記 号 処 理 45－2
（1988. 3. 23）

PROLOGコンパイラにおける非決定性処理の最適化方式

松本一夫　砺崎賢一　上原邦昭　豊田順一

大阪大学産業科学研究所

本報告では、PROLOGの非決定性処理を最適化する2つの方式を提案する。PROLOGの非決定性処理は、後戻り処理によって複数の節を試行することにより実現されている。したがって、非決定性処理を最適化するためには節間の関係を利用する必要がある。提案する2方式は、節を単位とする従来の最適化方式と異なり、述語を単位として最適化を行うものである。

まず初めに、述語単位での変数分類による述語内ローカル変数の導入と、分類情報を利用して最適化を実現するための処理系アーキテクチャを提案する。この方式は、後戻り処理時の単一化処理を効率化することによって非決定性処理を最適化するものである。

次に、カットや組込み述語の特性を利用したプログラム変換によって非決定性処理を最適化する方式を提案する。この方式は、本質的に決定性の述語でありながら、PROLOGの言語仕様上の制約から非決定的に実行されている述語を、インデキシング手法により決定的に実行される形式に変換するものである。変換された述語は、後戻り処理が削除されるため高速な実行が可能になっている。

Optimization Method for Non-determinate Predicate of PROLOG

by Kazuo MATSUMOTO, Ken'ichi KAKIZAKI, Kuniaki UEHARA and Jun'ichi TOYODA

The Institute of Scientific and Industrial Research

Osaka University, Mihoga-oka, Ibaraki, Osaka 567, Japan

The main computational mechanisms of PROLOG are "unification" and "backtracking". These mechanisms are usually implemented by use of WAM(Warren's Abstract Machine) architecture. However, optimization methods used in WAM works well only for determinate and backtracking-free predicates. To execute any PROLOG programs efficiently, optimization method for non-determinate predicates should be added on.

In this paper, we will, first, present an optimization method for non-determinate predicates consist of a new variable classification method and an new architecture for performing efficient backtracking. This classification method classifies the variables which take the same value in some clauses into "predicate local variables". Unification of these variables in non-deternimate predicates can be removed except for the first time in this architecture. Second, we will discuss an optimization method by using program transformation technique. In this method, a non-determinate predicate is transformed into determinate predicates. With this optimization, a non-determinate predicate can be executed so fast by indexing technique without backtracking.

1．はじめに

PROLOG プログラムの多くは非決定性処理によって実行される。非決定性処理は、後戻りによって複数の節を試行する処理であるため、その最適化を行うためには節間の関係を利用する必要がある。しかしながら、従来の最適化方式は節を単位としたものであるため、非決定性処理の効率化のためには有効でないという問題があった。この問題を解決するために、本報告では、述語を単位として非決定性処理の最適化を行う２つの方式を提案する。

第１に、述語を単位とする新たな変数分類法と、この分類によって得られた情報を用いて後戻り処理を効率化するためのアーキテクチャを提案する。この変数分類法は、述語中の複数の節において同一の値をとる変数を「述語内ローカル変数」[4]に分類するものである。また、提案するアーキテクチャにより、後戻り処理時の述語内ローカル変数の単一化処理を削除することで、非決定性処理の効率化を図っている。

第２に、プログラム変換の手法を用いて、非決定性処理が行われる述語を決定性処理が行われる述語に変換することにより最適化を行う方式を提案する。この方式は、非決定性処理が行われる述語が特定の条件を満たしている場合に適用される。変換された述語では、節の選択がインデキシングにより決定的に行われるため、後戻り処理を伴わない高速な実行が可能になる。

2．非決定性処理のオーバーヘッド

PROLOG 処理系では、非決定性処理を実現するために以下に示す４つの処理を行っている。

〈1〉選択点の生成と削除処理の実行
〈2〉後戻り処理の実行
〈3〉単一化処理情報の破棄
〈4〉単一化処理の再実行

従来の処理方式では、これらの処理が無駄に行われることがあった。たとえば、図１に示す partition/4 では、第１引数がリストで、変数 X の値が Y の値以下である場合には、次の順に処理が行われる。

1) 節 A) の実行の失敗に備えて、選択点を生成
2) すべての引数の単一化処理
3) 変数 X, Y の大小比較
4) カットによる選択点の削除
5) partition/4 の再帰呼び出し

この場合は、後戻り処理が行われることはないため、1), 4) での選択点の生成および削除は無駄な処理となっている。

```
:- mode partition(+,+,-,-).
A) partition([X|L],Y,[X|L1],L2) :-
       X =< Y,!,
       partition(L,Y,L1,L2).
B) partition([X|L],Y,L1,[X|L2],) :-
       partition(L,Y,L1,L2).
C) partition([],_,[],[]).
```

