Vol. 36 No. 2 情報処理学会論文誌 Feb. 1995

トランザクション論理におけるプログラム変換

磯崎秀樹† 勝野裕文†

データベースや論理型プログラムで起きる状態変化を明快かつ宣言的に記述する論理として，Bonner と Kifer はトランザクション論理を提案した．トランザクション論理には逐次連言と古典連言という二種類の連言がある．古典連言はプランニングなどに有用であるが，その実装方法はまだ提案されていない．そこで我々は古典連言に対処するため逐次連言だけを扱える Prolog 風のインタプリタを拡張して古典連言も扱えるようにした．しかし，古典連言を使用すると効率が悪くなりやすい．本論文ではプログラム変換を用いてプログラムの実行効率を向上させる方法を示す．Prolog では展開／畳み込みで再帰的述語を変換できたが，トランザクション論理では古典連言が畳み込みを阻止してしまう．そこで「綴じ合わせ」という新しい操作を導入して畳み込みを可能にする．また展開で生成されるルールのうち無駄なものを発見し，除去する方法を示す．そしてこれらの操作がプログラムの等価性を保存することを証明する．実験によってこの変換を適用したプログラムでは，実行時間が数倍向上することが確認できた．本手法により，宣言的表現で生じる計算コストを減らせるので，状態変化を伴う世界について宣言的で読みやすいプログラムが書きやすくなる．したがって，動的世界と相互作用する様々な推論システムの実装や解析が容易になる．

Transformation of Programs in Transaction Logic

HIDEKI ISOZAKI† and HIROFUMI KATSUNO†

Transaction Logic, proposed by Bonner and Kifer, clearly represents phenomena of state changes in logic programs and databases. Transaction Logic has two kinds of conjunctions: serial and classical. Classical conjunction is useful for planning and other applications, but its implementation has not yet been proposed. Therefore, we extended a Prolog-like interpreter for serial conjunction to accept classical conjunction. Since the use of classical conjunction often leads to inefficiency, we show methods for improving the efficiency of programs through program transformation. In Prolog, unfolding and folding are useful to transform recursive predicates. In Transaction Logic, however, classical conjunction impedes the application of folding. Therefore, we introduced a new operation called *zipping*, which enables the application of folding. We also developed a method to detect and remove meaningless rules generated by unfolding. We prove that these operations preserve the semantics of programs. Experiments show that program transformation reduces execution time considerably. Our method enables us to write declarative and readable programs. The transformation system reduces extra computation costs caused by declarative representation. Together these facts ease the implementation and analysis of various advanced reasoning systems.

1. はじめに

データベースや論理型プログラムで起きる状態変化を明快かつ宣言的に記述する論理として，Bonner と Kifer[1)~3)] はトランザクション論理 $\mathcal{TR}$ を提案した．そして，彼らは様々な領域へトランザクション論理が応用可能であることを示した．我々はマルチエージェント環境で各エージェントの推論と動作を制御するための核言語の一つとして $\mathcal{TR}$ を採用することを検討している．

トランザクション論理には逐次連言と古典連言という二種類の連言がある．古典連言はプランニングなどに有用[3)]であるが，その実装方法はまだ提案されていない．そこで我々は逐次連言だけを扱える Prolog 風のインタプリタを拡張して古典連言も扱えるようにした[6)]．古典連言を用いるとより柔軟な記述ができるが，宣言的記述なので実行効率はよくないことが分かった[6)]．

本論文では，プログラム変換の手法を使うことによってトランザクション・プログラムの実行効率を改善する方法を示す．本論文で提案する変換操作は，展開，畳み込み，綴じ合わせの3種類である．これらの中で，展開と畳み込みは Prolog プログラムに対して提案されたものの拡張である．特に，展開には，プロ

† NTT 基礎研究所 情報科学研究部
NTT Basic Research Laboratories

グラムを解析してトランザクションを実行するのに必要な状態遷移回数を予測することによって，実行時に発生する無駄な規則の組み合わせを除去する機能を組み込む．また Prolog プログラムでは展開と畳み込みを用いて再帰的プログラムを効率よい形に変換できたが，$\mathcal{TR}$ では再帰的プログラムを展開しても，古典連言と逐次連言が入り組んで，畳み込めない場合がある．そこで**綴じ合わせ**という新しい操作を導入し，古典連言と逐次連言の適用順序を入れ替えることによって，畳み込みができるように書き換える．そして展開，畳み込み，綴じ合わせ変換がプログラムの等価性を保存することを証明する．実際のプログラムに変換を施して実行効率の改善を計測したところ，5倍程度の改善が得られた．

以下，2章では Bonner と Kifer が提案した $\mathcal{TR}$ の概要と，本論文で仮定するいくつかの制限について述べる．3章では $\mathcal{TR}$ の意味論を展開する上で必要な**実行含意**の概念と $\mathcal{TR}$ の証明木を定義する．4章では $\mathcal{TR}$ のプログラムを実行するインタプリタ (TRIAS) の概要を示し，どのような無駄をプログラム変換で除こうとしているかを説明する．5章ではこの無駄を除くのに必要な技法として，何回の状態遷移を経てトランザクションの実行が成功するかを予測する方法を示す．6章ではなぜ再帰的プログラムを展開した後に畳み込めないことがあるのかを示し，**綴じ合わせ**を導入するとともに，展開／畳み込み／綴じ合わせ変換によってプログラムの意味が変わらないことを示す．7章では本論文で提案した手法に基づくプログラム変換の実例と実行効率の改善結果を示す．最後に8章でまとめと今後の研究課題を示す．

2. トランザクション論理の概要

$\mathcal{TR}$ を説明するには，**データベース，遷移ベース，トランザクションベース**の三つの概念が必要である．データベースとは状態を示す一階述語論理式の任意の集合 **D** のことで，$\mathcal{TR}$ はその変更を扱う．一階述語論理式の集合を変更する時に普遍的に使える基本的な変更操作の存在は期待できないので，$\mathcal{TR}$ では基本的な変更操作を応用ごとに遷移ベース $\mathcal{B}$ として定義[1)~3)]する．また，複雑な変更操作は基本変更操作をもとにトランザクションベース **P** でトランザクションとして論理式の形で定義する．

様相記号を除くと，$\mathcal{TR}$ の言語は通常の一階述語論理の諸記号に2項の論理記号 $\otimes$ を加えて構成される論理式の集合で，この論理式をトランザクション式と呼ぶ．ここで $\phi\otimes\psi$ は ϕ と ψ の**逐次連言** (serial conjunction) と呼ばれ，直観的には ϕ の実行後に ψ が実行可能なことを表す．これに対し，$\varphi\wedge\psi$ は φ と ψ の古典連言と呼ぶ．

遷移ベース $\mathcal{B}$ は $\langle \mathbf{D}_0, \mathbf{D}_1\rangle A$ という形の式の集合である．ここで，A は基本変更操作を表す基礎原子式であり，データベース $\mathbf{D}_0$ に A を施すとデータベース $\mathbf{D}_1$ になることを意味する．

トランザクションベース **P** は $H\leftarrow\beta$ という形のトランザクション式の集合である．H は原子式，β は任意のトランザクション式で，この式は H を実行するには β を実行すればよい，ということを表す．これをトランザクション規則，H をヘッド，β をボディと呼び，また $H\leftarrow\beta$ を r で表した場合は $\mathbf{head}[r]=H$, $\mathbf{body}[r]=\beta$ とする．なお，DEC-10 Prolog の記法に従い，変数を大文字から始まる文字列で，定数を小文字から始まる文字列で表す．

P, $\mathcal{B}$ で定義されている述語記号の全体集合をそれぞれ $\mathcal{P}(\mathbf{P})$, $\mathcal{P}(\mathcal{B})$ で表す．また，データベースに出現する述語記号の全体集合を $\mathcal{P}(DB)$ で表し，トランザクション式 β に含まれるすべての述語の集合を $\mathbf{preds}(\beta)$ で表す．

