インクリメンタル PROLOG コンパイラの設計と実現†

磁崎賢一††* 上原邦昭††† 豊田順一†††

本論文では，研究開発や実用的なシステムの構築に適した，高速で利用しやすい PROLOG 処理系の実現方式とその評価について述べる．開発した処理系では，インタプリタによって実行される述語とコンパイルされ機械語によって直接実行される述語を混在して実行することができる．また，処理系が稼動している状態で，プログラムのコンパイルとリンクを述語単位で順次行うことができるインクリメンタル・コンパイル機能を実現している．これらの機能は，述語間の相互呼び出しを管理するディスパッチャと呼ぶ機構によって実現されている．さらに，DEC-10 PROLOG などで問題となっていた，コンパイラとインタプリタの単一化のセマンティックスの相異を解消することができる仮想 PROLOG マシンの命令セットを提案している．この命令セットは，従来の最適化方式のほかにリスト処理に対する最適化方式を導入しており，高速なコンパイルドコードを生成できるという特徴を持っている．コンパイルドコードは，この命令セットを中間コードとして利用し，最適化を施した後にターゲットマシンの機械語に変換して生成している．本処理系は UNIX ワークステーション上に実現されており，同じデータ表現法である構造共有法を採用した PROLOG マシンと比較して，約 2.5 倍の 85 K LIPS の性能が得られている．

1. はじめに

人工知能の研究が広く行われるようになるにつれて，記号処理言語 PROLOG が多用されるようになってきている．PROLOG は単一化処理や非決定性処理などの高度な能力を特長として持っているが，インタプリタでこれらの能力を実現すると負荷が大きくなるために，従来の PROLOG 処理系は性能の低いものが多い．このため，コンパイラを装備した高性能な PROLOG 処理系の実現が強く望まれている．このような現状を踏まえて，本研究は，インタプリタとコンパイラが一体となり，全体として高い性能を発揮する，使いやすい PROLOG 処理系の実現方式の研究を行ったものである．

われわれが開発した PROLOG 処理系は，C-Prolog インタプリタ[1]をベースとして，インコア方式のコンパイラを結合したもので，インタプリタ上でプログラムをコンパイルし，コンパイルドコードをそのままインタプリタの処理環境中に読み込んで実行するインクリメンタルな機能を実現している．コンパイルドコードは，ディスパッチャ[2)-4)]と呼ぶ機構によってインタプリタ本体と結合され，インタプリタによって実行される述語（インタプリテッド述語と呼ぶ）とコンパイルされた述語（コンパイルド述語と呼ぶ）の相互呼び出しが可能になっている．また，コンパイルは，DEC-10 PROLOG[5] などの従来のコンパイラがファイル単位でしか指定できないのに対して，ファイル単位のみでなく任意の述語単位で指定できるために，プロトタイピングなどに適した自由度の高いものになっている．

一方，コンパイルドコードは，インタプリタとコンパイラのセマンティックスを統一することと，高度な最適化を施すことを目標として，仮想 PROLOG マシン命令セットを新たに提案し，この命令セットを利用して生成している．このため，DEC-10 PROLOG で問題となっていたセマンティックスの相違が生じないという特徴がある．また，従来の最適化方式のほかに，多用されるデータ構造のリストを高速に処理するための最適化方式を導入しており，高性能なコンパイルドコードを生成することができるという特徴を持っている．なお，本研究で提案する実現方式を評価するために，ワークステーション Sun-3/260 上に PROLOG 処理系を開発し，85 K LIPS の性能を得ている．

2. コンパイラの実現方式

2.1 処理系の構成

C-Prolog インタプリタは，DEC-10 PROLOG の実現にあたって Warren が提案した PLM[6] (DEC-10 PLM と呼ぶ) に基づいて構築されている．したがって，インタプリタと結合するコンパイラも，両者間での整合性をとるために，DEC-10 PLM に基づいた手

† The Design and Implementation of an Incremental PROLOG Compiler by KEN'ICHI KAKIZAKI (Faculty of Engineering Science, Osaka University), KUNIAKI UEHARA and JUN'ICHI TOYODA (Institute of Scientific and Industrial Research, Osaka University).
†† 大阪大学基礎工学部情報工学科
††† 大阪大学産業科学研究所
* 現在 九州工業大学情報工学部電子情報工学科

法を採用することが望ましい．本研究の初期段階[2]で開発したコンパイラは，このような理由で，DEC-10 PLM の概念を用いてコード生成を行っていた．しかしながら，この方式では，

1) テールリカージョンの最適化[7]

コンパイル時に，再帰処理を繰り返し処理に変換してコード生成を行うもので，再帰呼び出しのたびに，必要とされる新たなスタックフレームを確保する操作を除去して効率を向上させる手法．

2) スタックフレーム操作の最適化[8]

スタックフレームに格納されるデータを用途別に分類し，スタックフレームを2種類に分割することによって，スタックフレームの作成と操作を，必要最小限に抑えて効率を向上させる手法．

などの，処理速度とメモリ効率を大幅に向上させるために不可欠な最適化手法が利用できないという問題があった．この問題は，DEC-10 PLM が設計された時点では，上述の最適化手法を利用することが考慮されていなかったために，引数の受渡し方式やスタックフレームの構造と操作方式が障害となって生じたものである．

