記 号 処 理 48－4
（１９８８．１１．２９）

最適化PROLOGインタプリタの構成法

碇崎賢一　打浪清一
九州工業大学　情報工学部

本報告では，コンパイラの最適化手法を取り入れ，処理速度とメモリ効率を向上させたPROLOGインタプリタの構築法について述べる．このインタプリタでは，インデッキシング，スタックフレームの分割管理，テールリカージョンの最適化などの手法を取り入れて性能の向上をはかっている．とくに，インデッキシングを実現するために，述語を動的，静的の２種類に分類し，この分類に基づいて最適化を行なっている．また，組み込み述語の特性を利用して，ヒープスタックへの構造の複写を抑制する方式や，複数のサブゴールがある場合でも環境フレームの生成を抑制する方式を新たに導入し，処理速度とメモリ効率の向上をはかっている．これらの手法を評価するために試作した処理系では，従来のインタプリタの４倍程度の処理速度が得られると共に，メモリ効率が大幅に向上している．

Construction Methods of an optimized PROLOG interpreter

Ken'ichi KAKIZAIKI and Seiichi UCHINAMI

Faculty of Computer Science & Systems Engineering
Kyusyu Institute of Technology
680-4, Kawazu, Iizuka, Fukuoka 820, Japen

Abstract

This paper describes construction methods of an optimized PROLOG interpreter. In this interpreter, improvements in terms of the time and space efficiency are accomplished with various optimization methods for the compiler, such as clause indexing, separated stack frame management, and tail recursion optimization. Specially, we present a clause indexing method, which makes use of the declaration about predicate categories such as static and dynamic. We also propose an optimization method, which avoids structure copies from the database into the heap stack. The methods described above increase both the speed and memory efficiency.

The performance of the optimized PROLOG interpreter is four times faster than conventional interpreters, and the memory efficiency is extraordinary improved.

1　はじめに

知的情報処理システムの記述言語として PROLOG が広く用いられている．PROLOG は，プログラムの宣言的記述が可能で，単一化と後戻りを基本とした柔軟で強力な処理能力を備えているという特徴を持つが，C言語などの手続き的な言語と比較して，処理速度やメモリ効率が低いという問題点をかかえている．このため，一般的に，PROLOG はプロトタイピングには適しているが，実用システムには適していないとされている．しかしながら，最近になって，さまざまなコンパイル手法が研究開発され，コンパイルされたプログラムの処理速度やメモリ効率が改善されてきているために，実用システムへの応用も増えつつある．

一方、インタプリタに関する研究はそれほど行われておらず，コンパイラのような著しい性能の向上は得られていない．これは，

1）プログラムが最終的にコンパイルされるために，インタプリタの性能はそれほど重要でない．

2）インタプリタは，PROLOG 自身で記述でき，コンパイルすることによって性能を上げられる．

3）インタプリタをコンパイラと独立して開発するのにはコストがかかる．

などの理由により，インタプリタが重要視されていないためである．しかしながら，インタプリタは，プログラミング環境として見た場合にも，PROLOG プログラムの特性を考慮した場合にも重要である．

PROLOG はプロトタイピングに適しているために，さまざまな試作やデバッグなどの開発過程に費やされる時間が非常に長い．開発過程では，試行錯誤が多いために多くの作業がインタプリタ上で行われ，開発効率がインタプリタの性能に大幅に左右されてしまう．たとえば，処理速度が低ければ，簡単なテストやデバッグにさえ多大な時間が浪費されてしまう．また，メモリ効率が悪い場合にはメモリが不足しテスト実行さえできない可能性がある．このため，プロトタイピングに適しているとされながら，十分なテストや新たなアイディアの検証を行えない可能性がある．

さらに，知的情報処理システムでは，PROLOG の特徴であるデータとプログラムが同一形式で記憶されていることを利用して，知識をデータとしてデータベースに蓄えておき，その知識の適応が必要になった時点でプログラムとして利用する手法が多く用いられている．この場合コンパイルされたシステムでもインタプリタが不可欠になる上に，その性能がシステム全体の性能を左右することになる．したがって，PROLOG の特性を最大限に活用しようとする場合には，インタプリタの性能を向上させることが重要である．

本報告では，多くの PROLOG コンパイラに利用されている WAM[1] と基本的に同じ実現方式を採用した上で，さらにいくつかの新たな最適化手法を導入し，処理速度とメモリ効率を向上させた最適化 PROLOG インタプリタの構築法とその評価を述べる．