例 1 ハノイの塔では，状態を各ポールのディスクの並び方で表せる．そこで，ポール1にディスクが上から a, b, c の順で置かれていることをデータベース述語 **stack** を用いて $\mathbf{stack}(1,[a,b,c])$ と表す．あるディスクをあるポールから別のポールに移す動作が基本変更とし，その変更操作を **move** という述語を使って表すことにする．すると，ディスク a をポール3に移す操作 $\mathbf{move}(a,3)$ に対して，$\mathcal{B}$ は $\langle \mathbf{D}_0, \mathbf{D}_1\rangle\mathbf{move}(a,3)$ を要素に持つ．ただし，$\mathbf{D}_0$, $\mathbf{D}_1$ は下に示すような基礎原子式を真とするデータベースである．

$\mathbf{D}_0$: {$\mathbf{stack}(1,[a,b,c])$, $\mathbf{stack}(2,[])$, $\mathbf{stack}(3,[])$},
$\mathbf{D}_1$: {$\mathbf{stack}(1,[b,c])$, $\mathbf{stack}(2,[])$, $\mathbf{stack}(3,[a])$}

移動の繰り返しを表すには，以下の **rmove** という述語を **P** で定義する．引数はどのディスク X をどのポール Q に動かしたかの記録である．**state** は任意のデータベースで成り立つ0引数のデータベース述語であり[1)]，以後データベースに陽には含めない．

$\mathbf{rmove}([])\leftarrow\mathbf{state}$.

$\mathbf{rmove}([X-Q|L])\leftarrow\mathbf{stack}(P,[X|S_1])\otimes\mathbf{stack}(Q,S_2)\otimes Q\neq P\otimes\mathbf{move}(X,Q)\otimes\mathbf{rmove}(L)$.

我々が目標とするエージェント指向プログラミン

グ[10]への有効性を確認するには，制限された $\mathcal{TR}$ を扱えば十分なので，ここでは $\mathbf{D}$, $\mathcal{B}$, $\mathbf{P}$ に以下の条件を課す．

1. $\mathcal{P}(\mathcal{B})$, $\mathcal{P}(DB)$, $\mathcal{P}(\mathbf{P})$ は互いに素である．
2. $\mathbf{D}$ は閉世界仮説を満たす．すなわちデータベース述語の基礎原子式 A に対して，$\mathbf{D}\models A$ または $\mathbf{D}\models\neg A$ が成り立つ[8),9)]．
3. $\mathbf{P}$ は連言規則の集合とする．ただし連言規則は以下のように定義される．

原子式の逐次連言と原子式を**一次式**と呼び，一次式の古典連言と一次式を**二次式**と呼ぶ．例えば $p(x,a)\otimes q(y)$ は一次式で，$(p(x,a)\otimes q(y))\wedge p(x,b)$ は二次式である．さらに，二次式の逐次連言と二次式を**三次式**と呼ぶ．例えば $((p(x,a)\otimes q(y))\wedge p(x,b))\otimes(q(y)\wedge p(a,b))$ は三次式である．同様に逐次連言と古典連言によって構成される式を一般に**連言式**と呼ぶ．そして α が一次式（二次式あるいは三次式あるいは連言式）である時に，$A\leftarrow\alpha$ という形の規則を**一次規則**（**二次規則**あるいは**三次規則**あるいは**連言規則**）と呼ぶ．

3. トランザクション論理の意味論

本章では，トランザクション実行の論理的な意味を**実行含意** (executional entailment) という概念で定義する[1)~3)]．また，与えられた遷移ベース，トランザクションベースの下であるトランザクションが実行含意の意味で成功するかどうかを判定するために**証明木**を定義する．

3.1 実行含意

$\mathcal{TR}$ の意味論では最初に「パス構造」を定義し，それを元にして実際のトランザクションの実行の論理的な意味を表す実行含意を定義する[1)~3)]（以下では $\models$ という記号で表す関係を使って実行含意を定義する）．本論文では扱うデータベースとトランザクションベースを制限しているので，パス構造を定義せずに，直接実行含意を定義する．この定義が本来の意味と等価であることは容易に示せる．

さて，二つのデータベースの列 $\mathcal{D}_1=\langle\mathbf{D}_0,\cdots,\mathbf{D}_{i-1},\mathbf{D}_i\rangle$，$\mathcal{D}_2=\langle\mathbf{D}_i,\mathbf{D}_{i+1},\cdots,\mathbf{D}_n\rangle$ があるとき，$\langle\mathbf{D}_0,\cdots,\mathbf{D}_{i-1},\mathbf{D}_i,\mathbf{D}_{i+1},\cdots,\mathbf{D}_n\rangle$ を $\mathcal{D}_1$ と $\mathcal{D}_2$ の合成といい，$\mathcal{D}_1\circ\mathcal{D}_2$ で表す．また以下ではデータベースの列をパスとも呼ぶ☆．

遷移ベース $\mathcal{B}$ とトランザクションベース $\mathbf{P}$ が与えられた時に連言式 α がデータベース $\mathbf{D}_0$ を $\mathbf{D}_0,\cdots,\mathbf{D}_n$ というデータベース列を通じて $\mathbf{D}_n$ に変更することを $\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\alpha$ と書く．形式的には，関係 $\models$ は次の条件 (E1),…,(E5) を満たす最小の関係と定義する．

(E1) $\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\alpha\wedge\beta\Longleftrightarrow\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\alpha$ かつ $\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\beta$.

(E2) $\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\alpha\otimes\beta\Longleftrightarrow$ ある $i(0\leq i\leq n)$ が存在して $\mathcal{B}\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_i\models\alpha$ かつ $\mathcal{B},\mathbf{P},\mathbf{D}_i,\cdots,\mathbf{D}_n\models\beta$.

(E3) $\alpha\leftarrow\beta$ が $\mathbf{P}$ の要素で，$\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\beta$ ならば $\mathcal{B},\mathbf{P},\mathbf{D}_0,\cdots,\mathbf{D}_n\models\alpha$.

(E4) α が基本変更を表す基礎原子式ならば $\mathcal{B},\mathbf{P},\mathbf{D}_0,\mathbf{D}_1\models\alpha\Longleftrightarrow\langle\mathbf{D}_0,\mathbf{D}_1\rangle\alpha\in\mathcal{B}$ が成立する．

(E5) α が基礎原子式ならば $\mathbf{D}_0\models\alpha\Longleftrightarrow\mathcal{B},\mathbf{P},\mathbf{D}_0\models\alpha$.

補題 3.1 $\mathbf{P}$ と $\mathbf{D}$ が2章で示した条件 1, 2, 3 を満たせば (E1),…,(E5) を満たす最小の関係 $\models$ が存在する．

補題3.1の証明は後で示す定理3.1の証明の中で同時に行う．

(E1)と(E2)より，$\mathcal{B},\mathbf{P},\mathbf{D}_0\models\alpha\wedge\beta$ と $\mathcal{B},\mathbf{P},\mathbf{D}_0\models\alpha\otimes\beta$ は等価なので，従来の Prolog プログラムの古典連言は逐次連言に置き換えられる．

3.2 証明木

木 T が次の条件をすべて満たす時に T をデータベース列 $\mathcal{D}$ とトランザクションベース $\mathbf{P}$ の下☆での**基礎原子式 A の証明木**と呼び，この関係を T: **prt** $(A;\mathbf{P};\mathcal{D})$ と略記する．

