並列論理型言語 KL 1 のクローズインデキシング方式†

木村康則†† 近山 隆††

本論文では，並列論理型言語 KL 1 のクローズインデキシング方式を提案し，評価結果を報告する．KL 1 では，ゴールの実行のために試みる候補クローズの選択の順は言語では規定されておらず，コンパイラあるいは処理系で自由に決めてよい．そこで，本論文では，コンパイル時に個々のクローズが選択されるための条件を求め，選択される可能性のあるクローズ群をまとめてコンパイルしてオブジェクトコードを生成するクローズインデキシング方式を提案する．本方式では，引数のデリファレンス，ヘッド引数として現れた構造体の分解や，組込述語などクローズ間で共通した処理は，重複して実行されないようにコンパイルされる．また，クローズインデキシングに用いる命令の設計方針，命令の概要について説明する．次に，本方式の効果を調べるために汎用計算機上に KL 1 のエミュレータを作成して行った性能評価の結果を報告する．その結果，本方式は，クローズヘッドの引数の構造が複雑であるほど，サスペンションの回数が多いほど，候補クローズの数が多いほど，効果が大きいことがわかった．具体的には，クローズインデキシングを行わない場合と比較して，本方式では，静的なコード量は増大せず，命令実行数，実行時間も 1 割から 4 ～ 5 倍向上する．さらに，クローズインデキシングを行うことにより，ガード部の実行時の動的な命令分岐の数を 5 割近く減らせる場合があることがわかった．動的な命令分岐数の減少は，命令先取りバッファや，命令パイプライン機構をもったマシンにおいて命令ストリームの乱れを減らす効果をもたらすと考えられる．したがってこの結果は，KL 1 をこのような機構を備えたマシン上に実現する上で極めて好都合な結果である．さらに，本方式により，プログラマは，クローズの順に関して処理系の詳細を知らなくても，クローズを最適に並べた場合以上の速度を得られることを示した．

1. はじめに

近年，人工知能等の分野における情報処理量の増大に伴い，並列計算機や並列言語の研究開発が活発になっている．ICOT では，第五世代コンピュータプロジェクトの一環として，並列論理型言語 KL 1[2] を実行する並列推論マシン PIM[14] の研究開発を進めている．

KL 1 は，フラット GHC[12] に基づいて ICOT で設計されたコミッテドチョイス型言語 (Committed Choice Language, 以下 CCL) の一つで，シンタックスの単純さ，並列を自然に記述できることなどに特徴を持つ言語である．

KL 1 を PIM 上で効率的に実現するためには，言語とマシンの接点とも言える機械語命令セットを KL 1 の実行特性に合致し，PIM のハードウェアを生かすように設計，最適化することが必要である．KL 1 の基となった GHC の処理系については，既にソースプログラムを Prolog や Lisp のプログラムに変換して各々の処理系上で動作するものが提案されている[16),18]．しかし，これらは，Prolog や Lisp コンパイラによる最適化を期待しており，機械語レベルでの最適化は考えられていない．CCL の一つの FCP (Flat Concurrent Prolog) については，提案がある[7]が，この提案でも，命令セットの最適化などについては述べられていない．

本論文では，KL 1 の高速実行を目指した最適化の一方式であるクローズインデキシング手法について提案する[22]．クローズインデキシングによる最適化については，論理型言語 Prolog において多く行われているが[1),13]，そのままでは KL 1 に適用できない．なぜなら，クローズ選択の試みにおいて，Prolog では順序性があるのに対し KL 1 ではないこと，ヘッドユニフィケーションにおいて，KL 1 では方向性があるのに対し Prolog ではないこと，などの違いがあるからである．

次章以下では，まず，クローズインデキシングの基本的な考えかたを述べ，次にコンパイラでの処理方式について説明する．またインデキシングのための命令について，その設計方針と機能を説明する．さらに，PIM 上での実装に先立ち汎用計算機上に構築したエミュレータを使って行った評価結果について報告し，考察を加える．

2. KL 1 のクローズインデキシング

2.1 並列論理型言語 KL 1

並列論理型言語 KL 1 は，フラット GHC に基づいて ICOT で設計された言語である．KL 1 プログラム

† Clause Indexing Scheme for Parallel Logic Programming Language KL 1 by YASUNORI KIMURA and TAKASHI CHIKAYAMA (The Fourth Laboratory, ICOT Research Center Institute for New Generation Computer Technology).

†† (財)新世代コンピュータ技術開発機構研究所第四研究室

は，次のようなシンタックスを持つクローズ（または節）の集合として表される．

$H:-G_1,\cdots,G_m|B_1,\cdots,B_n.\quad(m\geq 0,n\geq 0)$

ここで，H, G_i, B_i は，各々，クローズヘッド，ガードゴール，ボディゴールと呼ばれる．'|' は，コミットメントオペレータと呼ばれ，クローズ中でこれに先立つ部分を受動部（またはガード部），これに続く部分を能動部（またはボディ部）と呼ぶ．ここで，受動部には組込述語しか書けない．これは，GHC の言語記述能力を保ちながら，効率的な実現を考慮して採用された制限である．

ゴールの実行は，ゴールと同じ述語名，引数個数のクローズ群の中からガード部の実行が成功したクローズを一つ選択し，そのクローズのボディゴールを実行するという操作の繰り返しで行われる．すべてのゴールの実行が成功すると実行の終了である．ただし，ガード部の実行において，ゴール側（呼び出し側）の変数を具体化しようとすると，その実行は中断させられ，そのクローズの選択の試みは棄てられる．そして，別のクローズ（候補クローズ，またはオルタナティブクローズと呼ぶ）に対して同じ操作を繰り返す．もし，すべての候補クローズの選択の試みに失敗すると，そのゴールの実行は中断（サスペンションと呼ぶ）し，他ゴールの実行などによってゴール変数が具体化されることにより，実行可能になるまで待たされる．

2.2 従来のコンパイル方式

KL 1 では，Prolog と違ってクローズ選択の試みを行う順は言語としては決めていない．したがって KL 1 コンパイラは，読み込んだクローズをその順で一つ一つ独立にコンパイルして KL 1-B[6),20)] と呼ばれる KL 1 の抽象機械語命令列を生成すればよい．一つの述語に対する命令列の最後には，実行が中断した時の処理を行う命令が置かれる．KL 1-B は，Prolog における WAM (Warren's Abstract Machine)[13)] の命令セットに相当するもので，KL 1 の実行を規定する抽象命令セットである．WAM は，D. H. D. Warren が Prolog の逐次実行モデルとして提案した抽象マシンである．これが従来用いられてきた最も素直なコンパイル方式である．例えば，**図 1** に示したプログラムは，**図 2** のようにコンパイルされる．

この方式では，コンパイラは対象となったクローズのみに注目すれば良いため，コンパイル時間が短く，かつコンパイラの作成が簡単であるという利点がある．一方で，クローズ間にわたった情報を考慮にいれていないため，実行時に次のような無駄な実行が行われる．

```
filter([X|Xs1], P, Ys0) :- X mod P =\= 0 |
                           Ys0 = [X|Ys1],
                           filter(Xs1, P, Ys1).    (1)
filter([X|Xs1], P, Ys0) :- X mod P =:= 0 |
                           filter(Xs1, P, Ys0).    (2)
filter([],      P, Ys0) :- true | Ys0 = [].        (3)
```

図 1 フィルタソースプログラム
Fig. 1 KL 1 program of 'filter.'

```
filter/3:              /* クローズ(1) のコード */
    try_me_else  filter/3/1
    wait_list  A1      /* この命令から7命令が  */
    read_car  A1,  X4  /* クローズ(2) のコード */
    read_cdr  A1,  X5  /* と機能的に同じ       */
    integer  X4
    integer  A2
    modulo  X4, A2, X6
    put_constant  0,  X7
    not_equal  X6,  X7      /* X mod P =\= 0  */
    collect_list  A1
    put_list  A1
    write_car_value  A1,  X4
    write_cdr_variable  A1,  X4
    get_list_value  A1, A3
    put_value  X5,  A1
    put_value  X4,  A3
    execute  filter/3
filter/3/1:            /* クローズ(2) のコード */
    try_me_else  filter/3/2
    wait_list  A1
    read_car  A1,  X4
    read_cdr  A1,  X5
    integer  X4
    integer  A2
    modulo  X4,  A2,  X4   /* X mod P =:= 0 */
    put_constant  0,  X6
    equal  X4,  X6
    collect_list  A1
    put_value  X5,  A1
    execute  filter/3
filter/3/2:            /* クローズ(3) のコード */
    try_me_else  filter/3/3
    wait_constant  □,  A1
   collect_value  A2
    get_constant  □,  A3
    proceed
filter/3/3:         /* サスペンション */
    suspend  filter/3
```

図 2 フィルタコンパイル結果（従来方式）
Fig. 2 Compiled code of 'filter' without indexing option.

