データベース・システム 66 － 5
（１９８８．７．２２）

項関係における高速検索手法

横田 治夫† 山崎 光彦‡ 北上 始†
†（株）富士通研究所
‡（株）富士通ソーシアルサイエンスラボラトリ

本稿では、単一化 (Unification) を使って知識ベースの検索をする RBU(Retrieval By Unification) 演算の高速化の手段として、項関係に対するインデクスの実現方法について報告する。項関係とは、変数も取り扱うことが可能な構造体である項を格納したテーブルのことで、 RUB 演算とは関係代数演算に単一化を導入して項関係から適当な項を検索するものである。ここで提案するインデクスは、ハッシングとトライ (Trie) 構造と呼ばれる一種の木構造を組み合わせて RBU における比較処理ならびにバックトラック処理の発生を抑えるものである。試作したプロトタイプの検索と更新に要する時間を計測し、インデクスの検索処理における高速化の効果を確かめると共に、更新処理におけるインデクスの維持のためのオーバヘッドがわずかであることを示す。

An Accelerated Retrieval Method for Term Relations

Haruo Yokota † Mitsuhiko Yamazaki ‡ Hajime Kitakami †
† FUJITSU LABORATORIES LTD.
1015, Kamikodanaka Nakahara-ku, Kawasaki 211, Japan
‡ FUJITSU SOCIAL SCIENCE LABORATORY Ltd.
Shinosaki Bldg. 6-4 Osaki 1-Chome Shinagawa-ku, Tokyo 141, Japan

This paper presents a method for indexing terms in a knowledge-base retrieval-by-unification (RBU) system. The term is a well-defined structure capable of handling variables to represent knowledge. RBU operations are an extension of relational database operations using unification and backtracking to retrieve terms from term relations. The term indexing we propose uses hashing and trie structures to reduce the number of comparisons between elements of a search condition and of an object term relation. Unification on a trie structure is suited to backtracking bindings of variables. The search and updating speed of an RBU prototype is measured to evaluate the indexing method. This method is effective in fast term retrieval for a large number of similar and varied form terms. The overhead for maintaining indexes in updating is low.

1. はじめに

RBU(Retrieval By Unification) 演算は、第五世代コンピュータ・プロジェクトにおいて、知識ベースシステムの演算の一つとして提案された。[1] 関係データベースにおける関係代数演算を知識検索用に拡張している。変数を含んだ構造体である項で知識を表現し、項の集合である項関係と呼ばれるテーブルから検索条件に単一化 (Unify) 可能な項を選択してくる。関係データベースが互いに関連し合った大量のデータに対して検索しやすい環境を与えたように、RBU もそれぞれに関係を持った大量の知識の塊をうまく取り扱うことを目指している。

RBU 演算による知識ベース処理の例の1つとして、RBU 演算の繰り返しによる SLD 演繹の実現がある。[1] SLD 演繹は、一階述語論理の一部であるホーン節論理の演繹アルゴリズムの一つで、Prolog の実行メカニズムの基にもなっている。[2] このため、ホーン節で意味ネットワーク的な知識を表現する DCKR[3] などを利用することにより、RBU 演算の組み合わせで階層的な知識を検索することができる。また、並列論理型言語の一つである GHC(Guarded Horn Clauses)[4] と RBU 演算を組み合わせて並列プロダクション・システムを実現する方法も提案されている。[5] GHC はストリームを使った並列処理を記述するのには強力な言語であるが、知識ベースのようなグローバルなデータを扱うのは得意でない。プロダクション・ルールやワーキング・メモリの内容を項関係に格納して RBU 演算で検索し、検索結果を GHC で並列に処理することにより効率の良い並列プロダクション・システムを実現することができる。

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、単一化エンジンを用いる方法[1,6,7] など専用ハードウェアを使った方法も提案されているが、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について検討する。

Prolog システムでは、インデクスとしてハッシングを使っている。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。[8] 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 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。

本稿では、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。まず第2章で、項関係と RBU 演算に関する定義を行い、第3章でインデクスの実現方法について述べる。また、インデクスの検索ならびに更新における影響を見るために試作したプロトタイプの評価について第4章で報告する。