図１　非決定性処理が行われる述語

次に、変数 X の値が Y の値より大きい場合の処理を示す。

1) 節 A) の実行の失敗に備えて、選択点を生成
2) すべての引数の単一化処理
3) 変数 X, Y の大小比較
4) 後戻り処理
5) 単一化処理によって得られた情報の破棄
6) 選択点の削除
7) すべての引数の再単一化処理
8) partition/4 の再帰呼び出し

この場合、2) の単一化処理で第１引数に与えられたリストの分解を行っているが、同様の処理が 8) でも行われている。また、2) では第３引数の単一化処理によってリストを作成するが、節 B) が選択される場合にはこの処理は無駄な処理となる。本報告で提案する方式は、これらの処理のオーバーヘッドを取り除くことにより、非決定性処理を最適化するものである。

3．述語単位の変数分類と処理系アーキテクチャ

非決定性処理の最適化方式として、述語を単位とした新たな変数分類法を提案し、その分類を利用して高速な処理を実現するための処理系アーキテクチャについて述べる。

3.1 述語内ローカル変数

PROLOG コンパイラは述語内の変数を出現状況によって分類し、この分類に基づいた情報を用いて効率のよい単一化処理を行うコードを生成している。従来は節を単位として変数分類を行い、節のヘッド部または第１番目のサブゴールのみに出現する変数をテンポラリ変数、それ以外の変数をローカル変数に分類している。しかしながら、このような節単位の変数分類では、非決定性処理の最適化に利用できる情報は得られないという問題がある。たとえば、図１の節 A), B) における変数 X, L, Y は、どちらの節においても単一化処理によって同じ値をとる。しかしながら、従来の節単位の変数分類法では、これらの変数が同じ値をとることを判別できないために、各節ごとに単一化処理によってあらためて値を求めなおさなければならなかった。

このオーバーヘッドを取り除くことによる最適化を実現するために、述語を単位とする新たな変数分類法を提案し、述語内ローカル変数を導入する。述語内ローカル変数とは、同じ述語中の異なる節にあっても単一化処理によって同一の値をとる変数であり、以下の条件を満たすものである。

1) 入力モードの変数である
2) 同一の引数の、共にレベル１の変数であるか構造中の同じ位置にある変数である
3) 節内で初めて出現する位置が共に等しい変数である

3.2 述語内ローカル変数を利用する最適化方式

述語内ローカル変数を利用する最適化方式によって生成されたコンパイルドコードでは、単一化処理によって一度求められた述語内ローカル変数の値は、最後の候補節の試行が終了するまで述語単位で保持される。候補節の再試行で、述語内ローカル変数に分類される変数の値が再度必要になった場合には、従来の方式で

は再び単一化処理を行わなければならないが、本方式では保持されている値を単に参照するのみでよい。この結果として、重複した単一化処理を除去することができ、再試行処理を高速化することができる。

後戻り処理をともなう非決定性処理は、通常以下のような順序で実行される。

(1) 選択点の生成
(2) ヘッド部の単一化処理あるいはボディー部のサブゴールの失敗
(3) 失敗した節において束縛した変数の解放
(4) 選択点の修正（または削除）
(5) 引数レジスタの再設定
(6) 候補節のヘッド部の単一化処理
(7) 候補節のボディー部のサブゴールの実行

(1) の処理では、(5) の処理で引数レジスタの再設定を行うために、述語が呼び出された時点での引数レジスタの内容を選択点に保存している。しかしながら、述語内ローカル変数を導入することにより、次の場合には引数レジスタの再設定の必要がなくなるため、レジスタの内容を保存する必要がなくなるという利点がある。

1) 引数が述語内ローカル変数である
2) 引数が複合項で、そのすべての要素が述語内ローカル変数である

また、単一化処理の部分実行により (6) の手間も減ることになるために、従来の処理に比べて、(1), (5), (6) の部分で高速化が図れる。

3.3 抽象マシンアーキテクチャとスタックの操作方式

述語内ローカル変数の特性を有効に利用し、後戻り処理を効率的に実行する抽象マシンアーキテクチャについて述べる。従来の抽象マシン WAM[1] は、複数のスタックとレジスタ群を持っており、レジスタの一部はスタックを管理するために利用されている。ローカルスタックと呼ばれるスタック（図2）には、抽象マシンの実行状態を示すデータとして、選択点のデータ構造（選択点フレーム）とローカル変数を保持するのデータ構造（ローカル変数フレーム）が格納されている。また、ローカルスタックを管理するためのレジスタは v レジスタと呼ばれ、スタックの先頭へのポインタが格納されている。