一方，これらの最適化手法を利用するために，Warren によって新たに提案されている WAM (Warren's Abstract Machine)[8] では，

1′) 引数レジスタを用いて述語の引数を受け渡す方式

2′) スタックフレームを用途別に分類して個別に操作する方式

などが導入されている．これらの方式を DEC-10 PLM に導入すれば，本研究の意図に適した処理系を構成することができると考えられる．以後の議論では，この拡張された DEC-10 PLM を E-PLM (Extended PLM) と呼ぶ．E-PLM では C-Prolog インタプリタとの整合性を保つために，DEC-10 PLM と同一のデータ表現法である構造共有法[6]を採用している．なお，構造共有法とは複合項をモレキュール (molecule) と呼ぶデータ構造で表現する方式で，モレキュールは複合項の構造を表すスケルトン (skeleton) と，単一化処理によって変数に束縛される値を格納する変数フレームから構成されている．

E-PLM は DEC-10 PLM と異なる処理方式になっているために，両者間の処理方式の相違を補完する操作が必要になる．しかしながら，前述の最適化手法を利用できるために，DEC-10 PLM の概念を直接使用した場合と比較して，コンパイルコードの速度を約2.6倍ほど向上させることができる[3),4)]という利点がある．なお，両者間での補完については3章で詳しく議論する．

2.2 コンパイル処理

本コンパイラは，高速処理を実現するために，ターゲットマシンのマシンコードを生成する方式を採用している．コンパイルは以下の3段階の処理によって行っている．コンパイル処理の第1段階では，PROLOG プログラムを E-PLM の命令コード (E-PLM コード) に変換する．第2段階では，E-PLM コードをマクロアセンブラによってターゲットマシンのマシンコードに変換する．この処理を行うために，すべての E-PLM コードをあらかじめアセンブラのマクロとして定義している．第3段階では，インタプリタに組み込んだローダによってマシンコードをインタプリタ内部に読み込み，インタプリタの各種の処理ルーチンとリンクして実行可能な状態にする．

マシンコードに含まれている入出力やデータベース操作などのシステム組込み述語の多くは，インタプリタ内部に用意されている処理ルーチンを直接用いている．また，E-PLM のデータ表現法には C-Prolog インタプリタと同一の構造共有法を採用しているため，インタプリタの単一化処理や入出力処理のルーチンをまったく変更せずにシステムを実現することが可能になっている．

3. インタプリテッド述語とコンパイルド述語のインタフェース

2.1 節で述べたように，E-PLM では，引数の受渡し方式とスタックフレームの操作方式がインタプリタの方式とそれぞれ異なったものになっている．したがって，インタプリテッド述語とコンパイルド述語を共存させるためには，これらの相異点を解消しなければならない．この問題の解決の例として，呼び出された述語が，呼び出した述語と自分自身の実行形式が同じであるか否かを判定し，異なっている場合には，相違点を解消する処理を行うという方式が考えられる．しかしながら，この方式では，すべての述語に前述の処理を行う機能が必要であり，各述語内部の処理方式が複雑になるという問題がある．

本研究では，この問題を解決するために，まずメッセージ交換の概念を用いて PROLOG の実行メカニズムをモデル化し，述語間の制御の流れを検討した．

さらに，この実行モデルに基づいて両述語間のインタフェースを制御するディスパッチャと呼ぶ機構を導入した．以後の議論では，この実行モデルを特にメッセージモデルと呼ぶ．

3.1 PROLOG の実行モデル

メッセージモデルにおける PROLOG 述語は，メッセージを受け取り，メッセージに対応した処理を行い，処理結果を再びメッセージとして呼び出し側の述語に返す処理を行うものである．したがって，このモデルでは，異なる実行形式の述語内部での処理方式が異なっていても，両者間でメッセージ形式とメッセージに対応する処理の結果を統一さえしていれば，システムとして一貫した処理を行うことができることになる．

メッセージモデルでは，各述語が以下の３つのメッセージを持っているものとする．

① call（コール処理）

呼び出される述語の名前とその述語に渡す引数をメッセージ引数として持つ．

② continuation（継続処理）

③ fail（後戻り処理）

これらのメッセージは，それぞれ，①ゴールとなる述語を起動するコール処理，②起動された述語が成功した場合に呼び出し側の述語に戻る継続処理，③起動された述語が失敗した場合に再試行を行う後戻り処理に対応している．

メッセージモデルでの述語の呼び出し処理は以下のように行われる．まず，述語が他の述語を呼び出す場合は，呼び出し側が呼び出される側に call メッセージを送る．次に，呼び出された述語はメッセージの引数に従って単一化処理などを行い，その処理結果に応じて continuation か fail メッセージのいずれかを呼び出し側に返す．さらに，呼び出し側の述語は，返されたメッセージに基づいて継続処理か後戻り処理を行う．