- クローズの読み込まれた順で，(i) 番目のクローズの選択の試みが失敗したことにより，($i+j$) 番目 ($j>0$) のクローズの選択が失敗すると分かってしまう場合でも，($i+j$) 番目のクローズの選択の試みが行われてしまう．例えば，図 1 では，ゴールの第一引数がリスト以外のものでも，クローズ(1)，(2)，(3)の順で実行が試みられる．

- (i) 番目のクローズの選択の試みが失敗し，$(i+j)$ 番目のクローズの選択の試みを行う時，(i) 番目のクローズの選択の試みの時に行われた両クローズに共通する処理，例えば，変数のデリファレンスや，ヘッド引数の構造体の分解など，が重複して行われてしまう．例えば，図1では，クローズ(1)の 'X mod P=\=0' で失敗して，クローズ(2)の実行が行われる時，再び第一引数のリストの分解や，'X mod P' の計算が行われる．ここで，変数のデリファレンスとは，その値（未定義の場合も含む）が幾つかのポインタを経由して指されている場合に，そのポインタを手繰って値を求めることである．

2.3 クローズインデキシングの方針

本論文で提案するクローズインデキシング方式では，同じ述語名，引数をもつクローズ群を一纏めにして扱い，各クローズが選択されるためのヘッド引数の条件およびガード部の組込述語を調べて，クローズ間にわたった情報を利用してコンパイルする．具体的には以下のような方針でクローズインデキシングを行う．これは2.2節で述べた欠点を除くものである．

- 実行するゴールに対して，選択される可能性のあるクローズ群のみの選択の試みを行うようにする．この結果，実行が中断するゴールに関しては，実行の中断が早く検出できる．
- 選択される可能性のあるクローズ群のガード部に共通の処理は重複して行わないようにする．ガード部に共通の処理とは以下のような処理である．
  - 変数のデリファレンスおよび具体化済かどうかのチェック．
  - データタイプのチェック．
  - 値一致のチェック．
  - ヘッド引数として現れる構造体の分解．
  - ガード部の組込述語．

また，以下のような方針でインデキシングの対象とする引数を選ぶことにする．

1. ヘッドおよびガード部のユニフィケーションとガード部の組込述語を対象とし，各クローズが一意に決まるまでヘッドの引数，ガード部組込述語の順でインデキシングの対象を拡げていく．
2. 引数のデータタイプは，整数，アトム，リスト，ベクタ，ストリング，'タイプ付変数'，通常の変数，とその他*，に分類する．ここで 'タイプ付変数' とは，'変数であるがガード部の解析により，ある決まったデータタイプの値に具体化されていなければならない変数' と定義する．
3. インデキシング対象の引数がベクタの場合には，まずその要素数で分類し，それで分類できない場合にはその要素をインデキシングの対象とする．リストの場合も同様にリスト要素をインデキシングの対象とする．これは，KL 1 ではゴール間のデータの受け渡しをリストを使ったストリームで行い，そのストリームには，ベクタによるメッセージが多く流れると予想され，リストやベクタの要素をインデキシングの対象とすることが性能向上に寄与すると期待されるためである．

* 例えば，コードへのポインタなどがある．

3. KL 1 コンパイラの処理

KL 1 コンパイラに実装したクローズインデキシングの処理手順について説明する．本方式の基本的な考え方は，Parlog[4]，FCP[9] など他の並列論理型言語にも適用可能である．KL 1 コンパイラは，まず各々のクローズが選択されるためのクローズのヘッド引数の条件を求める．次にこの条件をもとに，クローズをツリー状に展開していく．このツリーをインデキシングツリーと呼ぶ．その後，このツリーをトラバース (traverse) していきながらコード生成を行う．

3.1 ヘッド引数の条件の抽出

コンパイル対象のクローズ群に対して，各クローズが選択されるために必要なヘッド引数の条件を求める．以下，この条件を HCC (Head arguments Condition for Commitment) と呼ぶ．例えば，図1のようなプログラムに対しては図3のような HCC を求める．ここで，$\langle d_1, d_2, \cdots, d_n \rangle$ で一つのクローズの HCC を表し，d_i が引数位置 i の条件である．この時，2.3節で述べたタイプ付変数については，そのタイプも HCC に反映させる．

図1の例では，'var (int)' は変数が整数に具体化されていなければならないこと，'[…]' はリストに具体化されていること，'atom([])' は，アトムの [] が具体化されていること，'any' は，何でも良いことを

```
(1) : < [var(int)|any], var(int), any >
(2) : < [var(int)|any], var(int), any >
(3) : < atom([]),       any,      any >
```

図3 フィルタプログラムの HCC
Fig. 3 HCC for 'filter' program.

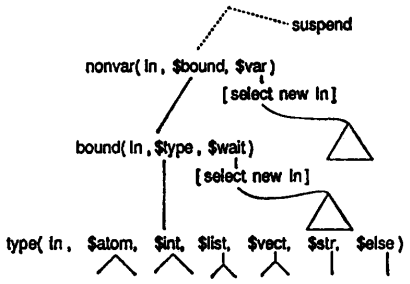

図 4 インデキシングツリーの基本形
Fig. 4 Basic structure of Indexing Tree.

示す.

3.2 インデキシングツリーの作成

3.2.1 基本アルゴリズム

HCC をもとに，各ノードが引数の条件による分岐，リーフ(葉)が分類されたクローズに対応するツリーを作る．以下に，第 *In* 番目の引数からインデキシングを行う場合の基本アルゴリズムを示す．この時，第 *In* 引数は引数レジスタの第 *In* 番目に，あらかじめ置かれていると仮定している．これは，WAM[13] と同様に KL 1-B 命令セットでは，ユニフィケーションはレジスタ上のデータに対して行われると考えているからである[6]．また，**図 4** にその場合のインデキシングツリーの基本形を示す．図 4 に現れる三角形は，与えられた *In* に対し，このアルゴリズムを適用することによって作られるサブツリーを表す．

