計算機アーキテクチャ　81－3
（1990．3．23）

並列論理型言語 KL1 の抽象機械語の改良

平野 喜芳[1], 後藤 厚宏[2]

1: (株) 富士通ソーシアルサイエンスラボラトリ
2: (財) 新世代コンピュータ技術開発機構

現在開発中の並列推論マシンPIMのために、抽象機械語KL1-Bの改良を行なったので報告する。改良されたKL1-Bでは、ゴールの実行方式として、引数を必要になるまでメモリ上に置いたままとし、レジスタを全て作業用として利用するメモリベース方式を採用した。これにより、(1) レジスタ数によりゴール引数の数が制限されなくなった。(2) レジスタの使用効率を良くすることができた。(3) コンテキストスイッチの処理を軽くすることができた。また、これに合わせて、いくつかの最適化を導入することにより、KL1の実行速度を高速化した。

Optimal Design of the Abstract KL1 Instruction Set

Kiyoshi HIRANO[1], Atsuhiro GOTO[2]

1: Fujitsu Social Science Laboratory Ltd.
1-6-4 Oosaki, Shinagawa-ku, Tokyo, 141, Japan
2: Institute for New Generation Computer Technology

This paper describes the design of new KL1-B: an abstract KL1 instruction set. The new KL1-B takes memory based execution scheme, i.e., goal arguments are fetched from memory to working registers only when their contents are required for unification, while previous KL1-B fetches all arguments beforehand. By this scheme, a number of goal arguments are not restricted by that of registers, so that physical registers can be used effectively. Also, cost for switching goal contexts can be minimized. In addition, several compile-time optimizations techniques are adopted in the new KL1-B.

1 はじめに

現在、ICOTでは第5世代コンピュータ・プロジェクトの一環として、並列推論マシンPIMの研究開発を進めている。このPIMのために、抽象機械語KL1-B [1] の改良を行なったので報告する。

KL1-Bは、並列論理型言語KL1を実行するための仮想的なハードウエアの機械語として定義したものであり、PrologにおけるWAM [2] コードに相当するものである。PIMではKL1プログラムを一旦このKL1-Bにコンパイルし、さらに、これを実際のPIMハードウエアで用意された機械語に変換してから実行することになっている。従って、KL1-Bは言語とハードウエアの接点となるものであり、これをどのように設計するかにより、KL1の処理効率に大きな影響を与える。

KL1-B に関しては、今までにも、MRB (Multiple Reference Bit) によるインクリメンタルGCのサポート[4]や、クローズインデキシングによる高速化等の改良[3]が行なわれてきた。しかし、Multi-PSI等の開発等の経験から、いくつかの問題点が知られるようになり、今回、PIM用KL1-Bとして、さらに改良を行なった。

2 従来のKL1-Bの問題点

Multi-PSI等で採用されている従来のKL1-Bとその処理方式には幾つかの問題点があることが分かっている。今回の改良は、これらを解決するのが主目的になるので、まず、これらの問題点の内容について説明する。

2.1 ゴール引数の数の制限

従来のKL1-Bでは、ゴール引数や作業用データを全てレジスタ上に置いて実行する処理方式 — レジスタベース方式を採用している。

このレジスタはハードウエアで提供されるものであり、数が限られている。従って、この方式の処理系では、KL1のゴール引数の数が制限を受けることになる。レジスタ個数は多くて数十個であり、処理系により違っているので、KL1でプログラムを書く場合の制約になると考えられる。

KL1の上位言語や暗黙の引数等の高機能マクロを使って記述されたプログラムを機械変換した場合には、ゴール引数の数が多くなる傾向があり、レジスタ数の制限により実行不可能になる場合が多くなっている。また、複雑なユニフィケーションがある場合には、作業用のレジスタを多く使うので、その分、引数個数の制限がきつくなる。