2. 項関係とRBU演算

項の定義は一階述語論理[9] のそれと同じであるが、ここでは項関係を定義するため集合を使う。

【定義】F_n をn引数の関数記号の有限集合、 V を次の式を満足する変数の可算無限集合とする。

$$\forall n,\ F_n \cap V = \emptyset.$$

【定義】次のような T を項集合 (Term Set) と呼び、その要素 t を項 (Term) 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 $t_1, \ldots, t_n \in T$, $f \in F_n$ $(n \geq 1)$ ならば

$$f(t_1,\ldots,t_n) \in T$$

【定義】 $T_1, T_2, \ldots, T_m$ を項集合としたとき、次のような TR_m を m 属性の項関係 (Term Relation) と呼ぶ。

$$TR_m \subset T_1 \times T_2 \times \cdots \times T_m \ (m \geq 1).$$

このとき次のような tt_m を項タプル (Term Tuple) と呼ぶ。

$$tt_m = (t_1, t_2, \ldots, t_m) \in TR_m$$

項タプル tt の i 番目のアイテムを $tt[i]$ で表す。

6つの項タプルからなる2属性の項関係の例を図1に示す。本稿では、変数を大文字で始まる記号で表すことにする。図の先頭の項タプルの第1属性にある項 $p(X,g(Y))$ の中の変数 X が、項 $f(a,b)$ に束縛された場合、変数のスコープからこの項タプルの第2属性の項 $r(X,Y)$ は $r(f(a,b),Y)$ となる。このような変数束縛 $\theta = \{f(a,b)/X\}$ を代入 (Substitution) と呼ぶ。

$p(X,g(Y))$	$r(X,Y)$
$q(f(a,X),g(X))$	$r(f(a,X),X)$
$p(X,g(b))$	$r(h(a,b),f(a))$
$q(f(X,Y),g(c))$	$s(X,g(Y,c))$
$p(f(a,b),h(X))$	$s(a,g(b,c))$
$p(f(a,X),h(X))$	$s(a,X)$

図1 項関係の例

項関係上の演算は、関係代数[10]を基にして、単一化を導入して拡張している。関係代数には、集合和 (Union) 、制約 (Restriction)、射影 (Projection)、結合 (Join)、などの演算があり、RBU のプロトタイプではそれらのほとんどを実現し、結合と制約について単一化による拡張を行った。また、更新系の演算や定義系の演算についても実現した。ここでは、インデクスに焦点を絞るために、単純な選択演算である制約について述べることにする。

【定義】代入 θ は、$t_1\theta=t_2\theta$ となるとき、またそのときに限って、項 t_1 と t_2 の単一化作用素 (Unifier) と呼ばれる。

【定義】項 t_1 と t_2 の単一化作用素 σ は、それらの項の任意の単一化作用素 θ に対してある代入 λ が存在し、$\theta=\sigma\bullet\lambda$ となるとき、またそのときに限ってそれらの項の最汎単一化作用素 (Most General Unifier) と呼ばれる。

【定義】単一化制約 (Unification-Restriction) とは、ある項関係 TR_m とその i 番目の属性に対する検索条件の項 t から次のような新しい項関係 TR'_m を作る演算である。

$$tt'_m \in TR'_m \Leftrightarrow \exists tt_m \in TR_m,\ \exists\sigma,\ tt_m[i]\sigma=t\sigma,\ tt'_m[k]=tt_m[k]\sigma \quad (1\leq k\leq m)$$

ここで、σ は $tt_m[i]$ と t の最汎単一化作用素である。

図1の項関係に対して、第1属性が検索条件 $p(f(A,c),B)$ と単一化可能であるような項を検索してくる単一化制約を行なった場合、1、3、6番目の項タプルが導かれ、図2に示すような項関係が生成される。ここで、それぞれの項タプルに対する最汎単一化作用素は、$\{f(A,c)/X,\ g(Y)/B\}$、$\{f(A,c)/X,\ g(b)/B\}$、$\{c/X,\ a/A,\ h(c)/B\}$ である。

$p(f(A,c),g(Y))$	$r(f(A,c),Y)$
$p(f(A,c),g(b))$	$r(h(a,b),f(a))$
$p(f(a,c),h(c))$	$s(a,c)$