1. まず，各クローズから求めた HCC の d_{In} を調べ，'any' とそれ以外の二つにクローズを分類する．'nonvar (In, $ bound, $var)' でこのノードを示す．ここで *In* はインデキシング対象の引数，$ bound, $ var は各々サブツリーのノードへのポインタである．$ bound は，d_{In} が 'any' 以外のクローズに対してインデキシングを行った結果のツリーへのポインタが入る．$ var には，この引数では分類できず，別の引数でインデキシングを行った結果のツリーへのポインタが入る．サブツリー $ bound を作成するために 2 に進み，$ var を作成するために 3 に進む．
2. 第 *In* 番目の引数をデリファレンスする命令を生成するためのノードを作る．図 4 では，'bound (In, $ type, $ wait)' で示す．ここで $ type は，*In* 番目の引数のタイプと値を調べるためのサブツリーへのポインタ，$ wait は，*In* 番目の引数がガード組込述語 'wait/1' の引数として現れているクローズを分類した結果のツリーである．ガード組込述語 'wait/1' は，その引数が具体化されていれば，そのタイプ，値に係わりなく成功するため，これら 'wait/1' を含むクローズ群は，第 *In* 番目の引数が $ type で分類できなかった場合のオルタナティブ（次候補）節となる．したがって，これらのクローズ群について，新たなインデキシング引数によって分類した結果のサブツリーを $ wait とする．サブツリー $ type を作成するためには 4 へ進み，$ wait を作成するためには 3 へ進む．
3. 新しいインデキシング対象の引数を決め（決め方については後述），これを新たな *In* として 1 から繰り返す．この時，古い *In* に対しては，インデキシング処理が終わったことを示すために，HCC にその旨書いて置く（例えば，d_{In} を DONE (d_{In}) などと書き換える）．新しいインデキシング対象の引数がない場合には，5 へ進み，組込述語による分類を試みる．
4. 引数のタイプを調べるためのノードを作る．'type (In, $ atom, $ int, $ list, $vect, $ str, $ else)' で示される．それぞれのタイプに一つしかクローズがなければ，その枝に関してはこれでおわり．複数個ある場合には，個々のタイプ別に以下に示す処理を行う．
   - 4-1. $ atom, $ int, $ str サブツリーは，各々値の一致を調べるためのツリーと 'タイプ付変数' を処理するためのツリーから構成される．**図 5** に，$ atom サブツリーの構造を示す．値が同じ，あるいは 'タイプ付変数' を持つクローズが複数ある場合は，次のインデキシング対象の引数を決め，1 から再帰的に繰り返し実行し，ツリーを作成していく．
   - 4-2. $ list の場合は，まずその 'car' をレジスタに読みだし（後述），それをインデキシング対象引数と考え，1 から実行する．'car' で分類できない場合は，'cdr' をインデキシング対象とする．**図 6** に，$ list サブツリーの

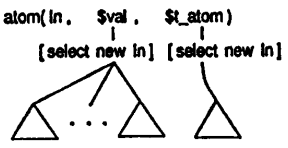

図 5 アトムサブツリーの構造
Fig. 5 Structure of 'atom' node.

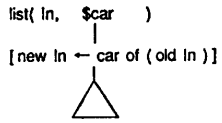

図 6 リストサブツリーの構造
Fig. 6 Structure of 'list' node.

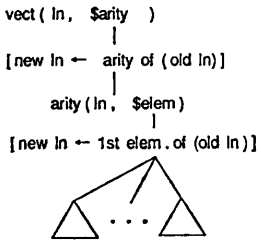

図 7 ベクタサブツリーの構造
Fig. 7 Structure of 'vect' node.

構造を示す.

4-3. $vect の場合は，まずその要素数で分類する．要素数を求め，それに基づき $int の場合と同様なツリーを作る．これで分類できない場合は，$list の場合と同様に要素を第一要素，第二要素,…の順で取り出してインデキシングの対象とする．図7に，$vect サブツリーの構造を示す.

4-4. $else の場合は，逐次に値の一致を調べるノードを作る．すなわち，インデキシングとして特に工夫しない．$else のデータは出現が極めて稀であると考えたからである.

5. 組込述語による分類では，まず解析をやりやすくするために，分類するクローズに番号をふり，その数 n，それらのクローズ中に出現するガード述語の最大個数 m からなる，n 行 m 列のテーブルを作り，対応する位置に組込述語を埋めていく．そして，(1, 1)の位置の述語から始めて，同じ述語名，引数（ゴールから渡される引数位置も同じである必要がある）を持つ述語を探す．背反条件の述語については，背反関係にある述語のテーブルをコンパイラで持ち，これを利用する．KL 1 ではガード部に記述できる組込述語の種類は限定されているので，このような処理が可能となる．図8にテーブルの例を示す.

このような処理の結果，共通，あるいは背反関係にある述語が見つかったならば，その述語を纏めて実行するためのノードを作る．見つからなかった場合には，テーブルのインデックスを変えて探す.

6. 以上の処理によって分類できない場合には，逐一クローズを実行してみるようなコード，すなわちこれ以降の実行はインデキシングを行わない場合と同じコードを生成するようにする.

● サンプルプログラム

```
p :- a, b | ...        (1)
p :- c, d, e | ...     (2)
p :- f | ...           (3)
```

● 組込述語テーブル

	1	2	3
1	a	b	true
2	c	d	e
3	f	true	true

図 8 組込述語による分類
Fig. 8 Classification by builtin predicates.

```
p([{f,X}|foo]) :- true | ...        (1)
p([{g,X}|bar]) :- true | ...        (2)

p([{_,X}|foo], f) :- true | ...     (1)'
p([{_,X}|bar], g) :- true | ...     (2)'

p([{_,_}|foo], f, X) :- true | ...  (1)''
p([{_,_}|bar], g, X) :- true | ...  (2)''
```

図 9 インデキシング対象引数の決定方法の図
Fig. 9 Selection of argument for Indexing.

3.2.2 インデキシング対象引数の決定方法

本方式では，クローズの第一引数，第二引数の順でインデキシングの対象とする．この場合，第一引数，第二引数などに出力変数が置かれた場合にはコンパイル時間が多くかかる．このような場合には，コンパイラで効率的なインデキシングができる引数を探すといったことも考えられるが，コンパイル時間の増大を招く．一方で一般的にプログラマは，引数位置の若い所にクローズを区別するためのパターンを書くことが多いと予想されるので，このようにインデキシング対象の引数の決め方を固定的にしても効果に影響はほとんどないと考えられる.

3.2.3 構造体要素によるインデキシング

本方式によるインデキシングでは，インデキシング対象のデータは，レジスタ上に乗っていると仮定している (3.2.1 項)．したがって，構造体の要素によるインデキシングを本アルゴリズムに適用するには，その要素をレジスタに読み出す必要がある．そのために，本コンパイラでは，必要に応じて要素を空いている引数レジスタに読み出して 3.2 節のアルゴリズムを適用している．例えば，図9の(1)，(2)では，プログラ

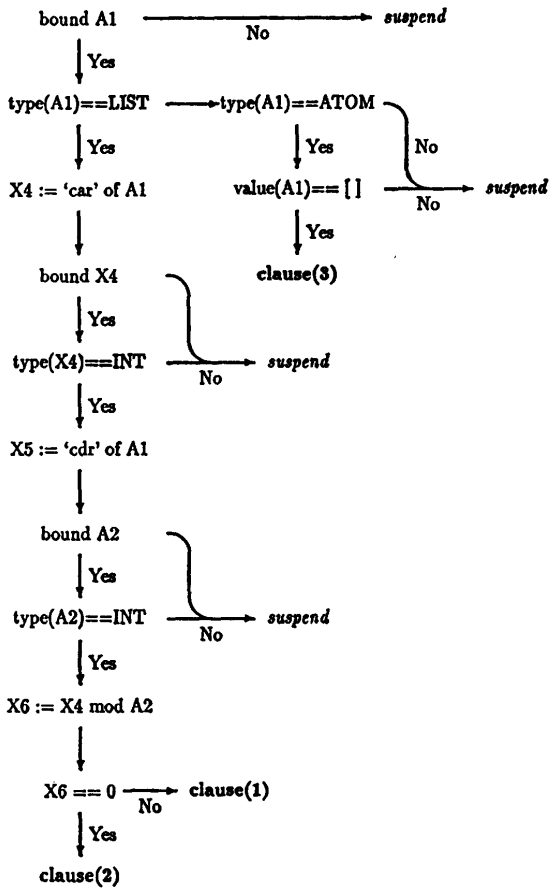

図 10 フィルタプログラムのツリー
Fig. 10 Tree for 'filter' program.

