第1属性	第2属性
q(c,\$(1))	\$(1)
p(f(c,d),e)	s(c,e)
p(f(\$(1),d),\$(1))	r(h(c,d),\$(1))
s(b,g(\$(1),\$(2)))	\$(2)
s(b,g(\$(1),d))	s(\$(1),d)

を使うことによりメタのレベルからオブジェクトの変数を操作することができるようになる．

GHCのユーザプログラムから知識検索モジュールを使う場合には，

rbu_stream(C1)

という専用の組込述語中の変数 *C1* に，

C1 = [urs(tr1,[1=p(a,$(10))],[1, 2],X) | C2]

のように検索コマンドをストリームとして渡す．

ここで，*urs(tr1,[1=p(a,$(10))],[1, 2],X)* はRBUのコマンドの１つの単一化制約のコマンドで，tr1という項関係の第１属性が *p(a,$(10))* と単一化可能であるタプルを探しだし，その第１属性と第２属性を取り出してくることを示している．この場合には，tr1の第１タプルと第３タプルが得られることになる．検索結果は，検索されたデータが見つかり次第，コマンド中の変数 *X* にストリームとして返ってくることになる．このように，メッセージの中に未束縛変数を入れて渡し，変数へのバインドとしてその返事を受ける方法が未完成メッセージによる通信と呼ばれる．

要求駆動を実現するため，変数 *X* に入れ物として，

X = [X1 | X2]

のような空のリストをデータが必要になるたびに用意しておき，知識検索モジュール側ではその要求に対して検索結果を返すようにする．最初に条件に単一化可能なタプルが検索されると，検索処理の終了を待つことなく，変数 *X1* に

X1 ← [p(a,g($(2))), r(a, $(2))]

のようにその最初の検索結果がバインドされる．これは，第１タプルの変数 *$(1)* に検索条件の *a* がバインドされ，検索条件の変数 *$(10)* に *g($(2))* がバインドされたものである．

さらに，次の入れ物を，

X2 = [X3 | X4]

と用意しておくと，次のタプル（第３タプル）が検索されたところで，

X3 ← [p(a,g(b)), r(h(a,b),f(a))]

というバインディングが得られることになる．実際には，*rbu_stream* という述語はGHCで記述されていて，出力ストリーム用の変数にリスト構造がバインドされるまでフックしていて，バインドされた時点でRBUに *get* コマンドを発生するようになっている．このような２つもモジュール間で要求駆動に基づくインタフェースをとる場合には，入れ物を１度に２つ以上用意しておいて，ダブルバッファリングすることが有用である（第６章参照）．最終的に，すべてのタプルがRBUからGHCに渡ると，結果ストリームの最後を示すために，転送要求である

X4 = [X5 | X6]

に対して，

X5 ← -1

のように，EOF が返る．これに対して，

X6 = []

のように空リストをバインドことによりストリームを切断することができる．もしさらに転送要求を出すと EOF を返し続ける．また，全部転送する前にストリームが切断されると，処理はそこで終了する．

並列知識検索モジュールでは，コマンド単位で複数の検索処理を並列に実行する．つまり，上のコマンドストリームの変数 *C2* に次の RBU コマンドをバインドしておくことにより，そのときの知識検索モジュールに割り当てられたプロセッサ分だけ，上の *urs* の検索処理と並列にコマンドを実行する．例えば，

C2 = [ujs(tr1,[2],tr2,[1],[1,4],Y) | C3]

といった検索コマンドを同時に実行することができる．ここで，*ujs(tr1,[2],tr2,[1],[1,4],Y)* は項関係 tr1 の第 2 属性と項関係 tr2 の第 1 属性の中から単一化可能な組み合わせを見つけてきて，2 つの項関係をつけた場合の第 1 属性と第 4 属性を取り出してくる処理を行う単一化結合というコマンドである．この場合には，tr1 の第 4 タプルと tr2 の第 4 タプルの組み合わせが検索される．この検索結果の転送は，その前の単一化制約の結果の転送とは別に，並列に行うことができるようにすべきである．複数の検索コマンドの結果を要求駆動にしたがって並列に行うために，それぞれのコマンドに対応させて RBU 側にバッファを用意する．このバッファは，未完成メッセージのための変数（X や Y）に対応していると考えてもよい．

GHC プログラムの中で，

Y= [Y1 | Y2]

とすることにより，

Y1 ← [p(f(b,$(10),g(c)), c]

という結果が得られる．

以上の GHC と RBU の通信の様子を図 5-2 に示す．

5．3　並列 RBU と GHC による並列演繹

並列知識ベース指向処理システム上で，第 3 章で示したRBU を使った SLD 演繹および SUD 演繹を実現する方法について述べる．第 3 章で用いた述語の二進木表現はリストで実現する．アルゴリズム中の単一化制約は，並列効果を増すために単一化制約の並列実

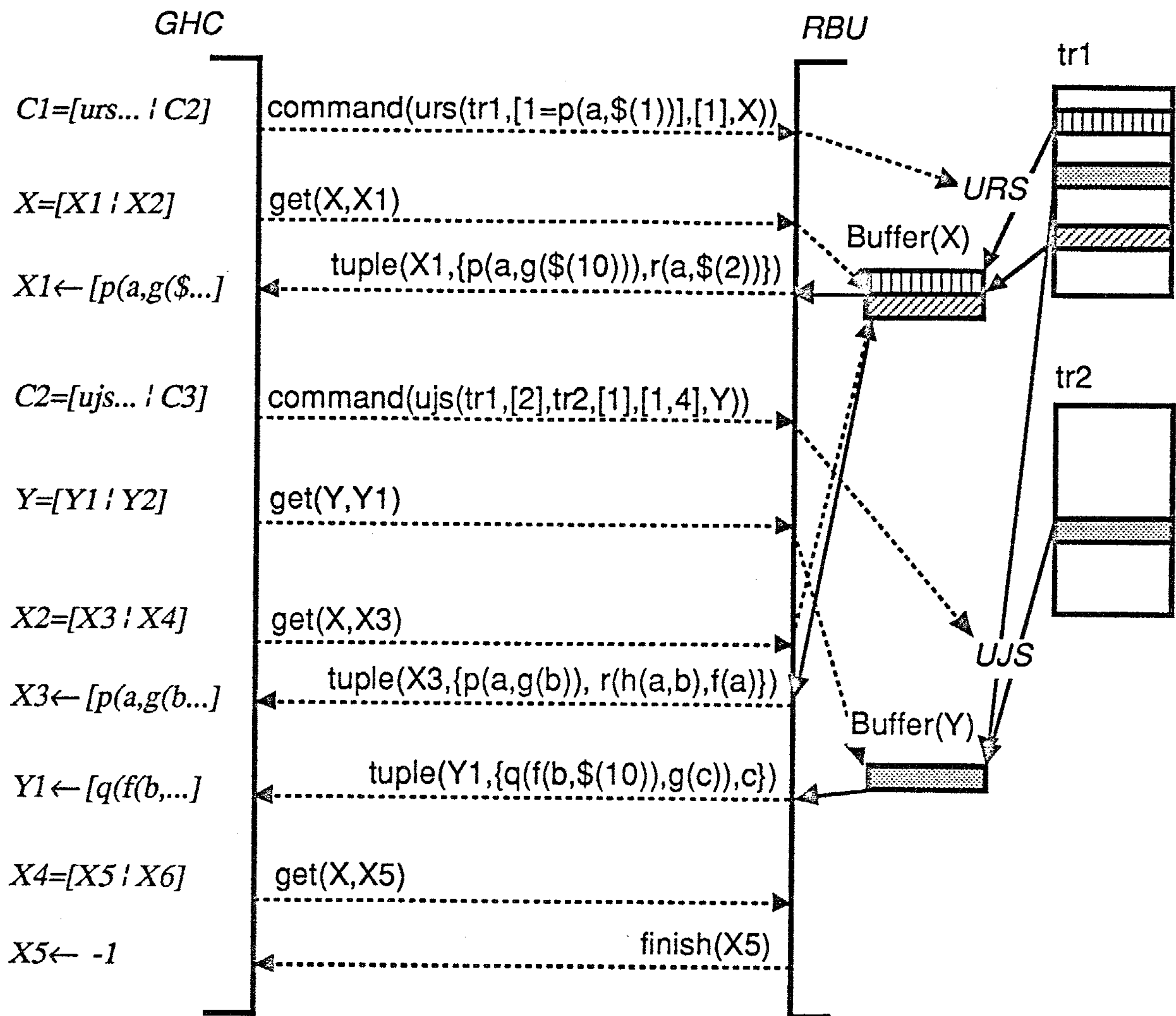

図5-2　GHCとRBUの通信

行に置き換える．つまり，結合操作は制約処理の繰り返しに分解することができ，並列知識ベース指向処理システムではその分解された制約処理を並列に実行できる．さらに，アルゴリズム中の変数制約は検索結果のストリームを GHC でチェックすることで実現し，並列性を高めるようにした．このようにしても，第３章で証明した完全性と停止性には影響はない．

５．３．１　並列 SLD 演繹

例として，第２章で用いた親子関係から祖先関係を導き出す問題を使う．祖先関係についての非常に小さい知識ベースを項関係に格納したものを表 5-3 に示す．単一化結合を単一化制約に分解したことにより，繰り返し中に初期ゴールを保持するための属性（第３属性）を付加している．この第３属性には，第１，第２属性に現われない変数を入れて置く．

表5-3　子孫関係に関する項関係 (anc_kb)

第１属性	第２属性	第３属性
[ancestor($(1),$(2))|$(3)]	[parent($(1),$(2))|$(3)]	$(4)
[ancestor($(1),$(3))|$(4)]	[parent($(1),$(2)),ancestor($(2),$(3))|$(4)]	$(5)
[parent(kenichi,hanako)|$(1)]	$(1)	$(2)
[parent(kenichi,tarou)|$(1)]	$(1)	$(2)
[parent(tarou,jirou)|$(1)]	$(1)	$(2)

図 5-3 に RBU と GHC による並列 SLD 演繹のためのプログラムを示す．プログラム中の，*Stream = [_|_]* や *L = [_|_]* は，要求駆動で検索結果を得るために未束縛変数を入れたリストセルを用意しているところである．また，*file_stream* は入出力のための組み込み述語である．

表5-3 の項関係を anc_kb と呼ぶことにすると，

sld(anc_kb, ancestor(kenichi,$(10))).

を実行することにより，*sld_loop* という述語が再帰的に呼び出されることになる．この *sld_loop* の中で作られた RBU コマンドが GHC 変数を介して *rbu_stream* という組込述語に渡されることになる．まず最初に，

sld_loop(anc_kb, [[[ancestor(kenichi,$(10))][ancestor(kenichi,$(10))]],-1],RBU,OUT)

という呼び出しが起こり，その中で，

urs(anc_kb, [1=[ancestor(kenichi,$(10))], 3=[ancestor(kenichi,$(10)]], [2, 3], Stream)

という単一化制約のコマンドが生成され，変数束縛を通して *rbu_stream* に渡される．この検索結果は，ストリームとして導出形と初期ゴールに対する変数の代入結果が，

S = [[[parent(kenichi,$(10))],[ancestor(kenichi,$(10))]],

[[parent(kenichi,$(2)), ancestor($(2),$(10))], [ancestor(kenichi, $(10)]]

のように得られる．この２つの結果を使って，２つの *sld_loop* が並列に呼ばれ，それぞれの中で，

urs(anc_kb, [1=[parent(kenichi,$(10))], 3=[ancestor(kenichi,$(10)]], [2, 3], Stream)

urs(anc_kb, [1=[parent(kenichi, $(2)), ancestor($(2), $(10))], 3=[ancestor(kenichi,$(10)]],

[2, 3], Stream)

という単一化制約コマンドが生成される．これらに対して，さらにまた検索結果がストリームとして得られ，その内要に対してそれぞれ新たな検索コマンドが生成されて行くことになる．

このようにして並列に演繹が進み，*Resolvent* という変数が *nil* になると，そのときの変

```
sld(KB,Goal) :- true |
        rbu_stream(RBU),
        file_stream(OUT),
        sld_loop(KB,[[[Goal],[Goal]],-1],RBU,OUT).

sld_loop(KB,[[Resolvent, Goal]|L],RBU1,OUT1) :- Resolvent ≠ nil |
        RBU1 = [urs(KB, [1=Resolvent,3=Goal],[2,3],Stream) | RBU2],
        Stream = [_|_],
        sld_loop(KB,Stream,RBU3,OUT2),
        L = [_|_],
        sld_loop(KB,L,RBU4,OUT3),
        merege(RBU3, RBU4, RBU2),
        merege(OUT2,OUT3,OUT1).
sld_loop(KB,[[nil, Goal]|L], RBU1, OUT1) :- true |
        L = [_|_],
        sld_loop(KB,L,RBU,OUT2),
        merege([writeterm(Goal)],OUT2,OUT1).
sld_loop(KB,[-1|L],RBU,OUT) :- true |
        L = [],
        RBU = [],
        OUT = [].
sld_loop(KB,[],RBU,OUT) :- true |
        RBU = [],
        OUT = [].
```

図5-3　GHCとRBUによる並列 SLD 演繹プログラム

数バインドの内容を含んだゴールを *writeterm* に入れて *file_stream* に渡す．その結果，最終的に，

ancestor(kenichi,hanako)

ancestor(kenichi,tarou)

ancestor(kenichi,jirou)

という 3 つのゴールが出力されることになる．

5．3．2　並列 SUD 演繹

RBU と GHC による並列 SUD 演繹のためのプログラムを図 5-4 に示す．これは 3 章のアルゴリズム 3-3 を基にしている．つまり，増加分の単位節と非単位節のとの間の単一化結合を繰り返すものである．前述したように，単一化結合を単一化制約の並列実行で実現している．プログラム中の *join_loop* がこれにあたる．単一化結合の一方を項関係とし，もう一方をストリームとしている．両方ストリームの場合には，$ 変数に対する単一化処理を GHC で記述すると非常に遅くなるので，一方を作業用の項関係（*work*）に格納して *join_loop* で実現している．また，変数制約に対応する処理が，*stream_vrs* である．なお，プログラム中の *ins_loop* はストリームを項関係に追加するための述語であり，*deferred_bind(F,X,Y)* は F の値が確定してから X と Y をバインドさせる次のような述語である．

deferred_bind(F,X,Y) :- wait(F) | X = Y.