- **ラベル** 葉以外の各節点 v は $\mathcal{D}$ の部分列 $\mathcal{D}_v$ と連言式 α のペア $\mathcal{D}_v:\alpha$ をラベルとして持つ．特に T の根のラベルは $\mathcal{D}:A$ であり，葉のラベルは空である．
- **逐次連言** α が $\phi_1\otimes\cdots\otimes\phi_k$ ならば v は子節点 $v_1,\cdots,v_k$ を持つ．v_i のラベルは $D_i:\phi_i$ であり，$D_v=D_1\circ\cdots\circ D_k$ である．
- **古典連言** α が $\phi_1\wedge\cdots\wedge\phi_k$ なら，v は k 個の子節点 $v_1,\cdots,v_k$ を持ち，v_i はラベル $\mathcal{D}_v:\phi_i$ を持つ．
- **基礎原子式** α が基礎原子式 B ならば次のいずれかが成り立つ．
  - **基本変更** $\mathcal{D}_v=\langle\mathbf{D}_p,\mathbf{D}_{p+1}\rangle$，$\langle\mathbf{D}_p,\mathbf{D}_{p+1}\rangle B\in\mathcal{B}$ であり，v は唯一の子節点 w を持つ．w のラベルは

☆ 厳密には，$\mathcal{TR}$ におけるパスの概念は単なるデータベース列を表すのではなく，より一般的な意味で用いられている[1)~3)]．

☆ 厳密には，遷移ベース $\mathcal{B}$ にも依存するが，以下の議論では $\mathcal{B}$ は固定して考えるので省略する．

空である．

○**規則による展開** **P** の中の連言規則の基礎代入例 $B \leftarrow \phi$ が存在し，v は唯一の子節点 w を持つ．w のラベルは $\mathcal{D}_v : \phi$ である．

○**データベースによる証明** $\mathcal{D}_v = \langle \mathbf{D}_P \rangle$，$\mathbf{D}_P \models B$ であり，v は唯一の子節点 w を持つ．w のラベルは空である．

例1の **rmove**($[a-2, b-3]$) の証明木を**図1**に示す．ただし，$\mathbf{D}_0$，$\mathbf{D}_1$，$\mathbf{D}_2$ は以下のとおりである．

$\mathbf{D}_0$: {**stack**(1, $[a, b]$), **stack**(2, $[]$), **stack**(3, $[]$)}

$\mathbf{D}_1$: {**stack**(1, $[b]$), **stack**(2, $[a]$), **stack**(3, $[]$)}

$\mathbf{D}_2$: {**stack**(1, $[]$), **stack**(2, $[a]$), **stack**(3, $[b]$)}

さて，変数を持たない連言式（基礎連言式）α に対して証明木が存在することと実行含意が成立することは等価である．

定理 3.1 基礎連言式 α に対し，証明木 T : **prt**(α; **P**; $\mathbf{D}_0, \cdots, \mathbf{D}_n$) が存在するための必要十分条件は $\mathcal{B}$, **P**, $\mathbf{D}_0, \cdots, \mathbf{D}_n \models \alpha$ である．

証明は付録に示す．

4. トランザクション論理の処理

本章ではマルチエージェント系シミュレータ TRIAS[6] で用いた $\mathcal{TR}$ のプログラムの処理方法を概説する．Prolog では，ゴール $A_1 \wedge \cdots \wedge A_n$ を左から順に規則で展開してゴールを書換えていく作業を繰り返す．TRIAS でも Prolog と同様に古典連言の左の要素から順に展開していくが，最初に $\mathcal{B}$ の述語による状態遷移が起きた段階でその要素の処理を中断し，古典連言の右隣の要素の展開を始める．そして，各遷移に対して同じデータベースが得られたか確認し，違えばバックトラックする．

例 2 $p, q \in \mathcal{P}(\mathcal{B})$ とし，次のような **P** を考える．

$a \leftarrow p \otimes a_1$. $a_1 \leftarrow q \otimes a_2$.

$b \leftarrow b_1 \otimes b_2$. $b_1 \leftarrow q \otimes b_3$.

インタプリタは初期データベース $\mathbf{D}_0$ だけを与えられてゴール $a \wedge b$ を以下のように書き換える．まず，左の要素 a を最初の基本変更が起きるまで展開する．

$a \wedge b \Rightarrow (p \otimes a_1) \wedge b$

$\mathbf{D}_0$ に基本変更 p を施した結果を $\mathbf{D}_1$ とする．状態変化が起きた位置を示すため，p を **done** という定数で置き換えて，a に関する展開を中断し，b を展開する．

$\Rightarrow (\mathbf{done} \otimes a_1) \wedge (b_1 \otimes b_2) \Rightarrow (\mathbf{done} \otimes a_1) \wedge (q \otimes b_3 \otimes b_2)$

ここで $\mathbf{D}_0$ に q を施した結果を $\mathbf{D}_2$ とする．この時，$\mathbf{D}_1$ と $\mathbf{D}_2$ が等しければ，次のように **done** を **state** に置き換えて a_1 の展開に戻る．

$\Rightarrow (\mathbf{done} \otimes a_1) \wedge (\mathbf{done} \otimes b_3 \otimes b_2) \Rightarrow$

$(\mathbf{state} \otimes a_1) \wedge (\mathbf{state} \otimes b_3 \otimes b_2) \Rightarrow a_1 \wedge (b_3 \otimes b_2)$

$\mathbf{D}_1$ と $\mathbf{D}_2$ が等しくなければ，実行含意の定義（E1）よりこの展開は失敗したことになるので，バックトラックして他の可能性を調べる．この例では引数の処理を考えなかったが，変数については Prolog とまったく同じユニフィケーションが行われる．

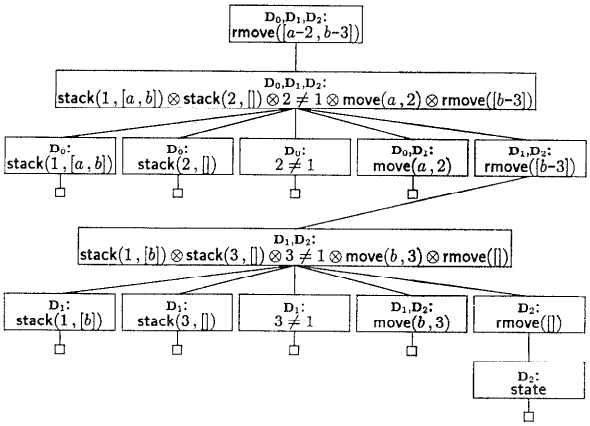


図 1 トランザクション論理における証明木の例
Fig. 1 A proof tree in Transaction Logic.

上記の古典連言の処理方法では，頻繁にバックトラックが生じ，実行効率がよくない場合がある．例えば，$a \wedge b$ というゴールを考え，a と b を展開する規則がトランザクションベース **P** 中にそれぞれ多数与えられているものとする．a に対し状態遷移なしで成功する規則を選んだ時に，b に対して状態遷移の生じる規則を選ぶと，a が成功するパスと b が成功するパスの長さが違うので必ず失敗する．この場合，規則のボディを満たすパスの長さが予測できればその規則の組み合わせは必ず失敗することが分かる．あらかじめそのような無駄な組み合わせを排除できれば，処理の高速化が期待できる．そこで次章ではパス長を予測する方法を示す．これを利用して6章では無駄な探索をしないプログラムに変換する方法を示す．

5. パス長予測

ある連言式が何回の状態遷移の後に成功するかを概

算するパス長*予測関数 l を定義する．この関数により不要な規則の除去や後述の綴じ合わせが可能になる．厳密に評価するためのコストと効果のトレードオフを考慮して，ここで定義する l は述語名だけに依存し，引数の中身は無視する．

以後，m 以上のすべての自然数の集合を $m^{\geq}$ と表す．また自然数の集合 A, B に対し $A+B=\{m+n \mid m\in A$ かつ $n\in B\}$ とする．次に各述語のパス長のデータ **pl**[・] が与えられているものとして，パス長を予測する関数 $l(\alpha)$ を以下のように定義する．

- α が述語記号 p を持つ原子式で **pl**[p] が定義されていれば $l(\alpha)=$**pl**[p].
- $\alpha=\phi_1\otimes\cdots\otimes\phi_m$ なら $l(\alpha)=l(\phi_1)+\cdots+l(\phi_m)$.
- $\alpha=\phi_1\wedge\cdots\wedge\phi_m$ なら $l(\alpha)=\cap_{1\leq i\leq m} l(\phi_i)$.