メッセージモデルでは，インタプリテッド述語とコンパイルド述語間の引数の受渡し方式とスタックフレームの操作方式の相違は，単にメッセージの引数形式の相違とみなすことができる．このため，両者の処理方式の相違は，メッセージの変換操作を行うことで解決することができる．このようなメッセージの変換操作を専門に行う機構として，３つのメッセージに対応してそれぞれメッセージ変換操作を行うディスパッチャを導入している．ディスパッチャは，メッセージを受け取る述語の実行形式に応じて適切な変換を行い，実行形式が異なる述語間で相互呼び出しを可能にするものである．ディスパッチャを導入したメッセージモデルでは，述語間での直接的なメッセージの交換は行わず，ディスパッチャを介してのみ（図１）メッセージの交換が行われる．このように，インタプリタの内部処理をコンパイルドコードの内部処理から隔離して影響を受けないようにしているために，ディスパッチャを導入すれば C-Prolog インタプリタの変更が少なくてすむという利点が生じる．

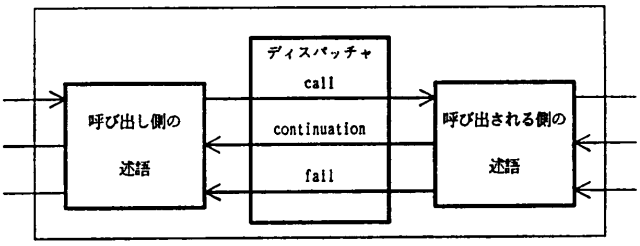
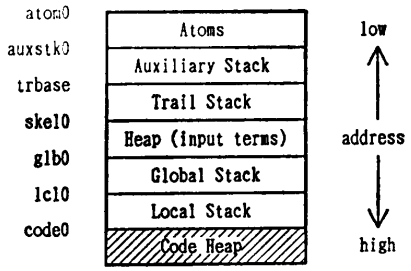


図 1 PROLOG の実行モデル
Fig. 1 PROLOG execution model.



斜線部分はコンパイラ用の拡張

図 2 C-Prolog コンパイラのメモリマップ
Fig. 2 Memory map of C-Prolog compiler.

3.2 ディスパッチャの処理

3.2.1 述語の実行形式の判定

呼び出される述語の実行形式に応じてメッセージ形式を変換するためには，プログラムの実行時に述語の実行形式を判定する必要がある．この判定は，C-Prolog インタプリタが用途に応じて複数の領域に分割して管理しているメモリの配置情報（図２）を利用して行っている．本システムでは，このメモリ領域を拡張して，コンパイルドコードを格納するためにコードヒープ領域を新たに設けている．インタプリテッド述語の定義となるデータ（スケルトン）とコンパイルド述語のマシンコードは，それぞれヒープ領域とコードヒープ領域に分離して格納しており，述語定義を指し

ているポインタはいずれか一方の領域を示している．したがって，ディスパッチャは，述語の実行形式を判定するために，ポインタがどちらの領域を指しているかを調べるのみでよい．

3.2.2 引数の受渡し

C-Prolog インタプリタは，サブゴールを示すモレキュールのアドレスを用いて引数を受け渡す，参照渡し方式を採用している．一方，E-PLM は引数レジスタを設け，引数レジスタを用いて引数を受け渡す，値渡し方式を採用している．したがって，互いに異なる実行形式の述語から呼び出された場合には，引数の受渡し方式が異なるために，引数を参照できないという問題がある．このような問題を解決するために，ディスパッチャが行う引数の受渡し方式の変換操作を，ゴール pred(1, X, Y) の変数がそれぞれ $X=[\ ]$, $Y=2$ に束縛されている場合（**図3**）を例として説明する．

インタプリテッド述語からコンパイルド述語が呼び出される場合は，ゴールに対応するモレキュールから引数レジスタに引数がコピーされる（図3，1）．この操作では，引数が定数の場合にはスケルトンから，引数が変数の場合には変数フレームから値が取り出され，対応する引数レジスタにコピーされる．この結果，コンパイルド述語に引数レジスタを通して引数が受け渡されることになる．逆に，コンパイルド述語からインタプリテッド述語が呼び出される場合には，ゴールに対応するモレキュールが生成される（図3，2）．このモレキュールに使用されるスケルトンは，すでにコンパイル時に作成されていたもので，すべての引数が変数となったものである．コンパイルド述語によって設定された引数レジスタの内容は，変数に束縛される値として対応する変数フレームにコピーされる．この結果，インタプリテッド述語にモレキュールによって引数が受け渡されることになる．

3.2.3 スタック管理

スタックフレームに格納されるデータは，複数のサブゴールを持つ節の実行を管理するための，環境を示すデータと，後戻り処理を行うために必要な選択点を示すデータに分類することができる．C-Prolog インタプリタでは，環境と選択点を1つのフレームとしてまとめ，述語が呼び出された時点で，処理内容に関係なく選択点と環境を対にしたフレームを作成する方法を採用している．一方，E-PLM では環境と選択点を示すフレームを分離して，必要な時点で必要なフレームのみを作成する方式を採用している．したがって，互いに異なる実行形式の述語から呼び出された場合には，フレーム構造が異なるために，継続処理や後戻り処理を行うためのフレーム操作が行えないという問題がある．このような問題を解決するために，ディスパッチャによって行われるフレームの変換操作を説明する．

