記号処理 46－4
（1988. 7. 1）

高速Prologインタプリタの構築法とその評価について

小長谷 明彦
日本電気（株）C&Cシステム研究所

本報告では動的コンパイル機能と多機能命令セットを備えた高速Prologインタプリタの実現法とその評価について述べる。動的コンパイル法ではクローズデータベース登録時にクローズをコンパイルする。このとき、クローズインデクシングならびに最終呼び出しの最適化等の最適化技法を適用することにより、インタプリタにおいても手続き単位にコンパイルしたプログラムの３０～８０％の性能を達成できることを示した。また、多機能命令セットを用いると同一の命令列をプログラムの実行とソースイメージの取り出しの両方に使用することができ、インタプリタにおけるメモリ消費量を大幅に削減することができることを示した。このような命令を実現するためには実行時にモードをチェックする必要があるが、そのオーバーヘッドは小さく、適当なハードウエアサポートがあれば無視できる範囲である。

The Implementation and Evaluation of a Fast Prolog Interpreter

Akihiko Konagaya
C&C Systems Research Laboratories, NEC Corporation
1-1, Miyazaki 4-chome, Miyamae, Kawasaki, Kanagawa
213 Japan

This paper reports a fast Prolog interpreter with dynamic compilation facility and multi-interpretable instructions. According to our evaluation, the dynamic compilation makes it possible for the interpreter to achieve 30-80 percent of procedure-compiled programs by means of compiling a clause as soon as it is asserted into an internal clause database. Well-known optimization techniques, such as clause indexing and tail recursion optimization, are also applicable with some modification. The multi-interpretable instruction set greatly reduces the size of interpretive codes by using the same instruction sequence for both execution and clause-image construction. To achieve this, the instsructions have to check mode at run time, but the overhead is very small.

1. はじめに

第五世代計算機プロジェクトが発足していらい、知識処理用言語としてPrologに代表される論理型言語が注目されている。その理由の一つとして、Prologのクローズが実行可能なデータ構造としての性質を備えていることが挙げられる。すなわち、「知識」をクローズで表現することにより、知識を単なるデータ構造としてではなく、プログラムとして実行することが可能となる。これにより、知識処理システムの知識ベースや推論機構としてPrologの持つクローズデータベース機能や、ユニフィケーションによるマッチング機能、バックトラックを用いた推論機能をそのまま利用することが可能となる。

一方、近年、Prologをより効率よく実行するために、Warren Abstract Instruction Set(WAM)[1]に代表されるようなクローズのコンパイル方法が研究され、性能を飛躍的に向上させることが可能となってきた[2]。ただし、このような命令セットを用いた言語処理系ではコンパイルしたクローズは動的なクローズの追加、削除ができず、インタプリタ上のプログラムと動作が異なるという問題点があった。この問題を解決する一つの方向は動的なクローズの追加、削除を禁止し、インタプリティブコードの仕様をコンパイルドコードに合わせることである。このようなアプローチは高速な処理系を作成するという観点からは極めて合理的であり、また、インタプリタはデバッグ専用となるので、インタプリタの速度低下はあまり問題ならないという利点がある。しかしながら、知識処理システムにおけるPrologのクローズの使われ方、すなわち、「知識ベース＝クローズデータベース」という図式を考えると、このアプローチはある意味ではPrologの本質的な機能を損なう恐れがある。

本稿では、上記で述べたような問題点を解消する方式として、動的コンパイル法ならびに多機能命令をもちいた高速Prologインタプリタを提案する。動的コンパイル法は、クローズをクローズデータベース登録時に動的にコンパイルする方法である。これにより、動的に追加、削除が可能な述語についても、クローズインデクシングや終端呼び出しの最適化、ならびに引数渡しの最適化等の最適化技法を適用でき、高速実行が可能となる。我々の性能評価結果によれば、手続き単位にコンパイルしたときの約３０～８０％の性能を引き出すことが可能である。また、多機能命令セットはクローズの構造体イメージの生成とクローズの実行が同一命令で実現できるような命令セットである。このような命令セットを導入することにより、動的コンパイルを採用した際に構造体イメージを別途保持する必要がなくなり、インタプリティブコードのメモリ消費量を半減できるという利点を持つ。