2.2 レジスタの使用効率

レジスタベース方式は、レジスタの使用効率が悪いということもできる。即ち、この処理方式では、たとえ使わない引数であっても、ゴール引数は全て無条件にレジスタ上に置いている。ところが、KL1プログラムでは、引数に対して何の操作もせずに、そのまま子ゴールに渡すだけの場合が多くある。例えば、以下のプログラムはウィンドウドライバの一部であるが、window/9 の引数のうち、O, P, U, S, Lの5個は何の操作も行なわれずに、子ゴール wait/11 に渡される。

```
window([get(X)|R],D,O,P,U,S,B,L,C):-
    true |
        flush(D,[get(Y,F)|ND],B,NB,C),
        wait(R,ND,O,P,U,S,NB,L,X,Y,F).
```

このように、当面使われない引数がレジスタを占有してしまうため、ユニフィケーション等の作業用としてレジスタを有効に使用することができなくなっていると考えられる。

2.3 コンテキストスイッチ処理

従来のKL1-Bでは、現在実行中のゴールの環境(引数等)は全てレジスタ上に、それ以外のゴールの環境は全てゴールレコードと呼ばれるメモリ上の構造体に置くという処理方式になっている。そのため、図1のように、サスペンド処理等のコンテキストスイッチの時には、現在の環境を全てメモリ上に書き出し、次に実行すべきゴールの環境を全てレジスタに読み込んでくる操作が毎回必要になっている。