2　インタプリタの実現方式

2．1　コンパイラベースのインタプリタの問題点

コンパイラを実装する PROLOG 処理系の実現方式は，以下の2種類に分類することができる．

1）コンパイラに依存しないインタプリタを個別に構築する．

2）コンパイラを構築しその上で稼働するインタプリタを PROLOG で記述する．

コンパイラを装備する PROLOG 処理系のほとんどは，2の方式で実現されている．これは，コンパイラができあがれば，比較的容易にインタプリタを作成することができるためである．たとえば，基本的な PROLOG インタプリタは，図1に示すように4個の節で記述することができる．しかしながら，このようにして作成されたインタプリタには，以下に述べるような本質的な問題点がある．ここでは，図2に示す partition/4 を使用してコンパイラベースのインタプリタの問題点を示す．

```
execute(true) :-
        !.
execute((Goal, Goals)) :-
        !,
        execute(Goal),
        execute(Goals).
execute(Goal) :-
        clause(Goal, Body),
        execute(Body).
execute(Goal) :-
        builtin(Goal).
```

図1　PROLOG による PROLOG インタプリタ

PROLOG インタプリタは，データベースに格納されているプログラムに基づいて処理を行うために，データベースからプログラムの定義を取り出さなければならない．図1の execute/1 では，この操作を第3節の clause/2 で行っている．clause/2 が成功すると，選択された節の本体の定義がデータベースからヒープスタック[1]に複写され，次に実行すべき新たなゴールとして変数 Body に代入される．ここで，1個のポインタを格納する領域をオブジェクトセルと呼ぶと，図2の partition/4 の第1節が選択された場合には，図3に示すオブジェクトセル14個分の構造が毎回ヒープスタックに複写される．この複写は，partition/4 の実行には本質的に不必要なものであるが，本来プログラムとして取り扱われなければならない partition/4 の定義が，execute/1 により単にデータとして処理されているめに行われてしまう．

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

図2　インタプリタによって実行される述語の定義

一方，partition/4 の本来の処理では，第1節が選択された場合には頭部の第3引数のリスト [X| L1] の構造が新たにヒープスタックに格納され，第2節が選択された場合には頭部の第4引数のリスト [X| L2] の構造が新たにヒープスタックに格納される．リストは2個のオブジェクトセルを占有するので，メモリ管理の観点では，partition/4 は1回の呼び出しで2個のオブジェクトセルを消費する述語であると考えることができる．実際に，partition/4 のコンパイルドコードでは，この割合でヒープスタックを消費する．

partition/4 の実行に必要な構造は，頭部の単一化で作成されたリストが占める2個のオブジェクトセルのみなので，PROLOG で記述されたインタプリタの構造複写によってメモリ効率は8倍ほど悪化している．ガーベッジコレクタ

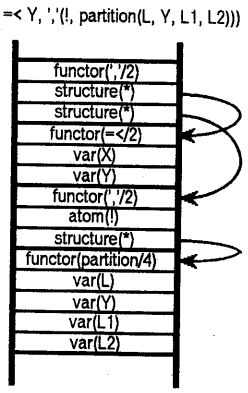


図３　ヒープスタック上に複写されたプログラムの構造

を実装している処理系では，ヒープスタックに複写されたプログラムが占める領域を回収することができるが，実質的な処理速度が大幅に低下してしまうという問題が生じる．また，ガーベッジコレクタを実装していない処理系では，大規模なプログラムを動かすことができなくなるという問題が生じる．さらに，構造の複写は，各構造の引数を再帰的にたどり，ヒープスタックのオーバーフローをチェックしながら行わなければならないために，処理速度が低下してしまうという問題も生じる．

一方，コンパイラに依存せず，Ｃ言語やアセンブラなどで記述されたインタプリタでは，データベースに格納されているプログラムの定義をヒープスタックに複写せずに直接参照して処理を進めることができるために，プログラムの定義をヒープスタックに複写する必要はない．このような理由から，高性能な PROLOG 処理系が必要な場合には，インタプリタを PROLOG 自身で記述せず，Ｃ言語などで記述する必要があると考えられる．

２．２　項形式と中間コード形式

コンパイラに依存しないインタプリタの実現方式には，プログラムを項形式で格納する方式と，中間コードに翻訳して格納する方式がある．前者は，プログラムの構造がデータベース内部に素直に反映されており，処理系を作成しやすいという特徴がある．このため，インタプリタの基本的な構成方式として多用されている．一方，後者は，最近提案[2],[3]された方式で，インタプリタの性能を向上させるために WAM の命令セットを中間コードとして用いたものである．この方式は，内部的には中間コードコンパイラを備えたインクリメンタルコンパイラと考えられる．このため，節の追加，削除などでデータベースの内容を変更するためには，中間コードをいったん項形式へ変換しなければならないため，インタプリタの構造が複雑になり処理時間がかかるという問題がある．これらの理由で，本報告では，項形式のインタプリタの構築法に関して述べている．

３　最適化方式

３．１　インデッキシング

インデッキシング[4]は，ゴールの引数が定数である場合に，そのゴールと単一化できる節の頭部が限定されることを利用して候補節を絞り込み，失敗する節の選択をあらかじめ除去する最適化手法である．インデッキシングの重要な機能として，次の２つがあげられる．