述語内ローカル変数を利用する処理系アーキテクチャは、従来の抽象マシンのローカルスタックの操作方式を拡張したものになっている。このアーキテクチャでは、述語内ローカル変数の値を格納するための領域を述語内ローカル変数フレーム（p_local フレーム）と呼び、ローカルスタック上に確保している。また、p_local フレームの先頭へのポインタを格納するために、p レジスタを新たに設けている。なお、従来の節単位のローカル変数を格納する領域は節内ローカル変数フレーム（c_local フレーム）と呼ぶことによって p_local フレームと区別する。

述語内ローカル変数を利用した最適化を行うためのローカルスタックの操作法について述べる。従来の方式では、述語が呼び出されると、まず後戻り処理のための選択点フレームが生成され、

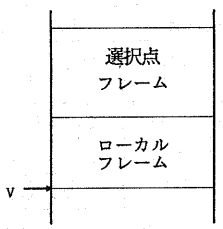


図2　従来のローカルスタック

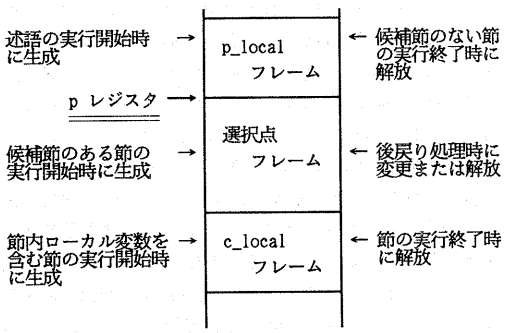


図3　ローカルスタックの操作

次に、選択された節で利用される c_local フレームが生成される。本方式では、このような操作に加えて、述語が呼び出された時点で選択点フレームが生成される前に p_local フレームを生成する操作（図3）を行っている。後戻り処理時には、選択点フレームと c_local フレームの解放などの操作が行われるが、p_local フレームは選択点フレームより前にあるために影響を受けず、複数の節にわたって値を保持することができる。p_local フレームは最終的に、最後の候補節の実行が終了した時点で解放される。本アーキテクチャでは、このようにして p_local フレームを管理し保持内容を利用することで、非決定性処理の高速化を実現している。

3.4 命令セット

述語内ローカル変数を利用し、最適化されたコンパイルドコードを生成するための命令セットの一部を表1に示す。拡張される命令コードは、以下の4種類に分類される。

1) control 系命令
　述語ローカル変数の格納領域を確保・解放する
2) get 系命令
　述語内ローカル変数と引数レジスタの単一化処理を行う。
3) unify 系命令
　複合項内の述語内ローカル変数の単一化処理を行う。
4) put 系命令
　述語内ローカル変数の値を引数レジスタに設定する。

従来の抽象マシンの命令セットでは、単一化処理用の命令は後戻り先を示す情報を持っていない。このため、単一化処理が失敗した場合の後戻り先は try_me_else 等の選択点を操作する命令によって節単位で指定されており、単一化処理が失敗した場合に

Get Instructions	Put Instructions
get_variable Pn,Ai get_value Pn,Ai,L	put_variable Pn,Ai put_value Pn,Ai
Unify Instructions	Control Instructions
unify_variable Pn,Ai unify_value Pn,Ai,L	p_allocate N p_deallocate N p_deallocate0

表1　命令セットの拡張

は、これらの命令によって指定された節が実行されるようになっている。この方式では、後戻り処理によって再試行される節のすべての単一化処理が行われることになる。これに対し、述語内ローカル変数を利用して最適化されたコンパイルドコードは、すでに単一化処理の終了している述語内ローカル変数を除く引数についてのみ単一化処理を行うことで高速化を図っている。このため、単一化が失敗した場合の後戻り先は、どの引数の単一化処理が失敗するかによって変えなければならない。このような機能を実現するために、本アーキテクチャに基づく単一化処理用の命令は、従来の命令と異なり単一化処理が失敗した場合の後戻り先を引数として指定できるようになっている。最適化されたコンパイルドコードでは、この引数で後戻り先を指定することによって、不必要な単一化処理を除去することが可能になっている。

3.5 構造共有法を採用している処理系への実装

3.3 節に示したアーキテクチャとスタック操作法は、複合項の表現法として構造複写法を採用している処理系を前提としている。このため、構造共有法を採用している処理系では、単一化処理によって作成されるデータ構造が問題となり、前述のローカルスタックとともに、データ構造が格納されるグローバルスタックの管理法を変更しなければならない。