図2　単一化制約の結果

3. 知識検索システムの実現方式

3. 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ことになる。図3に項 $p(f(a,b),h(X))$ に対応する木構造を示す。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。

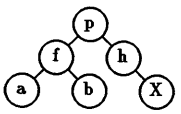

図3　項 $p(f(a,b),h(X))$ の木構造

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。[11] 図3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3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次のようなセルを使うことにする。

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

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。

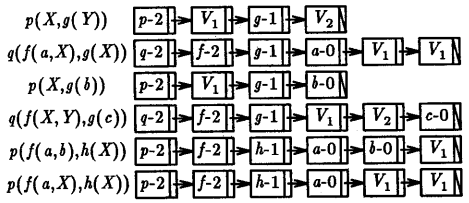

図4　LOSR に対するセル構造

図4は、図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。

3．2　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。[12] 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[5]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、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。[12] 一般にはトライ構造は、数値や文字の列を格納するのに使われるが、ここでは項の LOSR の要素に適用してインデクスに利用する。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に示した項の集合をトライ構造に変換すると図5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。

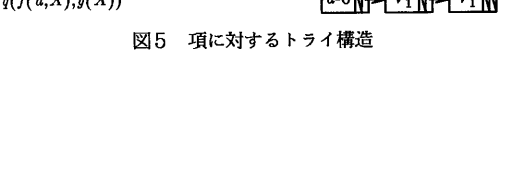

図5　項に対するトライ構造

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と図5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）]

<開始>

p-2, V_1(=f-2), g-1 <fail>

q-2 <fail>

p-2, V_1(=f-2), g-1 <fail>

q-2 <fail>

p-2, f-2, h-1, a-0, b-0, V_1(=c-0) <success>

p-2, f-2, h-1, a-0, V_1(=b-0), V_1 <fail>

<終了>

[トライ構造の場合（図5）]

<開始>

p-2, V_1(=f-2), g-1 <fail>

f-2, h-1, a-0, V_1(=b-0), V_1 <fail>

b-0, V_1(=c-0) <success>

q-2 <fail>

<終了>

このトレースで分かるように、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 $N+M$ 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6）。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。

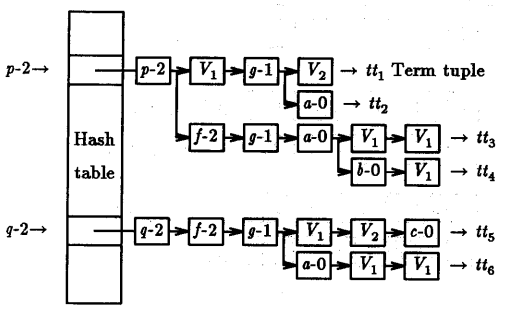

図6　ハッシングとトライ構造を組み合わせたインデクス

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トライ構造を使うわけである。

3．3　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、$[p\text{-}2,\ V_1,\ g\text{-}1,\ V_2]$ と $[p\text{-}2,\ f\text{-}2,\ g\text{-}1,\ a\text{-}0,\ b\text{-}0,\ c\text{-}0]$ となる。LOSR どうしで単一化を行うためには、1番目の変数 V_1 ($= X$) には、単に対応する位置にある $f\text{-}2$ だけでなく、その引数である $[a\text{-}0,\ b\text{-}0]$ も代入される必要がある。また、2番目の変数 V_2 ($= Y$) には、そのままでは対応する位置にない要素 $c\text{-}0$ を代入しなければならない。これらのためには、$[p\text{-}2,\ V_1,\ g\text{-}1,\ V'_1,\ V''_1,\ V_2]$ 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: IF W_A と W_B の両方が空 THEN
　単一化成功
　ELSE
　W_A と W_B の先頭からフラグ A と B を取る。