Ａ）選択可能な節を高速に検索できる．
Ｂ）選択点の生成を抑制することができる．

これらの機能によって，処理速度を向上させることができ，とくに，Ｂ）の機能によって，制御用スタックが消費されないためにメモリ効率を向上させることができるという利点がある．また，選択点が生成されないために，トレイルスタックの使用量も抑制できるという利点もある．しかしながら，以下に示す２つの理由でインタプリタでは実現することが困難であった．

１）インタプリタでは，実行時にプログラムを変更される可能性がある．したがって，プログラムが変更されるたびに，インデッキシング用のハッシュ表などのインデッキシング用の情報を作成すると，処理速度が大幅に低下する．
２）DEC-10 PROLOG[5]のインタプリタのセマンティックスに準拠すると，代替節の決定を後戻り処理の時点で行わなければならないために，節を選択した時点では代替節の有無を検出することができず，選択点の生成を抑制することができない．

本報告では，これらの問題を解決するために，処理時間が少なくてすむインデッキシング情報の作成法と，節を選択した時点で代替節を検出する方式を提案する．

3.1.1　インデッキシング情報の作成

インデッキシング用の情報は，節をデータベースに登録する時点で作成しておく方式と，実行時にデータベース内の残りの節を検索して得る方式が考えられる．しかしながら，後者の方式は，実行の際に毎回検索が必要で非効率であるという問題がある．したがって，節の登録時に述語の第１引数のデータタイプを利用してインデックスキーを作成し，そのインデックスキーをもとに代替節の有無を調べ記録する前者の方式を示す．

インデックスキーには，インタプリタが扱うデータタイプを用いており，たとえば，
１）アトム
２）整数
３）実数
４）リスト
５）構造
６）変数
などがある．partition/4 を使って，インデックスキーと

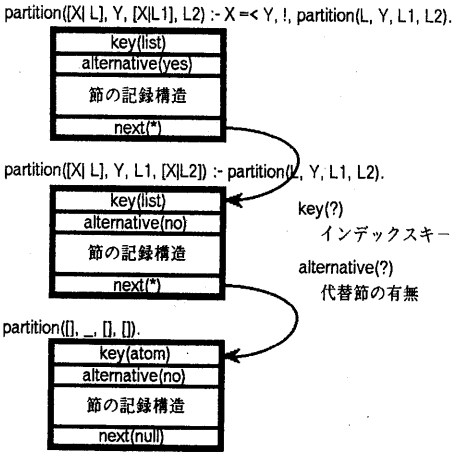


図４　インデックスキーと代替節の有無の記録

代替節の有無が記録されたデータベース内の節の記録構造を図4に示す．この例では，第1，第2節ではリストが，第3節ではアトムがインデックスキーとして記録されており，ゴールの第1引数がリスト場合には第1節と第2節が，nil の場合には第3節が候補として選択できることが示されている．したがって，ゴールの第1引数が nil の場合には，インデックスキーが合致しない第1，第2節を無視して，第3節を直接選択することができる．また，第2節が選択され失敗した場合には，代替節がないことが記録されているため，第3節を代替節として選択せずに即座に失敗とすることができる．このために，従来の処理系と比較して，処理速度を向上させることができるという利点がある．

以上のような最適化を実現するために必要な，データベースへの節の登録，削除の操作を簡単に示す．

1）asserta によって，節の先頭に節を登録する場合には，すでに登録されている節のどれかのインデックスキーが新たに登録する節のインデックスキーに合致するか，変数を示している場合には，登録する節に代替節が存在することを記録する．また，新たに登録する節のキーが変数を示している場合には，すでに1つでも節が登録されていれば，そのキーに関わらず代替節があることを記録する．

2）assertz によって，節の最後に節を登録する場合には，すでに登録されている節で，新たに登録する節のインデックスキーに合致するキーを持っているものは，代替節が存在することを記録する．また，新たに登録する節のインデックスキーが変数を示している場合には，すでに登録されている節のすべてに，代替節が存在することを記録する．

3）retract によって，節を削除する場合には，データベース内でその節より前のすべての節について，代替節があるかどうかを調べ代替節の有無を記録しなおす．

これらの操作の処理時間は，すでに記録されている節の個数にほぼ比例するが，各節に対する操作が単純なインデックスキーの比較と代替節の有無の記録のみなので，データベースに節を格納する処理時間より十分短い処理時間で行われ，処理速度に影響しないと考えられる．

3.1.2 静的述語宣言を利用した選択点の生成の抑制

PROLOG のシンタックスおよびセマンティックスは，蓄積されたソフトウェアを有効に利用し不用な混乱を避けるために，標準とされている DEC-10 PROLOG に準拠するのが適切であると考えられる．DEC-10 PROLOG のインタプリタでは，代替節の有無を後戻り処理を行う時点で判定している．したがって，節が選択された時点では代替節がなくても，プログラムのどこかで assertz などのデータベースを操作する述語によって新たな節が追加されている場合には，追加された節が代替節として利用されることになる．この機能を実現するためには，述語の最後の節を実行する場合にも選択点を残しておかなければならず，メモリ効率や処理速度が低下するという問題がある．