構造共有法では、単一化処理によって作成される新たな複合項はモレキュールと呼ばれるデータ構造で表現され、単一化処理が行われる時点でグローバルスタック上のもっとも新しい領域に作成される。この領域は選択点が生成された時点でのグローバルスタックよりも新しい領域に作成されることになるため、現在選択されている節の実行に失敗した場合には後戻り処理によって解放されることになる（図4 a）。従来の節を単位とするコンパイル方式で生成されているコンパイルドコードでは、異なる節で作成されたデータ構造を利用しないため、後戻り処理でグローバルスタック上のデータ領域を単純に解放することができた。

しかしながら、本方式では、単一化処理によって作成されたモレキュールが述語内ローカル変数の値となる場合には、後戻り処理で単純にグローバルスタック上のデータ領域を解放する操作を行うと、述語内ローカル変数に束縛されているモレキュールも解放されてしまうため、別の節で値を参照することができなくなるという問題がある。この問題を解決するために、述語が呼び出された時点で p_local フレームを生成する処理に加えて述語内ローカル変数用のモレキュールの領域を確保する処理を行うようにしている（図4 b）。また、この領域にアクセスするために述語内ローカル変数用ローカルスタックポインタ（レジスタ p1）を

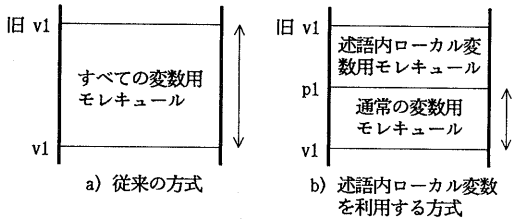


図4　グローバルスタックの操作

設けている。

3.6 従来の最適化方式との関係

述語内ローカル変数を用いた処理では、従来とは異なる単一化処理が行われる。このため、従来の最適化方式のうち、単一化処理を最適化するものを利用する場合には、それらの方式との関係を考慮する必要がある。ここでは、述語内ローカル変数の利用による最適化方式と、レジスタ割り当ての最適化およびインデキシング処理との関係について述べる。

3.6.1 レジスタ割り当ての最適化との関係

レジスタ割り当ての最適化は、データフローの解析に基づき単一化処理に用いるレジスタを適切に割り当てる方式である。この最適化により、変数の単一化処理や引数の設定処理を削除することができる。この方式は多用されているが、述語内ローカル変数を利用する最適化方式と両立しない場合がある。たとえば、図5に示す delete/3 の第1節の第1引数の変数 X の単一化処理は、レジスタ割り当ての最適化を行うことにより削除できる。しかしながら、述語内ローカル変数を利用した処理を行う場合には、この変数の単一化処理が必要になる。このように、これらの最適化方式が競合する場合には、どちらかの方式を選択しなくてはならない。

最適化手法の選択は、変数分類の結果に基づいて行っている。一般的に非決定性の述語では、レジスタ割り当ての最適化よりも述語内ローカル変数を利用した最適化の方が効果が大きい。しかしながら、最適化の対象となる変数がその述語における唯一の述語内ローカル変数である場合は、p_local フレームの生成および解放の処理を新たに行う必要があるため、最適化の効果が小さくなる。したがって、レジスタ割り当ての最適化を行えない述語内ローカル変数が他に存在する場合にのみ、述語内ローカル変数を利用する最適化を選択している。

```
:- mode delete(+,+,-).

delete(X,[X|Y],Y).
delete(X,[Y|Z],[Y|W]) :- delete(X,Z,W).
```

図5　delete/3 の定義

```
:- nth3(+,+,-).
A) nth3(1,[X,Y,Z],X).
B) nth3(2,[X,Y,Z],Y).
C) nth3(3,[X,Y,Z],Z).
D) nth3(X,[X,Y,Z],1).
E) nth3(Y,[X,Y,Z],2).
F) nth3(Z,[X,Y,Z],3).
```