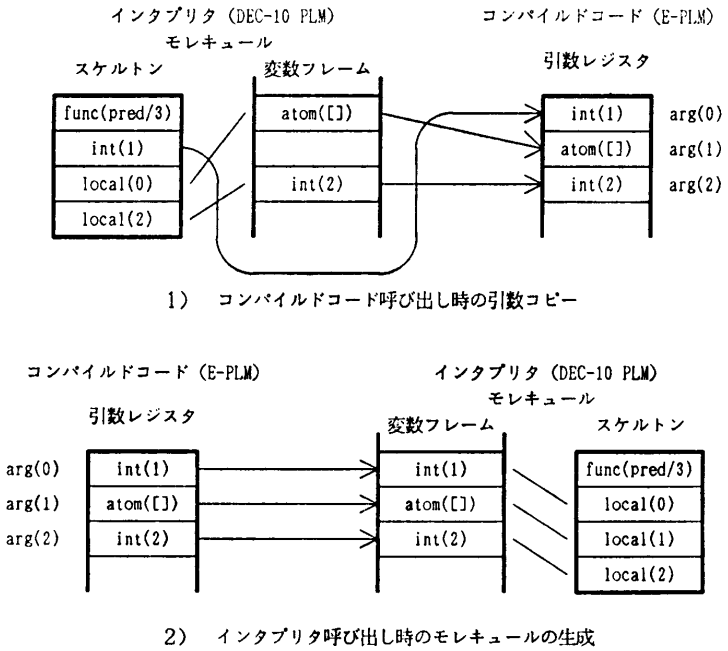
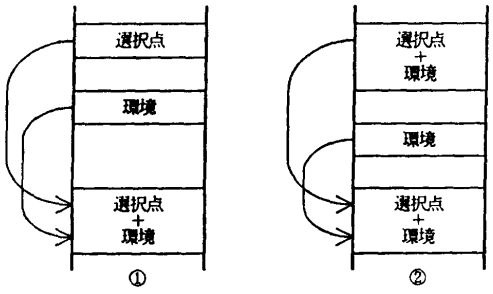
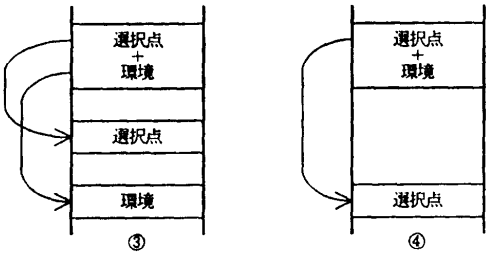


図3 引数の受渡し
Fig. 3 An example of arguments passing.

コンパイルド述語からインタプリテッド述語が呼び出される場合には，コンパイルド述語が作成したフレームを参照して，インタプリテッド述語が操作可能なフレームを作成する（**図4**，①）．この操作では，呼び出し側がコンパイルド述語であるために，選択点と環境のフレームのいずか一方，あるいは両者が作成されていないという状況が生じる可能性がある．このような場合には，ローカルスタック上の1つ前のフレームを利用して不足しているデータを補い，新たなフレームを作成するようにしている（図4，②）．逆に，インタプリテッド述語からコンパイルド述語が呼び出される場合には，インタプリテッド述語が作成したフレームを参照して，コンパイルド述語が操作可能なフレームを作成する（図4，③）．この操作では，呼び出し側が



1) コンパイルド述語からインタプリテッド述語



2) インタプリテッド述語からコンパイルド述語

図 4 ローカルスタック上のデータ変換
Fig. 4 Conversion of frame structures in a local stack.

インタプリテッド述語であるために，フレーム上の選択点と環境の情報のうち，どちらか一方あるいは両者が不必要になる情況が生じる可能性がある．したがって，この操作では呼び出し側のインタプリテッド述語が作成したフレームの必要な情報のみを利用して，新たなフレーム（図4，④）を作成するようにしている．

4. E-PLM コードの設計

命令セットの設計は，拡張を施した E-PLM に基づいて，最適化処理のしやすさとコンパイラのコードの生成しやすさを目標として行っている．E-PLM 命令は以下の6種類に分類できる．

1) インデッキシング処理
2) 選択点の生成と削除処理
3) 引数レジスタを用いた単一化処理
4) 複合項の単一化処理
5) 引数レジスタへの値の設定処理
6) 述語の呼び出しと継続処理

1) の命令は，節を選択する際に第1引数の値をキーとしてインデッキシング処理を行うもので，ハッシュ法[9]を利用することによって高速化している．2) の命令は，後戻り処理に必要な選択点の生成，更新，削除を行うものである．3) の命令は，引数レジスタによって受け渡された引数と呼び出された述語の引数を単一化するものである．4) の命令は，入れ子になった複合項を単一化するものである．5) の命令は，呼び出される側の述語の引数を引数レジスタに設定するものである．6) の命令は，コール処理や継続処理を行うものと，継続処理に必要な環境の生成，削除を行うものである．以後の節では，特に 1)，4) の命令に重点をおいて説明する．なお，他の命令については WAM の命令[8]に準じているので本論文では述べない．

4.1 単一化処理

単一化命令は，アトム，変数，複合項などの各タイプに対応するように設計しているため，単一化命令をプログラム中の各項と1対1対応させて並べることによって E-PLM コードを生成することができる．しかしながら，この方式を単純に使用すると，入れ子構造が深い複合項の場合には，生成されるコードの量が膨大になるという問題がある．このため，DEC-10 PROLOG では複合項のレベル2よりも深い構造の単一化処理を省略[6]しているが，インタプリテッド述語とコンパイルド述語のセマンティックスが異なるという新たな問題が生じている．このような問題を解決するために，E-PLM では，複合項のレベル2以上の入れ子になっている複合項のすべての単一化処理を1命令で行う unify_structure 命令を導入している．