この問題を解決する方法の1つとして，節を選択した時点で代替節の有無を判定しておく方式[3]が考えられる．以後の議論では，この方式を select 方式，DEC-10 PROLOG の方式を redo 方式と呼ぶことにする．redo 方式では，節を選択した時点で代替節の有無を検出することができるために，インタプリタにおいてもコンパイルドコードと同様に選択点の生成を抑制することが可能になる．しかしながら，この方式では，DEC-10 PROLOG のインタプリタとセマンティックスが異なるという問題がある．

select，redo のセマンティックスの差異は，図5のような動作の相違となって表れる．select 方式では，節が選択された時点で代替節が存在していないために，第1節が失敗すると再試行が行われずにゴールそのものが失敗する．一方，redo 方式では，後戻り処理が行われる時点では代替節が存在しているので，再試行が行われゴールが成功する．また，図5と異なり，図6のように実行前から代替節が存在していると，両方式とも実行前から存在している節のみでなく，実行時に追加されたされた節まで代替節としてゴールが成功する．このように，select 方式では，DEC-10 PROLOG のインタプリタとセマンティックスと異なると共に，実行時に追加された節の取扱が一定でないという問題がある．従来の DEC-10 PROLOG 準拠とされている処理系には，仕様の解釈の相違や最適化のために select 方式を採用しているものが多く，動作が異なるという問題が生じている．

```
goal(1) :-
        assertz(goal(2)),
        fail.

| ?- goal(X).
no
```

1）　select 方式

```
| ?- goal(X).
X = 2
yes
```

2）　redo 方式

図5　代替節の決定法

```
goal(1) :-
        assertz(goal(3)),
        fail.
goal(2).

| ?- goal(X).
X = 2 ;
X = 3
yes
```

図6　実行前から代替節がある場合の代替節の決定法

本報告では，DEC-10 PROLOG と同じ redo 方式を採用した上で，メモリ効率や処理速度が低下するという問題点を解決するために，述語をその動特性によって静的，動的の2種類に分類し，述語を定義する際にその動特性を宣言し最適化に利用する方式を提案する．静的述語は，プログラムの実行時にその定義が変更されない述語で，動的述語は，プログラムの実行時にその定義が変更される可能性がある述語である．静的宣言が行われている述語は，節の追加や削除が行われないことが保証されているので，節を選択した時点で代替節の有無を決定することができる．このため，コンパイルドコードと同様に選択点の生成を抑制し，処理速度とメモリ効率を向上させることが可能になる．

たとえば，partition/4 の第2節が選択された場合には，従来の方式では，節の追加に備えて選択点を除去することができないという問題があった．しかしながら，静的述語宣言が行われている場合には，節の登録時に代替節がないことが記録されているので，この情報を利用して選択点を速やかに除去することができるという利点がある．また，第3節が選択された場合には，代替節がないことが記録されているので，選択点の操作をまったく行わなくてよいと

いう利点がある．

述語の動特性の宣言には，選択点に関する最適化の他に２つの利点がある．１つめは，静的と宣言された述語が呼び出されたときに，選択された節が削除される可能性がなくなることである．このため，選択された節が実行中に削除されないようにするための保護処理が不必要になり，節の選択処理が高速化されるという利点がある．２つめは，節が定義されていない静的述語が呼び出された場合には，失敗ではなくエラーとして取り扱うことができるということである．また，静的述語に対して assert，retract が行われた場合にも，エラーとして取り扱うことができる．この結果，述語名の綴り間違いなどのバグを容易に検出できるという利点が得られる．

３．２　変数分類

変数以外の項は，ヒープスタック上とデータベース内で同じ表現形式で表しているが，変数は単一化処理を効率よく行うために表現形式を変更している．変数は，ヒープスタック上では自分自身をさすポインタとして表現されているが，データベース内ではその変数の各種の特性を表す情報の組合せで表現されている．変数を分類するための解析は，プログラムをデータベースに登録する際に行っており，WAM の方式と同様に以下に示す情報を得ている．

１）変数の識別情報（実行時の変数のアドレスに一致）
２）変数の種類（テンポラリ，ローカル，ボイド）
３）変数の出現順位（最初と最初以外）
４）変数のグローバライズの必要性の有無

これらの情報のうち３），４）は，同一の変数でも出現位置によって異なっている．最適化インタプリタは，これらの情報を利用してコンパイルドコードと同様な効率のよい処理を行う．

３．３　環境の生成の抑制

複数のサブゴールを持つ節を処理する場合には，以下に示す２種類の情報を保存しなければならないために，制御スタック上に環境と呼ばれるデータ構造を生成する．

１）親ゴールに復帰するための情報の保存
２）ローカル変数の保存

しかしながら，複数のサブゴールがある場合でも，最後以外のサブゴールのすべてが組み込み述語の場合には，以下に示す理由によって，それぞれの情報を保存する必要がなくなる．

