ゴール書換えモデルに基づく論理型プログラムの並列処理方式†

後藤厚宏††* 相田仁††
田中英彦†† 元岡達††

論理型プログラムは，プログラムの仕様と実行ストラテジを分離して検証できることから，知識情報処理における高度なプログラミングに対応できる能力をもつと期待されている．しかし従来の Prolog 処理系では，処理能力が不足しているだけでなく，逐次処理の制約から実行ストラテジの柔軟性が犠牲になっている．これらの問題に対処するために，論理式の導出過程に忠実なゴール書換え操作に基づいた OR 並列処理方式を提案する．本方式では，ゴールの導出において定義節のすべての選択肢を適用する OR 並列単一化プロセス間で独立性の高いゴールフレームを次々と受け渡しながら並列処理を進める．またゴールフレーム間の依存関係を関係木によって保持し，ゴールフレームの導出機構と制御を切り離している．この結果，プログラムに内在する並列性を活かした高並列処理が実現できると同時に種々の実行ストラテジに柔軟に対応できる．

1. まえがき

論理型プログラムは，プログラムの仕様（論理部）と実行ストラテジ（制御部）を分離して検証できる[1]ことから，知識情報処理[2]における高度なプログラミングに対応できる能力をもつものと期待されている．

しかし，現在用いられている多くの Prolog 処理系[3]には，次のような問題がある．

① プログラムの実行効率を大きく左右するリテラルの実行順序，定義節の選択順序等の実行ストラテジがプログラム中の式の配列によって固定されている．このため式の記述順序によって表現できる範囲でしか実行ストラテジが選択できない．

② 知識情報処理の分野は，その問題の性質から計算量が非常に大きく，現状のままでは計算能力が不足している．

②の問題に対して，われわれはすでに並列 Prolog の実験システム "Paralog"[4] を開発し，論理型プログラムの並列処理の可能性を示した．一方①の原因は，逐次マシン上で導出操作を効率的に進めるために実行ストラテジの柔軟な選択を犠牲にしたことにあり，並列処理によって解決できると考えられる．ただし，論理型プログラムの並列処理方式の検討は一部で行われつつあるが十分とはいえない．

本論文では，Horn 節の図式表現（証明図）を用いて，Horn 節の計算過程を書換え操作によってとらえたゴール書換えモデルを示し（2章），本モデルに基づいた OR 並列処理方式を提案する（3章）．続いて本並列処理方式の得失について議論し，本方式によって，プログラムに内在する並列性を活かした高並列処理が実現でき，同時に種々の実行ストラテジに柔軟に対処できること，さらに言語機能の拡張にも対応できることを示す（4章）．最後に今後の検討課題を列挙する（5章）．

2. ゴール書換え操作に基づく論理型プログラムの実行

論理型プログラムの実行を "論理式の導出過程[5]に忠実なゴール書換え操作" としてとらえたモデルを，ゴール書換えモデルと呼ぶことにする．本章では，Horn 節からなる論理型プログラム（仮に純 Prolog と呼ぶ）を想定し，証明図を用いて本モデルの基本概念を明確にする．

2.1 証明図による節のテンプレート表現

純 Prolog プログラムにおける初期ゴール節はゴールリテラルの AND 結合である．一方，定義節は頭部に示されたリテラルをボディ部リテラルの組（または空）へ書き換える指示であり，述語ごとに複数ありうる．

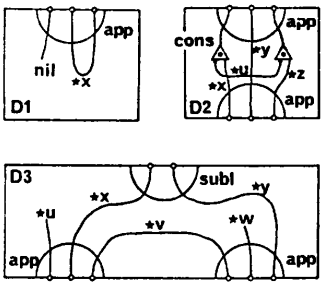
証明図では，この Horn 節を**図1**に示すような定義節テンプレートとゴールテンプレートによって表す．

ここで初期ゴール節内および定義節ボディ部内のゴールリテラルは，テンプレート内の上半円に対応

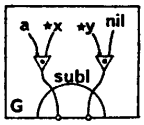
† Parallel Processing of Logic Programs Based on Goal-Rewriting Model by ATSUHIRO GOTO, HITOSHI AIDA, HIDEHIKO TANAKA and TOHRU MOTO-OKA (Department of Electrical Engineering, University of Tokyo).