以下、まず、２章で動的コンパイル法の課題と実現法について述べ、３章で多機能命令を用いたソースイメージの取り出し法および従来技法との比較について述べる。さらに、４章において、汎用計算機（ACOSシステム1000）上に作成したPrologマシンエミュレータによる処理性能およびメモリ使用量について報告する。

2. 動的コンパイル法

2. 1 設計課題

動的コンパイル法を実現するための課題は以下の２点である。

- 動的なクローズの追加／削除機能の実現。
- クローズのソースイメージの格納。

クローズの追加／除去を可能とするためには、クローズを節単位に格納、管理する必要

がある。このような動的なプログラムの管理を実現するために、動的コンパイル法では、itry_me_else, iretry_me_else, itrust_me_else_failという３つの命令を新たに導入している。これらの命令はそれぞれ先頭のクローズ、中間のクローズ、最後のクローズに対応し、クローズの格納、除去があると必要に応じて命令の書換えを行う。また、クローズインデクシングや最終呼び出しの最適化を行うための機構を備えている。これらの命令については次節で詳述する。

また、インタプリタを実現するためには、ソースプログラムの構造体イメージを確保しておく必要がある。ソースイメージの格納はクローズを構造体として保持しているときはほぼ自明であるが、動的コンパイル法ではその実現は重要な問題となる。一つの解決法はソースイメージとコンパイルしたコードの両方を保持するような方法であるが、このような二重管理は管理機構の複雑化とコード量の増加を招く。よりエレガントな解決法として本稿では多機能命令を提案する。多機能命令方式ではソースイメージの取り出しとクローズの実行を同一命令シークエンスで実現する。このような命令シークエンスの多重解釈を行うことにより、メモリ使用量を半減することができる。多機能命令方式については４章で詳述する。

２．２　動的コンパイル用命令

動的コンパイル用命令として以下の３命令を導入する。

- itry_me_else Key,Next
- iretry_me_else Key,Next
- itrust_me_else_fail Key

これらの命令の特徴はＷＡＭのインデクシング命令[1]とProlog-Xの先行検索方式[4]の両者の長所を兼ね備えている点にある。すなわち、ＷＡＭをベースとした命令体系を保持したまま、Prolog-Xの先行検索方式を実現した点にある。ＷＡＭのインデクシング機構の特徴はデータ型による分岐命令ならびにハッシュ表を用いた検索命令に用いて検索処理を最適化する点にある。ただし、このようなインデクシング方式は、コンパイル時に全てのクローズが定まっている場合はともかく、クローズの挿入、削除が頻発するインタプリティブコードの実現法としては適当でない。一方、先行検索方式ではクローズ毎に検索のための鍵を持たせ、クローズを実行する前にバックトラックの対象となるクローズ（代替クローズ）があるかないかを検索することにより、クローズ選択の高速化ならびに冗長な選択点の生成を抑えている。この方式では選択点の検索は線形となるがクローズの動的な追加削除が容易なためインタプリタのインデクシング方式として適している。しかしながら、Prolog-Xではこのような先行検索機構が述語の呼び出し命令ならびに「失敗」処理ルーチンに組み込まれているため、コンパイルコードとインタプリティブコードを混在させようとしたときにはうまくリンクできないという問題点がある。

動的コンパイル方式では、クローズの先行検索機能をitry_me_else,iretry_me_else,itrust_me_elseといったクローズのエントリ命令で実現することによりこの問題を解決する。これらの命令は先行検索を行うことを除けばＷＡＭのエントリ命令（try_me_else,retry_me_else,trust_me_else）と同様な働きをする。また、述語の呼び出し機構と先行検索機構が分離されているのでＷＡＭに基づく手続きコンパイルされた述語とも自然に混在実行することができる。各命令は制御が移ると与えられた検索鍵をもとにして、同一鍵を持つクローズを検索することでインデクシングを実現する。また、このとき、代替クローズが見