この関数の定義により，あとは各述語のパス長予測値 **pl** を求めるだけになる．$\mathcal{P}(DB)$ の各述語のパス長は 0，$\mathcal{P}(\mathcal{B})$ の各述語のパス長は 1 なので，$\mathcal{P}(\mathbf{P})$ の各述語に対して **pl** を定義する方法を考えればよい．本質的に再帰的に定義されている述語の場合と，そうでない述語の場合の 2 通りに分けて 2 段階で計算する．

まず，後者の場合を**図 2** のアルゴリズムに従って計算すると，C の述語記号に対しては **pl** の値が定まる．Y の述語記号に対しては未定である．Y の述語は本質的に再帰的に定義されているもので，C の述語はそうでないものである．

例 3 例 1 の **P** に規則

redundant$(X,P,Q)\leftarrow$**move**$(X,P)\otimes$**move**(X,Q) を加えると，**pl**[**stack**]$=\{0\}$，**pl**[**move**]$=\{1\}$，**pl**[**redundant**]$=\{2\}$ が得られ，再帰的述語は $Y=$ {**rmove**} で示される．一方，規則

mv$(X,P,L)\leftarrow$**rmove**$(L)\wedge$**move**(X,P)

を加えると，再帰的な **rmove** が使われているにもかかわらず，**pl**[**mv**]$=\{1\}$ であるとわかる．

次に**図 3** のアルゴリズムを実行する．**estimate 2** $(Y,C,\mathbf{P})$ の Y と C には **estimate 1** の実行後の Y と C を与える．なお，**call**[p] は述語 p の定義に直接的・間接的に使われている N の述語の集合で，正確には次のように定義される．p をヘッドに含む規則のボディに N の述語 q が現れていることを pDq で表し，D^* を D の反射推移閉包とすると **call**[p]$=\{q \mid pD^*q$ かつ $q\in N\}$.

たとえば例 1 の **P** では **pl**[**rmove**]$=0^{\geq}$ となる．ま

```
procedure estimate1(P, B);
  begin
    foreach p ∈ P(DB) do pl[p] := {0}; od;
    foreach p ∈ P(B) do pl[p] := {1}; od;
    Y := P(P); {Y はまだ pl の値が定まっていない述語の集合}
    C := P(B) ∪ P(DB); {C は既に pl の値が定まった述語の集合}
    repeat
      N := estimable1(Y, C, P); {N は新しく pl の値が定まった述語の集合}
      Y := Y − N; C := C ∪ N;
    until N = {};
  end;
function estimable1(Y, C, P) : set_of_predicates;
  begin
    N := {};
    foreach p ∈ Y do
      if estimable_pred(p, C, P) then N := N ∪ {p}; fi;
    od;
    return (N);
  end;
function estimable_pred(p, C, P) : boolean;
  begin
    sw1 := true; k := {};
    foreach (p(...) ← β) ∈ P do
      if (β = φ1 ∧ ··· ∧ φm) and (m ≥ 2) then
        sw2 := false; h := 0≥;
        for i := 1 to m do {どれか一つが成立すればよい}
          if preds(φi) ⊂ C then sw2 := true; h := h ∩ l(φi); fi;
        od;
        if sw2 then k := k ∪ h; else sw1 := false; fi;
      else {つまりβが原子式か逐次連言}
        if preds(β) ⊂ C then k := k ∪ l(β); else sw1 := false; fi;
      fi;
    od;
    if sw1 then pl[p] := k; fi;
    return (sw1);
  end
```

図 2 非再帰的述語のパス長評価
Fig. 2 A path-length evaluation algorithm for non-recursive predicates.

```
procedure estimate2(Y, C, P);
  begin
    repeat
      N := estimable2(Y, C, P); Y := Y − N; C := C ∪ N;
    until N = {};
  end;
function estimable2(Y, C, P) : set_of_predicates;
  begin
    N := {};
    foreach p ∈ Y do
      pl[p] := {}; sw := false; s[p] := ∞;
      foreach (p(...) ← φ) ∈ P do
        {p を定義する規則で，ボディがすべて C の述語からなるものがあれば
          p を N に加える．s[p] はそのようなボディの l の最小値である．}
        if preds(φ) ⊂ C then sw := true; s[p] := min({s[p]} ∪ l(φ)); fi;
      od;
      if sw = true then N := N ∪ {p}; fi;
    od;
    foreach p ∈ N do pl[p] := (min_{q∈call[p]} s[q])≥; od;
    return (N);
  end;
```

図 3 再帰的述語のパス長評価
Fig. 3 A path-length evaluation algorithm for recursive predicates.

* Bonner らはパス長を状態数としており[1),3)]，本論文の定義より 1 だけ多い．

た次の **P** の場合，

$p(x)\leftarrow b(x)$. $p(x)\leftarrow c(x,y)\otimes p(y)$.

$q(x)\leftarrow b(x)\otimes b(x)$. $q(x)\leftarrow c(x,y)\otimes q(y)$.

$\mathbf{pl}[b]=\mathbf{pl}[c]=\{1\}$ のとき $\mathbf{pl}[p]=1^{\geq}$, $\mathbf{pl}[q]=2^{\geq}$ となる．これに $q(x)\leftarrow p(x)$ を加えると $\mathbf{pl}[p]=\mathbf{pl}[q]=1^{\geq}$ となる．

次の補題は，α の実行が n 回の基本遷移の後で成功するなら，n が $l(\alpha)$ に含まれることを表している．証明は付録に示す．

補題 5.1（**パス長予測関数の完全性**） 任意の基礎連言式 α に対し $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models \alpha$ ならば $n \in l(\alpha)$ である．

以下の補題はボディのパス長推定が空集合である規則を削除しても意味が変わらないことを示しており，展開によって発生する無駄な規則の削除を正当化するために必要である．証明は補題 5.1 と定理 3.1 より容易に示せるので省略する．

補題 5.2（**無意味な規則の除去の正当性**） **P** の規則 r に対し $l(\mathbf{body}[r])=\{\}$ ならば，任意のデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$ と任意の基礎連言式 β に対して $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models \beta \Longleftrightarrow \mathcal{B}, \mathbf{P}-\{r\}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models \beta$ が成立する．

さて，後で提案する綴じ合わせの正当化のために次の補題と系が必要である．任意のデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$ に対して，$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\wedge\gamma)\otimes(\beta\wedge\delta)$ なら $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\otimes\beta)\wedge(\gamma\otimes\delta)$ が成立することは定義から容易に示せるが，逆は一般に成立しない．次の補題は，α と γ がいずれもちょうど m 回の遷移の後にのみ成功するなら，逆が成立することを示している．

補題 5.3（**綴じ合わせの正当性**） $l(\alpha)=l(\gamma)=\{m\}$ となる m があれば，任意のデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$ に対して $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\otimes\beta)\wedge(\gamma\otimes\delta)$ のとき $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\wedge\gamma)\otimes(\beta\wedge\delta)$ が成立する．

[証明] $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\otimes\beta)\wedge(\gamma\otimes\delta)$ を仮定すると，定理 3.1 と補題 5.1 により**図 4** の(a)のような $(\alpha\otimes\beta)\wedge(\gamma\otimes\delta)$ の証明木がある．これから図 4 の(b)に示す $(\alpha\wedge\gamma)\otimes(\beta\wedge\delta)$ の証明木を構築できる．したがって定理 3.1 により $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models (\alpha\wedge\gamma)\otimes(\beta\wedge\delta)$ が成り立つ． (証明終り)

次の系は補題 5.3 から容易に示せる．

系 5.1 $l(\alpha)=l(\gamma)=\{m\}$ で，**P** の中の規則 r が $A\leftarrow(\alpha\otimes\beta)\wedge(\gamma\otimes\delta)$ という形をしているとき $\mathbf{P}'=\mathbf{P}\cup\{A\leftarrow(\alpha\wedge\gamma)\otimes(\beta\wedge\delta)\}-\{r\}$ とする．このとき任意のデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$ と任意の連言式 ϕ に対して $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models \phi \Longleftrightarrow \mathcal{B}, \mathbf{P}', \mathbf{D}_0, \cdots, \mathbf{D}_n \models \phi$ が成立する．

6. プログラム変換

4 章で述べた無駄なバックトラックをあらかじめ除去するため，Prolog などで用いられている展開／畳み込みによってプログラムを変換することが考えられる．ところが $\mathcal{TR}$ のプログラム変換では問題が生じる．

次のような規則を含む Prolog プログラムを考えよう．

$p(X)\leftarrow r(X,Y)\wedge p(Y)$.