図5の述語 pred/3 の第2引数の複合項を例として，生成される E-PLM コードとその単一化処理方法を説明する．単一化処理は，呼び出し側の述語から受け渡された引数が定数であるか変数であるかによって内容が異なるために，引数が定数の場合を入力モード，変数の場合を出力モードと呼び，両者を区別して説明する．

出力モードの単一化処理は，基本的にスケルトンからインスタンス（instance，単一化処理によって生成される項）を作成するもので，以下に示す2つの処理が行われる．

1) インスタンスとなるモレキュールを作成し出力引数の値とする．

2) 複合項に含まれた各変数に束縛されている値を変数フレームに代入する．

図5では，1）の処理はA）の命令，2）の処理はC)，D）の各命令によって行われている．インスタンスとして作成されるモレキュールは，A）の命令の

```
pred(X, a(b, X, c(Y, d)), Y) :- ...

ユニフィケーション命令      命令の引数                  命令に対応する項
A) get_structure          arg(2), skelp, 2          a(
B) unify_atom             atom(b), skel(1)            b,
C) unify_value            temp(0), var(0), skel(2)    X,
D) env_variable           temp(1), var(1)               Y
E) unify_structure        skel(3)                     c(Y, d) )

skelp = skelton(functor(a/3),atom(b),var(0),skelton(functor(c/2),var(1),atom(d)))
                                                      skel(3)
```

図 5 複合項の単一化命令
Fig. 5 An example of the instruction sequence for the compound term.

引数として与えられるスケルトン（図5，skelp）と変数フレームから構成されている．構造共有法における出力モードの単一化処理では，複合項の定数部分がスケルトンによって表現されているために，定数項の単一化処理はモレキュールが作成された時点で終了していることになる．したがって，定数項に対応する単一化命令B），E）は何も処理を行う必要はなく，出力モードの単一化処理は，複雑な複合項であっても高速に行うことができる．

一方，入力モードの単一化処理は，基本的にスケルトンと引数を比較するもので，以下に示す2つの処理が行われる．

1）複合項のファンクタを比較する．

2）複合項内で入れ子になった各項（アトム，変数，複合項）の単一化を行う．

図5では，1）の処理はA）の命令，2）の処理はB），C），D），E）の各命令によって行われる．このうち，入れ子になっている複合項 $c(Y, d)$ の単一化処理は，D），E）の命令によって行われる．E）のunify_structure は複合項の完全な単一化処理を行う命令である．

unify_structure は，複合項のスケルトンを参照しながら単一化処理を逐次実行する機能を持っており，インタプリタの単一化ルーチンを利用して実現している．この機能によって，入れ子構造の深い複合項であっても，1命令で完全に単一化することができるようになっている．unify_stucture を実行するためには，単一化処理を行うためのスケルトンが必要であるが，A）の命令の引数として用意されているスケルトンの一部分（図5，skel（3））を利用できるため，メモリをほとんど消費しないという利点がある．また，処理対象の複合項内に変数がある場合には，unify_structure 命令がその変数に対応した変数フレーム上のセル（図5，var（1））を参照して束縛された値を得るようにしている．なお，D）の命令は，unify_stucture 命令が参照する変数の値を変数フレームに事前に設定しておく処理を行うためのものである．

4.2 リスト操作

リストは，汎用のデータ構造として PROLOG で広く使われている．このため，リストを効率よく処理することができれば，さらに処理速度の向上を図ることができる．リストを操作する述語の多くは，append/3 のように，リストを分解して要素を操作する節と，リストの終端での操作を行う節から構成されている．このようなプログラムに対して，一般のコンパイラでは，**図 6**，1）に示すように，①インデッキシング命令によって節を選択し，②リストあるいは③ NIL との単一化命令によって単一化処理を行う3つの PLM コードを生成している．①の命令では，引数のデータ型を判定するために，また②，③の命令では，引数の単一化処理を行うために，それぞれデリファレンス処理[6]が行われるが，これらは互いに重複した処理となっている．

E-PLM では，このような重複したデリファレンス処理を除去するために，リスト処理専用のインデッキング命令 switch_on_list[3),4)] を新たに導入している．

```
:- mode append(+, +, -).
append([X|Y], ...
append([], X, X).

    switch_on_term List, Nil, fail  ①
List: get_list                      ②
    unify_variable
    unify_variable
      :
Nil: get_nil                        ③
    get_value
      :
1)  従来のコンパイラのコンパイルドコード

    switch_on_list List, Nil
List: unify_variable
    unify_variable
      :
Nil: get_value
      :
2)  本コンパイラのコンパイルドコード
```

図 6 リスト処理命令
Fig. 6 An example of the instruction sequence for list processing.