SUD 演繹では，単位節の項関係とそれ以外の節の項関係に分ける必要があるため，表 5-3 の項関係は，表5-4 と表 5-5 の 2 つの項関係に分けられる．表 5-4 の項関係を anc_kb_u 表 5-5 の項関係を anc_kb_n と呼ぶことにすると，

sud(anc_kb_u, anc_kb_n, ancestor(kenichi, $(10)))

を実行することにより，前述した並列 SLD 演繹と同じ結果が得られることになる．

図 5-4 のプログラムでは，与えられたゴールの条件とは独立に，求められるすべての単位節を求めてから，ゴールに示された kenichi に関する単位節を抽出している．より効率的な方法として，setting（第 2 章参照）等による変数束縛を使うことにより，先に必要な

表5-4　祖先関係に関する単位節の項関係 (anc_kb_u)

第 1 属性	第 2 属性	第 3 属性
[parent(kenichi,hanako)|$(1)]	$(1)	$(2)
[parent(kenichi,tarou)|$(1)]	$(1)	$(2)
[parent(tarou,jirou)|$(1)]	$(1)	$(2)

表5-5　祖先関係に関する単位節以外の項関係 (anc_kb_n)

第 1 属性	第 2 属性	第 3 属性
[ancestor($(1),$(2))|$(3)]	[parent($(1),$(2))|$(3)]	$(4)
[ancestor($(1),$(3))|$(4)]	[parent($(1),$(2)),ancestor($(2),$(3))|$(4)]	$(5)

```
sud(U_KB,N_KB,Goal) :- true |
        rbu_stream(C1),
        file_stream(O1),
        C1 = [prs(N_KB,[1,2],S1)|C2], S1 = [_|_],
        join_loop(nonunit,U_KB,S1,C2,C3,US,[-1],NS,[-1],F2),
        deferred_bind(F2,C3,[crt(work,3,F3)|C4]),
        ins_loop(F3,work,US,C4,C5,F4),
        sud_loop(F2,F3,F4,N_KB,U_KB,US,NS,C5,C6,F5),
        deferred_bind(F5,C6,[urs(U_KB,[1=Goal],S2)|CT]), S2 = [_|_],
        portray(S2,O1,OT).

sud_loop(F1,F2,F3,N_KB,U_KB,[-1],[-1],C1,CT,FT) :-
    wait(F1), wait(F2), wait(F3) |
        C1 = CT, FT = term.
sud_loop(F1,F2,F3,N_KB,U_KB,US,NS,C1,CT,FT) :-
    wait(F1), wait(F2), wait(F3) |
        join_loop(unit,N_KB,US,C1,C2,NUS1,NUS2,NNS1,NNS2,F4),
        join_loop(nonunit,U_KB,NS,C2,C3,NUS2,NUS3,NNS2,NNS3,F6),
        join_loop(nonunit,work,NS,C3,C4,NUS3,[-1],NNS3,[-1],F8),
        ins_loop(F4,N_KB,NS,C4,C5,F5),
        ins_loop(F6,U_KB,US,C5,C6,F7),
        deferred_bind(F8,C6,[ers(work,F9)|C7]),
        deferred_bind(F9,C7,[crt(work,3,F10)|C8]),
        ins_loop(F10,work,NUS1,C8,C9,F11),
        sud_loop(F5,F7,F11,N_KB,U_KB,NUS1,NNS1,C9,CT,FT).

join_loop(nonunit,U_KB,[[H,B]|L],C1,CT,NUS1,NUST,NNS1,NNST,F1) :- true |
        C1 = [urs(U_KB,[1=B,3=H],[3,2],S)|C2], S = [_|_],
        stream_vrs(S,NUS1,NUS2,NNS1,NNS2,F1,FT),
        L = [_|_], join_loop(nonunit,U_KB,L,C2,CT,NUS2,NUST,NNS2,NNST,FT).
join_loop(unit,N_KB,[[H,B]|L],C1,CT,NUS1,NUST,NNS1,NNST,F1) :- true |
        C1 = [urs(N_KB,[2=H,3=B],[1,3],S)|C2], S = [_|_],
        stream_vrs(S,NUS1,NUS2,NNS1,NNS2,F1,FT),
        L = [_|_], join_loop(unit,N_KB,L,C2,CT,NUS2,NUST,NNS2,NNST,FT).
join_loop(_,KB,[-1|L],C1,CT,NUS1,NUST,NNS1,NNST,F) :- true |
        L = [], C1 = CT, NUS1 = NUST, NNS1 = NNST, F = term.

stream_vrs([[H,$(N)]|L],US1,UST,NS1,NST,F1,FT) :- true |
        US1 = [[H,$(N)]|US2],
        L = [_|_], stream_vrs(L,US2,UST,NS1,NST,F1,FT).
stream_vrs([[H,B]|L],US1,UST,NS1,NST,F1,FT) :- B ≠ $(N) |
        NS1 = [[H,B]|NS2],
        L = [_|_], stream_vrs(L,US1,UST,NS2,NST,F1,FT).
stream_vrs([-1|L],US1,UST,NS1,NST,F1,FT) :- rue |
        L = [], US1 = UST, NS1 = NST, F1 = FT.
```

図5-4 GHCとRBUによる並列 SUD 演繹プログラム

単位節をしぼり込むようにすることも可能である．

5．4　並列知識ベース指向システムの試作

並列知識ベース指向処理システムを Sequent 社製の共有メモリ型マルチマイクロ計算機 Symmetry 上に試作し，その試作システムを使って並列 SLD 演繹および並列 SUD 演繹の実験を行った．

5．4．1　試作システムの構成

今回実験に使ったSymmetry の構成は，インテル 80386 のプロセッサボードが共有バスに 18 台接続されているもので，このプロセッサやメモリ等の資源は UNIX ベースの並列 OS である DYNIX が管理する．システム全体は，C で記述した．

図5-5 に，試作システムの構成を示す．Symmetry の複数のプロセッサ上に，GHC* のプロセスと RBU プロセスを置いて，それぞれのプロセス間で通信をしながら処理を進める．利用するプロセッサの台数は，実験のため変化させる．この時，あるプロセッサは，並列推論モジュール用か並列知識検索モジュール用に振り分けられ，1つのプロセッサ上で2種類のプロセス両方を実行することがない．GHC は述語単位で，RBU は検索

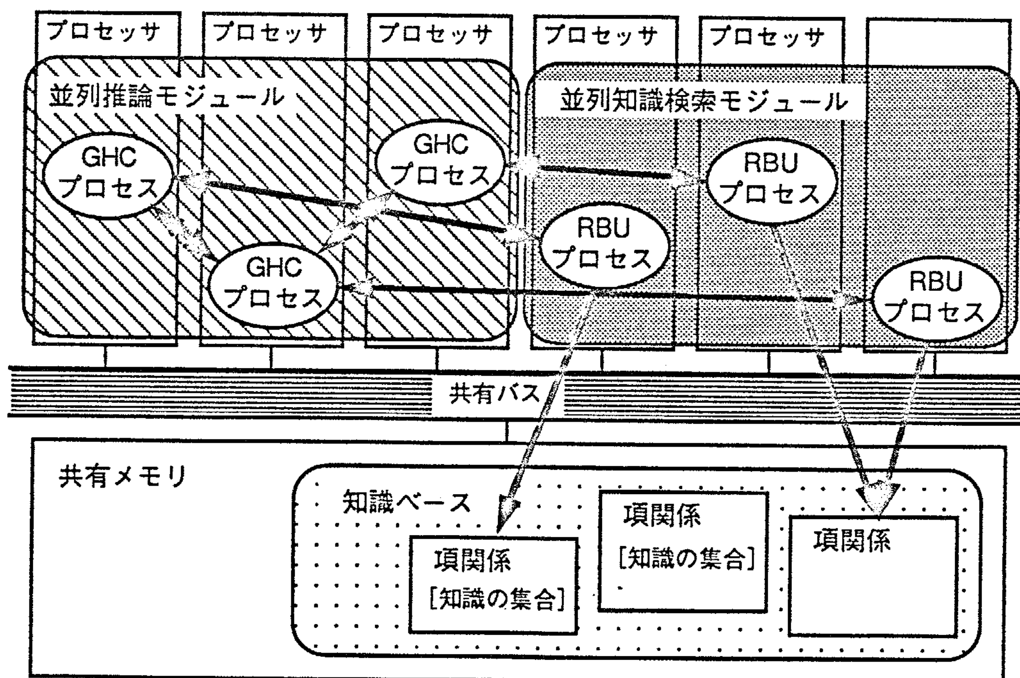

図5-5　試作システムの構成

* 試作システムにおける GHC 処理系は，小沢らによるFGHC 処理系[小沢他 89]に手を加えたものである．

／更新コマンド単位でプロセスとして実行される．実際に起動をかけられるプロセスは，負荷分散に従ってそれぞれのモジュール内で動的に決定される．

知識ベースは，共有メモリ上に置かれ，複数の RBU プロセスから同時にアクセスされる．コマンドレベルの排他制御は，前述したように項関係単位で行われ，セマフォは，Symmetry の持つスピンロックを用いて実現される．また，共有メモリのメモリ管理はページ単位で行い，排他管理機構を持つ管理テーブルを用いて管理する．知識検索モジュールは，第4章で述べたハッシュとトライ構造によるインデックスを持つ．

5．4．2　複数コマンドページによる動的負荷分散

並列推論モジュールと並列知識検索モジュールの間の接続も，共有メモリ上の共有領域を用いて実現する．共有メモリ上に，知識ベースの領域とは別に，コマンドページと呼ぶ領域を用意して，双方のモジュールからアクセスする．並列推論モジュールからは，検索や更新の指示をこのコマンドページを介して並列知識検索モジュールに渡す．並列知識検索モジュールで行われた検索の結果も，同様にコマンドページを介して並列推論モジュールに渡される．このとき，文字列で送るのではなく，ポインタでつながれたセル構造で情報が渡されるため，アトムテーブルはそれぞれのモジュールの間で共有している．アトムテーブルを共有することにより，通信容量や変換処理を大幅に減らすことができる（なお，将来的には並列推論モジュールで使うデータ構造全体も知識ベースとして捕え，推論も知識ベースも１つの枠組みの中で扱う方法も考えられる）．

コマンドや結果が上書きされたり，複数プロセスに重複して読み込まれたりしないように，アクセスするときにコマンドページ単位に排他制御を行う．このためコマンドページが１つだと，そのコマンドページへのアクセスが並列処理のボトルネックになる．そこで，コマンドページを複数設けて，アクセスを分散させている．試作システムでは，並列知識検索モジュールに割り当てられたプロセッサの台数と同じ個数のコマンドページを用意した．ただし，負荷が片寄らないよう，両モジュールの各プロセスはどのコマンドページにもアクセスできる（図 5-6）．つまり，並列知識検索モジュールの各プロセスは，検索／更新処理中でない場合，自分に対応するコマンドページに書き込まれたコマンドを優先的に読み取って実行するほか，自分に対応するコマンドページにコマンドが書き込まれていない場合には，他のコマンドページにコマンドを探しに行くことができる．また，並列推論モジュールは，コマンドを書き込む時点で書き込まれているコマンド数がなるべく少ないコマンドページを選択する．

この複数コマンドページを用いた動的コマンド割り振りにより，コマンド間の実行順序が保証されないことになる．並列推論モジュール側でコマンド間の順序関係を明示的

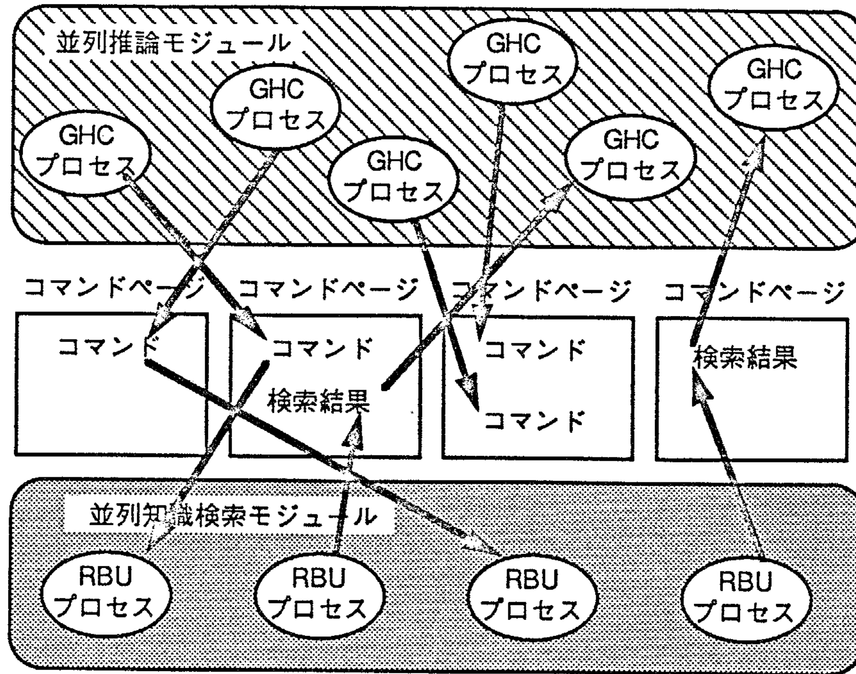

図5-6　複数コマンドページによる動的負荷分散

に制御したい場合には，検索／更新コマンド終了時に並列知識検索モジュール側から返されるステータス情報をコマンド生成に利用する．

GHC の処理系は実行中にヒープ領域が少なくなってくると，自動的にガーベージ・コレクション（GC）を開始する．この GC は世代別 GC [小沢他 89]となっており，内部の構造体の位置を移動する．このため，このモジュール間インタフェースもその移動に対応する必要がある．GC が起こると，GHC のプロセスだけでなくRBU のプロセスも止め，コマンドページ内のポインタの内要を更新する．

5．5　試作システム上の並列演繹の実行

試作システム上で，5．3で述べた並列 SLD 演繹と並列 SUD 演繹の実験を行った．実験では，まず5．3で例に用いた祖先関係を一人の親に二人の子供として5世代に渡って記述したもの知識ベースとした．つまり，ある1人の祖先から5世代後の32人の子孫までの親子関係を項関係に格納した．このとき，すべての祖先関係を出力する検索を実行させた．並列推論モジュール（GHC）用の割り当てプロセッサ台数と，並列知識検索モジュール（RBU）用の割り当てプロセッサ台数をそれぞれ1台から1台ずつ変化させて行った場合の実行時間（エラップス・タイム）について，並列 SLD 演繹の場合を表5-6に，並列 SUD 演繹の場合を表 5-7 に示す．また，それらをグラフにしたものを，それぞれ図 5-7，図 5-8 に示す．なお，前述した GC の影響をできるだけ小さくするため，ヒープの大きさをできるだけ大きく取って測定を行った．

表5-6　並列 SLD演繹におけるプロセッサ割り当てと実行時間　単位：秒

	RBU:1	RBU:2	RBU:3	RBU:4	RBU:5	RBU:6
GHC:1	25.19	25.18	24.84	25.71	25.14	25.60
GHC:2	22.88	12.77	12.72	12.61	12.61	12.80
GHC:3	17.51	9.21	8.58	8.57	8.40	8.44
GHC:4	17.36	8.80	6.74	6.42	6.40	6.35
GHC:5	17.54	8.82	6.04	5.36	5.26	5.13
GHC:6	18.51	8.83	6.01	4.68	4.51	4.40
GHC:7	18.33	8.88	6.00	4.98	4.23	3.84
GHC:8	19.04	8.89	5.99	4.58	3.80	3.50
GHC:9	18.06	8.78	5.93	4.52	3.75	3.26
GHC:10	18.01	8.90	5.97	4.56	3.77	3.17

表5- 7　並列 SUD演繹におけるプロセッサ割り当てと実行時間　単位：秒

	RBU:1	RBU:2	RBU:3	RBU:4	RBU:5	RBU:6
GHC:1	23.13	15.29	13.74	12.86	12.79	12.74
GHC:2	18.18	11.17	8.69	7.90	7.30	7.26
GHC:3	17.29	10.22	7.75	6.61	5.86	5.80
GHC:4	17.28	9.91	7.68	6.44	5.69	5.22
GHC:5	17.28	9.82	7.47	6.34	5.64	5.14
GHC:6	17.39	9.79	7.21	6.31	5.63	5.15

並列 SLD 演繹では，　GHC 用のプロセッサが 1 台の場合には，RBU 用のプロセッサ台数を増やしても処理時間にほとんど変化がない．GHC 用のプロセッサが 2 台になると，RBU 用のプロセッサが 1 台から 2 台になることにより，処理時間が約半分になる．これは，検索モジュール用のプロセッサが 2 台以上になった場合に，推論モジュール用のプロセッサが 1 台では不十分であることを示している．逆に，RBU が 1 台の場合には GHC が 3 台以上の時，2 台の場合には 4 台以上の時，3 台の場合には 5 台以上の時，実行時間がほとんど変化しなくなる．このように，推論モジュールと検索モジュールのプロセッサ台数を釣合を取って増加させて行かないと処理速度が向上しないことがわかる．最終的には，GHC 10台，RBU 6台で 3.17秒とそれぞれが 1 台ずつの時の 25.19 秒の約