†† 東京大学工学部電気工学科

* 現在 日本電信電話公社



(a) 定義節テンプレート
(b) Definition templates



(b) 初期ゴールテンプレート
(b) Initial goal template

[D1] append (nil, *X, *X) ←.
[D2] append ((*U . *X), *Y, (*U . *Z)) ← append (*X, *Y, *Z) .
[D3] sublist (*X, *Y) ← append (*U, *X, *V), append (*V, *W, *Y).
[G] ← sublist ((a . *X), (*Y . nil)) .

図 1 Horn 節のテンプレート表現
Fig. 1 Template representation of Horn clauses.

し，定義節の頭部リテラルは下半円に対応する．

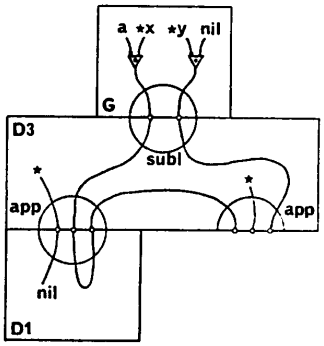
リテラルの引数は図中の小円（引数ポート）によって示され，値が決まっているときは関数式や定数とリンクで結ばれる．リテラル引数や関数式の引数が変数のときは，テンプレート内の同一変数ごとに引数間をリンクで結び，その相互関係を示す．

2.2 ゴールフレーム

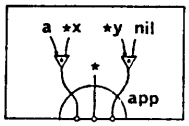
入力導出による純 Prolog の実行は，初期ゴールを出発点とし，ゴール節中のリテラルに定義節を適用（単一化）して新たなゴール節に書き換える操作を，ゴール節が空になるまで繰り返すことである．

証明図では，この実行過程における中間結果としてのゴール(中間ゴール)をテンプレートの組合せとして図式的に表現できる．たとえば，図1のプログラムにおいて，初期ゴール節に定義節 D3 を，続いて D1 を適用して得られる中間ゴールは**図2**(a)のようになる．

中間ゴールの導出において実行する単一化操作は，述語引数の照合操作，つまり接続したい両リテラルの引数ポートから両テンプレートをたどり，テンプレートの接続における矛盾がないことを検出する操作に相当する．単一化が成功したリテラルは円によって示され，まだ単一化が必要なゴールリテラルは上半円として残る．また，単一化の失敗はテンプレートの接続における矛盾として現れる．



(a) テンプレートの接続として表現したゴールフレーム
(a) Connection of templates



(b) 縮退を施したゴールフレーム
(b) Reduced form of the same goal frame

図 2 ゴールフレームの例
Fig. 2 Example of a goal frame.

テンプレートの組合せで表現した中間ゴール（初期ゴールテンプレートを含む）をゴールフレーム（goal frame）と呼ぶことにする。ゴールフレーム内には，ゴールリテラルとそれに付随した単一化の環境がすべて含まれている．ここで単一化の環境とは，それまでに行われた単一化によって生じた変数の束縛情報である．

2.3 ゴールフレームの縮退操作

上述のゴール書換え操作の本質はゴールフレームの縮退操作を導入すると，より明確になる．

縮退操作とは，テンプレートの接続によって表現されたゴールフレームから，それ以降の導出操作で利用しない情報を取り除く操作である．たとえば，図2(a)のゴールフレームは，ゴールリテラルの引数ポートから参照されない（リンクをたどれない）情報や，成功した単一化に関する引数ポートの接続情報を取り除くことによって図2(b)のようになる．

従来の逐次的な Prolog の実行[3]では，ゴールフレームの生成における“やり直し（後戻り）”を効率的に行う必要から，スタックを利用してテンプレートの組合せを記憶し，完全な縮退操作は行っていない．このため，変数の束縛情報が必要になった時点でリンクをたどる必要があった．しかし後述する定義節の選択肢のすべてを適用する処理方式では，やり直しの必要がなく，ゴールフレームを縮退した形で取り扱うことができる．