つからなければ選択点を生成しないことで最終呼び出しの最適化を実現する。インデクシングの鍵としてはヘッドゴールの第一引数のデータ型（変数、リスト）およびアトム、ファンクタを用いる。図１に鍵を用いたクローズの先行検索方式の概念図を示す。図１の例では、各クローズはａ，ｂ，ａを鍵として持つ。したがって、呼び出しゴールがｐ（ａ）のときは選択点を生成するが、呼び出しゴールがｐ（ｂ）のときは選択点を生成しない。

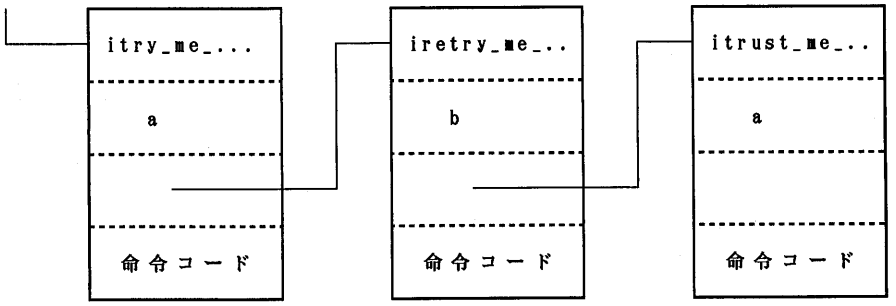

図１　クローズ先行検索方式

3.　ソースイメージの生成

3．1　従来方式との比較

前節で述べたように、動的コンパイル法の問題点の一つは、クローズのソースイメージをどのようにして保持するかにある。このような保持方式としては、以下の３種類が知られている[3]。

①　実行コードとソースイメージの両方を保持する方式。
②　実行コードから逆コンパイルによりソースイメージを取り出す方式。
③　ソース形式を表すユニットクローズをコンパイルして保持する方式。

しかしながら、各方式においては以下に述べるような問題点が存在する。まず、①の実行コードとソースイメージの両方を保持する方式では、単純にメモリ消費量が２倍必要となる。また、クローズの登録時間もソースイメージを登録する時間に加えて、クローズをコンパイルする時間がさらにかかることになる。次に、②の逆コンパイルする方式では、逆コンパイルに要する時間が大きいことや、ソースイメージの取り出しを可能にするために高度な最適化ができないという難点が挙げられる。さらに、③のユニットクローズとしてコンパイルする方式もメモリ消費量が増大する上に、登録時に２回コンパイルしなくてはならないという問題点がある。したがって、これら３種の方式は、いずれもクローズの追加、削除、ソースイメージの取り出しを頻繁に行うインタプリティブコードの実現法としては適当でない。

多機能命令方式では、モードにより、同一命令シークエンスをソースイメージの取り出しとクローズの実行の両方に使用する。すなわち、命令シークエンスを共有するので、コンパイルは一度で済み、メモリ使用量も少なくて済む。また、ソースイメージの取り出しも逆コンパイルではなく、コンパイルされた命令列をそのまま実行するだけで済むので極めて高速に実行できる。新たに導入した多機能命令を表１に示す。これらの命令はクロー

ズモード（組込述語 clauseを実行したとき）のときはソースイメージの生成を行い、実行モードのときはnoopまたは述語の実行を行う。

表１　多機能命令

命令	クローズモード	実行モード
ienter_unit Pred	Predを述語名とするヘッドゴールと本体部ゴールtrueの生成。	noop
ienter Pred	Predを述語名とするヘッドゴールの生成。	noop
icall Np,Pred	Predを第一引数とするＡＮＤゴールの生成。	述語Predの呼び出し。
iexec Pred	Predを述語名とする最終ゴールの生成。	最終ゴールの呼び出し。
icut Var	カットを引数とするＡＮＤゴール「 !,Rest 」の生成。	Varを用いた選択点のカット。
icut_last Var	本体部最終尾のカットゴール「 ! 」の生成。	Varを用いた選択点のカット。