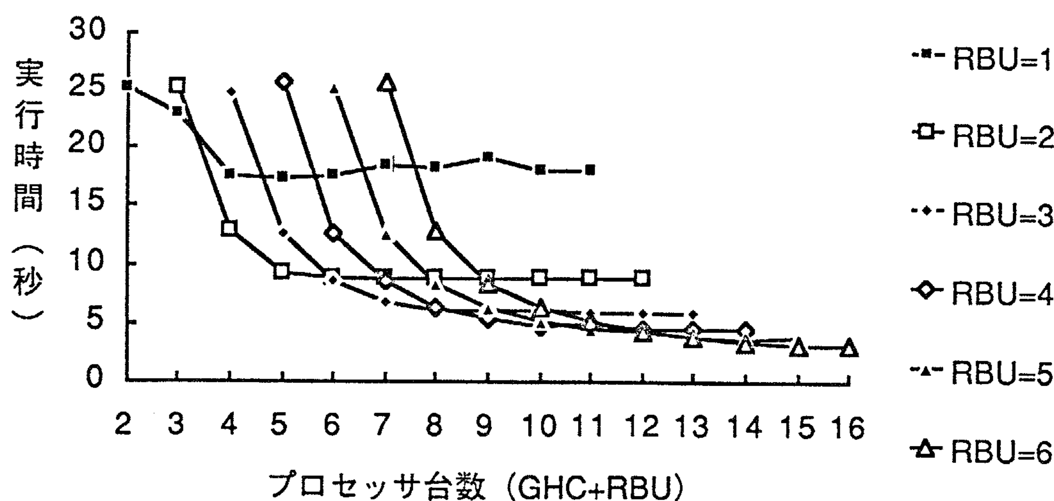

図5-7　並列 SLD演繹の実行時間の変化

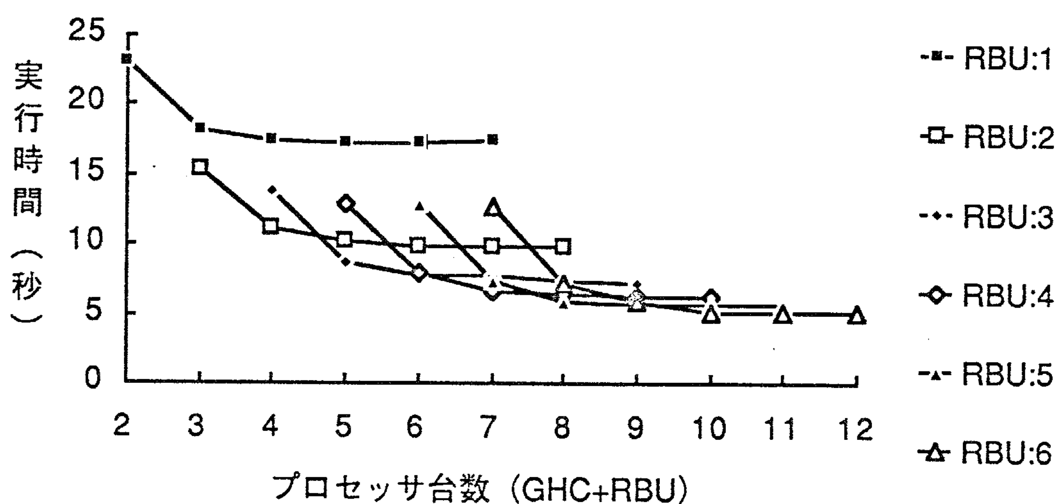

図5-8　並列 SUD演繹の実行時間の変化

8 倍のスピードで処理できている．これは，プロセッサ台数が 2 台から 16台と 8 倍に増えていることを考えると，理想に近い速度向上であるといえる．

図 5-7 から，並列 SUD 演繹も並列 SLD 演繹と同じ傾向を示していることがわかる．ただ，並列 SUD 演繹では，並列 SLD 演繹と比較すると並列化の効果は少し低めとなっており，GHC 5 台，RBU 6 台のとき 5.14 秒とそれぞれ 1 台ずつの時の 23.13 秒の 4.5 倍のスピードにとどまっている．これは，次の繰り返しにはいる前に同期を取る必要があるために全体として並列に動作できる部分が制限されるためと思われる．

SLD 演繹と SUD 演繹との比較と言う意味で見ると，この例の場合には SLD 演繹でもあまり冗長な枝を展開しないため，ほとんど差が出ていない（GHC 1 台，RBU 1 台の場合，SLD 演繹は 25.19 秒，SUD 演繹は 23.13 秒）．この差は，より冗長な枝を展開するよ

表5-8　並列SLD演繹と並列SUD演繹の比較　　単位：秒

	並列 SLD 演繹	並列 SUD 演繹
GHC=1, RBU=1	1141.69	97.94
GHC=3, RBU=3	374.48	33.70

表5-9　複数コマンドページの効果 (GHC=10, RBU =5)

コマンドページ数	1	2	3	4	5	6	7	8
実行時間（秒）	4.36	3.87	3.81	3.79	3.77	3.78	3.79	3.80

うな問題だと，はっきり現われる．例えば，子孫同士が友人であるような関係，

$\forall X, \forall Y, \forall A, \forall B$ (˜*ancestor(X,A)* ∨ ˜*ancestor(Y,B)* ∨ ˜*friend(A,B)*)

を求めるような場合には，その差が歴然と現われる．4 世代先の子孫が 1 組だけ友人関係であるような知識ベースを用意した場合の，並列 SLD 演繹と並列 SUD 演繹の実行時間を表 5-8 に示す．

次に，複数コマンドページの効果を測定した．並列プロセス間の通信が頻発する部分として，GHC 10台，RBU 5 台の場合に，コマンドページを 1 つから 8 まで変化させた時の並列 SLD 演繹の実行時間の推移を表 5-9 に示す．この結果から，コマンドページが 1 つから 2つに増えることによって 1 割以上の速度改善があることがわかる．また，知識検索モジュール用のプロセッサ台数よりコマンドページを増やしても効果のないことを確認し，試作システムで取った方法が適切であったことを示すことができた．

5．6　まとめと考察

並列知識ベース指向処理システムを構成するために，並列論理型言語 GHC から並列 RBU にアクセスする方法を提案し，共有メモリ型のマルチマイクロ計算機上に試作システムを作り，その上での並列演繹の実験の結果について述べた．

GHC と RBU の間のインタフェースを，未束縛変数による要求駆動のストリーム通信で構成することにより，知識検索モジュールを並列推論モジュールから使いやすくした．変数については，表記法を別に設け，メタとオブジェクトを切り分けた．また，実現に際しては，項関係単位で排他制御を行い検索コマンド毎に並列に実行する機構と，検索終了を待つことなく検索結果が得られたところでその結果を渡す機構を実現しと，モジュール間で複数コマンドページを使って動的付加分散を行うことにより，かなり理想に近い並列効果を得ることができた．実験の結果，このように機能が別れたシステムを並列化する場合には，それぞれのモジュールをバランスを取って並列化することが重要で

あることも示すことができた.

実験として行った並列知識ベース指向処理システム上での並列 SLD 演繹や並列 SUD 演繹の実現は，述語論理の探索木の OR 並列探索の実現とみなすことができる．GHC 自身は AND 並列に処理を進める言語であり，本システムが AND 並列と OR 並列を検索機構を持ち込むことにより融合したものという捕え方もできる.

また，並列知識ベース指向処理システムは，第 2 章で述べた演繹データベースシステムの機能を拡張し，さらに並列化をはかったものとみなすこともできる．特に，SUD 演繹では，単位節を分離していることから，演繹データベースとしての性質を強く持っている．しかし，格納できる内容が演繹データベースに比べ格段に強力になっている．単位節以外が増えた場合に対応できるだけでなく，単位節中に関数を持つこともできるようになった．なお，問題によっては，SUD 演繹が SLD 演繹よりかなり効率が良いことも，実例を使って示すことができた.

ここで示した並列知識ベース指向処理システムにより，これからの知識処理にとって基本的な並列処理環境の提供ができるものと思われる.

第6章　並列知識ベース指向処理システム上での意味ネットワーク探索

6．1　はじめに

ここまで，述語論理を基にした知識ベース指向処理について述べてきた．知識の表現方法としては，述語論理以外にも，意味ネットワーク，プロダクション・ルール，フレームなど，いろいろのものがある[Barr-Feigenbaum 81]．本章では，並列知識ベース指向処理システムの応用例の1つとして，意味ネットワークで表現されるような知識を知識ベースに格納して，並列に検索することを考える．

意味ネットワーク (Semantic network) は，Quillian らによって人間の連想記憶の心理モデルとして考案されたものである[Quillian 68]．意味ネットワークでは，節点 (node) と節点を互いに結ぶ弧(arc)で知識を表現する．通常，節点は対象(object)あるいは概念 (concept) や状態 (situation) を表わし，弧はそれらの間の関係 (relaion) を表わす[Barr-Feigenbaum 81]．

例として，計算機に関する知識を意味ネットワークを用いて表現したものを図 6-1 に示す．この意味ネットワークでは，計算機の持っている CPU や OS に関する知識，さらにその CPU や OS の階層的な関係に関する知識について表現している．例えば，symmetry と言うのは，computer であり，80386 を cpu として持ち，os は dynix であることを図 6-1 は示している．さらに，その 80386 は intel の product であり，dynix は unix の1種であるというようなことも表現されている．さらに，sun や fmR に関する知識も含まれ，CPU や OS に関する節点を供給している．

ここでは，第5章で紹介した試作システム上でこのような意味ネットワークで表現されるような知識ベースを並列検索する方法を示す．意味ネットワークは，述語論理と比

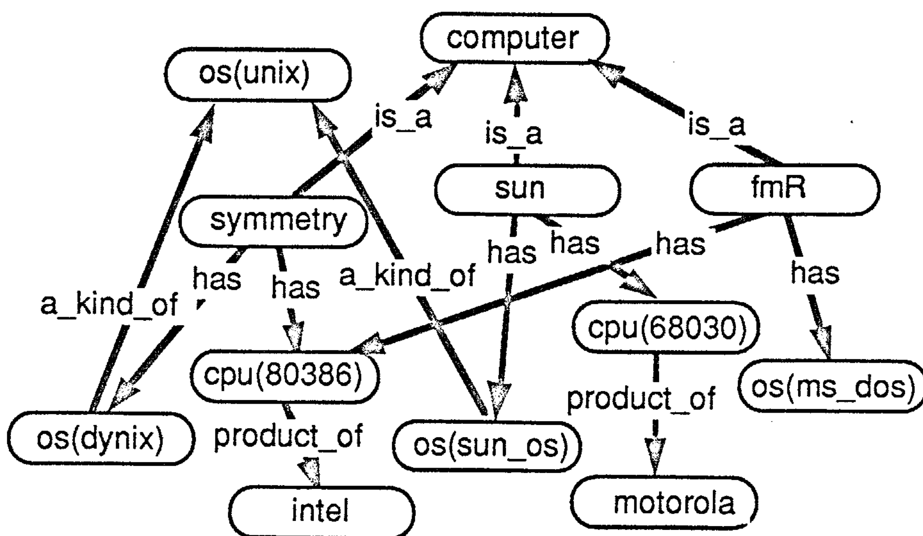

図6-1 計算機に関する意味ネットワーク

較すると，知識ベースとしての性質が強いと言える．ネットワークのノードを増やすことによって知識ベースのサイズが大きくなることから，比較的容易に大容量の意味のある知識ベースを準備することができる．特に，上述したような計算機に関する知識ベースについては，昨今各種の計算機がLANを介して多数接続されるようになり，人間の記憶だけで全体を把握することが困難となっており，メンテナンスの面からも計算機に付随する情報を知識ベースとして持つ必要性も高くなってきている．このため，本章では，第5章で述べた並列知識ベース指向処理システムの評価として，計算機に関する意味ネットワークの並列探索を行うことにする．

意味ネットワークを並列に検索するシステムとしては，意味記憶システムIX [樋口他 86] が提案されている．IXでは意味ネットワーク検索に特化した専用のハードウェアを使っているが，ここでは汎用の共有メモリ型マルチマイクロ計算機を利用しているため比較の対象としなかった．また，Prologで意味ネットワークを検索する方法としてDCKRが提案されている[小山-田中 85]． DCKRによって知識を表現し，第5章で述べた並列演繹方式をそのまま利用する方法もあるが，ここでは試作システムに取り入れたインデックスを生かすように意味ネットワークを項関係に格納し，処理速度の目安としてPrologの処理系との比較を行うことにする．

6．2　意味ネットワークの項関係への格納

意味ネットワークで表現された知識を項関係に格納する方法を考える．DCKRでは，ネットワークの弧を確定節（Prologの節）で表現して，論理変数へのバインディング情報で意味階層の保持をする方法を取っている．ここでも，同様に変数のバインディングで意味階層の履歴を保持するが，RBUのインデックスを有効に利用するために，オブジェクトを直接項関係のアイテムとして格納することにする．

図6-1の知識を項関係に格納したものを表6-1に示す．この項関係の第1属性と第3属性はオブジェクトを，第2属性はその間の関係を示している．第4，5，6属性は，意味階層をたどる場合に，問い合わせ中の変数のバインディング情報を保存するための内部変数である．第2章で述べた演繹データベースだと，関数を扱うことができないため，ここで示したような知識をそのまま関係データベースに格納することはできない．文字列で格納したとしても，ここで必要となるような検索機能を実現することは困難である．

6．3　並列トラバース

表6-1の項関係に格納された意味ネットワークをトラバースするための並列推論モジュ

表6-1　図6-1の知識の項関係

オブジェクトA	関係	オブジェクトB	ワーク用		
symmetry	is_a($(1))	computer	$(1)	$(2)	$(3)
symmetry	has($(1))	os(dynix)	$(1)	$(2)	$(3)
symmetry	has($(1))	cpu(80386)	$(1)	$(2)	$(3)
sun($(1),$(2))	is_a($(3))	computer	$(3)	$(4)	$(5)
sun($(1),$(2))	has($(3))	os(sun_os)	$(3)	$(4)	$(5)
sun(3,$(1))	has($(2))	cpu(68030)	$(2)	$(3)	$(4)
fmR($(1))	is_a($(2))	computer	$(2)	$(3)	$(4)
fmR($(1))	has($(2))	os(ms_dos)	$(2)	$(3)	$(4)
fmR(70)	has($(1))	cpu(80386)	$(1)	$(2)	$(3)
cpu(68030)	product_of($(1))	motorola	$(1)	$(2)	$(3)
cpu(80386)	product_of($(1))	intel	$(1)	$(2)	$(3)
os(dynix)	a_kind_of($(1))	os(unix)	$(1)	$(2)	$(3)
os(sun_os)	a_kind_of($(1))	os(unix)	$(1)	$(2)	$(3)
$(1)	$(1)	nil	empty	$(2)	$(3)

ール側のプログラムを示す．プログラム中の *rbu_stream* という述語が並列知識検索モジュールにコマンドを渡すための組込述語，*file_stream* は入出力のための組込述語である．*search_loop* という述語の第3引き数によって，処理ループ中で発生するRBUコマンドを *rbu_stream* にストリームとして渡している．なお，*urs* はRBUのコマンドの1つで，単一化制約のコマンドである(第3章参照)．なお，プログラムの中で，*S=[_,_|_]* あるいは *L=[_,_|_]* と記述してある部分は，ダブルバッファリングを使った要求駆動処理のために未定義変数の入ったリストを2つ分用意しているところである．

並列知識検索モジュールでは，コマンド単位で複数の検索処理を並列に実行する．このため，変数 *RBU* にバインドされるコマンド列は，そのときの並列知識検索モジュールに割り当てられたプロセッサ分だけ並列に実行される．

例えば，symmetry に関する知識を検索するため

traverse(kb,symmetry,$(10)).