3. ゴール書換えモデルの OR 並列処理方式

3.1 論理型プログラムの並列処理性

ゴール書換えモデルによる論理型プログラムの実行過程には，次の4種類の並列処理性が明らかである．

① リテラル内引数間並列処理：ゴールリテラルと定義節テンプレートとの照合操作において，複数の引数ポートがある場合に，それらのポートの照合操作を並列に行う．

② 定義節テンプレート間並列処理：ゴールフレーム内の特定のゴールリテラルと同じ述語名をもつ定義節テンプレートが複数あるときに，それらとの照合操作を並列に行う．

③ ゴールリテラル間並列処理：ゴールフレーム内にゴールリテラルが複数ある場合に，それらに対する照合操作を並列に行う．

④ ゴールフレーム間並列処理：複数のゴールフレームが処理系内に存在するときに，それらを並列に処理する．

3.2 OR 並列単一化プロセスによるゴールフレーム間並列処理

あるゴールフレーム中のリテラルについて定義節テンプレートの選択肢を並列に照合し(②)，照合が成功したおのおのについて新たなゴールフレームを生成すると，生成された複数のゴールフレームは非決定性の意味での OR 関係をもつ．これらについて同様の処理を繰り返せば，ゴールフレーム間並列処理へ自然に引き継ぐことができる．

上述の処理は，次の3種のサブプロセス（起動，照合，縮退）から成る OR 並列単一化プロセスとして考えることができる．

① 起動サブプロセス：入力したゴールフレームのなかからリテラルを選択し，それと対応する定義節テンプレート数の照合サブプロセスを起動する．

② 照合サブプロセス：各照合サブプロセスは，それぞれ {ゴールフレーム，定義節テンプレート} の組について照合操作を行う．成功した場合は変数の置換の組を生成し，縮退サブプロセスを起動する．また失敗した場合は即時に消滅する．

③ 縮退サブプロセス：縮退サブプロセスは {ゴールフレーム，定義節テンプレート，変数の置換の組} から新たなゴールフレームを生成し，消滅する．

ゴールフレームは OR 並列単一化プロセスによって消費され，新たに（複数の）ゴールフレームがプロセスによって生成される．生成されたゴールフレームは新たなプロセスを起動し，全体としては，多数のプロセスが次々と，生成/消滅を繰り返すことによりゴールフレーム間並列処理が実現される（**図3**）．

3.3 関係木によるゴールフレーム管理と制御

論理型プログラムに Horn 節を越えた言語セマンティクスが導入される場合や，5.1 節に示すような AND 並列処理へ拡張する場合は，ゴールフレーム間にいろいろな関係が生じる．

ゴールフレームは OR 並列単一化プロセスによって次々と生成/消費されてゆくので，互いの関係をゴールフレーム上に記録する方法は，相手の認識方法においてむずかしい．そこで，ゴールフレーム間の関係を表す関係木を別に作っておく．

関係木は根ノード，中継ノード，葉ノードによって構成できる．根ノードは，初期ゴールの入力プロセスに，また中継ノードは原則として各 OR 並列単一化プロセスに相当する．

ゴールフレーム間の関係は各プロセスから生成されるゴールフレームのグループごとに設定されると考えられる．そこで各中継ノードに，そのプロセスから生成されたゴールフレーム間の関係をノード属性として与えることにする(**図4**)．一方，ある時点でシステム内に存在する多数のゴールフレームを関係木上で特別な葉ノード（ゴールノード）と考え，そのゴールフレームの活性／不活性状態やメタ述語に起因するゴー

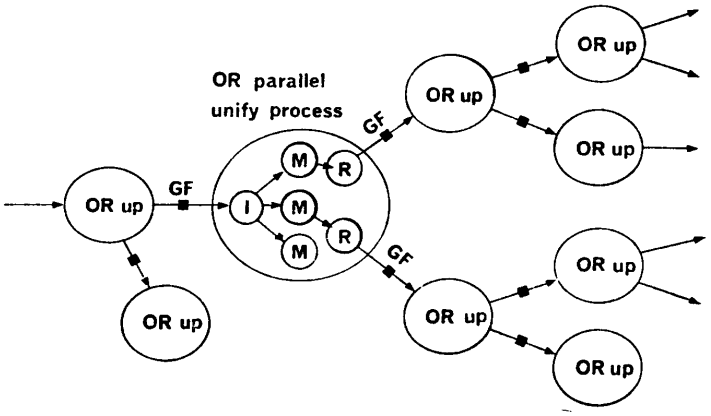


図3 OR 並列単一化プロセスによるゴールフレーム間並列処理
Fig. 3 Inter-goalframe parallel processing by OR-unify processes
I: initial sub-process, M: matching sub-process,
R: reduction sub-process