ムを意味的に (1)', (2)' のように書き換えてインデキシングをかけている．この操作は，上記のアルゴリズムの 4-2 で行われ，これに応じて HCC も同時に変更しなければならない．

この例では，第一引数がリストでその 'car' がベクタまで同じで，ベクタの第一要素によって分類されることになる．もし，ベクタの第一要素が同じだったら(例えば両方 'f' だった)，第二要素を読み出し ((1)", (2)")，処理を続行するが，両方変数 'X' で分類できないので，再びリストの 'cdr' を見て分類されることになる．

この時，構造体を一度に分解してその要素をレジスタに置いたほうが ((1)", (2)") コンパイラとしては容易であるが，作業レジスタを多く使う傾向があるため，採用しなかった．

3.2.4 フィルタプログラムの例

図 1 のプログラムのインデキシングツリーを**図 10** に示す．分かりやすくするために，図 4 から枝が一つしか出ていないノードや，対応するクローズのない枝を省略した．図で，'bound XX' とは，XX をデリファレンスし，具体化済かどうかを調べ，'type(XX)==TYPE', 'value (XX)==VAL' は，各々 XX のタイプ，値が TYPE, VAL と等しいかどうかを調べることを示す．

3.3 オブジェクトコードの生成

前節で述べたアルゴリズムに従って作ったインデキシングツリーを，ルートからたぐってコードの生成を行う．各ノードがインデキシング用の命令に対応し，リーフにおけるクローズのコンパイルでは，そこまでのツリーをたぐることによって既に生成された命令列は生成せず，ガード部の残りの部分，ボディ部のコードが生成される．これは，各ノードまでの分類で分かった引数の条件を HCC に反映させておくこと (3.2.1 項) により可能となる．また，図 5，図 6，図 7 で，[…] で囲まれた文に対しては，対応する命令を生成する．

コード生成の際，あるノードで枝が複数ある場合には，左の枝を先にたぐってコード生成を行う．例えば，'nonvar' ノードの $bound と $var の両方のツリーが実際に存在する場合には，まず $bound のツリーからコンパイルコードを生成し，$var のコードは，$bound のコードの後に生成する．ただし，'type' ノードの場合には，ノードに含まれるサブツリーを $list, $vect, $atom, $int, $str, $else の順でコード生成を行う．これは前述のように，ストリームとしてのリスト，メッセージとしてのベクタが頻繁に出現すると考えたからである．

また，あるサブツリーのコンパイルでは，実行時に，デリファレンスの結果変数が未定義であったり，値の不一致のためにそのサブツリーのコードの実行が続行できなくなった場合 ('失敗' と言う) に，別のオルタナティブクローズのコードに分岐するための分岐アドレスをコード中に置く必要がある．例えば，$bound サブツリーのオルタナティブクローズは $var であるから，$bound サブツリーのコードを実行して失敗した時には，$var サブツリーのコードの先頭に分岐するように分岐アドレスを設定しなければならない．同様に，'bound' ノードにおいて，$type のコードの分岐アドレスは $wait のコードの先頭，$wait コードの分岐アドレスは一つ上のノードの $bound の分岐アドレス，すなわち $var の先頭アドレスとなる，さらに，'atom' ノードの $val の分岐ア

ドレスは $t_atom の先頭を，$t_atom の分岐アドレスは 'bound' ノードの $wait の先頭を指す．以上のようにして，分岐アドレスを付け，実行時に失敗が起こった場合の制御を行う．

4. クローズインデキシング用の KL 1-B 命令

4.1 インデキシング命令の種類

2.3 節で述べたように，インデキシングの対象となった引数に対して，次の三つの操作がツリーの各ノードに現れてくる．

- 引数のデリファレンスと具体化の確認．
- 引数のタイプチェック．
- 値一致のチェック．

したがって，これに対応する命令を各データタイプごとに準備した．これらの種類と機能を**表1**に示す[21]．

表1で，(1)，(2)，(3)が各々デリファレンス，タイプチェック，値チェックに対応する．(4)，(5)，(6)は，(1)，(2)，(3)の命令を統合した命令である(4.2節)．また，引数が未定義のために実行が中断し，次候補クローズのコードへ分岐する場合と，値の不一致('失敗'と言う)などにより，次候補クローズのコードへ分岐する場合の分岐アドレスの指定方法を分け，前者は分岐アドレスを指定するための独立の命令である try_me_else 命令で指定し，後者は，命令オペランドの 'Lab' によって陽に指定した．このように命令に分岐アドレスを陽に持たせた理由は，インデキシングでは分岐命令が多く使われると予想され，この分岐先をすべて try_me_else 命令で設定すると，try_me_else 命令の数が増え，実行時間や静的コードサイズの増大を招くと考えたからである．例えば，命令に分岐アドレスを持たない場合には，図10の 'bound'，'type'，'value' などのノードに対応する命令の前にすべて try_me_else 命令を生成しなければならず，静的コードサイズの増大を招き，命令読みだし (fetch) の遅い計算機では，実行時間の増大を招いてしまう．

表 1 インデキシング用の命令の種類
Table 1 Instructions for clause indexing.

種類	機能	例
(1)	デリファレンス	wait Ai
(2)	タイプチェック	is_atom Ai, Lab
(3)	値一致チェック	test_constant C, Ai, Lab
(4)	(1)＋(2)＋(3)	wait_constant C, Ai
(5)	(1)＋(2)	jump_on_non_atom Ai, Lab
(6)	(2)＋(3)	check_atom C, Ai, Lab

以下にヘッド引数に定数が現れた場合を例に，インデキシング用の KL 1-B 命令について説明する[19]．

1. デリファレンス命令
   - wait Ai
     引数レジスタ Ai をデリファレンスし，未定義ならば try_me_else 命令で設定された次候補アドレスへ分岐する．それ以外の場合は次命令へ進む．
2. タイプチェック命令
   - is_atom Ai, Lfail
     デリファレンス済みの Ai のデータタイプがアトムであれば次命令へ進む．それ以外の場合は，Lfail で示される次候補アドレスへ分岐する．デリファレンス済みの引数レジスタのデータタイプを調べる命令で，他に is_integer, is_list, is_vector, is_string がある．
   - switch_on_type Ai, Latom, Lint, Llist, Lvect, Lstr, Lfail
     デリファレンス済みの Ai のデータタイプに応じて，Latom, Lint, … に分岐する．どのタイプでもなければ Lfail に分岐する．この命令は，Ai の取りうるデータタイプが三種類以上ある場合のみに生成される．
3. 値チェック命令
   - test_constant Const, Ai, Lfail
     デリファレンス済みの Ai の値が Const と等しければ，次命令へ進む．それ以外の場合は，Lfail で示される次候補アドレスへ分岐する．デリファレンス済みの引数レジスタの値の一致を調べる命令である．同様な命令に，ベクタの引数個数を調べる test_arity 命令がある．
   - branch_on_constant Ai, [(C 1, L 1), (C 2, L 2), …Lfail]
     デリファレンス済みで，データタイプがアトムか整数である Ai の値に応じて，L 1，L 2，… に分岐する．C 1，C 2，… のどれにも一致しなければ，Lfail に分岐する．同様な命令に，ベクタの引数個数で多方向分岐する branch_on_arity 命令がある．

4.2 命令の統合

インデキシングツリーをたぐってコードを生成する時には，各ノードでは対応する (4.1 節で述べた) インデキシング命令を使えばよい．しかし，これらの命

令は，一命令で単純な一つの機能を実行するもので，これだけでコードを生成すると静的コードサイズが大きくなり，実行命令数も多くなる．そこでいったん命令列を生成した後に，もう一度見直して命令の統合を行う．

命令の統合にあたっては，できるだけ多くの命令を統合することを考え，またできるだけ'前'の命令同士を統合することを考える．例えば，デリファレンス，タイプチェック，値チェックの命令が並んでいる場合には，まずこの三つの命令を一つに統合することを試み，できなければデリファレンスとタイプチェック命令の統合を試みるという順で統合を行っていく．また統合の仕方は，分岐アドレスが同じかどうかで以下に示すように変わる．