$q(X)\leftarrow s(X,Y)\wedge q(Y)$.

$n(A,B)\leftarrow p(A)\wedge q(B)$.

すると次のように展開 (unfolding) と畳み込み (folding) ができて再帰的な規則が得られる．

$n(A,B)\leftarrow(r(A,C)\wedge p(C))\wedge(s(B,D)\wedge q(D))$.

$n(A,B)\leftarrow r(A,C)\wedge s(B,D)\wedge p(C)\wedge q(D)$.

$n(A,B)\leftarrow r(A,C)\wedge s(B,D)\wedge n(C,D)$.

しかし，$\mathcal{TR}$ の場合はそれほど簡単ではない．**P** に次のような規則が含まれているとする．

$p(X)\leftarrow r(X,Y)\otimes p(Y)$.

$q(X)\leftarrow s(X,Y)\otimes q(Y)$.

$n(A,B)\leftarrow p(A)\wedge q(B)$.

この n を展開しても Prolog のように書き換えができないので畳み込みができない．

$n(A,B)\leftarrow(r(A,C)\otimes p(C))\wedge(s(B,D)\otimes q(D))$.

このように，$\mathcal{TR}$ の再帰的述語を変換するには，展開

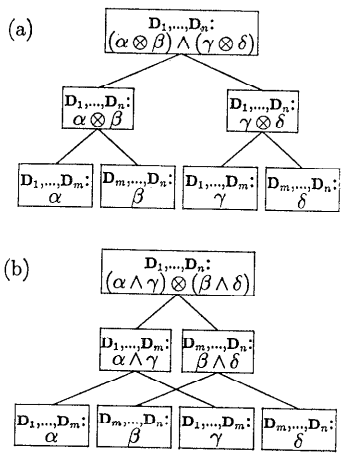


図 4 綴じ合わせは証明木(a)を(b)に変換する
Fig. 4 The zipping operation transforms the proof tree (a) into (b).

と畳み込みだけでは不十分なことがわかる．本論文で提案する**綴じ合わせ**は，パス長の情報を利用して畳み込みができるように規則を書き換えるものである．

なお本論文で提案する以下の各変換は任意の連言規則からなるトランザクションベースに対して拡張可能であるが，記述が煩雑になるので，最初のトランザクションベース **P** が次の条件を満たすと仮定する．

- 古典連言を含む規則は唯一 $r_0=(A_0\leftarrow A_1\wedge\cdots\wedge A_n)$ のみである．ただし A_i は原子式である．さらに A_0 の述語を定義する規則は他には存在しない．
- $\mathbf{P}^-=\mathbf{P}-\{r_0\}$ はすべて一次規則であって変換の必要はない．

3.1 節の終わりで述べたように，状態遷移のないとき古典連言は逐次連言で置き換えられるので，このような制限を課しても **P** の記述力は Prolog を上回る．

まず $\mathcal{GR}$ における展開，畳み込み，綴じ合わせの定義を述べたのち，それらを組み合わせて行う変換のガイドラインを示し，最後にその変換の過程でプログラムの意味が変化しないことを示す．

6.1 展　　開

我々は無駄な規則の組み合わせを発見し除去するため，原子式を展開して無駄な規則の組み合わせを明らかにする．そこで補題 5.2 による無駄な規則の除去も展開操作に含める．正確には以下のようにする．

一次規則 $r_1=(A_0\leftarrow A_1\otimes\cdots\otimes A_n)$ での i 番目の原子式 A_i の展開は，A_i と単一化可能なヘッドを持つ $\mathbf{P}^-$ の各規則 $r_2=(F_0\leftarrow F_1\otimes\cdots\otimes F_m)$ に対応して，r_2 の全変数をそれまでに使われていない新しい変数により 1 対 1 で置き換えた規則 $F_0'\leftarrow F_1'\otimes\cdots\otimes F_m'$ を一つ作り，

$$\mathbf{rep}(r_1,\langle i\rangle,r_2)=(A_0\leftarrow A_1\otimes\cdots\otimes A_{i-1}\otimes F_1'\otimes\cdots\otimes F_m'\otimes A_{i+1}\otimes\cdots\otimes A_n)\mathbf{mgu}(A_i,F_0').$$

の集合を作ることである．ただし $l(\mathbf{body}[\mathbf{rep}(r_1,\langle i\rangle,r_2)])=\{\}$ となる規則は削除する．同様に一般の連言規則での展開も定義できる．

例 4　例 3 に

$$\mathbf{foo}\leftarrow\mathbf{rmove}([X-Q])\wedge\mathbf{redundant}(X,R,S)$$

という規則を加え，ボディの各原子式を展開すると，

$$\mathbf{foo}\leftarrow(\mathbf{stack}(P,[X|S_1])\otimes\mathbf{stack}(Q,S_2)\otimes Q\neq P\otimes\mathbf{move}(X,Q)\otimes\mathbf{rmove}([]))\wedge(\mathbf{move}(X,R)\otimes\mathbf{move}(X,S))$$

が得られる．さらに **rmove**([]) は **state** に展開できるので

$$\mathbf{foo}\leftarrow(\mathbf{stack}(P,[X|S_1])\otimes\mathbf{stack}(Q,S_2)\otimes Q\neq P\otimes\mathbf{move}(X,Q)\otimes\mathbf{state})\wedge(\mathbf{move}(X,R)\otimes\mathbf{move}(X,S))$$

となるが，そのボディは古典連言の左側が $l(\cdots)=\{1\}$，右側が $l(\cdots)=\{2\}$ なので **pl**[**foo**]$=\{\}$ となり，展開した結果は空である．

6.2 畳 み 込 み

最初に与えられた規則 $r_0=(A_0\leftarrow A_1\wedge\cdots\wedge A_n)$ の全変数をこれまでに使われていない新しい変数で 1 対 1 で置き換えたものを $r_0'=(A_0'\leftarrow A_1'\wedge\cdots\wedge A_n')$ とする．三次規則 $r_1=(C\leftarrow C_1\otimes\cdots\otimes C_m)$ に対して $C_i=(A_1'\wedge\cdots\wedge A_n')\theta$ となる i と代入 θ があるとき，r_1 を

$$C\leftarrow C_1\otimes\cdots\otimes C_{i-1}\otimes A_0'\theta\otimes C_{i+1}\otimes\cdots\otimes C_m$$

で置き換えることを r_1 の r_0 による畳み込みという．

6.3 綴じ合わせ

一次式 $P=A_1\otimes\cdots\otimes A_n$ に対して，状態変化を伴う最も左の逐次連言要素を**実質先頭要素**といい $H[P]$ で表す．それより左の状態変化を伴わない部分を**0次成分**といい，$Z[P]$ で表す．また，残りを**後部**といい，$T[P]$ で表す．

正確には，$l(P)\neq\{\}$ であるとき*に以下のように定義する．$l(A_k)\neq\{0\}$ を満たす最小の k に対し $H[P]=A_k$ とする．さらに $k\geq 2$ なら $Z[P]=A_1\otimes\cdots\otimes A_{k-1}$，$k=1$ なら $Z[P]=$**state** とする．$k<n$ なら $T[P]=A_{k+1}\otimes\cdots\otimes A_n$，$k=n$ なら $T[P]=$**state** とする．