１）組み込み述語の呼び出しは，仮想 PROLOG マシンの述語呼び出しの手順を使わずに行われるために，復帰用の情報を保存する必要がない．
２）組み込み述語の内部処理が，テンポラリ変数レジスタの内容を破壊しないように記述されていれば，ローカル変数を使用する必要がない．

ただし，サブゴールのすべてが組み込み述語でも，clause/2 などの再試行が行われる述語がある場合には，後戻り処理で変数の代入を解かなければならないために，環境を作成しその述語の前後にわたって利用される変数をローカル変数として保存しなければならない．

最適化 PROLOG インタプリタでは，これらの条件が満たされる場合には，環境の生成を抑制し処理速度を向上させている．たとえば，partition/4 の第１節が選択された場合には，最後以外のサブゴールが組み込み述語なので環境は生成していない．このような最適化処理を実現するために，それぞれの組み込み述語に関連する情報として，環境を生成する必要の有無をインタプリタの内部に登録している．データベースに節を登録する際には，実行時に環境を生成する必要があるか否かをこの情報を利用して判定し，その結果を図４に示した節の記録構造の一部として記録している．実行時には，サブゴールの個数に関わらず，この情報にしたがって環境を生成するか否かを決定している．なお，この手法は，コンパイラでも利用することができる．

また，環境を生成した場合には，WAM と同様に最後のサブゴールを呼び出す前に環境を削除しているので，テールリカージョンの最適化[6]も実現されている．

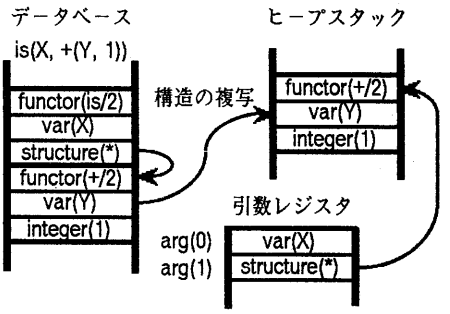
３．４　構造複写の抑制

構造複写法の欠点として，複合項を利用するためには，その構造をデータベースからいったんヒープスタック上に複写しなければならないことがあげられる．たとえば，図７に示す述語のサブゴール is/2 を実行する場合には，図８，１）に示すように構造 +(Y, 1) をヒープスタック上に複写し，その構造を指すポインタを第２引数として is/2 を呼び出す．この方式では，複写された構造は is/2 の処理でのみ使用され，以後２度と使用されないために，ヒープスタック上の塵としてメモリ効率を悪化させるという問題がある．この問題を解決するために，本報告では，述語の引数の項をその有効範囲によって分類し，その分類情報を用いて構造の複写を抑制する方式を提案する．

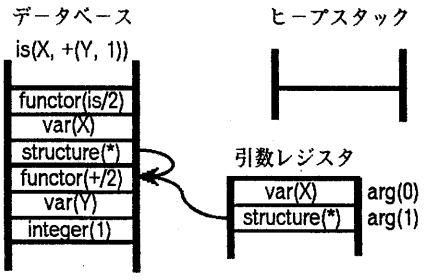
```
a(X, Y) :- X is Y + 1.
           ----------

                +(Y, 1)
```

図７　構造複写の対象の引数



１）構造を複写する従来の方式



２）構造を複写しない最適化方式

図８　データベース内部の構造と複写された構造

述語の引数として使用される項は，その述語が呼び出された時点で１回だけ使用される項と，その述語が成功した後も使用され続ける項に分類することができる．前者をローカル項，後者をグローバル項と呼ぶことにする．たとえば，図９の is/2 の第２引数 +(Y, 1) は，算術式として is/2 の処理のみで使用されるのでローカル項に分類される．また，clause/2 の第１引数 member(Q, W)も，変数 Q，W，E に対応する項を得るために，clause/2 の処理のみで使用されるのでローカル項に分類される．一方，append/3 の第２引数 [4, 5, 6] は，第３引数の変数 X に代入されるリストの一部になり，以後の処理で使用されるのでグローバル項に分類される．

```
X is Y + 1

clause(member(Q, W), E)

append([1, 2, 3], [4, 5, 6], X)
```

図９　複合項の有効範囲の分類

このような項の分類には，述語が引数をどのように利用するかという情報が必要であるために，ユーザ定義の述語では処理系が正確に識別することが困難である．このため，この最適化は，組み込み述語のみを対象に行っている．このような制約があるが，一般的に組み込み述語（特に is/2）の出現頻度が高いために，十分な効果があるものと考えられる．たとえば，プログラムの解析結果[7]として，プログラム全体に占める組み込み述語の割合が 50% 程度で，とくに is/2 などの算術演算述語は組み込み述語の 11% を占めていることが報告されている．なお，図９で示した述語の他にも retract/1, assert/1, functor/3, arg/3 などの組み込み述語の引数もローカル項に分類される．