1. デリファレンス＋タイプチェック＋値チェック命令
   引数レジスタの内容が，未定義の場合とタイプチェックや値チェックでの失敗の場合の分岐アドレスが同じ場合の統合命令である．例として，'wait_constant Const, Ai' がある．この場合の分岐アドレスは，先立つ try_me_else 命令で設定されているので命令には分岐アドレスを持たせる必要はない．またこの命令は，インデキシングを行わない場合に受動部のユニフィケーションのために生成される命令に一致する．表1の(4)の場合である．
2. デリファレンス＋タイプチェック命令（その1）
   引数レジスタが，未定義の場合とタイプチェックでの失敗の場合の分岐先が違う場合の統合命令である．例として，'atom Ai' や，'integer Ai' などの命令がある．分岐先は，先立つ try_me_else 命令によって設定される．
3. デリファレンス＋タイプチェック命令（その2）
   引数レジスタが，未定義の場合とタイプチェックでの失敗の場合の分岐先が違う場合の統合命令である．未定義の場合の分岐先は，先立つ try_me_else 命令で設定され，失敗の場合のそれは，命令オペランドとして与えられる．'jump_on_non_atom Ai, Lab' や，'jump_on_non_list Ai, Lab' などの命令がある．表1の(5)の場合である．
4. タイプチェック＋値チェック命令
   デリファレンス済みの引数レジスタ Ai のタイプチェックおよび値チェックを行うための統合命令である．例として，'check_constant Const, Ai, Lfail' 命令や 'check_vector Arity, Ai, Lfail' がある．表1の(6)の場合である．

表 2 ベンチマークプログラム
Table 2 Benchmark programs.

プログラム	リダクション	サスペンション	機能
prime	5876	0	500 までの素数生成
prime_dd	10747	10744	要求駆動型の素数生成
queen	38878	0	エイトクイーン
qlay	19419	0	レイヤード法[6]によるエイトクイーン
bup	34857 (ON) 35858 (OFF)	0	ボトムアップパーサ
kl1cmp	14919	51	KL1 で記述した KL1 コンパイラ
espascal	335115	0	パスカルの三角形による係数の計算(E. Tick[7] による)
etsmall	918520	131820	詰め込みパズル(同上)
tri	666235	0	パズル問題(同上)
semi	292309	1507	半群の要素の計算(同上)
pax	17530	6946	並列自然言語パーサ
bestpath	247384	58949	最短経路問題
parser	109505	22100	パーサ（マルチ PSI で使用）

4.3 コンパイル時間

インデキシングをかけてコンパイルするときの時間は，纏めてコンパイルするクローズの数，ヘッド引数のパターンの複雑さと，同じパターンの出現頻度に比例する．経験的には，クローズ数が多いほどクローズ中のヘッドパターンも複雑になり，同じパターンの出現頻度も大きくなる傾向にあるため，これらの積で時間がかかる．現在のコンパイラは，Prolog で記述しており，例えば，**表2**の 'prime' では2割程度，'qlay' ではガード部が複雑なため，3倍程度かかっている．ただし，Prolog ではテーブル類の破壊的な更新ができないため，コンパイラがインデキシング用に持つ管理テーブルの更新に時間がかかっている．もし，効率的なテーブル更新の機能をもつ言語で実現するならば，時間はかなり短縮されると予想される．

4.4 コンパイル例

図 11 に，図1のプログラムをインデキシングをかけてコンパイルした結果を示す．図1では，クローズ(1)と(2)に共通した処理，すなわち，第一引数のリストの分解，'modulo' の計算などが，1回しか行われ

```
filter/3:
    try_me_else  filter/3/1
    jump_on_non_list  A1, filter/3/3
    read_car  A1,  X4          /* この命令から7命令が */
    integer  X4                /* クローズ(1)と */
    read_cdr  A1,  X5          /* クローズ(2)に */
    integer   A2               /* 共通した処理  */
    modulo  X4,  A2,  X6
    try_me_else  filter/3/6
    put_constant  0,  X7
    not_equal  X6,  X7         /* X mod P =\= 0 */
    collect_list  A1           /* クローズ(1)に */
    put_list  X1               /* 固有のコード  */
    write_car_value  X1,  X4
    write_cdr_variable  X1,  X4
    get_list_value  X1,  A3
    put_value  X5,  A1
    put_value  X4,  A3
    execute  filter/3
filter/3/6:
    collect_list A1    /* クローズ(2)に */
    put_value X5,  A1  /* 固有のコード  */
    execute  filter/3
filter/3/3:            /* クローズ(3) */
    check_constant  □, A1, filter/3/1
    collect_value  A2
    get_constant  □,  A3
    proceed
filter/3/1:
    suspend  filter/3
```

図 11 フィルタコンパイル結果（インデキシングあり）
Fig. 11 Compiled code of 'filter' with indexing option.

ないようなコード列が生成されている．

5. 実　　験

5.1 実験処理系

実験は，UNIX マシン上に構築された KL 1 処理系である PDSS システム[5]を使って行った．PDSS システムでは，KL 1 のソースプログラムはまず KL 1-B 抽象命令にコンパイルされ，次にアセンブラによりバイトコードに変換される．そして，PDSS エミュレータのトップレベルがバイトコード列を次々に読み出し，解釈実行することにより，KL 1 プログラムが実行される．

5.2 ベンチマークプログラム

ベンチマークプログラムとして，表2に挙げたプログラムを使った[8),11),15)]．表2で，リダクションはプログラム実行に要したリダクション数で，サスペンションは，プログラム実行中に起こった実行中断（サスペンション）の回数を示す．なお，'bup' では非決定的な述語があり，インデキシング ON/OFF で選択されるクローズが変わるためにリダクション数が変わっている．

クローズインデキシングの効果を測定するために，インデキシングをかけてコンパイルした場合（以後 'ON' と略記）と，2.2 節で述べたインデキシングをかけないでコンパイルした場合（以後 'OFF' と略記）のふたつの場合を比較した．測定項目は，コンパイルされたコードの静的なサイズ，実行した KL 1-B 命令数と実行時間，である．

5.3 静的コードサイズ

表3に，表2のプログラムをインデキシング ON と OFF でコンパイルした時の KL 1-B 命令数と，バイトコードに変換したバイト数の静的コードサイズの比の平均と標準偏差を示す．インデキシング ON の場合の方が，サイズが小さくなっている．これは，ガード部を纏めてコンパイルすることにより，重複したコードが生成されていないためと考えられる．

5.4 実行命令数と実行時間

表4に実行命令数と実行時間を示す．実行時間の単

表 3 静的コードサイズ
Table 3 Static size of compiled code.

	平均	標準偏差
KL 1-B コード数比	0.92	0.042
バイト数比	0.94	0.041

表 4 実行命令数と実行時間
Table 4 Executed instructions and execution time.