さて，二次式 $S-P_1\wedge\cdots\wedge P_m$ に対して，その 0 次成分を $Z[S]=Z[P_1]\otimes\cdots\otimes Z[P_m]$ とする．$l(S)\neq\{\}$ かつ $l(S)\neq\{0\}$ であるとき，以下の定義に従って **front**[S]$\otimes$**res**[S] を S の**綴じ合わせ** (**zipping**) と呼ぶ．

- $l(H[P_1])=\cdots=l(H[P_m])=\{k\}$ となる正数 k があれば，**front**[S]$=Z[S]\otimes(H[P_1]\wedge\cdots\wedge H[P_m])$，**res**[$S$]$=T[P_1]\wedge\cdots\wedge T[P_m]$ とする．
- そのような k がない場合には **front**[S]$=Z[S]$，**res**[S]$=(H[P_1]\otimes T[P_1])\wedge\cdots\wedge(H[P_m]\otimes T[P_m])$ とする．

ただし **front**[S]$=$**state**$\otimes\cdots\otimes$**state** のときは **res**[S]$=S$ となって，この処理は無意味である．そこでこの場合は綴じ合わせが失敗したものとする．

また，三次規則 r に対して，**body**[r]$=S_1\otimes\cdots\otimes S_n$ (三次式) とする．S_i を **front**[S_i]$\otimes$**res**[S_i] で置き換える操作を，r の S_i に関する**綴じ合わせ**という．

6.4 変換のガイドライン

展開によって規則の数が増えてしまうと，かえって効率が悪くなる可能性がある．無駄な展開を防ぐため，展開には厳しい制約を課す．さいわい，展開すべ

---

* 補題 5.2 より，$l(P)=\{\}$ なら P をボディにもつ規則は不要である．

き原子式は綴じ合わせの情報を利用して絞りこめる．そこで変換は以下でおおまかに示すガイドライン（**図5**）に従って進める．

なお，これまで説明した変換以外にも，$((\alpha\otimes\beta)\otimes(\gamma\otimes\delta))$ を $\alpha\otimes\beta\otimes\gamma\otimes\delta$ と書き換えるような結合律の適用や，逐次連言にできる古典連言の置き換えや，冗長な **state** や **arc** の除去が実際には必要になってくる．ここで **arc** は任意のデータベース $\mathbf{D}_0$, $\mathbf{D}_1$ に対して，$\mathbf{D}_0$, $\mathbf{D}_1 \models$ **arc** が成り立つ0引数述語である[1]．このような表現の単純化処理を連言式 s に適用した結果を **reduce**(s) で表す☆．

まず r_0 のすべての原子式を展開して二次規則の集合を得る．その集合の要素である各規則 r が l(**body**[r])$=\{0\}$ を満たせば遷移のない古典連言を逐次連言で置き換えることや結合律によって r を一次規則にできる．一次規則になった規則は R_1 に移す．それ以外の場合は R_2 が空になるまで以下の処理を繰り返す．正常に終了すれば R_1 に r_0 の変換結果が得られる．しかし一般に終了する保証はないので，適当なところで打ち切る必要がある．

綴じ合わせする前の二次式を ϕ としよう．ϕ の綴じ合わせができた場合には，r の r_0 による畳み込みを **res**[ϕ] について試みる．畳み込みにより一次規則になった規則は R_1 に移す．綴じ合わせができない場合には，綴じ合わせに失敗する原因となった ϕ の原子式（後述）を展開（図5の **unfold_culprits**）したのち，再度綴じ合わせと畳み込みを試みる．綴じ合わせができても **res**[ϕ] が畳み込めない規則に対しては，**res**[ϕ] を新たな ϕ としてこの操作を繰り返す．

二次式 ϕ の綴じ合わせの失敗原因の原子式（culprit）とは，ϕ の実質先頭要素のうち以下のいずれかの条件を満たす原子式である．

- l 値が複数の要素を含むもの．
- l 値が一つの要素しか含まないが，他の実質先頭要素よりその値が大きいもの．

例5 ハノイの塔の問題では，目標状態に到達すれば終了してよい．この制約は目標状態でのみ真になる **achieved** というデータベース述語を用いて次のように定義できる．

search←**achieved**. **search**←**arc**⊗**search**.

各ディスクがそれより小さなディスクの上に載っていないことが **ordered**($\in\mathcal{P}(DB)$) で判定できるとする．するとこの制約を守り続けることは次の **order_check** で規定できる．

order_check←**ordered**⊗**oc**.

oc←**state**. **oc**←**arc**⊗**order_check**.

そこで r_0 として **soc**←**search**∧**order_check** という規則を作り展開すると，得られる二つの節のうち一つは次のようなものである．

soc←(**arc**⊗**search**)∧(**ordered**⊗**oc**)

このボディを ϕ として綴じ合わせると0次成分だけ前に出る．

front[ϕ]=(**state**∧**ordered**), **res**[ϕ]=((**arc**⊗**search**)∧**oc**)

front[ϕ] の **state** は不要なので削れる．**res**[ϕ] は r_0 で畳み込めず，また l(**arc**)$=\{1\}\neq l$(**oc**)$=0^{\geq}$ なので綴じ合わせもできない．そこで **oc** を展開すると次の規則が得られる．

soc←**ordered**⊗((**arc**⊗**search**)∧(**arc**⊗**order_check**))

ボディ部の三次式の第2要素を ϕ とすると，今度は

front[ϕ]=(**arc**∧**arc**), **res**[ϕ]=(**search**∧**order_check**)

で綴じ合わせでき，さらに **res**[ϕ] は r_0 で畳み込めるので，**soc**←**ordered**⊗**arc**⊗**soc** が得られる☆．

```
procedure transform(r0, P⁻);
begin
  R1 := {}; R2 := unfold_culprits(r0, P⁻);
  while R2 ≠ {} do
    foreach r ∈ R2 do
      if l(body[r]) = {0} then
        R1 := R1 ∪ {head[r] ← reduce(body[r])}; R2 := R2 − {r};
      else
        fr := front[body[r]]; rs := res[body[r]];
        if fr = state ⊗ ··· ⊗ state then {綴じ合わせに失敗したら展開}
          D := unfold_culprits(r, P⁻);
        else {綴じ合わせが成功したら畳み込み}
          if instance_of(rs, r0) then
            D := {head[r] ← reduce(fr ⊗ fold(rs, r0))};
          else
            D := {head[r] ← reduce(fr ⊗ rs)};
          fi;
        fi;
        foreach s ∈ D do
          if s が一次規則 then
            D := D − {s}; R1 := R1 ∪ {s};
          else
            D := D − {s}; R2 := R2 ∪ {s};
          fi;
        od;
      fi;
    od;
  od;
end;
```

図5 変換手続き
Fig. 5 The transformation procedure.

☆ これらの処理を行えば，このガイドラインに従う r_0 の変換結果は三次規則までに抑えられる．

☆ 冗長な arc∧arc は arc に書き換えた．

6.5 等価性の保存

展開／畳み込み／綴じ合わせのうち可能な操作をトランザクションベース $\mathbf{P}_0$ に対して次々と施して得られるトランザクションベースの系列を $\mathbf{P}_0, \mathbf{P}_1, \cdots$ とする．等価性の保存は次の定理で保証される．

定理 6.1 任意の基礎原子式 α，任意のデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$，任意の自然数 i に対して，$\mathcal{B}, \mathbf{P}_0, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha \Longleftrightarrow \mathcal{B}, \mathbf{P}_i, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$.