という問い合わせを行うと，

urs(kb,[1=symmetry,2=$(10),5=symmetry,6=$(10)],[3,4,5,6],S)

というコマンドが並列知識検索モジュールに渡される．このコマンドは，「kbと言う項関係の第1属性と第5属性を symmetry と単一化し，第2属性と第6属性同士を単一化できるタプルの3，4，5，6属性を取り出す」という意味で，表の 1，2，3番目と最

```
traverse(KB,Arg1,Arg2) :- true |
        rbu_stream(RBU),
        file_stream(OUT),
        search_loop([[Arg1,Arg2,Arg1,Arg2],-1,_],KB,RBU,OUT).

search_loop([[Q1,Q2,Q3,Q4]|L],KB,R1,O1) :- Q1 ≠ nil |
        R1 = [urs(KB,[1=Q1,2=Q2,5=Q3,6=Q4],[3,4,5,6],S)|R2],
        S = [_,_|_],
        search_loop(S,KB,R3,O2),
        L = [_,_|_],
        search_loop(L,KB,R4,O3),
        merge(R3,R4,R2),
        merge(O2,O3,O1).
search_loop([[nil,empty,Q1,Q2]|L],KB,R,O1) :- true |
        O1 = [writeterm([Q1,Q2])|O2],
        L = [_,_|_],
        search_loop(L,KB,R,O2).
search_loop([-1,_|L],KB,R,O) :- true |
        L = [],
        R = [],
        O = [].
```

図6-2　並列トラバースのためのプログラム

後の 4 つのタプルのが得られる．このときの変数バインディングの様子は次のようになる．

[computer, $(1), symmetry, is_a($(1))]

[os(dynix), $(1), symmetry, has($(1))]

[cpu(80386), $(1), symmetry, has($(1))]

[nil, empty, symmetry, symmetry]

この結果の内 1 つ（第 1 属性が nil になったもの）が出力され，残りの 3 つは新たなコマンド生成に使われる．それらの 3 つのコマンドは並列に動作することが可能である．この様子を図 6-3 に示す．RBUプロセスの左側の矢印が検索条件，右側がその検索結果を概念的に（検索条件や結果の正確な記述ではない）示している．ある RBU プロセスによ

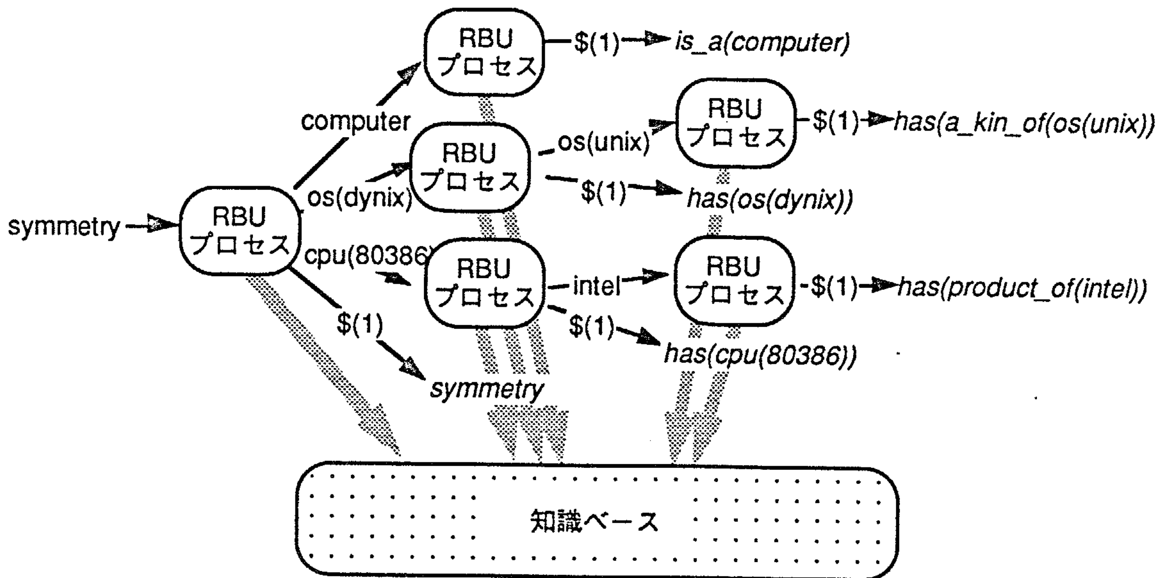

図6-3　並列検索の様子

って検索された結果は，次の RBU プロセスの検索条件となっている．これらのプロセス間の受け渡しは，GHC が行う．このため，動作順序等を記述することなく自然に並列に処理が進んでいく．また，RBU プロセスは検索が完了しないうちに，条件を満足するタプルが見つかったところから検索結果を返すため，結果的に図中の RBU プロセスは，ほとんど全てが並列に動作することになる．

図 6-3 で，斜体で書かれた部分が出力である．最終的に，

[symmetry,symmetry]

[symmetry, is_a(computer)]

[symmetry, has(os(dynix))]

[symmetry, has(cpu(80386))]

[symmetry, has(a_kind_of(os(unix)))]

[symmetry, has(product_of(intel))]

という単なるデータベース検索では得られない意味階層を反映した答えが出力される．

この他，構造が直接知識ベースの中に格納されているため，例えば sun3 シリーズの has 関係にだけ注目して，

traverse(kb,sun(3, $(20)), has($(30))).

のように問い合わせに構造を含ませることもできる．また，対話的な更新ができるため，検索結果を見ながら項関係に対するタプルの削除／挿入を行うことができる．

6．4　プロセッサ台数とインデックスの効果

性能を評価するため，表１にあるような計算機に関する項関係の知識を５５６タプル

分用意した．その項関係に対して，並列推論モジュールと並列知識検索モジュールのプロセッサの割り当て台数をそれぞれ1台から振らせた場合の，ある（64個の結果が得られる）問い合わせに対する検索時間を測定した．

第5章で述べたように，GHC はヒープ領域が少なくなってくると GC を行い，GHC のプロセスだけでなく，RBU のプロセスも止まることになる．そこで，以下の評価用の実験では，並列演繹のときと同様GC が起こらないよう十分広いヒープ領域を確保して行った．

プロセッサ台数と実行時間の関係を，項関係の第1属性にインデックスを張った場合と張らない場合について，それぞれ表 6-2と表 6-3に示す．表の横方向に RBU のプロセッサ数を，縦方向にGHC のプロセッサ数を変化させている．この表から，インデックスがない場合は，検索処理が全体の処理時間を制御することになり，検索モジュールのプロセッサを増やさないと，推論モジュールのプロセッサを増やしても処理速度が向上しないことがわかる．逆に，インデックスを張った場合には，推論処理が全体の処理時間を制御していることがわかる．

単に実行時間だけでは対象とした問題の難易度やサイズを示すことができないため，処理速度の目安として，完成度が高いと言われ現在最も広く使われている Prolog の処理系の1つであるQuintus Prolog のインタープリタと比較する．比較対象をインタープリタとしたのは，本システムが知識の対話的な更新を1つの特徴とし，更新の度にコンパイルし直すコンパイラでは比較の対象とならないためである．

また，すべてを GHC のみで記述したものと比較も考えられるが，現実的には不可能で

表6-2　プロセッサ台数と実行時間
（インデックスを張らない場合）

単位：秒

	RBU:1	RBU:2	RBU:3	RBU:4	RBU:5	RBU:6
GHC:1	4.239	3.207	3.178	3.198	3.126	3.154
GHC:2	4.080	2.362	1.870	1.837	1.822	1.830
GHC:3	3.997	2.261	1.677	1.484	1.405	1.384
GHC:4	3.986	2.219	1.624	1.403	1.257	1.227
GHC:5	3.976	2.176	1.613	1.359	1.217	1.143
GHC:6	3.969	2.165	1.600	1.324	1.170	1.097
GHC:7	3.964	2.159	1.585	1.301	1.158	1.081
GHC:8	3.966	2.170	1.600	1.307	1.148	1.070
GHC:9	3.963	2.197	1.562	1.308	1.125	1.061
GHC:10	3.966	2.192	1.584	1.321	1.120	1.061

表6-2　プロセッサ台数と実行時間
(インデックスを張った場合)

単位：秒

	RBU:1	RBU:2	RBU:3	RBU:4	RBU:5	RBU:6
GHC:1	2.983	2.959	3.033	3.024	3.018	2.977
GHC:2	1.589	1.548	1.553	1.576	1.582	1.600
GHC:3	1.117	1.096	1.080	1.115	1.090	1.115
GHC:4	0.871	0.838	0.846	0.833	0.846	0.850
GHC:5	0.747	0.687	0.689	0.685	0.678	0.705
GHC:6	0.678	0.595	0.582	0.584	0.597	0.602
GHC:7	0.652	0.532	0.523	0.521	0.537	0.514
GHC:8	0.623	0.484	0.476	0.468	0.479	0.468
GHC:9	0.627	0.455	0.441	0.433	0.430	0.423
GHC:10	0.636	0.440	0.401	0.394	0.390	0.390

ある．これは，GHCで同様の機能を実現するためには単一化処理自身をGHCで記述し，知識ベース全体をリスト等で保持しながら永久プロセス[淵他 87]で検索モジュールを実現する必要がある[北上他 88]ため，実行速度が比較にならないほど遅くなる上，メモリ使用量の関係から同程度のサイズの問題ではメモリ容量不足となり実測できないためである．

表 6-1 に対応する知識とその検索用の Prolog プログラムを図 6-4 示す．これと同様に評価用の知識ベースと同じ内容の知識ベース（５５６個分のホーン節）を SUN3/60 上の Quintus Prolog 1.6 上に乗せ，setof で同様の問い合わせで検索した場合の実行時間は，11.283 秒であった．SUN3/60 と Symmetry の 1 プロセッサの処理速度が Dhrystone 1.1 で比較すると，オプションなしの場合で SUN3/60 が 3582，Symmetry が 4258 であったことから，Symmetry 上の 1 プロセッサで Quintus Prolog 1.6 を使って実行した場合を想定すると，9.492 秒かかる計算になる．

この値を 1 とした場合に，試作システムでインデックスを張らなかった場合（表6-3）と，張った場合（表6-4）のプロセッサ数の変化に対する処理速度比の推移グラフをそれぞれ図 6-5 と図 6-6 に示す．処理性能を抑えているモジュールを明確にするために，図6-5では GHC プロセッサ台数を固定して RBU プロセッサ台数による変化を折れ線とし，図 6-6では RBU プロセッサ台数を固定して GHC のプロセッサ台数による変化を折れ線とした．

グラフから，インデックスを張って，並列知識検索モジュールと並列推論モジュールがそれぞれ 1台の場合にQuintus Prolog の約 3 倍，並列知識検索モジュール 4 台，並列推

```
test :- statistics, setof(trav(X,Y), trav(X,Y), S), statistics.

trav(X, Y) :- kb(X, Y, nil, empty).
trav(X, Y) :- kb(X, Y, XX, YY), trav(XX, YY).

kb(symmetry, is_a(X), computer, X).
kb(symmetry, has(X), os(dynix), X).
kb(symmetry, has(X), cpu(80386), X).
kb(sun(X,Y), is_a(Z), computer, Z).
kb(sun(X,Y), has(Z), os(sun_os), Z).
kb(sun(3, X), has(Y), cpu(68030), Y).
kb(fmR(X), is_a(Y), computer, Y).
kb(fmR(X), has(Y), os(ms_dos), Y).
kb(fmR(70), has(X), cpu(80386), X).
kb(cpu(68030), product_of(X), motorola, X).
kb(cpu(80386), product_of(X), intel, X).
kb(os(dynix), a_kind_of(X), os(unix), X).
kb(os(sun_os), a_kind_of(X), os(unix), X).
kb(X, X, nil, empty).
```

図6-4　比較用 Prolog プログラム

論モジュール 10台，計 14 台の場合に約 24 倍の処理速度で検索可能であることがわかる．また，インデックスを張らない場合は並列知識検索モジュール側の，張った場合には並列推論モジュール側のプロセッサ割り当てを相対的に増やすことにより，プロセッサ台数に対応して直線的に処理速度が向上している．このように，検索と推論の処理負荷の比率に合わせてプロセッサ台数を増やすことが重要である．ここで用いた例の場合，インデックスを張らない時には，並列知識検索モジュール 2 に対して並列推論モジュール 1 ，インデックスを張った時には並列知識検索モジュール 1 に対して並列推論モジュール 4 程度が適当である．ただし，この比率は知識ベースの大きさや問題の性質に依存するため，問題によってカスタマイズする必要がある．

また，インデックスを張ってない場合に，ある程度以上並列知識検索モジュールのプロセッサを増やすと，並列推論モジュールの台数比を上げても一様に処理速度の向上率

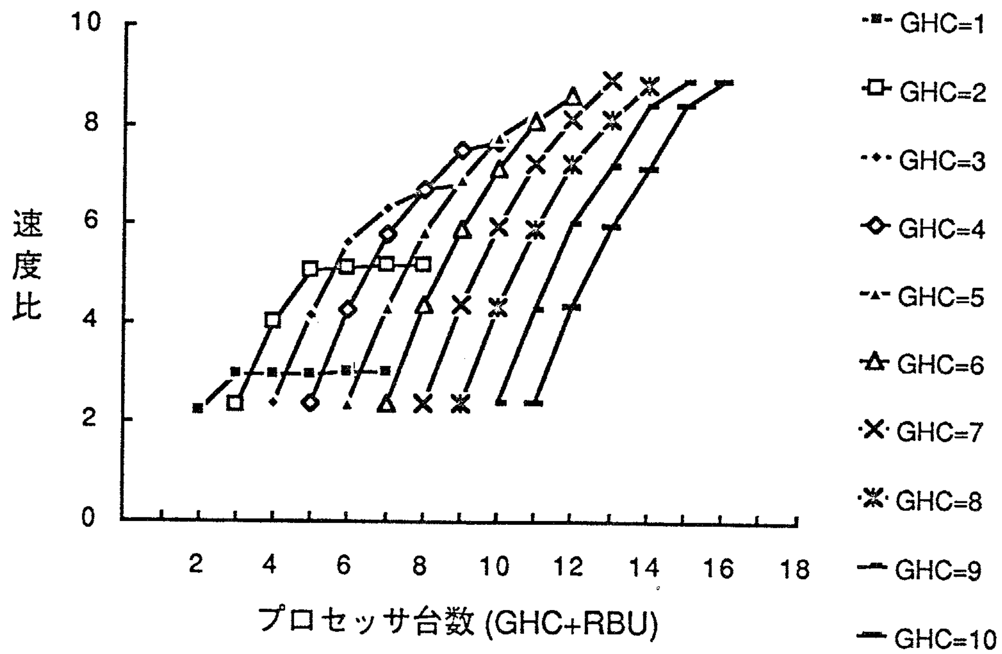

図6-5　プロセッサ台数による処理速度比の推移（インデックスを張らない場合）

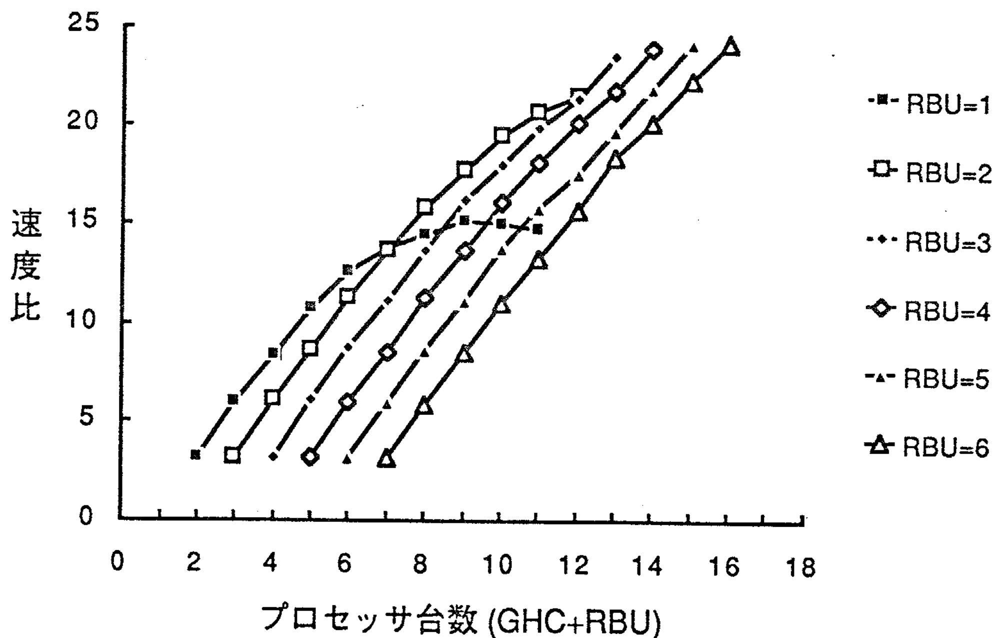

図6-6　プロセッサ台数による処理速度比の推移（インデックスを張った場合）

が低下している．この原因としては，インデックスが張ってない場合にはすべてのタプルを読み込む必要が生じるため，並列キャッシュのヒット率が下がり，共有メモリのアクセスネックになっている可能性が考えられる．並列知識ベース処理の場合，共有メモリのアクセス頻度を下げる上でも，ますますインデックスの重要性が高くなるといえる．