Step3: Case1: 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
　IF $E_A = F_A\text{-}K_A$ かつ $E_B = F_B\text{-}K_B$ THEN
　　IF $F_A = F_B$ かつ $K_A = K_B$ THEN
　　K 個の n を W_A と W_Bに入れ、Step2 へ
　　ELSE
　　単一化失敗
　ELSE IF $E_{A/B} = F\text{-}K$ かつ $E_{B/A}$ が変数 V_i THEN
　　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
　ELSE IF E_A と E_B 両方とも変数 THEN
　　変数どうしで代入して、Step2 へ
　Case2: 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
　IF $E_{A/B} = F\text{-}K$ THEN
　　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
　ELSE IF $E_{A/B}$ が $F\text{-}K$ を代入された変数 V_j THEN
　　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
　ELSE IF $E_{A/B}$ が何も代入されていない変数 V_j THEN
　　V_j に V_i を代入し、Step2 へ
　Case3: 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

[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7に示す。

3．4　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7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。

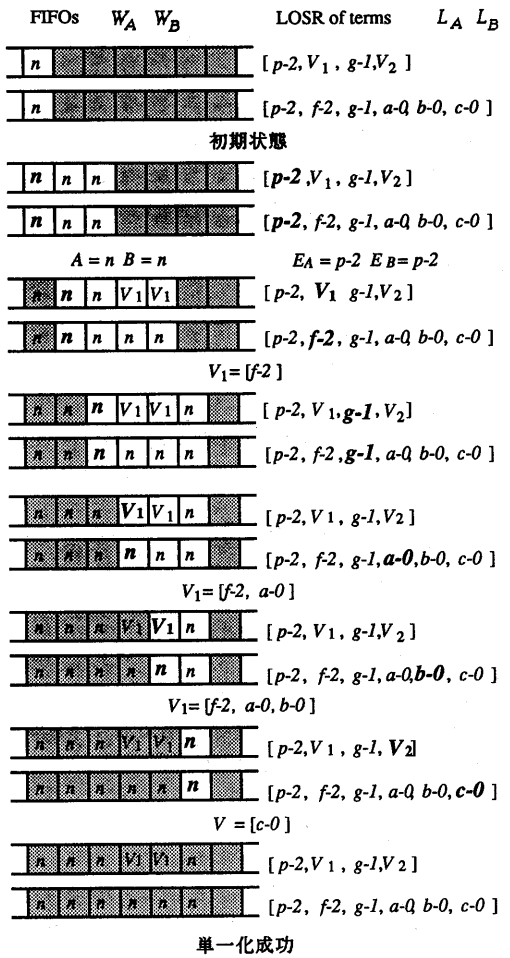

図7　ＬＯＳＲ上の単一化過程

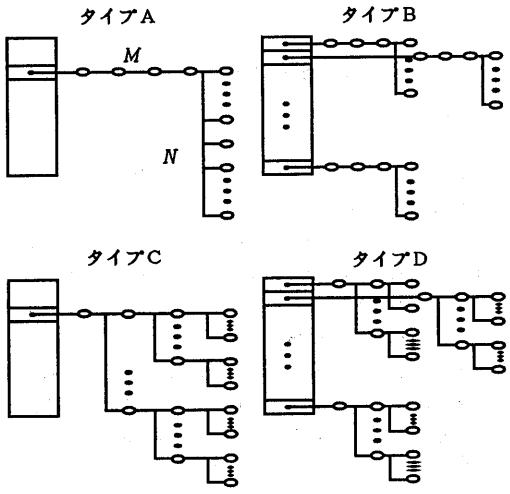

図8　各タイプに対するインデクスの形態

4．性能評価

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8

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。

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9、10、11、12に示した。

図9は、3．2節で述べた、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10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11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、