[証明] 証明の本質的な部分は，Prolog の展開／畳み込み変換の等価性の証明[12]で使われているアイデアに依存している．$i-1$ まで定理が証明できたと仮定して以下の性質を示す．

$\mathcal{B}, \mathbf{P}_i, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha \Longrightarrow \mathcal{B}, \mathbf{P}_{i-1}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$

変換に展開／畳み込みを使った場合は玉木[12]の補題1と同様にして容易に示せる．綴じ合わせの場合は系5.1から示せる．

定理を証明するためには，上の性質の逆を示さねばならない．玉木[12]は Prolog プログラムに対して**重み完全**という性質を定義し，変換の過程でその性質が保たれれば，$\mathbf{P}_0$ の下での証明木を $\mathbf{P}_i$ の下での証明木に変換できることを示している．$\mathcal{TR}$ でも，α の証明木 T の重みを「旧述語の原子式でラベル付けされた，T の節点の数」と定義すれば，玉木の証明を流用できる．ただし，我々は展開後にパス長チェックで不要な規則を除去しているので，これらの不要な規則が玉木の証明で使われる**下降インスタンス**にならないことを補題5.2を用いて示さなければならない．また，綴じ合わせが重み完全性を保存することを系5.1を用いて示す必要がある．これらは容易に証明できるので，詳細は省略する． 〈証明終り〉

7. プログラム変換の具体例

本章ではハノイの塔を解くための**P**がどのように変換されるかを示す．また，変換前後の実行時間がどのように改善されたかを実測値で示す．

7.1 ハノイの塔のプログラム

ハノイの塔のプログラムを完成させる．すでにデータベース述語として **stack**, **ordered**, **achieved** を用意したが，さらに目標状態を表すため，ある **stack**(P, L) が目標であることを表す述語 **goal**(P, L) を導入する．このうち **stack** だけが基本操作で直接書き換えられる．さらに以下の述語を **P** に加える．

• **maxsteps**(N) は実行を打ち切るためのトランザクションで，長さ N 以下の任意のデータベース列で成功するが，N を越えると失敗する．

maxsteps$(N) \leftarrow N \geq 0$.

maxsteps$(N) \leftarrow N > 0 \otimes M$ is $N-1 \otimes$ **arc** $\otimes$ **maxsteps**(M).

• **no_repeat**(X) は同じディスクを二度続けて動かす無駄を禁示するトランザクションで，引数 X は直前に動かしたディスクを表す．

no_repeat$(X) \leftarrow$ **state**.

no_repeat$(X) \leftarrow$ **stack**$(P, [Y|L]) \otimes Y \neq X \otimes$ **arc** $\otimes$ **stack**$(P, L) \otimes$ **no_repeat**(Y).

• **hanoi**(N, L, X) はハノイの塔を解くトランザクションで，N がディスクを動かす回数の上限，L が目標状態に達するまでのディスクの移動の系列，X が直前に動かしたディスクを表す変数である．

hanoi$(N, L, X) \leftarrow$ **maxsteps**$(N) \wedge$ **rmove**$(L) \wedge$ **search** $\wedge$ **order_check** $\wedge$ **no_repeat**(X).

パス長 l を計算すると，**P** で定義された述語記号を持つ **hanoi**, **maxsteps**, **no_repeat**, **rmove**, **search**, **order_check**, **oc** のいずれも $0^{\geq}$ である．

hanoi の規則が今回の r_0 である．展開すると考えられる $2\times2\times2\times2\times2=32$ の組み合わせ*のうち30がパス長チェックで取り除かれ，変換結果として次の2個の規則が残る．

hanoi$(A, [\,], B) \leftarrow$
$A \geq 0 \otimes$ **achieved** $\otimes$ **ordered**.

hanoi$(A, [B-C|D], E) \leftarrow$
$A > 0 \otimes F$ is $A-1 \otimes$ **stack**$(G, [B|H]) \otimes$
stack$(C, I) \otimes C \neq G \otimes$ **ordered** $\otimes$
stack$(J, [K|L]) \otimes K \neq E \otimes$ **move**$(B, C) \otimes$
stack$(J, L) \otimes$ **hanoi**(F, D, K).

7.2 実行効率

SPARCserver 670 MP 上の Quintus Prolog 3.1.4 で作成した TRIAS により変換前後の **hanoi** の CPU 時間を計測した．ゴールは **hanoi**$(16, L, X)$ で，最初のデータベースは

{**goal**$(3, [a, b, c, d])$, **stack**$(1, [a, b, c, d])$,
stack$(2, [\,])$, **stack**$(3, [\,])$}

とした．変換前は解

$L=[a-2, b-3, a-3, c-2, a-1, b-2, a-2, d-3,$
$a-3, b-1, a-1, c-3, a-2, b-3, a-3]$

を発見するのに 54.0 秒かかっていたが，変換後は 10.8 秒ですむようになり，5倍の高速化が達成され

* 実際には各述語を順次展開するので，32もの節が生成されることはない．

た．別の実験[4]では鉄道網上の探索プログラムを用いて計測を行ったが，やはり6倍弱の高速化が得られている．

8. おわりに

本論文では $\mathcal{TR}$ の変換操作として従来と同様の展開，畳み込みに加え，綴じ合わせという操作を導入し，いずれもプログラムの意味を保つことを示した．また，トランザクションの実行が成功するときのデータベース列の長さ（パス長）を推測して展開で生じる不要な規則を除去する方法を示した．パス長予測はコンパイラにも有用な技術である．

さらに基本変更の排反性を用いても規則の除去や簡単化ができる．例えば，二つの基本変更 p, q に対して $\mathbf{D}_0$, $\mathbf{D}_1 \models p \wedge q$ を満たす $\mathbf{D}_0$, $\mathbf{D}_1$ がないことがあらかじめ分かっていれば，$(p \otimes \phi) \wedge (q \otimes \phi)$ は必ず失敗するので削除できる．このような拡張は稿をあらためて議論する予定である．

本論文ではトランザクションベースが連言規則からなる場合のみを考察した．しかし一般の $\mathcal{TR}$ では仮説推論／回顧推論の様相記号なども現れる．特に，マルチエージェント環境での推論には，仮説推論が有用である[5]．これらを含む変換は今後の研究課題である．また，本論文では玉木，佐藤[11),12)]に従い，実行が成功する基礎原子式の集合が不変性を持っていることを示した．川村，金森[7]は Prolog プログラムの展開／畳み込み変換において答となる代入の集合が不変性を持っていることを示している．本論文の結果を，このより強い意味での等価性の保存へ拡張することも今後の課題の一つである．

謝辞 本論文の草稿に貴重なコメントを下さった平田圭二氏と Ron van der Meyden 氏に感謝します．また，査読者の方々からは数々の重要なコメントをいただきました．ここに記して感謝いたします．

参考文献

1) Bonner, A. J. and Kifer, M.: Transaction Logic Programming, Technical Report CSRI-270, University of Toronto (1992), csri. toronto. edu: csri-technical-reports/270/report. ps.
2) Bonner, A. J. and Kifer, M.: Transaction Logic Programming, Warren, D. S. (ed.), *Proceedings of the Tenth International Conference on Logic Programming*, pp. 257-279, MIT Press (1993).
3) Bonner, A. J. and Kifer, M.: Applications of Transaction Logic to Knowledge Representation, *Proceedings of the First International Conference on Temporal Logic*, pp. 67-81, Springer-Verlag (1994), Lecture Notes in Artificial Intelligence 827.
4) 磯崎秀樹：トランザクション論理におけるプログラム変換，人工知能学会全国大会論文集，pp. 423-426 (1994).
5) 磯崎秀樹：トランザクション論理における知識推論，人工知能学会研究会資料 SIG-FAI-9401-4, pp. 25-32 (1994).
6) 磯崎秀樹：トランザクション論理プログラミングの実装とプランニングへの応用，情報処理学会研究報告，94-AI-92-1 (1994).
7) Kawamura, T. and Kanamori, T.: Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation, *Theoretical Computer Science*, Vol. 75, pp. 139-156 (1990).
8) Reiter, R.: On Closed World Data Bases, Gallaire, H. and Minker, J. (eds.), *Logic and Data Bases*, pp. 55-76, Plenum Press (1978).
9) Reiter, R.: Nonmonotonic Reasoning, Traub, J. F., Grosz, B. J., Lampson, B. W. and Nilsson, N. J. (eds.), *Annual Review of Computer Science*, pp. 147-186, Annual Reviews Inc. (1987).
10) Shoham, Y.: Agent-oriented Programming, *Artif. Intell.*, Vol. 60, pp. 51-92 (1993).
11) Tamaki, H. and Sato, T.: Unfold/fold Transformation of Logic Programs, *Proceedings of the Second International Logic Programming Conference*, pp. 127-138 (1984).
12) 玉木久夫：論理型言語におけるプログラム変換，古川康一，溝口文雄（編），プログラム変換，3章，pp. 39-62，共立出版 (1987).