6．5　まとめと考察

実用に近い知識ベース指向処理の例として，計算機に関する意味ネットワークの並列検索を試作した並列知識ベース指向処理システム上で実行した．処理時間を計測した結果，Prolog の処理系と比較して十分実用になる速度で検索可能なことを示すことができた．また，検索と推論の負荷バランスを取りながら並列化することにより，プロセッサ台数に見合った速度向上が得られることが確認できた．第4章で示したように，インデックスの効果は知識ベースのサイズが大きければ大きいほどよく現われるため，ここで示した並列知識ベース指向処理システムは，さらに大きな知識ベースに対してより強力な検索環境を提供できる．

ここでは詳細には説明しなかったが，知識ベース指向処理の場合には検索処理のみでなく，その内容の更新処理も重要な要因である．例えば，計算機に関する知識ベースでも，新しい計算機が加わったり，仕様が変更になったりする度に知識ベースを更新する必要が有る．項関係を基にした並列知識検索モジュールは，項関係単位で排他制御が可能であり，更新処理の管理が容易である．さらに，試作システムでは第4章で示したようにインデックスを張ってあっても更新処理を高速に行うことができる．これは，Prolog や GHC のみで処理を行う場合と比較して，非常に強力な面であろう．

第7章　知識ベース並列指向処理システム上の並列状態遷移探索

7．1　はじめに

第5章および第6章で，RBU と GHC を使った並列知識ベース指向処理システム上での処理として，並列確定節演繹と意味ネットワークの並列探索について述べてきた．本章では，並列知識ベース指向処理のもう1つの例として，並列に状態遷移探索を行う方法を提案する．確定節演繹や意味ネットワーク探索は，知識ベースに対する問い合わせ処理を主とするものであったが，状態遷移探索では知識ベースを利用した診断型や計画型の意思決定支援システムを構築することを目指す．

作用素（プロダクション・ルール／ IF-THEN ルール）を用いて与えられた初期状態から次々と状態を遷移させて行き，終了条件を満足する終了状態に至るような手順を導く処理を状態遷移探索と呼ぶ．一般にプロダクション・システム[Barr-Feigenbaum 81]と呼ばれるものは，状態遷移探索を行って問題を解いている．プロダクション・システムでは，状態とプロダクション・ルールとの間のパターンマッチが全体の処理のかなりの部分を占めることが知られている[Forgy 82]．さらに，プロダクション・システムでは，状態探索木の枝刈りを行うために，コンフリクト・リゾリューションと呼ばれる処理をしており，これが並列の効果を抑える原因と考えられる．

そこで，本章では，状態とプロダクション・ルールとの間のパターンマッチに並列知識検索モジュールを利用すると同時に，コンフリクト・リゾリューションを行わない優良優先探索戦略[Yokota et al. 88, Itoh et al. 87]を導入する．また，GHC の永久プロセス[渕他 87]を使った木型プロセス構成によって優良優先探索の並列処理を効率良く管理する方法を示す．

並列処理に注目したプロダクション・システム・マシンとしては，CMU の PSM[Gupta et al. 86]，慶応大学のMANJI [Miyazaki et al. 87]，コロンビア大学のDADO[Stolfo 87]，NON-VON[Hillyer-Shaw 86]などがあり，DADO や NON-VONは木構造のプロセッサ構成を取っているが，いずれのマシンもコンフリクト・リゾリューションを行うことを基本としているところが異なる．なお，ここで示す方法はコンフリクト・リゾリューションを行わないので，プロダクション・システムと呼ばず状態遷移探索と呼ぶことにする．

7．2　状態遷移探索の並列性

状態遷移探索においては，ある状態から複数の状態に遷移する可能性があるため，状態空間全体としては1つの探索木を形成する（図 7-1）．この探索木上で，初期状態から終了状態へ至るパスを求めることが目標となる．しかし，多くの場合は，探索木の大き

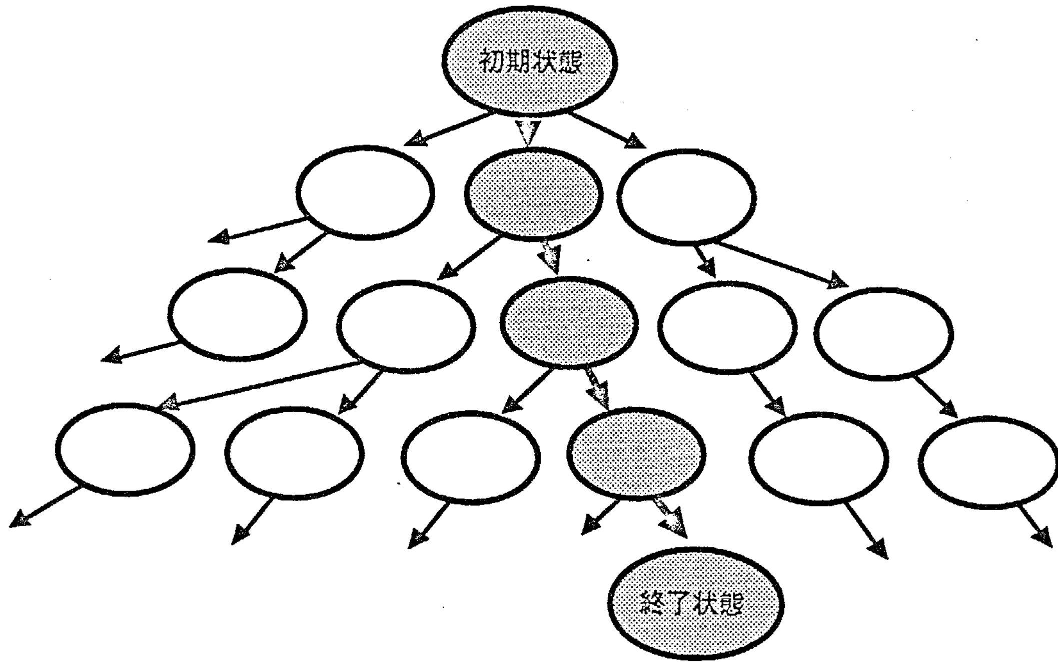

図 7-1　状態遷移図（探索木）

さが問題のサイズに対して指数関数オーダーで広がるため，何らかの探索制御を用いない限り多大な時間を要し，実際に答のパスを導くことは不可能となる．

そこで，従来のプロダクション・システム等では，各状態毎に評価関数を計算して，自分の枝の中で最も評価関数値の高い（良い）枝を選択し，それ以外の枝は刈り取ってしまう（コンフリクト・リゾリューションを行う）方法が取られてきた[Barr-Feigenbaum 81]．しかしコンフリクト・リゾリューションを行うと，ある時刻に存在する状態は１つ（あるいは高々自分から遷移する次の状態の数分）であるため，ほとんど並列性がないことが知られている．効率の良い探索手法で有名な Rete アルゴリズム[Forgy 82]を用いたシステムで測定した結果，高々一桁前後の並列性しかでないという報告がされている[Gupta 87]．また，初期状態から終了状態に到達可能なパスが存在するのに必ずしも解が得られないという欠点がある．つまり，答えのパスの評価関数値がその分岐点で必ず最高になるとは保証されていないため，答えのパスを枝刈りしてしまうことがありうるわけである．一度枝刈りをされてしまうと，その枝は二度と展開されないため，終了状態に到達不可能となってしまう．このため，必要な枝が刈り取らないようにプロダクション・ルールを記述する必要があった．

並列性を高め，プロダクション・ルールの記述を容易にするため，コンフリクト・リゾリューションを行わずに，各状態を１つのプロセサに対応付けて，並列に実行させるという方法が考える．このすると，同時に存在しうる状態数が多い上に，基本的に既に生成されてしまった状態間には相関関係がないため，非常に高い並列性を得ることがで

きる．しかし，この場合には，状態数が探索木の深さに対して指数関数的に増加することが一般的であるため，単純に各状態をプロセサに割り当てていったのでは，プロセサ数がいくら在っても足らないことになってしまう．

そこで，各状態をプロセスに対応付け，プロセス数を制限しながら各状態の評価関数値を実行優先度の管理に使って並列に状態空間を展開していくことにする[Yokota et al. 88, Itoh et al. 88]．つまり，評価関数値の高い状態を優先度の高いプロセスとしてディスパッチし，評価関数値の低い状態は実行を中断したプロセスとして保存する方法をとる．ただ，この場合にも評価関数値の管理が一元的になると，実行可能なプロセス数を増やした場合に管理プロセスの負荷が大きくなり，全体の処理のネックとなって並列性が上がらなくなってしまう．

７．３　木型プロセス構成上での優良優先状態遷移探索

評価関数値に従って並列に状態を展開していく場合に，管理が集中するのを避ける方法を考える．特に，実行優先度の管理を全部の優先順位の情報を集めてから決定するのではなく，部分的（ローカル）な情報から適当に定め，全体的に見ると比較的優先順位の高いプロセスが優先的に実行されているような状態を実現する．このため，展開する状態の管理を一元的に厳格に行うのではなく，階層的に緩やかに行う．一般に探索戦略として，展開された探索空間の中で評価関数値の最良の枝から展開していく方法を最良優先探索（Best First Search ）と呼ぶ[Barr-Feigenbaum 81]．ここでは，探索空間の中で最良であるかどうかは明確でないが，その状態のまわりの中で評価関数値がよさそうな枝から先に展開するので，優良優先探索（Better First Search ）と呼ぶことにする[Yokota et al. 88]．

優良優先探索の実現方法として，木型プロセス構成上での実現方法を考える．まず，状態遷移を行うプロセスと実行優先順位を管理するプロセスとを複数作り，それらを木構造にマッピングする．木はノードと枝からなり，一番根元のノードを根ノード，一番先端のノードを葉ノード，それ以外のノードを分岐ノードと呼ぶことにする．１つのノードが１つのプロセスに対応する．枝で接続されたプロセス間はメッセージ通信ができるようにする．ある状態から次の状態へ遷移させる処理は，各葉ノードのプロセスで行われる．生成された状態の情報は，メッセージとしてプロセス間を流れる．分岐ノードは，流れてくる状態情報の中の状態の評価関数値から，次に処理すべき状態を決定して自分の下のノードに渡すようにする．

図7-2にプロセス構成と状態空間の対応を示す．図の上半分は木型のプロセス構成を示しており，1～8の各ノードはそれぞれプロセスに対応している．この内，1が根ノー

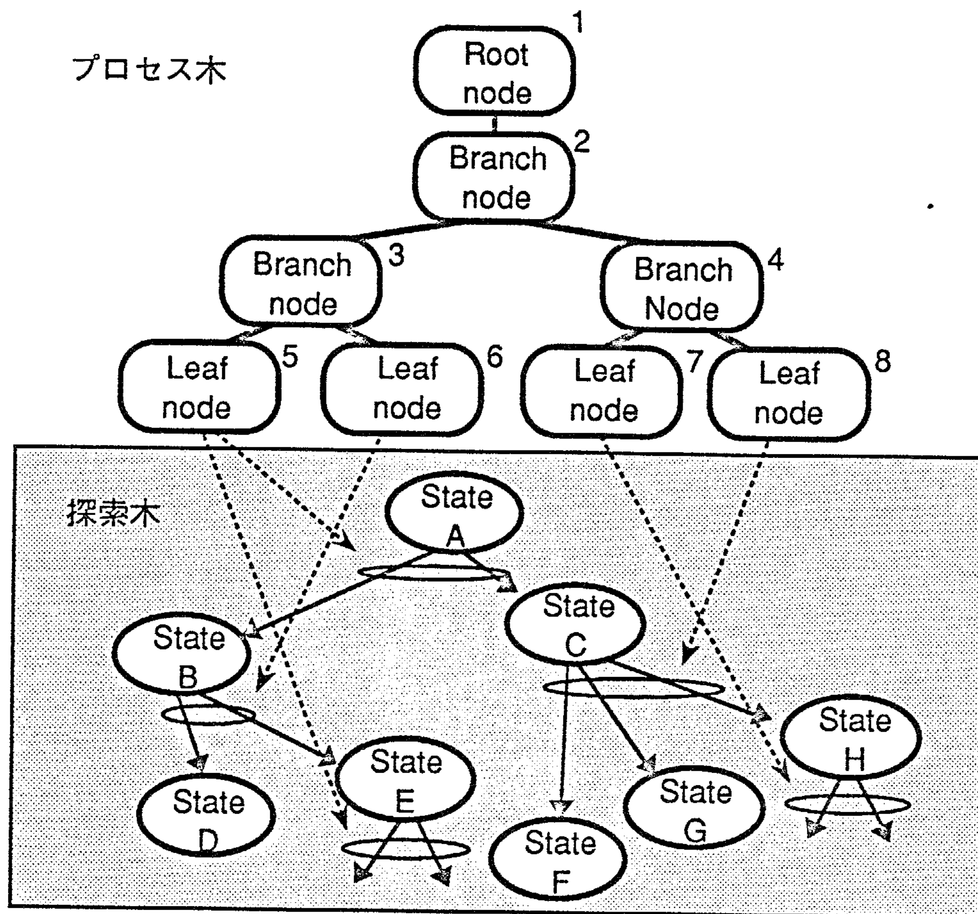

図 7-2 木型プロセス構成と状態木

ド， 2～ 4が分岐ノード， 5～ 8が葉ノードである．メッセージは，枝でつながれたプロセス間のみで転送される．図の下半分は，このプロセス構成上で処理されていく状態の遷移の様子を示している．状態 A から状態 B，状態 C へ遷移し，さらに状態 B から状態 D，状態 E に，状態 C から状態 F，状態 G，状態 H に遷移していくものとする．また，各葉ノードから状態間の遷移を示す矢印への破線の矢印は，その状態遷移がどの葉ノードで行われるかを示している．

例えば，図7-2では状態Aから状態 B および状態 C への遷移が， 5の葉ノードプロセスで処理される．次に，状態 B と状態 C に関する状態情報は 3の分岐ノードに送られることになる．この状態情報は， 3の分岐ノードで評価関数値で優先順位管理され，状態 B は 6の葉ノードに，状態 C は 2と 4の分岐ノードを通って 8の葉ノードに渡される．

緩やかな管理を行うために， 5の葉ノードで次に展開する状態を選ぶ場合，ローカルに状態 E と状態 D を比較して状態 E を次に展開する．この時，全体からすると状態 E より評価関数値が良い状態（例えば，状態 F ）が他に存在するかもしれないが，ローカルに判断して状態 E の処理を先行させる．つまり，並列性を高めるため，必ずしも最良の状態を選択する訳ではないが，なるべく評価関数値の良い状態を選択するようにする．

もし，厳格に管理しようとすると，全ての状態情報を根ノード（あるいは一番上の分岐ノード）まで転送して，そこで全体の中から一番評価関数値の良い状態を集めなければならない．すると，根ノードに転送するまでの処理に時間がかかる上，根ノードに評価関数値の管理処理が集中してしまい，解決しようとしている（管理処理の負荷が大きすぎるため，十分に並列処理できないという）問題点がそのまま残ってしまう．そこで，それぞれの分岐ノードでローカルに管理することにより，状態の評価関数の管理が一箇所に集中してボトルネックとなることを回避するようにする．