	Indexing	命令数	比	時間	比
prime	ON	95267	1.02	1690	1.01
	OFF	93324		1680	
prime_dd	ON	153791	0.93	3570	0.95
	OFF	165750		3740	
queen	ON	574499	0.95	12330	0.99
	OFF	603677		12420	
qlay	ON	433201	0.65	8340	0.70
	OFF	664138		11960	
bup	ON	464622	0.79	11360	0.90
	OFF	581555		12590	
kl1cmp	ON	255743	0.95	6230	1.00
	OFF	269056		6240	
espascal	ON	4598162	0.94	108990	0.93
	OFF	4889898		117520	
etsmall	ON	17250751	0.77	667950	0.87
	OFF	22351835		772140	
tri	ON	13751029	0.96	322070	0.96
	OFF	14366902		333250	
semi	ON	4570589	0.72	217450	0.93
	OFF	6352156		233970	
pax	ON	255014	0.15	6910	0.22
	OFF	1725475		31680	
bestpath	ON	5936163	0.91	119220	0.93
	OFF	6533653		127600	
parser	ON	1848921	0.68	47110	0.79
	OFF	2712670		59690	

位はミリ秒である．また，インデキシングOFFを基準とした場合の比を示す．実行命令数，実行時間とも値の大小はあるもののインデキシングの効果がでていることがわかる．実行命令数で見ると，'qlay'，'semi'，'parser'で3割，'bup'，'etsmall'で2割，'pax'で8割減少している．その他のプログラムでは，数%の減少である．プログラムによって効果に大小があるが，この点については次章（6章）で考察を加える．

また，実行命令数の減少の割合に比べて実行時間の減少の割合が小さい傾向がある．これは，ガードの命令が実行失敗時の分岐先をオペランドとして持つことにより(4.1節)，減少した命令の多くが，try_me_else命令のような比較的'軽い'（実行時間の短い）命令であったためと考えられる．また，インデキシング用の命令には，ヘッドの引数に書かれた定数をキーにしてハッシュ値を計算し，分岐するといった'重い'命令が増えたとも考えられる．

1. 例1

```
p(1) :- true | true.
p(2) :- true | true.
          ...
p(100) :- true | true.
```

2. 例2

```
p(100,100,100,100,100,100,100,100,100,1) :- true | true.
p(100,100,100,100,100,100,100,100,100,2) :- true | true.
                  ...              ...
p(100,100,100,100,100,100,100,100,100,100) :- true | true.
```

3. 例3

```
p([[[[[[[[[[1|_]|_]|_]|_]|_]|_]|_]|_]|_]|_]) :- true | true.
p([[[[[[[[[[2|_]|_]|_]|_]|_]|_]|_]|_]|_]|_]) :- true | true.
                  ...              ...
p([[[[[[[[[[100|_]|_]|_]|_]|_]|_]|_]|_]|_]|_]) :- true | true.
```

図 12 ヘッドパターン別のプログラム
Fig. 12 KL1 sample programs with various head patterns.

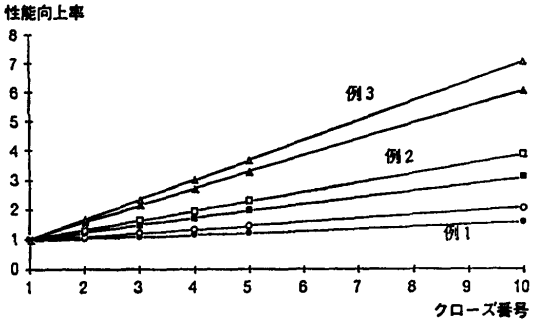

図 13 ヘッドパターン別の実行結果
Fig. 13 Execution results of the sample programs.

6. 考　　察

第5章の実験結果から，クローズインデキシングにより実行命令数が減り，速度向上にも効果があることが分かった．本章では，提案したインデキシング方式の性能特性について調べ，ベンチマークプログラムの結果について考察する[17]．具体的には，クローズのヘッドパターンの複雑さ，サスペンションの有無，候補クローズ数，ガード部での分岐回数，クローズのプログラム上での並び方などが性能向上に及ぼす影響について考察する．

6.1 クローズのヘッドパターンによる性能比較

本方式では，ヘッド引数として現れた構造体は，その引数個数，要素もインデキシングの対象としている(2.3節)．したがって，ヘッド引数に複雑な構造体が書かれている場合にインデキシングによる効果が期待される．そこで，クローズのヘッドパターンの複雑さによる性能を測定比較するために，**図 12** に示す3種類の述語を用意し，各々1番目から100番目のクローズが選択されるようなゴールを実行させて，実行時間と実行命令数を測定した．結果を**図 13** に示す．この図で横軸は選択されるクローズの番号，縦軸はインデキシング OFF の場合と比較した時の実行時間（黒い点で表示），実行命令数（白い点で表示）の割合（比）である．実行時間より，実行命令数の比のほうが変化が大きいが，これは，5.4 節で述べたのと同様の理由による．

例3，例2，例1の順でインデキシングの効果が大きい．これは，例3では，整数の比較に先だってリストを分解し，'Car' 要素をユニフィケーション用のレジスタにリストがネストしている回数だけ（例3では10回）読み出してくるという操作が必要となる．インデキシング OFF では，この操作がクローズが選択されるまでの候補節の数だけ繰り返されるのに対して，インデキシング ON では，1回で済むからである．一方，例1のプログラムでは，クローズの選択のためには引数レジスタ上の整数の比較だけでよく，インデキシング OFF の時でも，分岐回数は増えても実行命令数，時間はあまり増えないからである．図で，選択

```
filter([[I|_]|Ins], I, K, Out) :- true | ...
filter([[I|_]|Ins], I, K, Out) :- K := I-J | ...
filter([[I|_]|Ins], I, K, Out) :- K := J-I | ...
filter([[I|In]|Ins], J, K, Out) :- I =\= J | ...
```

図 14 'Qlay' プログラム (一部)
Fig. 14 KL 1 program of 'qlay' (part).

```
n_subj_zz2([],V,Out,L) :- true| Out=[].
n_subj_zz2([msg1(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg2(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg3(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg4(In,LCL)|Z],A,Out,LCU) :- ...
n_subj_zz2([msg5(In,A,LCL)|Z],B,Out,LCU) :- ...
                    :
```

図 15 'Pax' プログラム (一部)
Fig. 15 KL 1 program of 'pax' (part).

表 5 サスペンション回数と性能向上率
Table 5 The number of suspensions and its effects on the efficiency.

プログラム	リダクション	サスペンション	命令比	時間比
Prime	5876	0	1.02	1.01
Prime_dd	10747	10744	0.93	0.95

されるクローズが同じ場合の，例1と例2の比の差が次の引数をインデキシングの対象とするときのオーバヘッド，例1と例3の差が構造体から，要素を読み出すためのオーバヘッドと考えることができる．

実験の結果でインデキシングの効果の大きかった 'qlay' や，'pax' のプログラムを検討してみると，図 14 や，図 15 のようになっている．すなわち，両者ともヘッド引数のパターンが複雑なクローズが多いことが性能向上に寄与している．したがって，ヘッド引数のパターンが複雑なほどインデキシングの効果が大きいと言える．

6.2 サスペンション回数と性能向上率

2.3 節で，本方式のインデキシングではサスペンションが早く検出できることを述べた．本節では，同じ結果をもたらす，2種類のプログラムを使ってサスペンション回数と性能向上率について検討する．

'Prime_dd' は，'prime' と同じく素数生成のプログラムであるが，'prime' のように始めに求める素数までの整数のリストを作り，倍数をふるい落とすのではなく，'ふるいプロセス' からの要求に対して '整数生成プロセス' が整数を一つ作って渡すといった要求駆動型のプログラムである．したがって，このプログラムでは，実行の中断（サスペンション）が頻繁に起こる．実行結果を表 5 に示す．'Prime_dd' は，クローズ

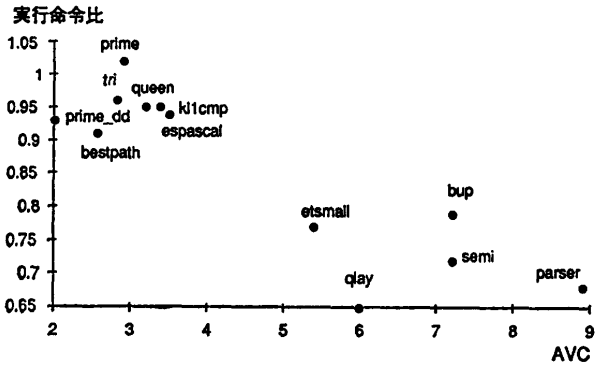

図 16 平均クローズ数と実行命令比
Fig. 16 The number of average clauses (AVC) and the ratio of executed instructions.