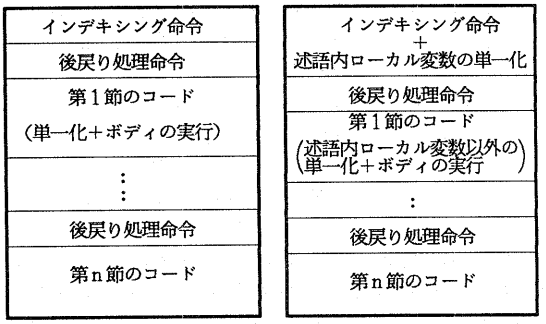
実行順序	述語内ローカル変数を単一化する節
A B C D E F	A

インデキシングしない場合

第1引数の値	実行順序	述語内ローカル変数を単一化する節
1	A D E F	A
2	B D E F	B
3	C D E F	C
上記以外	D E F	D

インデキシングする場合

表2　インデキシングと述語内ローカル変数の単一化処理の関係



a) 従来のコンパイルドコード　　b) 述語内ローカル変数を利用するコンパイルドコード

図6　コンパイルドコードの構成

3.6.2 インデキシング処理との関係

3.2 節で述べたように、述語内ローカル変数の値を求める処理は最初の単一化処理によって行われ、それ以降で値が必要になった場合は p_local フレーム内の値が参照される。このため、述語内ローカル変数の最初の単一化処理が、どの節で行われるかということが重要になる。しかしながら、インデキシングを行う場合には、表2に示すように第1引数に与えられる値によって述語内ローカル変数の単一化処理を行う節が変化するという問題がある。この問題は図6に示すように、同一のキーでインデキシングされる節の最初の節に、述語内ローカル変数の最初の単一化処理を行う命令を配置することで解決している。このように命令を配置することにより、インデキシング処理が終了し試行すべき節が確定した時点で、試行する節に出現する述語内ローカル変数の単一化処理を行うことが可能になっている。

4. プログラム変換を利用した最適化方式

非決定的に実行される述語は、複数の節が選択されるものと、ただ1つの節しか選択されないが、 PROLOG の言語仕様上の制約のために後戻り処理によって複数の節を試行することにより節選択が行われるものがある。このうち後者については、いくつかの条件を満たしている場合に限り、インデキシングによって決定的に実行される述語に変換することによって実行速度を大幅に改善することができる。本章では、プログラム変換の手法を用いてこのような最適化を行う方式について述べる。

4.1 プログラム変換

一般にプログラム変換とは、与えられたプログラムをより高速に実行されるプログラムに変換する手法である。これは、PROLOG におけるプログラム変換についても同様である。PROLOG は、宣言的解釈と手続き的解釈の双方が可能な言語である。この二面性を活かし、わかりやすい宣言的なプログラムから、わかりにくいかもしれないが手続き的に見た場合により効率の良いプログラムを得ようとするのが、PROLOG におけるプログラム変換である。

通常の変換はソースレベルでの効率化を目指したものであり、計算量を少なくすることにより実行形態によらないで高速に実行されるプログラムを得る手法である。これに対し、本報告で述べるプログラム変換は、変換されたプログラムがコンパイルされた場合にのみ高速に実行されることを目的としたものであり、非決定性処理が行われる述語をインデキシング手法によって節選択が決定的に行われる述語に変換するものである。したがって、変換によって計算量のオーダが変わるような通常のプログラム変換とは目的が異なる。オーダが変化しなくてもインデキシング処理を行うことにより選択点の操作にかかるコストを大幅に削減することができるため、高速な処理が可能になる。インデキシングはコンパイラにおいて利用される手法であり、変換の結果得られたプログラムをインタプリタで実行した場合の効率は保証されない。

4.2 節選択のメカニズムと変換可能性

複数の節から実行対象となる節を選択する処理は、基本的にヘッド部の単一化処理によって行われる。節の選択が単純な引数のマッチング処理で行える場合には、インデキシング処理を利用することで決定性処理として高速に実行することができる。しかしながら、節が選択される条件をヘッド部だけで記述できない場合には、節が選択されるための制約条件をボディー部のサブゴールによってテストするように記述しなければならない。このような述語は、図7, 1) に示す構成をとる。a1, ..., an と c1, ..., ci は、制約条件をテストするためのサブゴールでガードゴールと呼ぶ。また、b1, ..., bm と d1, ..., dj は、制約条件が満たされた場合に実行されるサブゴールである。

制約条件が満たされた場合に、別解を生成しない決定性処理を行いたい場合には、ガードゴールの最後にカットが置かれる。とくに、ガードゴールが背反する制約条件の場合には、節の選択が逐次的に実行されることを利用して、図7, 2) のように後者のガードゴールを省略することが多い。このようにして記述された述語は、概念的には決定的に実行できるはずであるが、PROLOG

の記述能力の制約から後戻り処理をともなって実行される。

```
goal :- a1, ..., an,        goal :- a1, ..., an, !,
        b1, ..., bm.                b1, ..., bm.
goal :- c1, ..., ci,        goal :- d1, ..., dj.
        d1, ..., dj.
        1)                          2)
```

図7　ガードゴールとカット

図7、1）に示されるような述語で、ガードゴールが背反する制約条件をテストしている述語や、図7、2）に示されるような述語は、手続き型言語の if ... then ... else ... 構文のように、ガードゴールの実行結果によって実行するゴールを選択する形式に変換できれば、決定性処理として高速に実行することができる。このような変換は、ガードゴールが実行の履歴に影響されず、決定性処理を行う述語である場合に可能である。非決定性処理を行う述語である場合には、ガードゴールによって生成と検査を行うプログラムになるので、単純な条件判断によってゴールを選択する形式に変換することはできない。