葉ノードへの処理の割当は，要求駆動で行う．つまり，葉ノードが暇な時は，上の分岐ノードに仕事の要求を出して処理すべき状態を受け取るようにする．新たに葉ノードで生成された状態の情報は，出来たものから分岐ノードに渡すようにする．分岐ノードは，下からの要求があって自分で状態情報を持っている場合のみ，自分の中にある状態の内評価関数値の良いものを下に渡す．下から状態情報が送られてきた場合には，その情報を蓄えて置き，適当な戦略で上の分岐ノードに転送する．このようにに，葉ノードが暇なときだけ要求を出して処理を行うようにすることによって，資源の管理が効率良く行うことが期待できる．

根ノードは，初期状態を与えたり全体の処理の終了を指示するなど全体の管理を行う．全ての葉ノードからの要求が根ノードに届いた場合には，処理すべき状態がなくなったことになるため，処理を終了する．

木構造を作って各分岐ノードで管理を分散して行うことにより，管理が集中して処理全体で管理処理の負荷が大きくなりすぎることがなくなる．ここで，注意すべきことは，全体から見た優先順位によって処理すべき状態を決めるような厳格な管理を行うのではなく，ローカルな優先順位を判定して緩く管理する点である．

7．4　永久プロセスによるノードの構成

RBU と GHC からなる知識ベース指向並列処理システムの上で優良優先遷移探索を行う場合，木型プロセス構成のノードを GHC で記述し，その葉ノードから RBU を使ってプロダクションを行う方法を考える．各ノード（プロセス）は，GHC の永久プロセス[淵他87]によって実現する．

例えば4つ葉ノードからなる木型プロセス構成を実現するための GHC プログラムを図7-3 に示す．このなかの，*root*, *branch*, *leaf* がそれぞれ根ノード，分岐ノード，葉ノードに対応する永久プロセスである．それぞれのプロセス間で，GHC の論理変数を使ったメッセージ通信を行う．また，葉ノードで RBU のコマンドストリームを生成し，そのストリームをマージしてrbu_stream 述語に渡している．

```
four_leaf_tree :- true |
        root(B1u,B1d),
        branch(1, 2, B1d,B2u,B3u,[],[],B1u,B2d,B3d),
        branch(1, 2, B2d,L1u,L2u,[],[],B2u,L1d,L2d),
        branch(1, 2, B3d,L3u,L4u,[],[],B3u,L3d,L4d),
        leaf(L1d,L1u, C1),
        leaf(L2d,L2u, C2),
        leaf(L3d,L3u, C3),
        leaf(L4d,L4u, C4),
        merege(C1,C2,C5),
        merege(C3,C4,C6),
        merege(C5,C6,C7),
        rbu_stream(C7).
```

図 7-3　4つの葉ノードを持つ構成

GHC の永久プロセスは，再帰的呼び出しによって実現されており，引き数として渡される変数の内容によって動作が規定される．変数は構造体の中で次々と再帰的に利用され，メッセージはストリームとして扱われる．

葉ノードに対応する永久プロセスは，次のように記述される．

```
leaf([state(E,S)|I],O1,C1) :- true |
        state_trans(state(E,S),O1,O2,C1,C2),
        O2 = [request|O3],
        leaf(I,O3,C2).
leaf([],O,C) :- true |
        O = [],
        C = [].
```

第1引き数がメッセージを受け，第2引き数がメッセージを送るようになっている．また，第3引き数は rbu のコマンドストリーム用である．

メッセージの先頭が状態を示す構造体 *state(E,S)* であった場合には，それを受け取り，状態遷移処理（*state_trans*）を起動する．*S* は状態の内容，*E* はその状態の評価関数値である．状態遷移処理については，後述する．状態遷移の結果および次の状態遷移を行うための要求駆動用の *request* はメッセージとして出力ストリームに加えられる．２番目の節は，入力ストリームが閉じられた場合に，出力ストリームを閉じるためのものであ

る．

分岐ノードはそれぞれにノードの状態を持ち，オートマトンによって動作が規定される．状態遷移探索の状態と，分岐ノード自信の状態と区別をつけるため，後者をノード状態と呼び，単に状態と記述する場合は探索空間の状態を指すものとする．

分岐ノードのノード状態は，状態を保持している場合，保持していない場合，下のノードからの要求を保持している場合，保持していない場合に場合分けできる．これらのノード状態は，永久プロセスのメッセージ用の引き数以外の引き数によって実現される．

もし，ある分岐ノードが状態も要求も保持していないノード状態にあって，メッセージとして要求を受けた場合には，その要求をキューに保持して，上のノードに要求のメッセージを送る．この要求キューが，状態を待つノード状態であることを示している．要求を保持する時に，次に状態を転送する先を知るために，下のノードのどちらから要求を受けたかを保持する必要がある．次の２つの節がそのための処理を示している．

```
branch(T, ZD, IU,[request|IL],IR,RQ,SQ,OU,OL,OR) :- SQ = [] |
        OU = [request|OUx],
        branch(T, ZD, IU,IL,IR,[req(left)|RQ],SQ,OUx,OL,OR).
branch(T, ZD, IU,IL,[request|IR],RQ,SQ,OU,OL,OR) :- SQ = [] |
        OU = [request|OUx],
        branch(T, ZD, IU,IL,IR,[req(right)|RQ],SQ,OUx,OL,OR).
```

IU，*IL*，*IR* はそれぞれ上，左下，右下のノードからの入力ストリームを示し，*OU*，*OL*，*OR* はそれらのノードへの出力ストリームを示している．また，*RQ* および *SQ* は，それぞれ要求用と状態用のキューを示す．この場合は，状態を持たないため，ガード部が *SQ* = [] となっている．なお，*T* および *ZD* は，ある頻度で状態を上のノードに転送するために，頻度を設定用の変数である．要求に関しては，頻度の設定と独立であるため，そのまま次の呼び出しに渡している．

もし，状態も持つ時に，つまり状態用のキュー *SQ* が空でない時に，要求メッセージを受けた場合には，キューの中から評価関数値に従って適当な状態を選択してきて，その状態を要求メッセージを送ってきたノードに送り返す．一般的には，最も評価関数の良い状態を選択する．

```
branch(T, ZD, IU,[request|IL],IR,RQ,SQ,OU,OL,OR) :- SQ = [_|_] |
        select(SQ,State,SQx),
        OL = [State|OLx],
        branch(T, ZD, IU,IL,IR,RQ,SQx,OU,OLx,OR).
```

```
branch(T, ZD, IU,IL,[request|IR],RQ,SQ,OU,OL,OR) :- SQ = [_|_] |
        select(SQ,State,SQx),
        OR = [State|ORx],
        branch(T, ZD, IU,IL,IR,RQ,SQx,OU,OL,ORx).
```

select は状態キューの中から一番評価関数値の良い状態を選択するための述語である．

既に要求キュー *RQ* に要求を保持していて，他のノードから状態を受け取った場合には，その状態は要求キューの情報に従って要求を発生しているノードに転送する．このとき，以前要求キューに要求をつなげた際に上に送った要求は不要となるため要求キャンセルを上に送る．また同時に，状態を上に送る頻度を設定するための変数 *ZD* で *T1* の剰余をとる．次に示すのは，左下のノードからの要求がキューの先頭にあった場合の処理である．

```
branch(T1, ZD, [state(E,S)|IU],IL,IR,[req(left)|RQ],[],OU,OL,OR) :-
    T2 := T1 + 1, modulo(T2, ZD, T3) |
        OL = [state(E,S)|OLx],
        OU = [cancel_req|OUx],
        branch(T3, ZD, IU,IL,IR,RQ,[],OUx,OLx,OR).
```

要求キャンセルを受けた場合には，要求キューの中から対応する要求を削除し，上のノードにさらに要求キャンセルを伝える．．

```
branch(T, ZD, IU,[cancel_req|IL],IR,RQ1,SQ,OU,OL,OR) :-  SQ = [] |
        rm_element(RQ1, req(left), RQ2),
        OU = [cancel_req|OUx],
        branch(T, ZD, IU,IL,IR,RQ2,SQ,OUx,OL,OR).
branch(T, ZD, IU,IL,[cancel_req|IR],RQ1,SQ,OU,OL,OR) :-  SQ = [] |
        rm_element(RQ1, req(right), RQ2),
        OU = [cancel_req|OUx],
        branch(T, ZD, IU,IL,IR,RQ2,SQ,OUx,OL,OR).
```

要求キュー *RQ* が空の時に他のノードから状態を受け取り，*T1* が 0 でない場合には，その状態を状態キューに保持しておく．

```
branch(T1, ZD, [state(E,S)|IU],IL,IR,[],SQ,OU,OL,OR) :-
    T1 ≠ 0, T2 := T1 + 1, modulo(T2, ZD, T3) |
        branch(T3, ZD, IU,IL,IR,RQ,[state(E,S)|SQ],OU,OL,OR).
```

要求キュー *RQ* が空で，*T1* が 0 の場合には，その状態を上のノードに転送する．このように *ZD* で示される値の個数の状態を受け取るうち 1 回，上のノードに状態を送る．

```
branch(0, ZD, IU, [state(E,S)|IL],IR,[],SQ,OU,OL,OR) :- true |
        OU = [state(E,S)|OUx],
        branch(1, ZD, IUx,IL,IR,[],SQ,OU,OL,OR).
```

以上のように入力に対応してノード状態をそのまま記述することにより，分岐ノードのオートマトンを構成することができる．

7．5 RBU を用いた状態遷移処理

次に，各葉ノードで行う状態遷移処理について考える．状態遷移（プロダクション・システム）においては，状態と作用素の間のパターンマッチ操作が，全体の処理に対して多くの部分を占めていることが知られている[Forgy 82]．そこで，第5章で述べた GHC と RBU のインタフェースを使って状態遷移処理を行う．項関係に格納する対象としては，初期状態，中間状態，作用素（プロダクション・ルール）が考えられる．作用素自身を GHC で記述する方法[淵他 87]も考えられるが，ここでは作用素を知識ベースと捕え，頻繁に更新されるものとして，初期状態，中間状態と同様に項関係に格納するものとする．

例として，"Monkey and Banana" 問題に対する作用素と，初期状態を項関係に格納したものをそれぞれ表7-1 と表 7-2 に示す．"Monkey and Banana" 問題とは，「Monkey が banana 1房が天井からぶら下がり， box が1つ置いてある部屋にいて，そのままでは monkey が banana に届かないような場合に，monkey はどうしたらbanana を手にすることができるか」という問題である[Nilsson 80]．

表 7-1 の第1，第2属性が作用素の条件部，第3属性が実行部に対応する．作用素の条件部は，状態との単一化のみで判別できる部分（第 1 属性）と，不等号など単一化だけでは判別できない部分（第 2 属性）とに分けられる．実行部は，状態から取り除くもののリスト，状態に新たに加えるもののリスト，出力するもののリストの3要素のリストからなる．

例えば第1タプルは，monkey が floor の上の座標 ($(2), $(3)) に示される場所にいて，$(1) で示されるものを持ち，座標($(6), $(7)) に示される場所に object の $(4) で示されるものがある時に，monkey が座標 ($(2), $(3)) から座標 ($(6), $(7)) に移動する場合の作用素を示している．つまり，状態の中に monkey(on(floor), hold($(1)), $(2), $(3)) という項と，object($(4),$(5), $(6), $(7)) という項および history([$(8)|$(9)]) という項があったら，monkey(on(floor), hold($(1)), $(2), $(3)) と history([$(8|$(9)])) の項を削除し，代わりに monkey(on(floor), hold($(1)), $(6), $(7)) と history([move($(6), $(7), $(2), $(3)),$(8)|$(9)]) という項を状態に加えることを示している．ただし，同じ場所では移動しないこと，前

表7-1　作用素（プロダクションルール）の例

<table>
<tr><th>IF 部 (1)</th><th>IF 部 (2)</th><th>THEN 部</th></tr>
<tr><td>[monkey(on(floor),hold($(1)),$(2),$(3)),
object($(4),$(5),$(6),$(7)),
history([$(8)|$(9)])]</td><td>[$(2) ≠ $(6),
$(3) ≠ $(7),
$(8) ≠ move($(2),$(3),
$(6),$(7)]</td><td>[[monkey(on(floor),hold($(1)),$(2),$(3)),
history([$(8)|$(9)])],
[monkey(on(floor),hold($(1)),$(6),$(7)),
history([move($(6),$(7),$(2),$(3)),$(8)|$(9)])],
[]]</td></tr>
<tr><td>[monkey(on(box),hold(nil),$(1),$(2)),
object($(3),on(ceiling),$(1),$(2)),
history([$(4)|$(5)])]</td><td>[]</td><td>[[monkey(on(box),hold(nil),$(1),$(2)),
object($(3),on(ceiling),$(1),$(2)),
history([$(4)|$(5)])],
[monkey(on(box),hold($(3)),$(1),$(2)),
history([hold($(3)),$(4)|$(5)])],
[]]</td></tr>
<tr><td>[monkey(on(floor),hold(nil),$(1),$(2)),
object($(3),on(floor),$(1),$(2)),
history([$(4)|$(5)])]</td><td>[$(4)≠ drop($(3))]</td><td>[[monkey(on(floor),hold(nil),$(1),$(2)),
object($(3),on(floor),$(1),$(2)),
history([$(4)|$(5)])],
[monkey(on(floor),hold($(3)),$(1),$(2)),
history([hold($(3)),$(4)|$(5)])],
[]]</td></tr>
<tr><td>[monkey(on(floor),hold($(1)),$(2),$(3)),
history([$(4)|$(5)])]</td><td>[$(1) ≠ nil,
$(4) ≠ hold($(1))]</td><td>[[monkey(on(floor),hold($(1)),$(2),$(3)),
history([$(4)|$(5)])],
[monkey(on(floor),hold(nil),$(2),$(3)),
object($(1),on(floor),$(2),$(3)),
history([drop($(1)),$(4)|$(5)])],
[]]</td></tr>
<tr><td>[monkey(on(floor),hold(nil),$(1),$(2)),
object($(3),on(floor),$(1),$(2)),
history([$(4)|$(5)])]</td><td>[$(4)≠ down($(3))]</td><td>[[monkey(on(floor),hold(nil),$(1),$(2)),
history([$(4)|$(5)])],
[monkey(on($(3)),hold(nil),$(1),$(2)),
history([climb($(3)),$(4)|$(5)])],
[]]</td></tr>
<tr><td>[monkey(on($(1)),hold($(2)),$(3),$(4)),
history([$(5)|$(6)])]</td><td>[$(1) ≠ floor,
$(5) ≠ climb($(1))]</td><td>[[monkey(on($(1)),hold($(2)),$(3),$(4)),
history([$(5)|$(6)])],
[monkey(on(floor),hold($(2)),$(3),$(4)),
history([down($(1)),$(5)|$(6)])],
[]]</td></tr>
<tr><td>[monkey(on($(1)),hold(banana),$(3),$(4)),
history([$(5)|$(6)])]</td><td>[]</td><td>[[monkey(on($(1)),hold($(2)),$(3),$(4)),
history([$(5)|$(6)])],
[],
[$(5)|$(6)]]</td></tr>
</table>

に移動してきた場所にすぐ戻らないことを第２属性で制約している．

同様に，第２タプルは box の上で天井に下がっているものをつかむための作用素，第３タプルは床の上にあるものを拾うための作用素，第４タプルは持っていたものを落とすための作用素，第５タプルは物に上るための作用素，第６タプルは物から降りるための作用素，第７タプルは banana を手にしたらそれまでの履歴を出力するための作用素である．

次に，表 7-2 の項関係の第１属性が初期状態を示し，第２，第３，第４属性は，実行時の作業領域として使われる．第１タプルは，初期状態では monkey は座標(5,7) に何も持たずにいることを，第２タプルは banana が座標 (2,2) に天井から下がっていることを，第３タプルは座標(9,5) の床の上に box が置いてあること，そして第４タプルは履歴としての初期状態を示している．状態が遷移して作られる中間状態も，この初期状態の項関係と同様の項関係が処理の途中で作成される．