3．２　多機能命令

（１）多機能命令の生成

多機能命令の生成にはＷＡＭのコンパイル技法をそのまま適用することができる。ユニットクローズ、本体部のゴールが１つまたは２つのときの命令生成パターンを以下に示す。

```
P                    P：－Q.              P：－Q，R.

ienter_unit P        ienter P             ienter P
get args of P        get args of P        get args of P
proceed              put args of Q        put args of Q
                     iexec Q              icall Np,Q
                                          put args of R
                                          deallocate
                                          iexec R
```

（２）　実行モード

実行モードのときの多機能命令の動作はienter命令がnoopとなる他は、ＷＡＭと同様で

ある。すなわち、iexecは最終ゴールへのジャンプ命令、icallはゴールの呼び出し命令となる。クローズモードのときの多機能命令の動作は以下の通りである。例として、リストの連接を行うappendプログラムを考える。

```
append([],X,X).
append([X|Y],Z,[X|YZ]) :- append(Y,Z,YZ).
```

動的コンパイル法によるコンパイル結果は以下のようになる。ＷＡＭと全く同じ最適化が適用できる点に注意されたい。

第一クローズ

```
ienter_unit   append/3
get_nil       A0
get_value     A1,A2
```

第二クローズ

```
ienter          append/3
get_list        A0
unify_variable  X
unify_variable  A0    % Y
get_list        A2
unify_value     X
unify_variable  A2    % YZ
iexec           append/3
```

（３）クローズモード

クローズモードにおける多機能命令の動作は以下の通りである。ヘッドゴールの生成は（ＷＡＭの）get命令がヘッドゴールの引数に関する情報を全て含んでいることを利用する。すなわち、呼び出し側の引数を全て変数にしてget命令を実行するとヘッドゴールの引数のソースイメージが自動的にユニファイされる。したがって、ヘッドゴールの生成はienter_unit命令およびienter命令において述語名をファンクタに持つ構造体を生成し、構造体の各変数を引数レジスタにロードすれば良い。後の引数の構造体イメージの生成は、get命令を実行することにより達成される。

本体部ゴールの生成もput命令が本体部ゴールの引数イメージを引数レジスタにセットすることを利用する。すなわち、put命令において引数イメージがセットされているので、icall命令はＡＮＤゴール（','/2）をファンクタとする構造体の第一引数として、述語名をファンクタとし、引数レジスタの内容を要素とする構造体をユニファイすればよい。また、ＡＮＤゴールの第二引数には以下に続く命令により生成されたＡＮＤゴールおよびiexecute命令で生成された最終ゴールがユニファイされる。

4.　性能評価

動的コンパイル法ならびに多機能命令の効果を測定するために、ＡＣＯＳシステム１０００上にＷＡＭをベースとしたPrologマシンエミュレータを作成し、種々のインタプリタ方式（コード／ソース分離方式、構造体格納方式）ならびに手続きコンパイル方式との比較を行った。

（１）実行性能

実行性能を比較するための評価用プログラムとしては、Prologコンテスト[5]から以下の

ベンチマークプログラムを採用した。評価結果を表２に示す。

ＲＥＶ３０　　３０要素のリストの逆転
ＳＯＲＴ５０　５０要素のリストのソート
ＤＢ－１　　２００要素のデータベース検索
ＦＩＢ　　フィボナッチ数列

動的コンパイル方式（コード／ソース分離方式、多機能命令方式）は手続きコンパイル方式の約３０～８０％の処理性能を達成している。ユニフィケーション命令は共通なので、動的コンパイル方式の性能劣化の要因としては、①インデクシング方式の違い、②組込述語の呼び出し方式の違い、③多機能命令のオーバーヘッドの３点が考えられる。特に、ＲＥＶ３０、ＤＢ－１のように引数のＷＡＭのインデクシング命令（switch_on_term,switch_on_constant）が効率よく働くときは性能差が大きいが、複雑な処理を行うような述語においては手続きコンパイル法と動的コンパイル法との性能差はあまりないといってよい。