この命令を使用した E-PLM コードを図6，2）に示す．この命令はリスト処理において，第1引数によるインデッキシング処理と，それに続く単一化処理を1命令で行うものである．この命令によって，前述の3命令に重複して用意されていた冗長なデリファレンス処理を除去することができ，処理を高速化するとともにコードサイズを縮小することができる．

4.3 カット

カットの機能は，代替節を示す選択点フレームをローカルスタックから除去することによって実現できる．3.2.3 項で述べたように，C-Prolog インタプリタでは述語が呼び出された時点でかならずフレームが作成されるために，このフレーム以後に作成されたすべてのフレーム内の選択点の情報を除去することでカットの機能が実現されている．しかしながら，E-PLM では述語が呼び出された時点でフレームが作成されるとはかぎらないために，除去しなければならない選択点フレームを決定する基準になるフレームがなく，カットの機能を実現できないという問題がある．

この問題を解決するために，E-PLM では，カットを含む述語が呼び出された時点で，ローカルスタックの先頭アドレスを記録する操作を行っている．カットの機能は，このアドレスを基準にして除去する選択点フレームを決定し実現している．一方，カットを含まない述語では，このような操作は必要ないため，アドレスを記録するためのコードを生成せずに効率の向上を図っている．なお，同様な概念に基づき，PROLOG マシンに適したカットの実現方式[10]が提案されている．この方式では，パイプラインの空きを利用して見かけの処理時間をなくすために，呼び出し側の述語でスタックアドレスを記録している．しかしながら，この操作は，呼び出される述語にカットが含まれるか否かにかかわらずに行われるために，カットが含まれていない場合には無駄になる．このため，本システムのように，パイプラインの空きを積極的に利用することができない汎用プロセッサ上の処理系では，オーバヘッドを生じてしまうという問題がある．したがって，汎用プロセッサ上の処理系では，E-PLM で採用している方式の方が効果的であると考えられる．

4.4 組込み述語

組込み述語は，機能によって以下のように分類することができる．

1) 算術演算
2) 算術比較
3) 項の分類（タイプチェック）
4) データベース操作
5) 入出力操作
6) デバッガ

組込み述語は一般的に使用頻度が高いため，その高速化は処理系の性能向上に大変有効である．しかしながら，すべての組込み述語の使用頻度が高いわけではなく，同様な高速化を行っても述語によって効果は大きく異なる．したがって，組込み述語の処理は，それぞれの述語の特性に合わせて行わなければならない．

コンパイルドコードで組込み述語を処理する方法として，次に示すような方式が考えられる．

a）インタプリタを呼び出して，組込み述語の処理をインタプリタによって実行する方式

b）インタプリタ内部の組込み述語処理ルーチンを直後呼び出して実行する方式

c）アセンブリ言語で記述した組込み述語処理ルーチンを呼び出して実行する方式

これらの実現方式は，a），b），c）の順に開発時の負担が少なく，逆の順に性能が高いという特徴がある．本システムでは，1），2），3）に分類される組込み述語の処理は比較的単純であり，高速な処理が望まれるのでb）かc）の方式によって実現している．また，4），5），6）に分類される組込み述語の処理は，副作用を伴うために処理時間がかかり，インタプリタの呼び出し処理に要するオーバヘッドを無視することができるので，a）の方式によって実現している．

5. 評価と考察

処理系の性能を評価するために，ベンチマークテストを行った．

5.1 プログラム言語としての評価

ベンチマークには30要素のリストの反転を行うプログラム nreverse/2[11] を用いた．使用している最適化手法ごとに得られる実行時間と LIPS 値を表 1 に示す．最適化手法を利用したコードでは 85.0 K LIPS と高い速度（表1，3）を得ている．インタプリタに

表 1 コンパイルドコードの最適化手法別の性能
Table 1 Performance evaluation of a benchmark program.

	モード宣言	リスト	ms	K LIPS
1	×	×	9.9	50.0
2	○	×	7.8	63.6
3	○	○	5.8	85.0

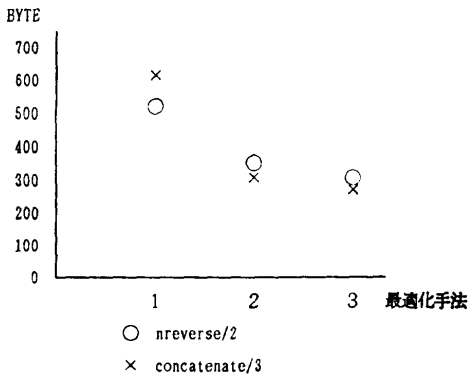


図 7 コンパイルドコードのサイズ
Fig. 7 Compiled code size.

よる実行は 5.9K LIPS なので，14.4 倍ほど高速化されており，コンパイラの有効性が示されている．

また，モード宣言[6]を行うことによって，コンパイルドコードの速度は 1.3 倍程度向上し，サイズは 0.6 倍程度に圧縮されている．本システムのようにマシンコードを直接生成するコンパイラでは，モード宣言は，速度を向上させるとともに，コードサイズを圧縮する大きな利点があると考えられる．ただし，モード宣言の効果は述語によって異なる．たとえば，concatenate/3 と nreverse/2 では，前者における効果の方が大きい（**図 7**）ことがわかる．これは，モード宣言によって最適化されるのが単一化命令であり，コンパイルドコードの中で単一化命令が占める割合が大きいほど効果が大きくなるためである．

5.2 リスト操作

リスト操作を高速化するために 4.2 節で導入した命令の効果を示す．この命令を使用した場合には，一般のインデッキシングと単一化命令を使用した場合に比べて速度は 1.3 倍程度に，コードサイズは 0.9 倍程度に改善されている．リストを操作する述語においては，新たに導入した命令を使用した最適化手法が，一般的なモード宣言による最適化手法と同様に効果的であることが示されている．また，引数がデリファレンス処理を多く必要とする場合には，この効果がさらに顕著になると考えられる．