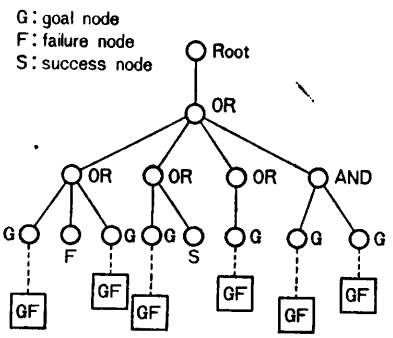


図 4 関係木によるゴールフレーム管理
Fig. 4 Management of goal frames using relation-tree

ルフレーム自体の属性を，そのノード属性とみなす．

OR 並列単一化プロセスにおいてゴールフレームの導出がすべて失敗した場合，ゴールノードは“失敗”の属性をもった葉ノードに置き代わる．新たなゴールフレームが（複数）生成された場合は，中継ノード（通常 OR 属性）と新ゴールフレームに相当するゴールノードから成る部分木に置き代わる．

このような関係木を導入することにより，実行途中における多数のゴールフレームは関係木の末端のゴールノードとして位置づけることができ，あるゴールフレームの状態によって他のゴールフレームを消滅／一時不活性化／活性化する等の影響を及ぼし合い，全体としてさまざまな機能を実現することが可能である．

4. 並列処理モデルの評価・検討

4.1 他の並列処理方式の研究例

Conery[6] らは，多数の独立したプロセス間でのメッセージ通信に基づいた並列処理モデル（AND/OR モデル）を提案している．AND プロセスは，ゴールリテラルの AND 結合を解くために，各リテラルについて ORプロセスを起動する．OR プロセスは，リテラルに適用可能なすべての定義節との単一化を行い，成功したものについてボディ部を解く AND プロセスを起動する．並列処理は木状に生成されたプロセス間で成功／失敗／再試行のメッセージを送受しながら進められる．

Minker[7] らが開発している実験システム PRISM は，定義節を格納して単一化を行うデータベース（EDB, IDB）と問題解決マシン（PSM）の 3 種のモジュールから成る．PSM が扱うゴール木の各ノードは，手続きとして呼び出された定義節のボディ部から成り，根ノードから各葉ノードまでを合わせると，縮退していないゴールフレームに相当する．

他には Darlington[8] らの研究等がある．

4.2 処理の独立性と高並列性

高並列処理を実現する上で最も重要な課題は，個々の処理単位の独立性である．

与えられた問題を構成する各処理単位間には当然なんらかの依存性があり，各処理単位間の独立性はある範囲に制限される．しかし，通常は処理方式が独立性を過大制限している場合が多い．

ゴールフレーム内には，一般に複数のゴールリテラルがあり，また，一つのリテラルの単一化操作においてゴールフレーム全体の情報は必ずしも必要としない．このため AND/OR モデルのように，単一化を行うリテラルとそれに付随する環境を切り出して，子プロセスを起動したり，PRISM のようにゴール木を生成していくことも考えられる．しかし，この方式には以下のような問題がある．

① ゴールフレーム内の各リテラルは共有変数を介して強く依存し合う場合が多いため，子プロセスにおける照合結果をまち，それを用いて親プロセスが残りの環境に操作を施す必要がある．

② 照合が複数成功した場合は，共有している残りの環境に対して操作が集中し，親プロセスが隘路となる危険性がある．

③ 高並列マシンのモジュール間結合網では遅延が問題となりやすいため，木状に生成されたプロセス群間の頻繁なメッセージ通信は，負荷分散をむずかしくする．

これに対し，前章で述べた並列処理方式では，ゴールフレームを渡してプロセスを起動するため，1 回の通信量は AND/OR モデルに比して大きいが，

① プロセス間の通信は単方向であり，プロセス間の待合せが不要である，

② 共有部分をもたないため，隘路が生じにくい，

③ 各ゴールフレームは，それ自身が一つの問題として独立しているため，負荷分散の自由度が大きい，

等の点において高並列処理に適する．

4.3 シミュレーション評価

本並列処理の並列度を調べるシミュレータを作り，それが非常に大きいことを確かめた（**図 5**）．ここで depth（横軸）はゴール入力時からのプロセス起動回数を示し，その各時点における単一化の成功数（縦軸）は次に起動されるプロセス数つまり並列度を示す．