付録

定理 3.1 基礎連言式 α に対し，証明木 T: $\mathbf{prt}(\alpha; \mathbf{P}; \mathbf{D}_0, \cdots, \mathbf{D}_n)$ が存在するための必要十分条件は $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models \alpha$ である．

[証明] まず最初に $\models_0$ を以下のように定義する．

$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models_0 \alpha \Longleftrightarrow$

証明木 T: $\mathbf{prt}(\alpha; \mathbf{P}; \mathbf{D}_0, \cdots, \mathbf{D}_n)$ が存在する

この時 $\models_0$ が条件 (E1), …, (E5) を満たす最小の関係であることを示す．これが示せれば，補題3.1と定理3.1の証明が同時に完了する．

$\models_0$ が (E1), …, (E5) を満たすことは容易に示せるので省略する．証明木 T: $\mathbf{prt}(\alpha; \mathbf{P}; \mathbf{D}_0, \cdots, \mathbf{D}_n)$ が存在すれば，(E1), …, (E5) を満たす任意の関係 $\models_1$ に対して $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \models_1 \alpha$ となることを示す．これは T の最小の高さを h として，h に関する帰納法を使う．

• $h=1$ の場合，α が基礎原子式 A で，次のいずれか

が成立する．

○$n=0$ かつ $\mathbf{D}_0 \vdash_0 A$.

○$n=1$ かつ $\langle \mathbf{D}_0, \mathbf{D}_1 \rangle A \in \mathcal{B}$.

したがって，(E5)または(E4)より $\mathcal{B}, \mathbf{P}, \mathbf{D}_0 \vdash_1 A$ または $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \mathbf{D}_1 \vdash_1 A$ が成立する．

●$h=m$ まで成り立つと仮定して $h=m+1$ の場合を示す．

○α が $\phi_1 \otimes \cdots \otimes \phi_k$ か $\phi_1 \wedge \cdots \wedge \phi_k$ の場合．

証明木の定義の2番目と3番目の条件，帰納法の仮定，(E1)と(E2)により容易に $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash_1 \alpha$ が示せる．

○α が基礎原子式 A なら，上と同様にして(E1)あるいは(E2)の代わりに(E3)を使えばよい．

補題 5.1 任意の基礎連言式 α に対して，$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$ ならば $n \in l(\alpha)$ が成立する．

[証明] $\mathcal{P}(DB) \cup \mathcal{P}(\mathcal{B})$ の述語だけからなる連言式 α に対して補題が成立することは，α の構造に関する帰納法により容易に示せる．

次に **estimate_pred**（図2）で C に属する述語記号のみを持つ連言式に対して補題の成立を仮定して，原子式 $\alpha=p(\cdots)$, $p \in N$ なら補題が成立することを示す．$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$ なら定理3.1から対応する証明木 T がある．$p \in \mathcal{P}(\mathbf{P})$ なので，T の根はある規則の基礎代入例 $p(\cdots) \leftarrow \beta$ で決まる子節点を持つ．

●β が $\phi_1 \wedge \cdots \wedge \phi_m$ の形をしている場合．

$p \in N$ なので，$\mathbf{preds}(\phi_i) \subset C$ を満たす ϕ_i が一つはある．その条件を満たす任意の ϕ_i に対して，$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \phi_i$ の証明木が T に含まれている．ϕ_i については補題が成り立つという仮定により $n \in l(\phi_i)$ が得られる．すると l の定義から $n \in l(\alpha)$ が成り立つ．

●β が原子式か $\phi_1 \otimes \cdots \otimes \phi_m$ の形をしている場合．

同様に定理3.1から $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \beta$ がいえる．$\mathbf{preds}(\beta) \subset C$ なので $n \in l(\beta)$ が言え，$n \in l(\alpha)$ が得られる．

この結果をもとに，図2の **estimable 1** の終了時点で，C に属する述語記号のみを持つ連言式に対して補題の成立を示すのは容易である．

最後に，Y の述語記号を持つ連言式に対して補題の成立を示す．上の図2の場合と同様に，図3の **estimable 2** において，C に属する述語記号のみを持つ連言式に対しては補題が成立すると仮定して，**estimable 2** の脱出時に原子式 α の述語記号が C に含まれた場合の補題の成立を示す．

図3の条件 $\mathbf{preds}(\phi) \subset C$ を満たす規則を N の終了規則ということにする．$\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$ とすると，定理3.1から対応する証明木 T がある．下に示す補題 A.1 により T のある節点 n で N の終了規則 $H \leftarrow \beta$ で展開されているものがある．このとき，n のラベルを $\mathbf{D}_i, \cdots, \mathbf{D}_{i+j} : H$ とすると，β の述語記号はすべて C の要素なので，$j \in l(\beta)$ が成り立つ．一方，$j \leq n$ と $j^{\succ} \subset l(\alpha)$ により $n \in l(\alpha)$ がいえる．

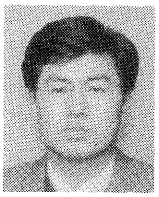
estimate 2 の実行後に $Y \neq \{\}$ であれば $p \subset Y$ に対して $\mathbf{pl}\lceil p \rceil = \{\}$ があるが，このとき p を述語記号として持つ原子式 A に対して，どのようなデータベース列 $\mathbf{D}_0, \cdots, \mathbf{D}_n$ をとっても $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$ とならないことは定理3.1を使って容易に示せる．

補題 A.1（終了規則の証明木） 原子式 $\alpha=p(\cdots)$, $p \in N$, $\mathcal{B}, \mathbf{P}, \mathbf{D}_0, \cdots, \mathbf{D}_n \vdash \alpha$ と仮定する．すると任意の証明木 $T: \mathbf{prt}(\alpha ; \mathbf{P} ; \mathbf{D}_0, \cdots, \mathbf{D}_n)$ は N の終了規則で展開されている節点を持つ．

[証明] T の高さに関する帰納法を用いる．T の高さが最小値である2のときは，基本変更またはデータベースによる証明に相当するので補題は成立する．

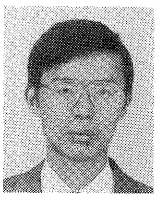
次に T の高さが k まで補題が成り立つと仮定して $k+1$ の場合を示す．T の根が $h(\cdots) \leftarrow \beta$ という形の規則 r で展開されているとする．$\mathbf{preds}(\beta) \subset C$ なら r は N の終了規則である．$\mathbf{preds}(\beta) \not\subset C$ かつ T の根以外に N の述語記号が出現しないとすると，N の定義に矛盾する．したがって証明木の定義から，T は N の述語記号を持つ原子式の証明木を部分木 T' として含む．帰納法の仮定より，T' のある節点は N の終了規則で展開されているので，T も N の終了規則で展開される節点を持つ．

(平成6年9月30日受付)
(平成6年11月17日採録)



磯崎　秀樹（正会員）

1983年東京大学工学部計数工学科卒業．1986年同大学院卒業．同年NTT入社．以後，人工知能の基礎研究に従事．1990年米国スタンフォード大学客員研究員．現在，NTT基礎研究所情報科学研究部主任研究員．電子情報通信学会，人工知能学会，ソフトウェア科学会各会員．



勝野　裕文（正会員）

1974年東京大学理学部数学科卒業．1976年同大学理系大学院修士課程修了．同年日本電信電話公社入社．以後，データベース，人工知能の基礎研究に従事．1988年カナダトロント大学客員研究員．現在，日本電信電話株式会社基礎研究所情報科学研究部主幹研究員．博士(数理科学)．電子情報通信学会，日本数学会，ACM，AAAI各会員．