引数がローカル項に分類される組み込み述語では，述語を呼び出す際に引数の複写を行わず，図８，２）に示すようにデータベース内の原始項をそのまま引数として利用する．したがって，ローカル項を持つ組み込み述語の処理ルーチンは，データベースの複合項が引数として直接与えられても処理できるように構成しておく必要がある．しかしながら，データベースに格納されているプログラムは，変数を除きヒープスタックに複写された場合と同じ項形式で記録されているので，変数を処理する部分を除いてほとんど変更する必要はない．

この方式では，複写に必要な処理時間と共に，不必要な領域を回収するためのガーベッジコレクションに必要な処理時間が除去されるために，処理速度を向上させることができる．特に，ガーベッジコレクションの影響は，ベンチマークテストなどの小さなプログラムでは問題にならないが，大規模で実用的なシステムを構築する際には，処理速度を左右する大きな要因となる．また，ガーベッジコレクションを行わない処理系では，この方式を採用することによってメモリ効率が大幅に向上される．

この方式は，インタプリタだけでなくコンパイルドコードにも適用可能である．一般的に，コンパイルドコードでは，is/2 の第２引数の算術式は特別に扱われ，算術演算命令に置き換えられている．したがって，構造の複写は行われず，処理速度が低下したりメモリ効率が悪化することはない．しかしながら，clause，retract などの組み込み述語では，データベースとの単一化のために複合項が必要なために，is/2 のような最適化を施すことができず，構造の複写が行われている．しかしながら，本報告で提案している方式は，述語の引数のローカル項を unify 系の命令列に変換せずに，その構造をコンパイルドコード内に埋め込み，引数として直接使用することで利用することができる．

一般的に，構造共有法[4]は構造を複写する必要がないために，構造複写法と比較してメモリ効率がよく処理速度も早いとされているが，モレキュール[4]を作成する必要があるためにメモリ領域や処理時間は必要である．一方，本報告で示した構造の複写を抑制する方式は，メモリを使用せず処理時間も必要としないために，構造共有法よりも効果的な実現方式であると考えられる．

４　評価と考察

４．１　処理速度

本報告で示した構築法を採用した PROLOG インタプリタを PC-9801VX21 上に試作した．このインタプリタは，移植性を高めるためにＣ言語で記述されており，MS-DOS 上で稼働している．処理速度の測定は，代表的な３種類のベンチマークプログラム[8]を使用して行った．nreverse，qsort の測定では，repeat，fail ループでプログラムを 100 回実行させ，その実行時間を 100 で割って処理時間を得ている．また，queen8 の測定では，すべての解を求めさせ，その実行時間を処理時間としている．なお，試作したインタプリタの処理時間には，スタックのオーバーフローチェックと最大使用量の記録，キーボード割り込みの監視時間などが含まれている．

また，比較のために PC-9801VX21 上の Arity/Prolog と SUN-3/50 上の C-Prolog の処理速度を測定した．表１，表２に示すように，試作したインタプリタの処理速度は，同じパーソナルコンピュータ上の処理系と比較して 30% ほど向上している．また，ワークステーション上のインタプリタと比較した場合にも，同等以上の処理速度が得られている．

３種類の処理系で，実現手法やプロセッサが異なり測定条件が一定でないために，直接比較することはできないが，同一の測定条件が得られた場合を仮定して比較を述べる．試作したインタプリタが稼働している MS-DOS では，メモリを高速に操作する必要がある場合は，NEAR ポインタと呼ばれる 16 ビットのポインタが使用される．このポインタで操作できるメモリ領域は 64K バイト以下に制限されているために，Arity/Prolog を含む MS-DOS 上のほとんどの PROLOG インタプリタでは，ヒープスタックなどのデータ領域が 64K バイト以下に制限されている．この制限は，実用的なプログラムの作成を困難にしており，パーソナルコンピュータ上で実用的な応用システムを構築できないとされる理由の１つになっている．試作したインタプリタでは，このような制限をなくすために，HUGE ポインタと呼ばれる 32 ビットのポインタを使用してメモリを操作している．この方式では，実装されているすべてのメモリを利用できるという利点があるが，NEAR ポインタを使用した場合と比較して処理速度が３分の１程度に低下する．したがって，Arity/Prolog などの従来の処理系と同一条件で比較するために NEAR ポインタを使用した場合には，処理速度が３倍ほど向上するものと考えられる．条件が異なる現時点でも本インタプリタの方が 30% ほど処理速度が高いので，結果的に４倍以上の処理速度が得られるものと考えられる．また，試作したインタプリタをワークステーションに移植した場合の処理速度の予測を行うために，本インタプリタと基本的な実現方式が共通な append/3 のコンパイルドコードに相当するプログラムをＣ言語で記述し，PC-9801VX21 と SUN-3/50 で実行させた．この結果，PC-9801VX21 では 20K LIPS 程度，SUN-3/50 では 90K LIPS 程度の処理速度が得られた．この性能比で単純に予測すると，SUN-3/50 で

は nreverse で約 10K LIPS，qsort で約 7.7K LIPS の処理速度が得られることになる．これは，C-Prolog の４～５倍の処理速度であり，本報告で示した各種の最適化手法の効果が示されている．