例えば、先ほどのウインドウドライバのプログラムの場合には、実行可能であるか、サスペンド

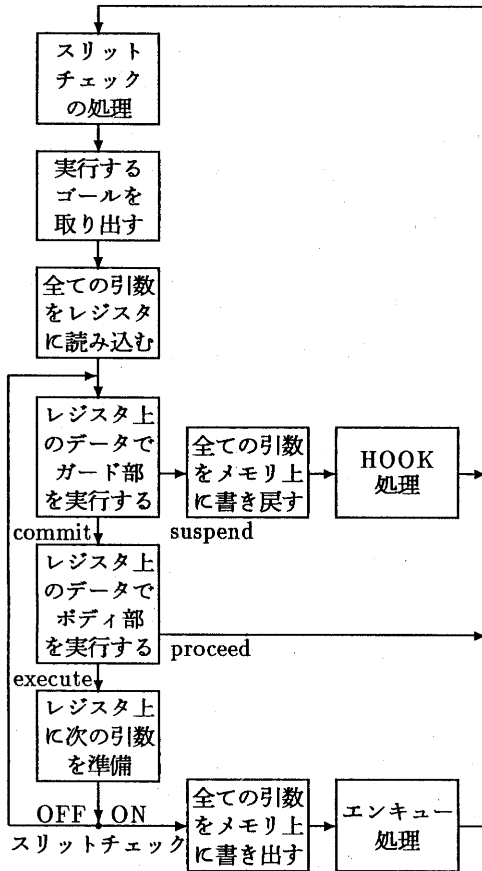

図 1: 従来の KL1-B のゴール実行方式

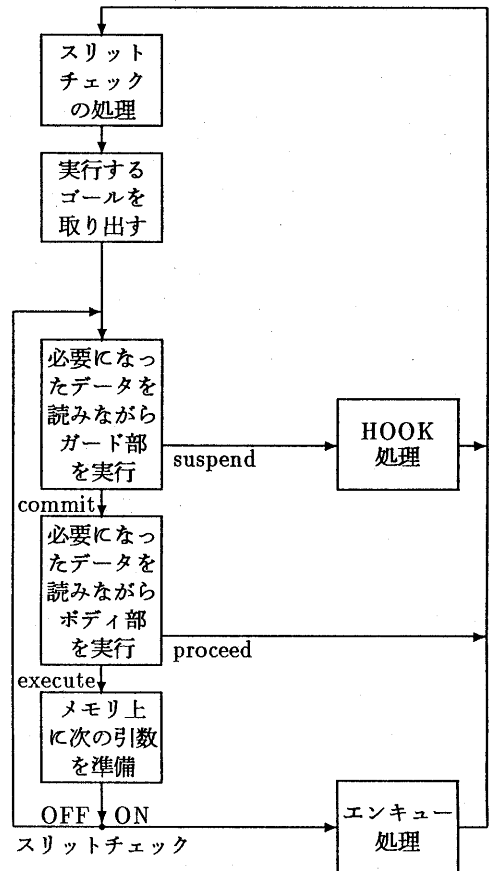

図 2: 新しい KL1-B のゴール実行方式

しなければならないかは、第1引数のみをチェックすれば分かる。しかし、従来の方式では、実行に先立ち、全ての引数を一旦レジスタ上に用意することにしており、もしサスペンドしなければならない場合には、それら全てを、メモリ上に書き戻すという処理が必要になっている。

マルチプロセッサ上で実行する並列プログラムでは、かなりの頻度でサスペンド等のコンテキストスイッチが発生すると考えられ、このコストが重い従来方式には問題があると考えられる。

3 KL1-B の改良

以上のような従来のKL1-Bの問題点を解決するために処理方式を変更し、また、さらに処理効率を上げるために最適化を進めた。

3.1 メモリベース方式

新しいKL1-Bでは、処理方式の基本として、メモリベース方式を採用した。これは、ゴールの引数環境を必要になるまでメモリ上に置いたままとし、全てのレジスタを作業用として利用する処理方式である。これにより、ゴール引数の数がレジスタ数により制限を受けなくなり、また、レジスタの有効利用とコンテキストスイッチの高速化を実現できた。

3.1.1 メモリベース方式による実行

従来の方式では、実行待ちやサスペンドしている時には、ゴールの環境をメモリ上の構造体、ゴールレコードに保持しているが、実行に先立

ち、ゴール環境をレジスタ上に読み込み、ゴールレコードは棄ててしまっていた。新しい方式では、実行中のゴールでもこのゴールレコードをそのまま利用するようにし、ゴール環境をレジスタ上になるべく読み込まないようにした。これにより、全てのレジスタを作業用として利用できるようにした。

図2に示すように、引数は必要なものだけを、必要になった時に、命令によりレジスタ上に読み込むようにしたので、ゴール実行に先立った準備の手間を最小限に押えることができ、サスペンドする場合にも、保存されているゴールレコードをそのまま使えるので、引数を書き戻す必要がなく、コンテキストスイッチを非常に軽くすることができた。

しかし、直後に(execute命令で)実行するゴールの場合には、従来は、レジスタ上で引数準備を行なえばよかったのが、一旦メモリ上に書かなければならなくなった。この欠点に関しては、現在は特別な対応をしていないが、将来、最適化によりある程度補うことができると考えられる。この最適化については、最後で説明する。

3.1.2 メモリベース方式のための命令

メモリベースの処理方式を実現するために引数をゴールレコードからレジスタ上に読み込む命令を用意した。

メモリベース方式は、ゴールレコードを普通の構造体と同じように扱う方式ということができる。即ち、ゴールレコードは引数と制御情報を要素とする構造体であり、この構造体を唯一の(物理的な)引数としてゴールを実行しているのだと考えることができる。従って、構造体の要素を操作する命令とほぼ同じ機能のゴールレコード操作命令を用意した。

なお、子ゴールの引数を書き込む命令は従来から用意されていた。

機能	構造体用	ゴールレコード用
read + wait	read_wait	**load_wait**
read	read	**load**
write	write	store

例えば、

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

というプログラムをコンパイルした結果、生成されるKL1-Bコードは以下のようになる。

```
 1  foo/3:
 2    reset_ssp
 3    load_wait      r0,0,r1,susp
 4    is_integer     r1,susp
 5    test_integer   1,r1,susp
 6    load_wait      r0,1,r2,susp
 7    is_atom        r2,susp
 8    test_atom      foo,r2,susp
 9    load_wait      r0,2,r3,susp
10    is_vector      r3,susp
11    test_arity     2,r3,susp
12    read_wait      r3,0,r4,susp
13    is_integer     r4,susp
14    test_integer   2,r4,susp
15    commit
16    read           r3,1,r5
17    alloc_goal     1,r6
18    reuse          r3,vector(2),list
19    write          r5,r3,car
20    put_atom       [],r7
21    write          r7,r3,cdr
22    store          r3,r6,0
23    enqueue        r6,baz/1
24    put_integer    1,r8
25    store          r8,r0,0
26    put_atom       bar,r9
27    store          r9,r0,1
28    execute        foo/2
29  susp:
30    suspend        foo/3
```