数が2の述語から書かれているが，インデキシングにより，サスペンションを速く検出できるために性能向上がみられる．

表2のプログラムでは，'prime_dd' のほかに，'etsmall', 'pax', 'bestpath' と 'parser' でサスペンション回数のリダクション数に対する割合が比較的大きい．これらのプログラムでは，サスペンションを早く検出できることが性能向上に貢献していると考えられる．

さらに，実際のマルチプロセッサ上では，ゴールは真に並列に実行されるため，サスペンションが起こる比率が高くなると予想される．したがって，インデキシングによる効果は，マルチプロセッサ上でより期待できる．

6.3 平均クローズ数と性能向上率

インデキシングでは，候補クローズの数が最も性能向上に寄与するものと考えられる．そこで，以下のようにして動的な平均クローズ数を求め，これと，インデキシング ON/OFF 時の実行命令数比の関係を調べた．

ベンチマークプログラムの実行において，各ゴールの呼び出し回数と，対応する述語のクローズ数から，動的な平均クローズ数（以下，AVC と呼ぶ）を求め，これを横軸，実行命令比を縦軸にとった結果を図 16 に示す．ここで，動的な平均クローズ数を以下のように定義する．

$$\text{動的な平均クローズ数 AVC} = \sum_{i=1}^{N} c_i * n_i / C$$

ここで，N：呼び出された述語の種類（個数）．
C：述語の総呼び出し回数．
c_i：述語 i の呼び出し回数．
n_i：述語 i の候補クローズ数．

このAVCは，プログラム中のすべてのゴールの実行にあたって，対応する述語が平均幾つの候補クローズから構成されていたかを示すものである．

図16では，各標本点が全体としてゆっくりとした右さがりの相関を持ち，AVCが大きいほどインデキシングの効果も大きいことがわかる．特に，'qlay'では，ヘッドの引数パターンが複雑なため，AVCの割に効果が大きく，'bup'では，ヘッドの引数パターンが単純なため，AVCの割に効果が小さい．この図から，AVCが4～5以上あるとインデキシングの効果を期待できることがわかる．'Prime_dd'や'bestpath'では，AVCが小さい割にインデキシングの効果があるが，これは，6.2節で述べたサスペンションの早期検出によるものと考えられる．一方で，'pax'では，AVCが約20にもなり，サスペンションも平均2.5リダクションに1回起こっている（表2）．このことからも特異点的にインデキシングの効果がでている理由が分かる．

6.4 ガード部での分岐回数

クローズインデキシングを行わないと，クローズの選択は逐次に行われるため，ガード部の分岐回数が増えるものと予想される．そこで，ガード部での動的な分岐回数を調べるために，ガード部で実行された分岐する可能性のある命令のうちで，実際に分岐した命令の数をインデキシングON/OFFの比で求めた．比を縦軸にとった結果を**図17**に示す．'Qlay', 'bup', 'etsmall', 'semi', 'pax'でインデキシングONの時の分岐回数がOFF時に比べて50%以下となっている．これは，クローズの選択にあたって，インデキシングOFFでは逐次的にクローズ選択の試みを行うのに対して，インデキシングONでは，ガード部の実行をまとめて行うため，クローズ選択の失敗による分岐回数が減っているものと考えられる．

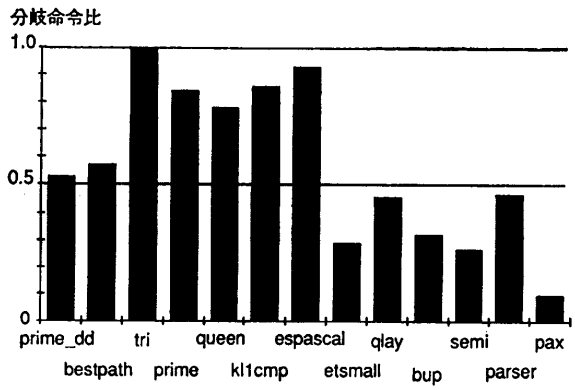

図17 ガード部での分岐回数比
Fig. 17 The ratio of instructions branched in guard execution between with and without indexing option.

KL1の実行では，分岐が起こるのは本節で述べたガード部での分岐と，リダクションの切れ目で実行される命令*による分岐と，実行中断（サスペンション）である．後二者による分岐の回数はインデキシングON/OFFにかかわらず同じで，これを考慮に入れた全体の分岐回数の比でも，図17で示された傾向は変わらなかった．

この図と図16から，インデキシングON時の実行命令数の減少には，ガード部でのクローズ選択の失敗による無駄な実行の減少が大きく寄与していると言える．さらに，この結果は，命令先取りバッファ機構や命令パイプライン機構をもったマシンにおいて，命令ストリームの乱れを減らす効果をもたらすと考えられる．したがって，このような機構を持ったマシンでは，さらにインデキシングによる性能向上が期待できる．

6.5 クローズ並べ替えとインデキシングの比較

クローズインデキシングをかけてコンパイルすると，ソースプログラムに書かれたクローズの順はコンパイルコードに反映しない．一方，頻繁に選択されるクローズをソースプログラム中で始めの位置に書くということが実際のプログラム開発でしばしば行われる．そこで，候補クローズの順とインデキシングの関係を調べるために，次のような実験を行った．

まず，表2のプログラムを実行させて，各々のゴールの実行で選択されたクローズを記録しておく．次にこの統計をもとに，選択された回数の多い順にクローズを並べかえ，新しいプログラムを作る．そして，この新しいプログラムをインデキシングなしでコンパイルし実行した場合と，インデキシングをかけた場合で，実行命令数の比較を行った．結果を**図18**に示す．図18では，元々のプログラムをインデキシングなしでコンパイル，実行した場合の実行命令数を1としている．'クローズ並べ替え'が，新しいプログラムの実行命令数の割合，'インデキシング'がインデキシングONの場合の実行命令の割合である．

図18より，候補クローズの数が多いほど'クローズ並べ替え'による効果が大きいことがわかる．これは，候補クローズの数が2，3個程度であれば，プログラマは，容易に頻繁に選択されるクローズを予想で

* 'Execute', 'proceed'命令などである．

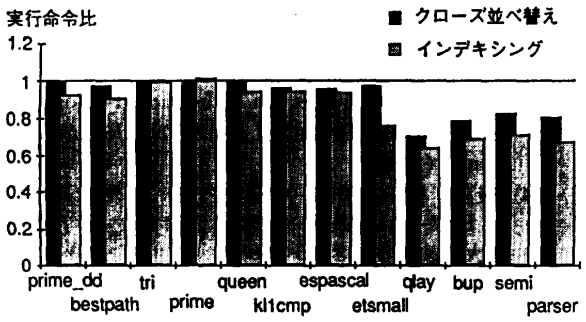

図 18 クローズ並べ替えとインデキシングの比較
Fig. 18 The ratio of executed instructions between when clauses are optimally sorted and are compiled with indexing option.

きること，そして万一，最後のクローズばかりが選択されても失敗の回数は高々数回でそのオーバヘッドも小さいことが理由として考えられる．

また，'prime' を除くすべてのプログラムで，インデキシング ON の場合の方が，クローズを並べ替えた場合よりも実行命令数が減っていることがわかる．実行時間で測定しても，両者の差は縮むものの同様の傾向を示した．これは，呼び出し回数の多い順にクローズを並べかえても，インデキシングなしでは，第二候補節以降が選択される場合には，選択の試みが逐次に行われるのに対して，インデキシング ON ではこれが最小限のコストでできるようにコンパイルされているからである．

この結果から，プログラマは，クローズの順に関して処理系の詳細を知らなくても，クローズインデキシングにより，最適の順にクローズを並べた時以上の性能を得ることができることがわかる．この結果は，個々のクローズが選択される頻度をあらかじめ予想できない場合，例えば，ゴール間にストリームを張り，そこを流れるメッセージによって処理を変えるような場合で，メッセージの種類が入力されたデータに依存するような場合，に特に効果的である．

6.6 本方式の他言語への適用可能性

本論文で提案したクローズインデキシング方式は，クローズの並びが意味を持たず，コンパイラでその順を変更して良いこと，ガード部でのユニフィケーションでは呼び出し側の変数を具体化することがないこと，ガード部には組込述語のみが許され，実行において多環境が不要であること，という KL 1 の特徴を利用してガード部の実行の最適化をねらったものである．したがってこのような条件を満たす言語であれば，本方式を適用することは基本的には可能である．