5.3 複合項の単一化処理

本コンパイラでは，複合項のレベル 2 よりも深い構造の単一化処理はインタプリタの単一化ルーチンを利用して行っている．この方式では，単一化ルーチンが C 言語で記述されているために，アセンブリ言語によって記述されているコードと比較して効率が低いという問題がある．しかしながら，以下のような理由により，複合項の単一化処理の多くはインタプリタの単一化ルーチンを使用することなく行われるために，一般的な状況では処理速度の低下は無視できると考えられる．

1）append/3 や member/2 の例からもわかるように，PROLOG プログラムはレベル 2 より深い項の単一化処理を必要とするものは少ない．

2）レベル 2 よりも深い構造を持つ複合項は，入力引数としてよりも出力引数として用いられることが多い．出力引数の単一化処理はモレキュールを生成するのみでよいので，インタプリタの単一化ルーチンは使用されない．

以上のように，問題点も実質的な影響がないために，本コンパイラで採用した複合項の単一化処理方式は，高い性能，統一されたセマンティックス，小さなコードサイズという，コンパイラに対する 3 つの要求を満たす方式であると考えられる．

5.4 データベース操作言語としての評価

ベンチマークには 256 個のデータからなるデータベースの 256 個めと，128 個めのデータを検索するプログラムを用いた．実行時間およびインタプリタとコンパイルドコードとの速度比を**表 2** に示す．コンパイルドコードは，インタプリタに対して 560 倍の高い速度を得ており，大規模なデータベースの参照に対して，インデッキシングによる最適化処理が非常に有効であることが示されている．また，異なる位置のデータでもほぼ一定の時間で検索されており，ハッシュ法の効果が示されている．

5.5 ディスパッチャのオーバヘッド

ディスパッチャのメッセージ変換によるオーバヘッドを，nreverse/2 と concatenate/3 のコンパイルドコードを使用して測定した．オーバヘッドの測定には，通常のメッセージ変換が行われない場合と，メッ

表 2 データベース検索の性能
Table 2 Performance evaluation of a database access program.

	Byte	256		128	
		ms	倍率	ms	倍率
1	—	8.3	1	4.3	1
2	41,652	0.017	500	0.017	250
3	20,620	0.015	560	0.015	280

1 インタプリタ
2 モード宣言を行っていないコンパイルドコード
3 モード宣言を行っているコンパイルドコード

セージ変換が行われるように設定した場合のベンチマークテストの実行時間を比較して行った．

30要素のリストの反転を行う場合は，2つの述語間で31回の制御の移行が行われ，1回の移行につき2回のメッセージ変換が行われる．このベンチマークテストの実行時間は 9.3 ms で，メッセージ変換が行われない場合に比べて 3.5 ms のオーバヘッドが生じている．したがって，1回あたりのメッセージ変換に 56 μs ほど要していることになる．しかしながら，一般的には述語の実行形式が交互に異なることはほとんどなく，特に速度が重要な場合にはほとんどの述語がコンパイルされ，メッセージ変換のオーバヘッドが生じないために，速度に対する影響はほとんどないものと考えられる．また，ディスパッチャを導入することによって，E-PLM の最適化手法を利用できるようになったために，DEC-10 PLM を直接利用した場合と比較して2.6倍のコンパイルドコードの性能を得ることができたことを考慮すると，オーバヘッドによる速度低下は無視できると考えられる．

5.6 組込み述語

組込み述語の処理方式を評価するために，リストの長さを測る述語 length/2 を使用してベンチマークテトを行った．length/2 の繰り返し処理では，使用頻度の高い組込み述語 is/2 が毎回呼び出されるので，組込み述語の処理方式が実行速度に与える影響を示すのに最適であると考えられる．

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

テストは，1,000 個の要素を持つリストを length/2 の引数として行った．コンパイルドコードは，4.4 節で示した3種類の組込み述語の処理方式を用いて生成した．表3でのa），b），c）は，4.4節で示した各処理方式に対応している．この結果，c）の方式は実現のコストは最も高くなるが，a），b）の方式と比較して大幅に高速化されており，コンパイルドコードの利点を生かす最良の方式であることが示されている．一方，a）の方式でほとんど性能が向上していないのは，毎回の組込み述語の呼び出しにディスパッチャのオーバヘッドが含まれているためと考えられる．

表 3 組込み述語の処理方式別の性能
Table 3 Performance evaluation of a built-in predicate.

	ms	倍率
インタプリタ	516	1
a)	420	1.23
b)	260	1.98
c)	24	21.5

6. 開発の経緯

本システムは，まず DG のスーパーミニコン MV/8000 II 上で開発[3),4)]し，Sun-3 上に移値したものである．一般的に，コンパイラはプロセッサへの依存度が高いために移植が困難である．しかしながら，本システムではプロセッサに依存する部分が E-PLM コードをマシンコードに変換する処理に限定されているために，移植性が高いという利点があり，基本的に2人月程度の短期間で移植することができた．また，本システムの開発当初はマクロアセンブラやローダにいたるまで，ほとんどの部分を PROLOG で記述していた．しかしながら，このシステムは，すべての処理をメモリ上で一括して行っていたためもあって，比較的大きな述語をコンパイルする場合にメモリ消費量とコンパイル時間が激増し実用的でなかった．たとえば，節が50個程度の述語をコンパイルする場合に，8 M BYTE 程度のメモリ空間と約10分の処理時間を要した．このため，現在のシステムでは，ほとんどの部分をC言語によって記述したものに変更してこの問題を解決している．