表１　ベンチマークテストの処理速度
（単位：　LIPS）

述語	本インタプリタ	Arity/Prolog	C-Prolog
nreverse	2.2 K	1.6 K	1.9 K
qsort	1.7 K	1.3 K	1.7 K

表２　ベンチマークテストの処理時間
（単位：　ms）

述語	本インタプリタ	Arity/Prolog	C-Prolog
nreverse	220	310	260
qsort	350	480	360
queen8	43,000	75,000	53,000

４．２　述語の動特性宣言とインデッキシング

静的述語の宣言によって，処理速度やメモリ効率がどの程度向上するかを測定した．メモリ使用量は宣言の効果が表れる制御スタックとトレイルスタックを測定している．測定に用いたプログラムは，静的，動的のそれぞれの測定について，プログラムを構成するすべての述語の動特性を宣言によって統一している．

表３に示されるように，３種類のプログラムすべてで処理速度とメモリ効率が向上しており，最適化の効果が明らかになっている．処理速度は，最大の nreverse で 30% 程度，最小の qsort でも 5% 程度向上している．また，制御スタックのメモリ使用量は，最高の qsort で 270 分の１以下，最低の queen8 で５分の１になっており，メモリ効率が大幅に向上している．

表３　静的述語と動的述語

述語	静的述語			動的述語		
	処理時間(ms)	メモリ使用量(Byte)		処理時間(ms)	メモリ使用量(Byte)	
		制御	トレイル		制御	トレイル
nreverse	220	0.7 K	0	310	20.4 K	1.8 K
qsort	350	0.0 K	0	380	13.5 K	1.2 K
queen8	43,000	0.9 K	0.1 K	45,000	4.3 K	0.2 K

表３の測定結果で，qsort に対して nreverse ほど宣言の効果がないのは，qsort から呼ばれている partition/4 の第１節にカットがあり，この節が選択された場合には動的述語であっても選択点が削除され，静的述語として最適化される効果が相対的に低くなるためと考えられる．また，queen8 で静的述語と動的述語にほとんど変化がないのは，生成検査法によって処理を行う本質的に非決定性プログラムとなっているために，静的述語とした場合でも選択点が生成されることが多いためと考えられる．

述語の動特性の宣言は，ユーザにとって面倒かもしれないので，試作したインタプリタでは，宣言しない述語は静的述語に分類されるようになっている．したがって，実行時に変更される述語以外は，宣言を特に意識する必要はない．また，コンパイラの最適化に利用されるモード宣言と比較した場合には，モード宣言は各述語の引数の働きを正確に把握しなければならないが，動特性の宣言は assert, retract されている述語のみを意識すればよいので簡単である．宣言が間違っている場合には，モード宣言では予期しない動作を生じるが，動特性の宣言ではエラーが検出されるか処理速度が低下するだけで安全である．さらに，プログラムを最終的にコンパイルするのであれば，コンパイルする述語であるかインタプリタによって実行する述語であるかをコンパイラに対して宣言する必要があるために，動特性の宣言をその宣言とみなせば問題はないと考えられる．

４．３　環境生成の抑制

環境の生成を抑制するかしないかだけを変更したインタプリタを用意して，環境の生成を抑制した場合に処理速度がどの程度向上するかを測定した．

表４に示されるように，nreverse 以外のプログラムで処理速度の向上がみられ，環境の生成の抑制効果が明らかになっている．nreverse で効果がないのは，プログラムに組み込み述語が含まれていないために，環境の生成を抑制できる部分がないためである．一般に，コンパイルドコードでは，選択点や環境などのフレームの生成を抑制することによって処理速度が大幅に向上されるが，表４の qsort, queen8 では処理速度がそれほど向上していない．この原因は，インタプリタのプログラムの解釈実行のオーバヘッドが大きいために，全体の処理時間に占めるフレーム生成の処理時間が相対的に小さくなっているためと考えられる．

環境に関する最適化手法として，サブゴールを呼び出すたびに，環境上のローカル変数領域を必要最小限に切り詰めて，制御スタックの利用効率を向上させる方式が WAM に導入されている．この方式は，インタプリタでも実現することが可能であるが，メモリ効率がほとんど向上されないことと，処理速度が低下することがわかったために採用していない．しかしながら，本報告で述べている方式は，環境フレーム自体の生成が抑制されるために，WAM の方式より大幅なメモリ効率の向上と処理速度の向上が実現されている．

表４　環境生成の有無
（単位：　ms）

述語	環境フレーム有	環境フレーム無
nreverse	220	220
qsort	370	350
queen8	46,000	43,000

４．４　構造複写の抑制

構造複写の抑制による効果を調べるために，図１０のアッカーマン述語の処理時間とメモリ使用量を測定した．図１０の定義より明らかなように，アッカーマン述語は数値演算を行うだけでデータ構造を作成しない述語である．し