対象となる述語がプログラム変換によって性能を向上させることができるか否かを判定するためには、ヘッドの引数のパターン、モード宣言、カット、ガードゴールなどの情報を利用する。変換対象となる述語は、選択点が生成されるプログラムで、ガードゴールが背反する制約条件をテストしているものか、ガードゴールの後にカットがあるものである。このような判定基準を使用することで、プログラム変換によって最適化が行える述語を比較的簡単に判定することができる。

ガードゴールとして扱う述語は、実行の履歴に影響されず決定性処理が行われることが判明していなければならないので、組込み述語に限定している。PROLOG プログラムの解析結果[7]として、コンパイルすることを前提としているプログラムでは、ほとんどの述語でモード宣言が行われていることと、組込み述語とカットの1つの節当りの出現頻度が、それぞれ 0.5 と 0.65 と高いことが報告されている。したがって、これらの情報をプログラム変換に利用する方式には一般性があると考えられる。

4.3 変換戦術

プログラム変換を効果的に行うためには、変換の方向付けが重要である。我々の最適化手法では、2章で述べたオーバーヘッドを除去することを目標として、以下に示す戦術に基づいてプログラム変換を行う。

(1) 変換対象となる述語の定義をもとにして補助述語を導入し、入力引数が同じパターンを持つ節を1つの節にまとめる。
(2) ガードゴールとカットを補助述語に置き換え、インデキシングができる形式にする。
(3) 1つの節からなる述語があれば、その述語をサブゴールに持つ述語を展開する。

戦術 (1) は、単一化処理の遅延実行と単一化処理情報の有効利用を目的としている。導入された補助述語のそれぞれの定義節は、まとめられた節の定義節と同じサブゴール列を持つ。変換前の節のヘッド部の入力引数が複合項である場合は、補助述語の定義節のヘッド部の対応する引数は、複合項中の変数とアトムを展開したものとなる。また、出力引数は変換前の節のものと同じである。補助述語の呼び出し側では、すべての引数は変数とする。

戦術 (2) は、選択点の操作の除去を目的としている。(1) で導入した補助述語は、ガードゴールの実行と後戻り処理によって節が選択される形になっている。このガードゴールとカットを、インデキシング用の値を出力する補助述語に置き換える。さらに、ガードゴール以降のボディー部を第1引数にインデキシング用のアトムを持つ別の述語で実行するようにする。

戦術 (3) は、冗長な述語呼び出しを除去するためのものである。

4.4 補助述語の定義

プログラム変換に利用するために定義している補助述語の一部を示す。

```
:- mode bool(+,-).
```
　第1引数の条件の真偽を第2引数に返す
```
:- mode type_of(+,-).
```
　第1引数の項のタイプを第2引数に返す

これらの補助述語は、数値比較や値のタイプチェックを行う組込み述語がガードゴールとして使用されている述語の変換に利用するものである。これらの補助述語を利用することで、後戻り処理によって節を選択するプログラムを、インデッキシングによって節を選択する効率のよいプログラムに変換することができる。補助述語の処理は、後戻り処理をともなわない手続き的な操作によって実現することができるので、結果的に後戻り処理のオーバーヘッドを除去することができる。

4.5 変換例

図7に示したようにガードゴールが明示的に記述されている述語と、そうでない述語に対して、プログラム変換を適用した例を示す。まず、明示的なガードゴールがある述語として partition /4 の変換を考える。

```
:- mode partition(+,+,-,-).
A) partition([X|L],Y,[X|L1],L2) :-
        X =< Y, !,
        partition(L,Y,L1,L2).
B) partition([X|L],Y,L1,[X|L2]) :-
        partition(L,Y,L1,L2).
C) partition([],_,[],[]).
```

まず、節 A), B) は入力引数のパターンが等しいので、変換戦術 (1) により、補助述語 partition1/5 を導入する。

```
:- mode partition1(+,+,+,-,-).
D) partition1(X,L,Y,[X|L1],L2) :-
        X =< Y, !,
        partition(L,Y,L1,L2).
E) partition1(X,L,Y,L1,[X|L2]) :-
        partition(L,Y,L1,L2).
```

partition1/5 の入力引数は、節 A), B) の入力引数中の変数を

並べたものとなり、出力引数は、A), B) と同じである。この partition1/5 を用いて、partition/4 は以下の２つの節によって再定義される。

```
F) partition([X|L],Y,L1,L2) :-
        partition1(X,L,Y,L1,L2).
G) partition([],_,[],[]).
```