表 7-1 の作用素と表 7-2 の初期状態から状態遷移を行うのためのプログラムを図 7-4 に示す．まず *ujs(rule,[1],State,[1].[5,2,3],X)* という単一化結合操作で，作用素の中から条件の１番最初の内容がそのときの状態とマッチするものをしぼり込む．しぼり込んだ作用素の内，２番目以降の条件すべてが満足しているか単一化制約を使って調べるのが，*trans_loop* である．単一化検索によって判定できる条件がすべて満足している作用素は，*trans_loop* の2番目の述語に渡される．ここで，単一化だけでは判定できない条件の部分を *check* でチェックし，それも満足している作用素の実行部分を *make_state* に送る．*make_state* では *Del* と *Add* から新たな状態を作る．その状態に対する評価関数値を計算した後，その状態が格納された項関係名と新しい評価関数値を組にして，上位の分岐ノードに転送する．また，その時何か出力するものがあるときは，*Out* の内容を出力する．

この状態遷移プログラムを実行することにより，[hold(banana), climb(box), drop(box), move(2,2,9,5), hold(box), move(9,5,5,7), start] という履歴が出力される．つまり，monkey は座標 (5,7) から座標 (9,5) に移動し，box を持って座標 (2,2) に移動し，持っ

表 7-2　初期状態

状態	ワーク用		
[monkey(on(floor),hold(nil),5,7)|$(1)]	$(1)	$(2)	$(3)
[object(banana,on(ceiling),2,2)|$(1)]	$(1)	$(2)	$(3)
[object(boxr,on(floor),9,5)|$(1)]	$(1)	$(2)	$(3)
[history([start])|$(1)]	$(1)	$(2)	$(3)

```
state_trans(state(Eval, State), O1, O2, C1, C2) :- true |
        C1 = [ujs(rule,[1],State,[1].[5,2,3],X)|C2],
        X = [_|_],
        trans_loop(X,Eval,State, O1,O2, C2, C3).

trans_loop([-1], Eval, State, O1, O2, C1, C2) :- true |
        O1 = O2,
        C1 = C2.
trans_loop([[[],Cond,Do]|L], Eval, State,O1,O3,C1,C3) :- true |
        check(Cond, true,X),
        make_state(X, Eval, State, Do, O1,O2, C1, C2),
        L = [_|_],
        trans_loop(L,Eval, State, O2,O3,C2,C3).
trans_loop([[Match, Cond, Do]|L], Eval, State, O1, O3, C1, C4) :- Match ≠ [] |
        C1 = [urs(State, [ 1=Match, 3= Cond, 4 = Do],[2,3,4],X)|C2],
        X = [_|_],
        trans_loop(X, Eval, State, O1, O2, C2, C3),
        L = [_|_],
        trans_loop(L, Eval, State, O2, O3, C3, C4).

make_state(true,Eval, State, [Del, Add, Out], O1, O2, C1, C4) :- true |
        copy_state(State,  Newstate, C1, C2, A1),
        del_tpl(A1, Del, Newstate, C2, C3, A2),
        add_tpl(A2, Add, Newstate, C3,C4, A3),
        eval_state(A3, Eval, Newstate, O1, O2),
        output(Out).
```

図 7-4　状態遷移のためのプログラム

ていた box を置いて，その上に乗り，banana をつかむことができる訳である．

プロダクションシステムで処理のネックとなっている状態と作用素の間のパターンマッチに単一化結合を利用することにより効率化をはかっている．また，木構造のプロセス構成で並列に状態遷移を行って行く場合に問題になるのが，プロセス間の通信量であ

る．できあがった状態の内容をすべてを葉ノードから分岐ノードへいちいち転送していたのでは通信量が膨大になり，処理のボトルネックになってしまう．そこで，共有メモリ上に項関係として状態を置いて，分岐ノードへはその共有メモリ上の状態へのポインタ（項関係名）のみを渡すことにより転送量を減らしている．

7．6 並列化と葉ノードの数

GHC の永久プロセスによる木構造のノード構成と RBU を使った状態遷移の方法を示したが，GHC は実際には述語単位で並列に実行するため，葉ノードがそのままプロセッサに固定されることはない．このため，葉ノードの数を一定にしてプロセッサ数を変化させることが可能である．では，葉ノードの数は実際何に影響を与えるのであろうか．第5章で紹介した試作システムを使って実験を行ってみた．

葉ノードの数を，2，4，8としたときの"Monkey and Banana"問題の解が得られるまでの実行時間を，並列知識検索モジュールと並列推論モジュールの割り当てプロセッサを変化させながら測定した結果を，表 7-3，7-4，7-5 に示す．表から，プロセッサ数が少ない場合には，葉ノードの数にはほとんど影響されないが，プロセッサ数が増えるにしたがって，葉ノード数が多いほうがより早く結果が得られるようになることがわかる．ここ

表 7-3 葉ノードが 2 の時の実行時間 単位：秒

	RBU:1	RBU:2	RBU:4
GHC:1	18.31	18.55	20.78
GHC:2	10.87	10.76	10.47
GHC:4	7.50	6.13	6.06
GHC:8	5.98	4.20	4.09

表 7-4 葉ノードが 4 の時の実行時間 単位：秒

	RBU:1	RBU:2	RBU:4
GHC:1	17.67	20.56	21.44
GHC:2	9.66	9.35	10.62
GHC:4	6.56	5.90	5.46
GHC:8	5.59	3.88	3.82

表 7-5 葉ノードが 8 の時の実行時間 単位：秒

	RBU:1	RBU:2	RBU:4
GHC:1	19.00	20.54	20.94
GHC:2	10.78	10.25	8.56
GHC:4	6.46	5.76	5.24
GHC:8	5.31	3.81	3.60

で用いた"Monkey and Banana" の場合は，答えが得られるまでに遷移する状態数は高々 20 前後であるため，それほど大きな影響は現われていないが，問題が大きくなって状態数が増えるほど葉ノードの数が影響すると期待できる．

7．7　まとめと考察

まず，状態空間を並列に効率よく探索するため優良優先探索（Better First Search ）戦略を提案した．これは，各状態の評価関数値により展開の優先順位管理を行う場合に，優先順位決定を空間全体から最良の状態を厳密に選ぶのでなく，ローカルに選定して緩やかに管理する方法である．このすることにより，管理が分散，軽減されるため，処理の並列性を引き出すことができる．

また，その実現方法として，木型のプロセス構成による階層的な優先順位管理機構を提案した．プロセス間で転送される状態情報の内容は，状態へのポインタ（項関係の名前），その状態の評価関数値からなり，プロセス間の転送量を抑えている．この状態情報は，分岐ノードから葉ノードには要求駆動で，葉ノードから分岐ノードにはデータ駆動で流れ，自然にパイプラインを構成する．分岐ノードは，状態情報のあるなし，要求のあるなしで，オートマトンを構成している．葉ノードは，RBUのコマンドを使って項関係と状態情報から状態遷移を行い，パターンマッチ処理の効率化をはかった．

状態の遷移は状態を表す変数への単一代入とみることができるため，更新のための衝突は起こらない．ただ，状態の内容をコピーする必要が生じる．この時，状態の内容の全部をコピーしていたのでは，コピー処理が重くなりすぎるため，データの共通部分を共有し，実際にはタプルのコピーは起こらないようにする方法も考えられる．

第8章　結論

計算機システムを人間により近づけるために，知識ベース指向で処理を進めるいくつかのアプローチについて述べた．全体として，機能の低い方法から高める方向へと議論を進めてきた．

まず最初に，関係データベース中に知識を格納して，関係データベースと一階述語論理の推論機構を組み合わせて知識ベース指向で処理を進める演繹データベースについて，２種類の処理方法を示した．１つは，与えられたゴールから推論を始め，反駁木を根から葉に向かって演繹を進める方法で，演繹データベースでSLD演繹を行う間送評価法である．もう１つは，逆に反駁木を葉から根に向かって演繹を進める方法で，演繹データベースで単位演繹を行う setting 評価法である．間送評価法では，途中結果を保持しておくことによりかなり効率の良い処理が可能となるが，全解を求めようとすると複雑な再帰呼び出しの場合には停止しない場合がある．一方 setting 評価法はホーン節集合に対して健全かつ完全であり，関数記号を含まない場合には全解探索の停止性も保証できる．また，関係データベース側の簡単な拡張により，推論処理との通信を極力抑えることが可能である．

演繹データベースの問題点は，知識をデータベースに格納しているため，構造体や変数を文字列としてしか格納できないことである．知識が構造を持たない単なるデータの場合には，演繹データベースは十分有効であるが，知識内部に構造を持たせるような場合には，機能的に十分とは言えない．そこで，知識として構造を持たせるために，項関係とその上の単一化検索演算を行うモデル（RBU モデル）を提案した．RBU モデルは，知識ベース処理を実現するための関係データベースに対する自然な拡張である．RBU の定式化のために，項関係にホーン節を格納して単一化検索演算を組み合わせることにより，SLD 演繹および選択関数を用いる単位演繹である SUD 演繹が行うアルゴリズムを提案した．いずれのアルゴリズムも，項関係に格納されたホーン節の集合が充足不能の場合には，反駁を求めて停止する．

次に，RBU モデルに基づく知識ベース指向処理システムを実現するために，専用ハードウエアを用いる場合の構成方法，および専用ハードウエアを用いずに汎用プロセッサの上でソフトウエアで実現する場合の高速化手段として専用インデックスの構成方法を示した．専用ハードウエアとしては，単一化エンジンと多ポートページメモリを利用する方法を提案した．特に，単一化結合をハードウエアで効率行うために項の間の順序づけを行う方法を導入した．また，専用インデックスとしては，一種の木構造であるトライとハッシングを組み合わせた方法を提案し，プロトタイプによってその効果を示し

た．知識ベースの場合には，検索だけでなく対話的な更新も重要となってくるが，このインデックスは更新時の維持コストがわずかで済むという特徴を持ち，知識ベースシステムに適していると言える．

さらに知識ベース指向処理を高速に行うために，システムを並列化する方法を提案した．まず，知識検索システム自体を並列化するために，検索／更新コマンド単位で並列実行する同時実行メカニズムを示した．次に，システム全体を並列に動作させるために，並列論理型言語 GHC から並列知識検索システムの並列アクセスを可能とした．このため，複数のコマンドページによるモジュール間の接続方法を導入した．また，GHC から知識検索処理を制御するため，GHC の論理変数を用いた要求駆動のストリーム通信機能を用意し，知識ベース指向処理の記述を容易にした．

そして共有メモリ型マルチマイクロ計算機上に知識ベース指向並列処理システムを試作し，各種の実験を行った．まず，その試作システム上で，並列 SLD 演繹および並列 SUD 演繹を行い，プロセッサ台数にともなって性能が向上することを示した．このとき，並列知識検索モジュールと並列推論モジュールの負荷バランスをとってプロセッサ台数を増やすことが重要であることを確認した．と同時に，SLD 演繹とSUD 演繹の性能上の違い，複数コマンドページの効果を示した．次に，実用処理に比較的近い意味ネットワークの並列探索を行い，処理性能の目安として Prolog システムとの実行時間の比較を行った．知識ベースが大きくなった場合に，特に共有メモリ型の並列システムで処理しようとする時には共有メモリのアクセスネックの面からも，インデックスが重要であることを再確認した．さらに，知識ベース指向並列処理システムの上に診断型や計画型の意思決定支援システムを構築することを目的に，並列状態遷移探索を行う方法を示した．状態遷移探索の並列性を増すために，優良優先探索戦略を提案し，GHC で記述した木型のプロセス構成上で実現した．試作システム上の実験により，その有効性を示した．

このように，知識ベース指向で処理を行う場合、要求される機能レベルによりいろいろな構成方法があることがわかる．本研究により，知識ベースの内容の複雑さ（構造体や変数を含むかどうかなど），知識を扱う制御の複雑さ（再帰呼び出しの複雑さ），利用できる資源（専用ハードウエア／汎用プロセッサ），処理速度に対する要求，および実現したい機能などに合わせてシステムを構成する方法を示すことができた．

謝 辞

本研究をまとめるにあたり，あらゆる面で常に暖かい御指導，御鞭撻を賜わりました東京工業大学工学部 当麻喜弘 教授に心から深く感謝の意を表します．

本論文をまとめるにあたり，適切な御教示ならびに御指導を賜わりました東京工業大学工学部 片山卓也 教授，田中穂積 教授，南谷崇 教授，米崎直樹 助教授に心から厚く感謝の意を表します．

本研究は，第五世代コンピュータプロジェクトの一環として行われた知識ベースシステムの知識ベース指向処理の研究をまとめたものであります．

本研究を進めるに際し，終始暖かい御指導ならびに御援助を賜わりました（財）新世代コンピュータ技術開発機構研究所 淵一博 所長，同研究所 古川康一 次長，同研究所 内田俊一 研究部長に心から厚く感謝の意を表します．

本研究の初期の段階で，適切な御指導と本研究の指針を与えて下さった神奈川大学 村上国男 教授に心から深く感謝の意を表します．

本研究に対しなみなみならぬ御理解と卓越した御見解とを示され，暖かい御指導，御鞭撻を賜わりました名古屋工業大学工学部 伊藤英則 教授に心から厚く感謝の意を表します．

本研究に対し御理解を頂き，終始暖かい御指導を賜わりました（株）富士通研究所 佐藤繁 取締役，同研究所情報処理研究部門 棚橋純一 部門長，同部門人工知能研究部 林弘 部長，富士通（株）宇宙開発推進室第二開発部 相馬行雄 部長に心から厚く感謝の意を表します．

本研究を推進するにあたり，終始暖かい御指導，御助言を頂いた（株）富士通研究所人工知能研究部第三研究室 服部彰 室長，同研究室 北上始 氏に心から感謝いたします．

本論文をまとめるにあたり，御助言を頂いた（株）富士通研究所システム研究部第一研究室 浅川和雄 室長に心から感謝いたします．

本研究の初期の段階で，（財）新世代コンピュータ技術開発機構研究所において，数多くの有意義な議論をして下さり，適切な御助言を頂いた富士通（株）国際情報社会科学研究所第二研究部第二研究室 國藤進 室長，日本電気（株）角田健男 氏，沖電機工業（株）宮崎収兄 氏，キヤノン（株）柴山茂樹 氏に心から感謝いたします．

SUD 演繹の完全性の証明に際し，御助言を頂いた米 Northwestern 大学のHencshen 教授，Setting 評価法および単一化検索モデルを使った演繹の論理的な定式化において，数多くの議論と適切な御助言を頂いた（財）新世代コンピュータ技術開発機構研究所 坂井

公 氏，単一化検索モデルについて熱心に議論をして下さった同研究所 佐藤健 氏，富士通（株）国際情報社会科学研究所 村上昌己 氏，沖電機工業（株）森田幸伯 氏，工業技術院電子技術総合研究所 西田健次 氏，知識ベース指向システムの試作，同システムの並列化において協力して下さった富士通ソーシアルサイエンスラボラトリ（株）山崎光彦 氏，竹中伸一 氏，関光彦 氏，北島由蔵 氏，本田秀行 氏に心から深く感謝いたします．

本研究を進めるにあたり，その他様々な面で御協力を頂いた（財）新世代コンピュータ技術開発機構研究所ならびに（株）富士通研究所情報処理研究部門人工知能研究部の関係諸氏に感謝いたします．

最後に本研究をまとめるにあたり，常に理解を示し，支えてくれた妻の横田かおるに心から感謝します．

参考文献

[Aho-Ullman 79] Aho, A. V. and Ullman, J. D. : Universality of Data Retrieval Language, *Proc. of ACM/SIGPLAN Conf. on Programming Languages*, pp. 110-117, (1979).

[Anderson-Bledsoe 70] Anderson, R. and Bledsoe, W.W.: A Linear Format for Resolution With Merging and a New Technique for Establishing Completeness, *Journal of the ACM*, Vol. 17, No. 3 (1970) .