また、コード／ソース分離方式と多機能命令方式の差が小さいことは、多機能命令の実行時のモードチェックのオーバーヘッドが全体の処理時間に比べて小さいことを示している。本性能評価はPrologマシンエミュレータ上で実験したので、手続きコンパイルで直接ネイティブコードまでコンパイルしたときとの比較はできないが、中間言語方式でのProlog処理系ならびにPrologマシンにおいてはインタプリティブコードでも高速実行が可能なことを示している。

表２　手続きコンパイル方式を１００としたときの性能比率

方式	Rev30	Sort50	DB-1	Fib
手続きコンパイル方式	100	100	100	100
コード／ソース分離方式	35.2	84.2	26.0	69.7
多機能命令方式	35.8	85.0	26.8	70.3

表３　メモリ消費量の比較　（内　相対指標）

方式	ヒープメモリ消費量
レコード方式	３０３７２語　（１００）
コード／ソース 分離方式	６１３１１語　（２０２）
多機能命令方式	３６００８語　（１１９）

（２）メモリ消費量の比較

メモリ消費量の測定においては、Prologで記述されたエキスパートシステム（約８０００行）をインタプリタを用いてロード（consult）し、ヒープの消費量を測定した。ヒープ消費量の比率を表３に示す。表３の通り、コード／ソース分離方式では、レコード方式に比べ２倍のメモリを消費するのに対し、多機能命令方式では約２０％のメモリ消費量増加で済んでいる。さらに、生成命令数で比較すると、多機能命令方式はコード／ソース分離方式の約４５％であり、多機能命令の効果が大きいことを示している。

また、現在のPrologマシンエミュレータはワードコード方式を採用しているが、バイトコードを用いたときには多機能命令方式のメモリ消費量をさらに減らせると予想される。

5.　終わりに

動的コンパイル法、多機能命令を備えたPrologインタプリタについてＡＣＯＳ１０００上に作成したPrologマシンエミュレータを用いて評価し、その有効性を確認した。動的コンパイルの考え方はPrologインタプリタに限らず、各種の言語処理系にも応用できると思われる。従来、コンパイラがサポートされている言語処理系ではインタプリタはデバッグ専用という意識が強かったが、インタプリタの持つ柔軟性は知識処理においては魅力的であり、インタプリタの高速化は十分価値があるといえよう。

また、動的コンパイル方式で問題となるソースイメージの管理方式として、同一命令シークエンスを命令実行とソースイメージの取り出しの両方に用いる多機能命令方式を導入した。このような方式が実現できた理由の一つにはPrologのユニフィケーションという特殊性があるが、一つの命令に複数の意味を持たす多機能命令の考え方は他にも応用できそうである。

謝辞

本論文の執筆にあたって、本研究の機会を与えてくださった日本電気（株）の大野部長、梅村課長、実験システムの作成、評価データの収集をして頂いた神戸日本電気ソフトウエア（株）の阪野氏ならびに本研究に関する討論に参加して頂いた方々に感謝の意を表します。

参考文献

[1] Warren D.H.,"An Abstract Prolog Instruction Set", Technical Note 309, SRI International, October 1983

[2] 幅田，中崎，新，小長谷，梅村：逐次型推論マシン：ＣＨＩ，情報処理学会記号処理研究会，ＳＹＭ４２－１，１９８７

[3] Clocksin W.F.,"Implementation Techniques for Prolog Databases", Software Practice and Experience, vol.15, no.7, July 1985, pp.669-675.

[4] Clocksin W.F.,"Design and Simulation of a Sequenctial Prolog Machine", New Generation Computing, vol.3, 1985, pp.101-120.

[5] Okuno H.,"The Report of the Third Lisp and the First Prolog Contest", 情報処理学会記号処理研究会，ＳＹＭ３３－４，１９８５