例えば，並列論理型言語として KL 1 以外に提案されている FCP[9] や Flat Parlog[3),4)] などでも本方式は適用できると考えられる．さらにこれらの言語の中には，ゴール引数変数の入出力を示すモード宣言を書かせるものがあり，この情報を利用すれば，さらに最適化が可能になると期待される．

一方で，Prolog のような逐次実行の論理型言語では，クローズの並びが意味を持ち，また呼び出し側の変数を具体化することがあるので，このままでは本方式を適用することはできない．

7. おわりに

クローズを一纏めにしてコンパイルし，選択される可能性のあるクローズのみの実行を試み，さらに余分なガード部での実行を行わないようにすることで，KL 1 の高速実行を目指すクローズインデキシング方式を提案した．実験評価の結果，以下のことが分かった．

1. ヘッド引数のパターンが複雑なほど (6.1 節)，サスペンション回数が多いほど (6.2 節)，動的な平均クローズ数が多いほど (6.3 節)，効果が大きい．特に 'pax' では，これらの条件をすべて満たすため，約5倍の速度向上が得られている．
2. 本方式でコンパイルすることにより，実行時の分岐回数を 50% 程度減らせる場合があることを示した (6.4 節)．これは，KL 1 を命令先取りバッファや，パイプライン機構を備えたマシン上に実現する上で極めて好都合な結果である．
3. クローズインデキシングにより，プログラマは，クローズの順に関して処理系の詳細を知らなくても，クローズを最適に並べた場合以上の速度を得られることを示した (6.5 節)．これは，個々のクローズの選択される頻度があらかじめ予想できない場合に効果的で，プログラムの生産効率を上げるのに役立つ．

本論文では，汎用計算機上のエミュレータ (PDSS 処理系) を使って評価を行った．エミュレータ方式では，命令の読み出し時間が実際の計算機に比べて比較的大きいため，命令の統合といった処理を行った．筆者らは現在，命令先取りバッファ，パイプライン機構を備えたマシン PIM/p[10] の開発を進めている．本クローズインデキシング方式を PIM/p に実現するためには，ターゲットマシンの特性を生かすように命令

セットを変更する必要が出てくるであろう．KL 1 コンパイラは，コンパイラ本体とインデキシング処理部を，独立性の高いプログラムとして作っているため，このような要求に対処することは容易である．

謝辞 日頃御指導いただく ICOT 第4研究室内田俊一室長（現 ICOT 研究部長）に感謝します．また，PDSS 処理系を使って実験を行ってくれた，富士通 SSL の西崎慎一郎氏，平野喜芳氏（現 ICOT），貴重なコメントを頂いた ICOT 第4研究室，後藤厚宏氏（現 NTT），中島克人氏（現 三菱電機）をはじめとする ICOT および関連メーカの方々，PIM/MPSI ワーキンググループ，PIM/MPSI 開発グループ，並列処理検討会のメンバの方々に感謝します．

参 考 文 献

1) Carlsson, M.: Freeze, Indexing, and Other Implementation Issues in the WAM, *Proc. Int. Conf. on Logic Prog. '87*, pp. 40-58 (1987).
2) Chikayama, T., Sato, H. and Miyazaki, T.: Overview of the Parallel Inference Machine Operating System (PIMOS), *Proc. Int. Conf. on FGCS '88*, pp. 230-251 (1988).
3) Clark, K. and Gregory, S.: PARLOG: Parallel Programming in Logic, *ACM Trans. Prog. Lang. Syst.*, Vol. 8, No. 1, pp. 1-49 (1986).
4) Gregory, S.: *Parallel Logic Programming in PARLOG*, Addison-Wesley (1987).
5) ICOT 第四研究室：PDSS—言語仕様と使用手引き—，TM-437, ICOT (1988).
6) Kimura, Y. and Chikayama, T.: An Abstract KL 1 Machine and Its Instruction Set, *Proc. Symp. on Logic Prog. '87*, pp. 468-477 (1987).
7) Kliger, S. and Shapiro, E.: A Decision Tree Compilation Algorithm for FCP (|, :, ?), *Proc. Int. Conf. on Logic Prog. '88*, pp. 1315-1336 (1988).
8) Okumura, A. and Matsumoto, Y.: Parallel Programming with Layered Streams, *Proc. Symp. on Logic Prog. '87*, pp. 224-231 (1987).
9) Shapiro, E.: Concurrent Prolog: A Progress Report, *IEEE Comput.*, Vol. 19, No. 8, pp. 44-58 (1986).
10) Shinogi, T., Kumon, K., Hattori, A., Goto, A., Kimura, Y. and Chikayama, T.: Macro-call Instruction for the Efficient KL 1 Implementation on PIM, *Proc. Int. Conf. on FGCS '88*, pp. 953-961 (Nov. 1988).
11) Tick, E.: Performance of Parallel Logic Programming Architectures, TR-421, ICOT (Sept. 1988).
12) Ueda, K.: Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, TR-208, ICOT (1986).
13) Warren, D. H. D.: An Abstract Prolog Instruction Set, Technical Note 309, SRI International (1983).
14) 後藤，杉江，服部，伊藤，内田：並列推論マシン PIM—中期構想—，第33回情報処理学会全国大会論文集，3B-5 (1986).
15) 寿崎，佐藤，杉村，赤坂，瀧，山崎，弘田：マルチ PSI における並列構文解析プログラム PAX の実現および評価，並列処理シンポジウム JSPP '89 予稿集，pp. 343-350 (1989).
16) 上田，竹内：GHC プログラミング講習会資料 (1986).
17) 西崎，平野，武井，森田，木村：KL 1 クローズインデキシング方式の評価，第38回情報処理学会全国大会論文集，6Q-7 (1989).
18) 藤村，栗原，加地：LISP 上の GHC コンパイラ，情報処理学会論文誌，Vol. 28, No. 7, pp. 776-785 (1987).
19) 木村，関田，近山：KL 1 におけるコード生成の最適化，第36回情報処理学会全国大会論文集，pp. 815-816 (1988).
20) 木村，近山，久門，中島(浩)：並列推論マシン PIM—KL 1 の抽象命令仕様とコンパイラー，第34回情報処理学会全国大会論文集，2P-1 (1987).
21) 木村，後藤，中島(克)，近山：KL 1 抽象命令セットの改良について，第38回情報処理学会全国大会論文集，6Q-5 (1989).
22) 木村，西崎，中越，平野，近山：KL 1 のクローズインデキシング方式，並列処理シンポジウム JSPP '89 予稿集，pp. 187-194 (1989).

(平成元年 6 月 14 日受付)
(平成2年 12 月 18 日採録)

木村 康則（正会員）

昭和31年生．昭和54年名古屋工業大学工学部電子工学科卒業．昭和56年東京工業大学大学院修士課程修了．同年(株)富士通研究所入社．Lisp マシン ALPHA のハードウェア，ファームウェアの開発に従事．昭和60年6月(財)新世代コンピュータ技術開発機構に出向．第四研究室にて KL 1 コンパイラの開発に従事．平成元年4月(株)富士通研究所に復職，現在に至る．人工知能研究部第三研究室に所属し，並列マシン，並列言語などの開発に従事している．電子情報通信学会会員．

近山 隆（正会員）

昭和28年生．昭和52年東京大学工学部計数工学科卒業．57年同工学系大学院情報工学専門課程博士課程修了．工学博士．同年富士通(株)入社．(財)新世代コンピュータ技術開発機構に出向，現在に至る．この間，手続き型言語，関数型言語，論理型言語，オブジェクト指向言語とこれらの逐次および並列処理系，プログラミング環境，論理型言語専用計算機とそのオペレーティングシステムの研究開発に従事．