かしながら，従来の方式である構造の複写を抑制しない方式では，演算が行われるたびにヒープスタックに２種類の複合項 +(X, 1)，-(X, 1) が大量に複写され，メモリ効率が悪化することが明らかになっている．一方，本報告で提案した構造の複写を抑制した方式では，ヒープスタックはまったく使用されず，処理速度も 12% ほど向上している．

```
ack(0, N, V) :-
        V is N + 1.
ack(M, 0, V) :-
        M1 is M - 1,
        ack(M1, 1, V).
ack(M, N, V) :-
        M1 is M - 1,
        N1 is N - 1,
        ack(M, N1, U),
        ack(M1, U, V).

| ?- ack(3, 3, X).
```

図１０　アッカーマン述語

表５　構造複写抑制の効果

述語	構造の複写有		構造の複写無	
	処理時間 (ms)	ヒープ使用量 (Byte)	処理時間 (ms)	ヒープ使用量 (Byte)
ack	3,300	43.6 K	2,900	0

４．５　プログラムの読み込み時間

フロッピーディスク上の 1900 行，38K バイト程度のプログラムを読み込んだ場合の処理時間は 12 秒程度であった．この処理時間には，フロッピーディスクのアクセス時間も含まれているので，実際にプログラムの読み込み処理のみに要した時間は 12 秒以下である．したがって，１秒当り 150 行以上の割合でプログラムが読み込まれている．本インタプリタでは，プログラムの読み込み時に最適化処理のためのさまざまな解析を行っているが，これらの解析が十分に短い時間で行われるために，インタプリタの会話性がそこなわないことが示されている．

この処理時間には，節を単に assert するだけでなく，字句解析や構文解析を行った上で読み込まれた項をいったんヒープスタック上に作成する処理時間が含まれている．したがって，プログラムの実行時に，入出力を介さずにヒープスタック上の項を assert する場合には，処理速度はさらに早くなると考えられる．

５　結論

インタプリタの重要性と，コンパイラベースのインタプリタの問題点を示し，その問題点を解決すると共に，各種の最適化手法を利用するインタプリタの構築法を述べた．また，本報告で提案した手法を採用したインタプリタを試作し評価することによって，従来のインタプリタと比較して，処理速度が４倍程度になり，メモリ効率も大幅に改善されることを示した．

本報告で述べた手法を採用したインタプリタは，32 ビットのプロセッサでは，1 MIPS あたり 5K LIPS 程度の処理速度が得られると考えられる．したがって，10 MIPS の処理速度を持つ最近のワークステーション上では，インタプリタでも PROLOG マシン PSI 以上の処理速度が得られることになる．この処理速度はプログラムの開発段階で得られるために，コンパイルドコードのみの性能が高い従来の処理系と比較して，効果的な開発環境を実現することができると考えられる．また，本報告で述べた構成法を利用することによって，高価なワークステーションではなく，従来不適当とされてきたパーソナルコンピュータ上で，実用的な応用システムを作成できる処理系を実現することができる．この結果，オフィスなどで大量に使用されているパーソナルコンピュータ上で PROLOG を利用した応用システムを構築することが可能になり，知識情報処理の実用化と適用分野の拡大に寄与することができると考えられる．

６　今後の課題と予定

今後の課題としては，メモリ効率や処理速度をさらに向上させるために，今回利用しなかった最適化手法を導入し評価を行うことが考えられる．たとえば，ハッシュ法を利用したインデッキシングや，レジスタ割当の最適化は，節の登録時の負荷が大きくなりインタプリタの会話性が損なわれるという理由で採用しなかったが，実際にどの程度の負荷が生じるかを評価する予定である．

また，試作した処理系をワークステーションに移植すると共に，コンパイラを実装し実用的な処理系として整備していく予定である．さらに，メモリ容量の少ないコンピュータでも実用的な応用システムを作成できるように，試作した処理系にメモリ効率を向上できる CDR コーディング方式[9]を導入し評価することを考えている．

参考文献

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

[2] Buettner, K.A.: Fast Decompilation of Compiled Prolog Clauses, Third International Conference on Logic Programming. Lecture Notes in Computer Science. Vol. 225, Springer-Verlag, pp.663-670, (1986).

[3] 小長谷: 高速 Prolog インタプリタの構築法とその評価について，情報処理学会，記号処理研究会資料，46-4，(1988).

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

[5] Pereira, L.M., et al.: User's Guide to DECsystem-10 Prolog, Dept. of Artificial Intelligence, University of Edinburgh, (1978).

[6] Warren, D.H.D.: An Improved Prolog Impremention Which Optimises Tail Recursion, Dept. of Artificial Intelligence, University of Edinburgh, Research Paper No.141, (1980).

[7] 尾内 他: 逐次型 Prolog プログラムの解析，ICOT, Proceedings of The Logic Programming Conference '84, 9-5, (1984).

[8] 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).

[9] 碓崎 他: PROLOG 処理系アーキテクチャの拡張と最適化方式の提案，ICOT, Proceedings of The Logic Programming Conference '88, pp.151-160, (1988).