また縮退の効果によってゴールフレームがそれほど

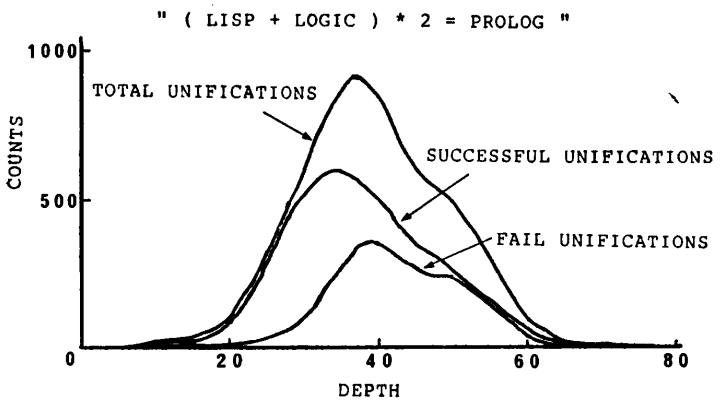


図 5 覆面算プログラム "(LISP+LOGIC)×2=PROLOG" の並列度
Fig. 5 Parallelism of an example program.

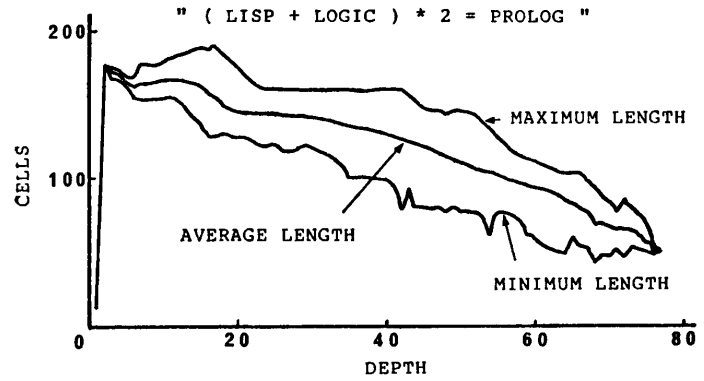


図 6 覆面算プログラムにおける縮退したゴールフレームの大きさの推移
Fig. 6 Length of goal frames in reduced form.

大きくならないこともわかった(図 6). これは, ゴールフレームを渡すことによって新たなプロセスを起動する負荷が, それほど大きくないことを示している. ただし大きな構造データについては, マシンアーキテクチャにおいて構造メモリを検討することも必要かもしれない.

4.4 実行制御の柔軟性

従来の Prolog 処理系では, 逐次マシン上で後戻りを用いた導出操作が効率的にできる半面, その処理方式によってリテラルの実行順序, 定義節の選択順序等が制限を受けてきた.

本並列処理方式では, 各プロセスが縮退操作を施したゴールフレームの全体を扱う. このため処理方式によってリテラルの実行順序が規定されず, 起動サブプロセスにおいて, ゴールフレーム内の全リテラルの中から実行すべきリテラルが選択できる.

また OR 並列単一化プロセスの環境では後戻り自体が不要である. このためゴールフレームを選択してプロセスを起動することにより, 縦型探索/横型探索等のストラテジが実現できる.

一方 PRISM では, リテラルの選択範囲がゴール木の各ノード (定義節のボディ部) に制限される.

4.5 言語機能の拡張への対応

AND/OR モデルではゴールの導出プロセスが, また PRISM ではゴール木が, ゴール間の依存関係に相当する情報を蓄える. 一方, 本処理方式では, ゴールフレームとその導出機構から, ゴールフレーム間の依存関係を関係木として切り離している. このため言語機能を拡張する場合は, 関係木のノードに必要に応じた属性を与え, ノードを操作するコマンドを導入すればよい (以下では, ノード間でコマンドを送受するとみなす. 4.6 節参照).

たとえば, メタ述語として not[9] を論理型プログラムに導入する場合は次のようになる. not を含むゴールフレーム

?—not (goal 1), goal 2.