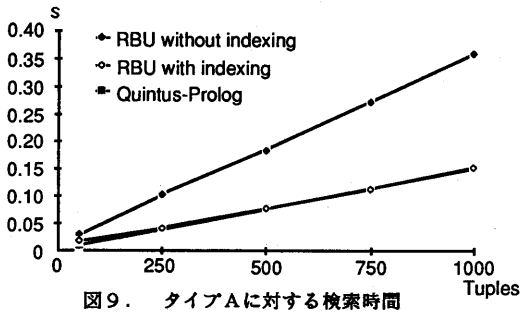

図9． タイプAに対する検索時間

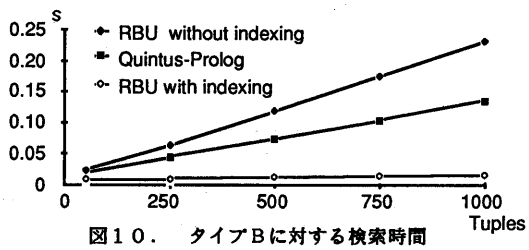

図10． タイプBに対する検索時間

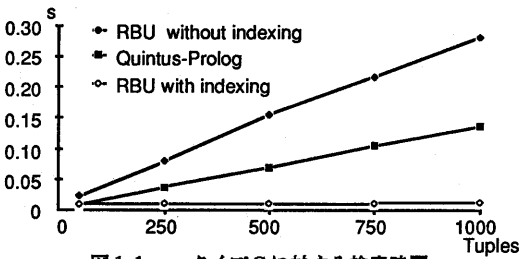

図11． タイプCに対する検索時間

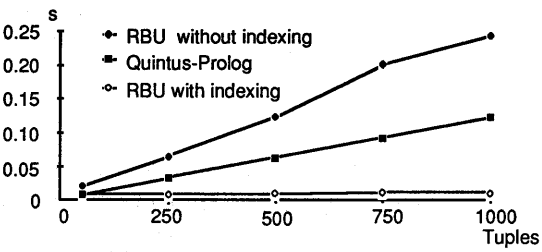

図12． タイプDに対する検索時間

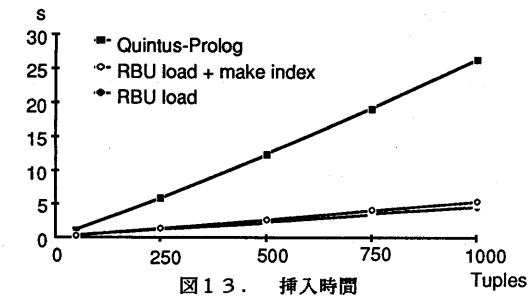

図13． 挿入時間

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１２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１３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。

5. まとめ

RBU システムの高速化の手段として、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。我々は 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。

謝辞

日頃、御指導御助言を頂くＩＣＯＴ伊藤第三研究室長、富士通研究所、林人工知能研究部長、服部人工知能第三研究室長、ならびにＩＣＯＴのＫＢＭメンバーに感謝いたします。

参考文献

1. H. Yokota and H. Itoh, "A Model and Architecture for a Relational Knowledge Base," *Proc. of the 13th International Symposium on Computer Architecture,* pp. 2-9,

2. J. W. Lloyd, *Foundation of Logic Programming,* Springer-Verlag, 1984.

3. 小山、田中：Definite Clause Knowledge Representation, *Proc. of the Logic Programming Conference '85,* pp95-106, 1985.

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

5. H. Yokota, H. Kitakami, and A Hattori, "Knowledge Retrieval and Updating for Parallel Problem Solving," *ICOT Technical Report TR-380,* 1988.

6. Y. Morita, H. Yokota, K. Nishida, and H. Itoh, "Retrieval-By-Unification Operation on a Relational Knowledge Base," *Proc. of 12th International Conference on VLDB,* pp. 52-59, 1986.

7. H. Itoh, T. Takewaki, and H. Yokota, "Knowledge Base Machine Based on Parallel Kernel Language," *Proc. of 5th International Workshop on Database Machines,* pp. 15-28, 1987.

8. T. Ohmori and H. Tanaka, "An Algebraic Deductive Database Managing a Mass of Rule Clauses," *Pro c. of 5th International Workshop on Database Machines,* pp. 291-304, 1987.

9. C. L. Chang and R. C. T. Lee, *Symbolic Logic and Mechanical Theorem Proving,* Academic Press, 1973.

10. J. D. Ullman, *Principals of Database Systems,* 2nd ed., Computer Science Press, Potomac, Md., 1982.

11. D. E. Knuth, *The Art of Computer Programming,* 3, Sorting and Searching, Addison-Wesley, 1973.

12. D. E. Knuth, *The Art of Computer Programming,* 1, Fundamental Algorithm, Addison-Wesley, 1973.