節 F) では、partition1/5 のすべての引数を変数として呼び出している。

この変換により、選択点の操作が除去される。また、レジスタ割り当ての最適化を行うことにより、単一化処理の回数を削減できる。さらに、出力引数の単一化処理は、ガードゴールの実行による節選択の確定後まで遅延されている。

次に、変換戦術 (2) により、節 D) のガードゴール "X =< Y" とカットを補助述語 bool/2 で置き換える。

```
H) partition1(X,L,Y,L1,L2) :-
        bool(X =< Y, B),
        partition2(B,L,Y,L1,L2).
:- mode partition2(+,+,+,-,-).
I) partition2(true,L,Y,[X|L1],L2) :-
        partition(L,Y,L1,L2).
J) partition2(false,L,Y,L1,[X|L2]) :-
        partition(L,Y,L1,L2).
```

この変換によって partition1/5 の定義節は、節 H) １つだけとなるため、変換戦術 (3) により、呼び出し側である節 F) を partition1/5 で展開する。

```
K) partition([X|L],Y,L1,L2) :-
        bool(X =< Y, B),
        partition2(B,L,Y,L1,L2).
L) partition([],_,[],[]).
```

結局、上例では (1)～(3) の３つの操作により、節 A), B), C) が K), L), I), J) に変換された。

次に、partition/4 と異なり、明示的なガードゴールを持たない述語として、nth/3 の変換例を以下に示す。

```
:- mode nth(+,+,-).
nth(0,[X|L],X) :- !.
nth(N,[_|L],X) :- N1 is N - 1,
                  nth(N1,L,X).
```

nth/3 では最初の節の第１引数が０でなくてはならないので、この制約をガードと考えて partition/4 と同様な変換を行うことができる。

```
:- mode nth(+,+,-).
nth(N,L,X) :-  bool(N == 0,B),
               nth1(B,N,L,X).
:- mode nth1(+,+,+,-).
nth1(true,_,[X|L],X).
nth1(false,N,[_|L],X) :-  N1 is N - 1,
```

4.6 プログラム変換による最適化の効果

4.5 節の partition/4 の変換の場合を例にとって、プログラム変換による最適化の効果を示す。

1) 節 A) における最適化の効果
   変換されたプログラムでは、インデキシング処理によって節の選択が行われる。したがって、後戻り処理のための選択点の生成処理が除去できる。
2) 節 B) における最適化の効果
   節 A) の実行の失敗による後戻り処理、後戻り処理にともなう選択点の削除処理、節 B) の第１引数の単一化処理、節 A) の第３引数の単一化処理などが除去できる。
3) 節 C) における最適化の効果
   節 C) は最適化されない。

変換によって生成されたプログラムでは、ヘッド部の第１引数を用いたインデキシング処理によって決定的に節の選択が行われるため、後戻り処理を伴わずに高速に実行することができる。最適化による効率の向上は A), B) の節でみられるが、とくに、B) の節が実行される場合の効率の向上が著しい。C) の節は最適化による変化はないが、終端条件で１度しか実行されないので、プログラムの実行性能にはほとんど影響しない。

5. 評価と考察

SUN-3/260 上で動作する C-Prolog コンパイラ[3]を用いてベンチマークテストを行い、提案した２つの方式の評価を行った。

5.1 新変数分類による最適化

提案した変数分類法と処理系アーキテクチャの評価を行うために、partition/4 を使用し、1000 要素からなるリストの分割を 100 回繰り返したときの処理時間 (cputime)を、以下の３種類の方法で測定した。

A) インタプリタ
B) 最適化方式を行わないコンパイルドコード
C) 述語内ローカル変数による最適化を利用したコンパイルドコード

また、それぞれの実行法ごとに、以下の３種類の場合について測定した。その結果を表４に示す。

(1) すべての要素に対し後戻り処理が行われる場合
(2) 半分の要素に対して後戻り処理が行われる場合
(3) 後戻り処理が行われない場合

述語内ローカル変数を利用する最適化では、後戻り処理が行われる回数が多いほど効果が大きくなる。このため、すべて後戻り処理を行う (1) の場合が最も効果が大きく、約 17 %高速化されている。後戻り処理が全く行われない (3) の場合は、約 5 %遅くなっているが、平均的な (2) では約 8 %高速化されており、一般的な状況での本方式の有効性が示されている。

5.2 構造共有法に固有のオーバーヘッド

最適化を行った場合に速度が向上する要因には、後戻り処理時の冗長な単一化処理のコストや、選択点フレームの生成コストの減少がある。従来の方式では、選択点フレームにすべての引数の

	A) インタプリタ	B) 従来のコンパイラ	C) 述語内ローカル変数	D) ｵｰﾊﾞｰﾍｯﾄﾞ除去	E) プログラム変換
(1)	56.25	5.67	4.72	4.25	2.00
(2)	45.25	4.62	4.25	3.88	1.95
(3)	36.50	3.58	3.77	3.45	1.90

(1) partition([1,2,3,4,5,6,7,8,9,10], 0, X, Y).
(2) partition([1,2,3,4,5,6,7,8,9,10], 5, X, Y).
(3) partition([1,2,3,4,5,6,7,8,9,10],10, X, Y).