において goal 1, goal 2 間に共有変数がないとする. このとき図 7 (a) のよ

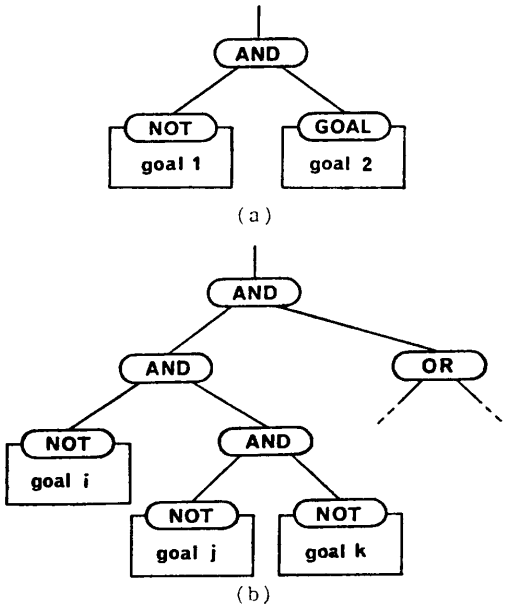


図 7 メタ述語 not の実現例
Fig. 7 Execution of meta-predicate "not".

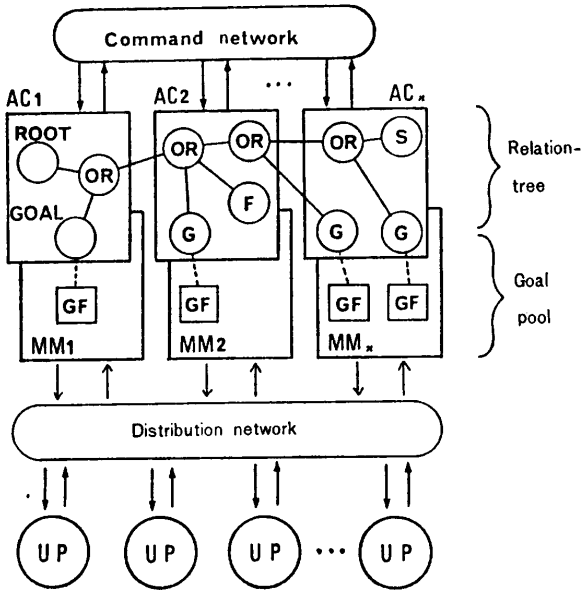


図 8 マシンアーキテクチャの概観
Fig. 8 Abstract machine architecture.

うに and ノード（属性 AND）と not ノード（属性 NOT）を用いて2個のゴールフレームに分け，goal 1 と goal 2 を並列に実行する．

このとき goal 1 の実行では，同図(b)のように中継ノードがすべて not ノードによって構成される．

not ノードは，OR 並列単一化プロセスによる導出が空節にいたると失敗ノードとなり，導出がすべて失敗したときに成功ノードとなる．子ノードは成功／失敗をコマンドによって親ノードに知らせる．

and ノードは，すべての子ノードが成功したときに自分も成功ノードとなる．一方，子ノードの一つが失敗ノードとなったときは，子ノードを強制終了させるコマンドを他の子ノードに送り，実行が不要となったゴールフレームを消滅させ，対応する関係木の末枝を刈り込む．

4.6 マシンアーキテクチャの抽象モデル[10]

OR 並列単一化プロセスによるゴールフレーム間並列処理の抽象モデルを図8に示す．

UP は OR 並列単一化プロセスに対応する処理要素である．また，本並列処理がもつ負荷分散の自由度を抑制しないように，各 UP にはすべての定義節をもつメモリ（DM）が付随していると考える（図 9）．

一方，goal pool は実行待ちの多数のゴールフレームを蓄える．

ゴールフレーム間の関係木は goal pool を構成する記憶モジュール（MM）のおのおのに用意されたコントローラ（AC）によって管理される．したがって先に述べたノード制御コマンドは，コマンドネットワークを介して AC 間で送受される．

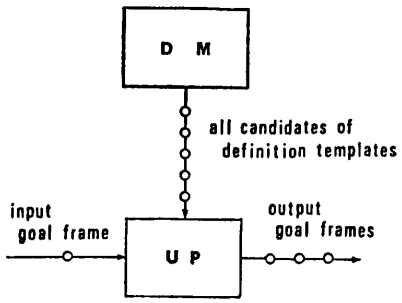


図 9 UP (unify processor) と DM (definition memory)
Fig. 9 UP (unify processor) and DM (definition memory).