[Apt-Emden 82] Apt, K.R. and van Emden, M.H.: Contributions to the Theory of Logic Programming, *Journal of the ACM*, Vol. 29, No. 3 (1982) .

[Barr-Feigenbaum 81] Barr, A. and Feigenbaum, E. A.: *The Handbook of Artificial Intelligence*,1, William Kaufmann, Inc. (1981).

[Boral et al. 82] Boral, H., DeWitt, D. J. Friedland, D. Jarrel, N. F., and Willkinson, W. K. : Implementaion of the Database Machine DIRECT, *IEEE Transcations on Software Engineering*, Vol. SE-8, No. 6, (1982).

[Bowen-Kowalski 82] Bowen, K. A. and Kowalski, R. A. : Amalgamating Language and Metalanguage in Loginc Programming, *Logic Programming*, ed. Clark K. L. and Tarlund S - A., Academic Press, (1982).

[Chang-Lee 73] Chang, C. L. and Lee, R. C. T.: *Symbolic Logic and Mechanical Theorem Proving*, p. 330, Academic Press (1973).

[Chang 81] Chang, C. L.: On Evaluation of Queries Containing Derived Relations in a Relational Data Base, *Advanced in Data Base Theory*, Vol. 1, ed. Gallaire et al., PLENUM, pp235-260 (1981).

[Chakravarthy et al. 82] Chakravarthy, U. S., Minker, J., and Tran, D. : Interfacing Predicate Logic Languages and Relational Databases, *Proc. of the First International Logic Programming Conference*, pp.91-98, (1982).

[Clark-Gregory 86] Clark, K. L. and Gregory, S.: PARLOG: Parallel Programming in Logic, *ACM Transaction on Programming Language System*, Vol. 8, No. 1, pp. 1-49 (1986).

[Coelho et al. 80] Coelho, H., Cotta, J. C., and Pereira, L. M. : *HOW TO SOLVE IT WITH PROLOG*, 2nd edition, Laborato'rio Nacional de Engenharia Civli, (1980).

[Earley 70] Earley, J.: An Efficient Contexfree Parsing Algorithm, *Communication ACM*, Vol. 13, No. 2, pp.94-102 (1970).

[Forgy 82] Forgy, C. L.: Rete: A Fast Algorithm for the Many Pattern / Many Objerct Pattern Match Problem, *Artificial Intelligence*, Vol.19, (1982).

[淵他 87] 淵一博監修，古川康一，溝口文雄供編：*並列論理型言語 GHC とその応用*，知識情報処理シリーズ第 6 巻，p. 277，共立出版 (1987).

[Gallaire 83] Gallaire, H.: Logic Data Bases vs. Deductive Data Bases, *Proc. of Logic Programming Workshop*, pp. 608-622, (1983).

[Gallaire et al. 84] Gallaire, H., Minker, J., and Nicolas, J-M.: Logic and Databases: A Deductive Approach, *Computer Surveys*, 16, 2 (1984) .

[Gupta et al. 86] Gupta, A., Forgy, C. L., Newell, A., and Wedig, R.: Parallel Algorithms and Architectures for Rule-Based Systems, *Proc on 13th International Symposium on Computer Architecture*, pp. 28-37 (1986).

[Gupta 87] Gupta, A.: *Parallelism in Production Systems*, Morgan Kaufmann Publishers, Inc., (1987).

[Halstead 85] Halstead, R.: MultiLisp: A Language for Concurrent Sybolic Computation, *ACM TOPLAS*, Vol. 7, No. 4, pp. 501-538 (1985).

[Henschen-Naqvi 84] Henschen, L. J. and Naqvi, S. A.: On Computing Queries in Recursive First-Order Databases, *Journal of the ACM*, Vol. 31, No. 1, pp. 47-85, (1984).

[Hillyer-Shaw 1986] Hillyer, B. K. and Shaw, D.E.: Execution of OPS 5 Produciton Systems on a Massively Parallel Machine, *Jurnal of Parallel and Distributed Computing*, Vol.3 , No. 2, pp. 236-268, (1986).

[樋口他 88] 樋口哲也，古谷立美，半田剣一，楠本博之，国分明男：意味ネットワークマシン（IXM）プロトタイプの開発，*情報処理学会「知識工学と人工知能研究会」資料*，61-4, (1988).

[Itoh et al. 87] Itoh, H., Takewaki, T., and Yokota, H. : Knowledge Base Machine Based on Parallel Kernel Language, *Proc. of 5th International Workshop on Database Machines*, pp. 15-28 (1987).

[Itoh et al. 88] Itoh H., Monoi H., Shibayama S., Miyazaki N., Yokota H., and Konagaya A.: Knowledge Base Subsystem in Logic Programming Paradigm, *Proc. of Int'l Conf. on FGCS 1988*, pp. 37-57 (1988).

[伊藤他 89] 伊藤英則，物井英俊，世木博久，柴山茂樹，宮崎収兄，横田治夫，小長谷明彦：ロジックプログラミングパラダイムにおける知識ベースシステム，人工知能学会誌，Vol. 4, No. 3, pp. 272-279, (1989).

[Kakuta et al. 85] Kakuta, T., Miyazaki, N., Shibayama, S., Yokota, H., and Murakami, K. : The Design and Implementation of Relational Database Machine Delta, *Proc. of the International Workshop on Database Machines '85*, (1985).

[Kim et al. 88] Kim, W., Ballou, N., Chou, H-T., Garza, J. F., and Woelk, D.: Integration an Object-Oriented Programming System with a Database System, *Proc. of Object Oriented Programming Systems, Languages and Applications* '88, pp. 142-152 (1988).

[Kitakami et al. 84] Kitakami, H., Kunifuji, S., Miyachi, T., and Furukawa, K. : A Methodology for Implementation of A Knowledge Acquisition system, *Proc. of the 1984 International Symposiumon Logic Programming*, (1984).

[北上他 88] 北上始，横田治夫，服部彰：知識処理向き並列推論エンジン，*電子情報通信学会研究会資料*，CPSY, 88-50(1988).

[Knuth 73a] Knuth, D. E. : *The Art of Computer Programming*, 1, Fundamental Algorithm, p. 634, Addison-Wesley (1973).

[Knuth 73b] Knuth, D. E.: *The Art of Computer Programming*, 3, Sorting and Searching, p. 723, Addison-Wesley (1973).

[Kowalski 79] Kowalski, R. A: Logic and Semantic Network, *Communication of the ACM*, Vol. 22, No. 3, pp185-192, (1979).

[小山-田中 85] 小山晴生，田中穂積：Definite Clause Knowledge Representation, *Proc. of the Logic Programming Conference '85*, pp95-106 (1985).

[Kunifuji-Yokota 82] Kunifuji, S. and Yokota, H. : PROLOG and Relational Data Base for Fifthe Generation Computer Systems, *Proc. of ONERA-CERT Workshop on "Logic Bases for Data Bases,"* ed. Gallaire et al., 1982.

[Lloyd 84] Lloyd, J. W.: *Foundation of Logic Programming*, p. 124, Springer-Verlag (1984).

[Loveland 78] Loveland, D.W.: *Automated Theorem Proving*, North-Holland, (1978).

[Miyachi et al. 84] Miyachi, T., Kunifuji, S., Kitakami, H., Furukawa, K., Takeuchi, A, and Yokota, H.: A Knowledge Assimulation Method for Logic Database, *Proc. of the 1984 International Symposium on Logic Programming*, (1984).

[Miyazaki et al. 87] Miyazaki, J., Amano, H., Takeda, K., and Aiso, H.: A Shared Memory Architecuter for MANJI Production System Machine, 5th International Workshop on Database Machines, pp. 354-368 (1987).

[Morita et. al 86] Morita, Y.,Yokota, H., Nishida, K., and Itoh, H. :Retrieval-By-Unification Operation on a Relational Knowledge Base, *Proc. of 12th International Conference on VLDB*, pp. 52-59 (1986).

[Murakami et al. 83] Murakami, K., Kakuta, T., Miyazaki, N., Shibayama, S., and Yokota, H. : A Relational Data Base Machine: First Step To a Knowledge Base Machine, *Proc of 10th Annual International Symposium on Computer Architecture*, pp. 423-426, (1983).

[Nilsson 80] Nilsson, N. J. : *Principles of Artificial Intelligence*, p. 476, Springer-Verlag, (1980).

[Ohmori-Tanaka 87] Ohmori, T. and Tanaka, H. : An Algebraic Deductive Database Managing a Mass of Rule Clauses, *Proc. of 5th International Workshop on Database Machines*, pp. 291-304 (1987).

[Okumura-Matsumoto 87] Okumura, A. and Matsumoto, Y.: Parallel Programming with Layered Streams, *Proc. 1987 Sympsium on Logic Programming*, pp. 224-231 (1987).

[Okuno 83] Okuno, H. : The Execution Mechanism for Logic Programming Language to Perform an Effieient retrieval of All Solutions, *Proc of the Logic Programming Conference '83*, (1983).

[小沢他 89] 小沢年弘，細井聡，服部彰：FGHC処理システムのメモリ使用特性と世代別ガーベージ・コレクション，*情報処理学会論文誌*, Vol. 30, No. 9, pp. 1182-1188 (1989).

[Reiter 78] Reiter, R. :Deductive Question-Answering on Relational Data Bases, *Logic and Data Bases*, ed. Gallaire and Minker, PLENUM, pp23-41, (1978).

[Robinson 65] Robinson, J. A. : A machine-oriented logic based on the resolution principle, *Journal of the ACM*, Vol. 12, No. 1, pp23-41 (1965).

[Sabbatel et al. 84] Sabbatel, G. B., Dang, W., Ianeselli, J. C., and Nguyen, G. T.: Unification for a Prolog Data Base Machine, *Proc. of the Second International Logic Programming Conference*, pp.207-217, (1984).

[Shapiro 86] Shapiro, E. Y. : Concurrent Prolog: A Progress Report, *Computer*, Vol. 19, No. 8, pp. 44-58 (1986).

[Shibayama et al. 87] Shibayama, S., Sakai, H., Monoi, H., Morita, Y., and Itoh, H. : Mu-X: An Experimental Knowledge Base Machine with Unification-Based Retrieval Capability, *Proc. on France-Japan Artificial Intelligence and Computer Symposium 87*, pp. 343-357, (1987).

[Stolfo 84] Stolfo, S. Five Algorithms for Production Systems Execution on the DADO machine, *AAAI'84*, pp.300-307, (1984).

[Tanaka 84] Tanaka, Y.: A Multiport Page-Memory Architecture and A Multiport Disk-Cache System, *New Generation Computing 2*, pp.241-260 (1984).

[Ueda 86] Ueda, K. : Guarded Horn Clauses, *Logic Programming '85*, E. Wada (ed). Lecture Notes in Computer Science 221, Springer-Verlag (1986).

[Ueda 87] Ueda, K.: Making Exhaustive Search Programs Deterministic, *New Generation Computing*, Vol.5, No. 1, pp.29-44 (1987).

[Ullman 82] Ullman, J. D. : *Principals of Database Systems*, 2nd ed., p. 484, Computer Science Press, Potomac, Md. (1982).

transaction on Database Systems, Vol. 10, No. 3, pp289-321, (1985).

[Vieille 87] Vieille, L. : A Database-complete Proof Procedure Based on SLD-resolution, *Proc. of the 4th International Conference on Logic Programming*, pp.74-103 (1987).

[Wada et al. 87] Wada, M., Morita, Y., Yamazaki, H., Yamashita, S., Miyazaki, N., and Itoh, H.: "A Superimposed Code Scheme for Deductive Databases", *Proc. of 5th International Workshop on Database Machines,* pp. 569-582, (1987).

[安浦他 85] 安浦寛人，大久保雅且，矢島修三：論理型言語の単一化操作のためのハードウェアアルゴリズム，*信学技報,* EC-84-67,pp.9-20,(1985).

[横田-西尾 90] 横田一正，西尾章治郎：演繹・オブジェクト指向データベース，*情報処理学会誌,* Vol. 31, No. 2 , pp.234-243(1990).

[横田他 83] 横田治夫，角田健男，宮崎収兄，柴山茂樹，村上国男：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（昭和58年後期）全国大会*，2K-7 (1983)

[Yokota et al. 84a] Yokota, H., Kunifuji, S., Kakuta, T., Miyazaki, N., Shibayama, S, and Murakami, K.: An Enhanced Inference Mechanism for Generating Relational Algebra Queries, *Proc. of the 3rd ACM SIGACT-ASIGMOD Sympo. on Principles of Database Systems,* pp. 229-238 (1984) .

[横田他 84] 横田治夫，国藤進，柴山茂樹，宮崎収兄，角田健男，村上国男：Prolog による推論機構と関係データベースの結合，*情報処理学会「知識工学と人工知能研究会」資料,* 32-2, (1984).

[Yokota et al. 84b] Yokota, H., Kakuta, T., Miyazaki, N., Shibayama, S., and Murakami, K.: Unification in A Knowledge Base Machine, *情報処理学会第29回（昭和59年後期）全国大会*, 3F-2 (1984).

[横田他 85a] 横田治夫，安部公朗，森田幸伯，伊藤英則：単一化による知識ベース検索，*昭和60年度電子通信学会情報システム部門全国大会,* (1985).

[横田他 85b] 横田治夫，物井秀俊，森田幸伯，伊藤英則：マルチポート・ページメモリの構成方法，*昭和60年度電子通信学会情報システム部門全国大会,* (1985).

[Yokota et al. 86a] Yokota, H., Sakai, K., and Itoh, H.: Deductive Database System based on Unit Resolution, *Proc. of the 2nd Int'l. Conf. on Data Engineering,* pp. 228-235 (1986) .

[Yokota-Itoh 86] Yokota, H. and Itoh, H.: A Model and an Architecture for a Relational Knowledge Base, *Proc. of 13th International Symposium on Computer Architecture*, pp. 2-9 (1986) .

[横田-伊藤 86] 横田治夫，伊藤英則：知識ベースシステム，*計測と制御,* Vol.25, No.4 ,pp. 53-58 (1986).

[Yokota et al. 86b] Yokota, H., Murakami, M., Morita, Y., and Itoh, H. : A Relational Knowledge Base with Retrieval-by-Unification Operation, *情報処理学会第32回（昭和61年前期）全国大会*, (1986).

[横田他 88a] 横田治夫，山崎光彦，北上始：項関係における高速検索手法，*情報処理学会「データベース・システム研究会」資料*，66-5, (1988). {情報処理学会論文誌投稿中}

[横田他 88b] 横田治夫，北上始，服部彰：項インデクスによる知識検索の高速化，*情報処理学会第37回（昭和63年前期）全国大会*，(1988).

[Yokota et al. 88] Yokota ,H., Kitakam,i H., and Hattori ,A.: Knowledge Retrieval and Updating for Parallel Problem Solving, *ICOT Technical Report TR-380* (1988).

[Yokota et al. 89] Yokot,a H., Kitakami, H., and Hattori, A. : Term Indexing for Retrieval by Unification, *Proc. of 5th Int'l Conf. on Data Engineering*, pp. 313-320 (1989).

[横田他 90a] 横田治夫，北上始，服部彰：知識ベース指向並列処理システム，*情報処理学会論文誌*，Vol. 31，No. 11，掲載予定 (1990).

[横田他 90b] 横田治夫，北上始，服部彰：項関係上での単一化検索を使ったホーン節推論アリゴリズム，*情報処理学会論文誌*，Vol. 31，No. 10，掲載予定 (1990).

[Yokote-Tokoro 88] Yokote, Y. and Tokoro, M.: Concurrent Programming in ConcurrentSmalltalk, *Object-Oriented Concurrent Programming*, Yonezawa A. and Tokoro M. (ed.), pp.129-158, MIT Press(1988).

[Yonezawa et al. 88] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y.: Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, *Object-Oriented Concurrent Programming*, Yonezawa A. and Tokoro M. (ed.), pp.55-90, MIT Press (1988).