(単位: sec)

表4 partition/4 の処理時間

初期値を保存し、候補節の再試行時にはこの情報を用いて引数値を初期化する。しかしながら、引数中の変数がすべて述語内ローカル変数である場合には、再単一化のための引数の初期化は不要になるため、選択点フレームに初期値を格納しておく必要がない。このため、選択点フレームの大きさと生成のためのコストが減少する。

逆に、処理速度が低下する要因としては、述語内ローカル変数を使用するために必要な p_local フレームの生成にかかるコストがあげられる。本来ならば、p_local フレーム生成に伴うコストの増加は、選択点フレームの生成コストの減少と相殺される程度のものであるため、後戻り処理が行われない場合でも処理速度が低下することはない。(3) における速度低下は、C-Prolog が構造共有法によって複合項を表現しているために、3.5 節で述べたグローバルスタック操作の増加が原因となっているものと考えられる。

この操作のオーバーヘッドを算出するために、partition/4 を用いてベンチマークテストを行った。オーバーヘッドの算出はゴールモレキュールの領域確保に関係する処理を2回ずつ行った結果と、5.2 の結果との差をとることによって行った。表4の D) に、このオーバーヘッドを取り除いた処理時間を示す。この結果から、述語内ローカル変数を利用した場合の処理時間の約 10 %が、構造共有法に固有のオーバーヘッドによるものであることがわかる。このオーバーヘッドを除くことにより、従来のコンパイルドコードに比べて、(1) の場合で約 25 %、(2) では約 16 %、後戻り処理が全く行われない (3) の場合でも、約 4 %高速化されている。

5.3 プログラム変換による最適化

プログラム変換を利用した最適化方式を適用した場合の partition/4 の処理時間を表4の E) に示す。

プログラム変換による最適化を行った場合は、partition/4 の処理速度は通常のコンパイルドコードに比べて約 2.8 倍、インタプリタに比べて約 28.1 倍高速化されている ((1) の場合)。これは、プログラム変換により、選択点の生成および削除処理、後戻り処理、冗長な単一化処理などが除去された結果である。インタプリタで実行する場合は後戻り処理を行うほど遅くなるため (1) が最も遅いが、プログラム変換による最適化を施したコンパイルドコードでは、すべての後戻り処理が削除されるため、どの節が選択されても同程度の速度で実行されている。

6. おわりに

本報告では、非決定性処理を最適化する方式として、述語内ローカル変数を利用して後戻り処理を効率化する方式と、プログラム変換の手法を利用して後戻り処理を除去する方式を提案した。さらに、我々が開発した C-Prolog コンパイラ上でのベンチマークテストにより、partition/4 の実行速度は最適化を行わないコンパイルドコードと比較して、述語内ローカル変数の導入により約 17 %、プログラム変換により約 2.8 倍に高速化されることが示された。しかしながら、前者の方式は構造複写法を採用している処理系での利用を前提としており、構造共有法を採用している C-Prolog 処理系では約 10 %のオーバーヘッドを生じている。今後は、構造複写法を採用している処理系上での評価を行う必要がある。

参考文献

[1] Warren, D.H.D.: Implementing Prolog - compiling predicate logic programs, Dept. of Artificial Intelligence, University of Edinburgh, Research Reports 39 & 40 (1977)

[2] Warren, D.H.D.: An Abstract Prolog Instruction Set, SRI International, Technical Note 309 (1983)

[3] 砒崎 他: C-Prolog コンパイラの開発, ICOT, Proceedings of the Logic Programming Conference '86, pp.159-166 (1986)

[4] 砒崎 他: PROLOG 処理系におけるコンパイル方式の改良, 日本ソフトウェア科学会, 第4回大会論文集, pp.23-26 (1987)

[5] 松本 他: PROLOG の非決定性処理を最適化するコンパイル方式の評価, 情報処理学会, 第36回全国大会講演論文集, 6H-7 (1988)

[6] 奥乃 : 第三回 LISP コンテストおよび第一回 PROLOG コンテストの課題案, 情報処理学会, 記号処理研究会資料, 28-4 (1984)

[7] 尾内 他: 逐次型 Prolog プログラムの解析, ICOT, Proceedings of The Logic Programing Conference '86, pp.159-166 (1986)