5. 今後の研究課題

5.1 AND 並列処理への拡張

ゴールフレーム内の複数ゴールリテラルを並列処理する方式（3.1節③）は，与えられた問題の AND 分割に相当し，原則としてリテラル間で共有する変数についての無矛盾性検査 (consistency check)[4]が必要である．ただし，この検査は容易ではなく，また分割したことによってその部分問題が停止しない場合もあるため，無矛盾性検査を必要としない範囲に並列処理を制限したほうがよい．

最も実現が容易な方法は，ゴールフレームが共有変数をもたないリテラルまたはそのグループに分割できるとき，それらを AND 関係にある別々のゴールフレームとして処理を進めるものである．先に述べた縮退操作によってゴールフレーム中の変数が未定義変数に限られていれば，このような分割の可否の判定は比較的容易である．たとえば4.3節のメタ述語 not の実現例では，not 内のリテラルと外側のリテラルの間に原則として共有変数をもたないため，並列処理が可能である．

さらに並列処理を進めるには，次のようにすればよい．まず単一化時にリテラル間の共有変数を参照または値を束縛するリテラルを一つに限り単一化を行う．このとき，その共有変数を参照しないことが確かなリテラルも並列に単一化を行う．ただし，単一化時に共有変数を参照するかどうかの判定は容易ではないため文献11）にあるような read-only annotation 等のプログラマによる指示が必要であろう．また分割された

ゴールフレームの部分間で値の受け渡しがあるため，負荷分散の自由度は制限されることになる．

5.2 処理の効率化

本並列処理方式の並列度は非常に大きく，高並列マシンにおいてもプロセッサ数を容易に上回る．そのためプロセスの起動を自由にしたままでは数の爆発を起こし，資源の枯渇を招く危険性がある．そこで縦型／横型探索に加えて，実際の有限資源の環境下で効率的に処理を進める探索ストラテジを導入してプロセスの起動制御を行う必要がある．

また，リテラルの実行順序は処理効率に大きく影響するため，最終的に失敗に至るようなゴールフレームを数多く生成しないものを優先して実行することが望ましい[12]．

6. おわりに

本論文では，論理型プログラムを高速に実行するマシンアーキテクチャの開発に向けて，ゴール書換えモデルとそれに基づいた OR 並列処理方式について述べた．本方式は論理型プログラムにおける論理と制御の分離をそのまま実行モデルに反映しているため，高並列処理と柔軟な実行ストラテジの両者が実現できる．

参考文献

1) Kowalski, R.: Algorithm＝Logic＋Control, *CACM*, Vol. 22, No. 7, pp. 424-436 (1979).
2) Moto-oka, T. (ed.): *Fifth Generation Computer Systems*, North-Holland, Amsterdam (1982).
3) Warren, D. H. D.: Implementing Prolog—Compiling Predicate Logic Programs, D. A. I. Research Reports 39-40, University of Edinburgh (1977).
4) 相田 仁他：並列 Prolog 処理システム "Paralog" について，情報処理学会論文誌，Vol. 24, No. 6, pp. 830-837 (1983).
5) Robinson, J. A.: Computational Logic—The Unification Computation, *Machine Intell.*, Vol. 6, No. 4, pp. 63-72 (1971).
6) Conery, J. S. and Kibler, D. F.: Parallel Interpretation of Logic Programs, Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, pp. 163-170 (1981).
7) Kasif, S., Kohli, M. and Minker, J.: PRISM—A Parallel Inference System for Problem Solving, Logic Programming Workshop 83, pp. 123-152 (1983).
8) Darlington, J. et al.: ALICE: A Multi-Processor Reduction Machine for the Parallel Evaluation of Applicative Languages, Proc. 1981 Conf. on Functional Programming Languages and Computer Architecture, pp. 65-76 (1981).
9) Clark, K. L.: *Negation as Failure*, *Logic and Data Bases*, pp. 293-322, Plenum Press, New York (1978).
10) 後藤厚宏他：高並列推論エンジン PIE について，Logic Programming Conf. '83, ICOT (1983).
11) Shapiro, E.: A Subset of Concurrent Prolog and Its Implementation, TR-003, ICOT (1983).
12) Warren, D. H. D.: Efficient Processing of Interactive Relational Database Queries Expressed in Logic, Proc. 7th International Conf. on VLDB, pp. 272-281 (1981).

(昭和58年 8 月 2 日受付)
(昭和58年10月11日採録)