7. おわりに

本論文では，高性能な PROLOG 処理系の実現手法として，インタプリタとコンパイラの整合性の高い結合方式と，最適化処理に適した E-PLM 命令セットの提案を行った．さらに，その方式を用いた処理系を実現し，提案方式の有効性を確認した．

今後の課題としては，PROLOG 処理系の問題として指摘されている，メモリ効率と後戻り処理をともなう処理の速度をさらに向上させるための研究があげられる．現時点では，前者に対しては，CDR コーディングを用いた手法[12),13)]を，後者に対しては，プログラム変換と新たな変数分類法を利用した最適化手法[13)-16)]を提案しており，さらに研究を進めていく予定である．また，本システムに関しては，大規模なプログラムを開発するためのモジュール化機能や，コンパイラを支援する各種のツールを実装していく予定である．

参考文献

1) Pereira, F. et al.: C-Prolog User's Manual, Dept. of Architecture, Univ. of Edinburgh (1984).
2) 砥崎賢一ほか: C-Prolog 上でのインタプリタとコンパイラの共存法について, 情報処理学会知識工学と人工知能研究会資料, 45-7 (1986).
3) 砥崎賢一ほか: C-Prolog コンパイラの開発, *Proc. of the Logic Programming Conference '86*, ICOT, pp. 159-166 (1986).
4) Kakizaki, K. et al.: Development of C-Prolog Compiler, In Wada, E. (ed.), *Logic Programming '86, Lecture Notes in Computer Science*, Vol. 264, pp. 126-138, Springer-Verlag (1987).
5) Pereira, L. M. et al.: User's Guide to DEC system-10 PROLOG, Dept. of Artificial Intelligence, Univ. of Edinburgh (1978).
6) Warren, D. H. D.: Implementing Prolog—Compiling Predicate Logic Programs, Research Reports 39 & 40, Dept. of Artificial Intelligence, Univ. of Edinburgh (1977).
7) Warren, D. H. D.: An Improved Prolog Implementation Which Optimizes Tail Recursion, Research Paper No. 141, Dept. of Artificial Intelligence, Univ. of Edinburgh (1980).
8) Warren, D. H. D.: An Abstract Prolog Instruction Set, Technical Note 309, SRI International (1983).
9) 砥崎賢一ほか: C-Prolog コンパイラの性能評価, 第33回情報処理学会全国大会論文集, pp. 497-498 (1986).
10) 黒沢憲一ほか: 内蔵型高速 PROLOG プロセッサ IPP (V) PROLOG 命令アーキテクチャ, 第34回情報処理学会全国大会論文集, pp. 197-198 (1987).
11) 奥乃 博: 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).
12) 砥崎賢一ほか: PROLOG 処理系における CDR コーディング方式, 第36回情報処理学会全国大会論文集, pp. 801-802 (1988).
13) 砥崎賢一ほか: PROLOG 処理系アーキテクチャの拡張と最適化方式の提案, *Proc. of the Logic Programming Conference '88*, ICOT, pp. 151-160 (1988).
14) 砥崎賢一ほか: 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).
15) 松本一夫ほか: PROLOG の非決定性処理を最適化するコンパイル方式の評価, 第36回情報処理学会全国大会論文集, pp. 803-804 (1988).
16) 松本一夫ほか: PROLOG コンパイラにおける非決定性処理の最適化方式, 情報処理学会記号処理研究会資料, 45-2 (1988).

(昭和63年 4 月 25 日受付)
(昭和63年12月 12 日採録)

砥崎 賢一 (正会員)

昭和35年生. 昭和60年大阪大学工学部造船工学科卒業. 昭和62年同大学院基礎工学研究科情報分野修士課程修了. 昭和63年同大学院博士課程退学. 同年九州工業大学情報工学部助手. 現在主として, 記号処理言語, 並列処理およびプログラミング環境に関する研究に従事している. 人工知能学会会員.

上原 邦昭 (正会員)

昭和29年生. 昭和53年大阪大学基礎工学部情報工学科卒業. 昭和58年同大学院基礎工学研究科博士後期課程退学. 同年, 大阪大学産業科学研究所勤務. 現在, 同研究所講師. 工学博士. 人工知能, 特に自然言語理解, および自動プログラム合成の研究に従事している. ACM, 電子情報通信学会, 計量国語学会, 人工知能学会, 日本ソフトウェア科学会各会員.

豊田 順一 (正会員)

昭和13年生. 昭和36年大阪大学工学部通信工学科卒業. 昭和41年同大学院博士後期課程単位取得退学. 同年大阪大学基礎工学部助手. 昭和44年助教授. 昭和57年大阪大学産業科学研究所教授. 工学博士. 現在, 主として, 自然言語理解, 画像理解, 文書画像処理, および ICAI システム等の研究に従事している. 電子情報通信学会, 日本認知科学会, 人工知能学会各会員.