ここで、2行目ではサスペンションスタックの初期化を行なう。3〜5行目ではそれぞれ、第1引数の読み込みと具体化チェック、型のチェック、値のチェックを行なう。6〜8行目、9〜11行目も同様の処理を行なう。12〜14行目では第3引数のベクタの第1要素をチェックする。ここまでで、このクローズの選択が確定するので、15行目にcommit命令が置かれている。

引続き、16行目では第3引数のベクタの第2要素をレジスタ上に乗せておく。17行目ではbaz/1のためのゴールレコードを割り付ける。18行目では不要になる2要素ベクタをリスト用に再利用する。19〜21行目ではそのリストの要素にデータを書き込む。22行目でリストをゴール引数に書き込

み、23行目でエンキューする。24〜27行目ではbar/2用の引数をゴールに書き込み、28行目でエクゼキュートする。30行目はサスペンドする場合の処理を行なう。

3.1.3 ゴールレコード上の引数の利用

新しい方式では、メモリ上にあるゴールレコードを保存しているので、子ゴールを呼び出す場合には、親の引数環境を積極的に利用する最適化を行なうことにした。

従来の方式でも、親の引数環境(レジスタ上にある)を子ゴールにそのまま渡せる場合があったが、それは、すぐ次に(execute命令で)実行するゴールに限られていた。

新しい方式では、複数の子ゴールを呼び出す場合には、親のゴールレコードをそれらのうちの任意のものに再利用することができる。そこで、コンパイル時の解析により、親の引数環境を最も有効に活用できる子ゴールを選ぶことにした。これにより、引数環境をゴールレコードに書き込む操作を減らすことができるようになった。

例えば、次のようなプログラムでは、foo/5 → bar/5 と bar/5 → foo/5 の組合せで、引数の多くが渡されている。このような場合、従来の方式では、親の引数環境を子ゴールにそのまま渡せるのは、foo/5 → bar/5 の場合だけで、bar/5 → foo/5 の場合には、新たに割り付けたゴールレコードに引数を書き込まなければならなかった。しかし、新しい方式では、コンパイラが最も適した組合せを選ぶので、どちらでも親の引数環境を利用できるようになる。

```
foo(A,B,C,D,E) :- true |
        bar(A,B,C,D,X), xxx(E,X).

foo/5:                                      親
  commit                                    |
  load              r0,4,r1                -+
  alloc_goal        2,r2          ---+      |
  store             r1,r2,0         -+      |
  alloc_variable    r3               |      |
  store             r3,r2,1         -+      |
  enqueue           r2,xxx/2     <---+      |
  store             r3,r0,4                 +
  execute           bar/5        <------+
```

```
bar(A,B,C,D,E) :- true |
        xxx(E,X), foo(A,B,C,D,X).

bar/5:                                      親
  commit                                    |
  load              r0,4,r1                -+
  alloc_variable    r2                      |
  store             r2,r0,4                -+
  enqueue           r0,foo/5 <-------------+
  alloc_goal        2,r0          ---+
  store             r1,r0,0         -+
  store             r2,r0,1         -+
  execute           xxx/2         <--+
```

3.1.4 メモリ上の作業用領域

新しい方式では、メモリ上にも作業用の領域を用意することにした。これは、どうしてもレジスタが不足する場合に使うもので、当面必要でないデータをレジスタ上からここに退避し、空いたレジスタを今必要なデータのために利用できるようにするものである。このために、次の2つの命令を用意した。

退避(レジスタ→メモリ)	save
復帰(メモリ→レジスタ)	restore

これにより、複雑なユニフィケーションを行なうプログラムでも、ハードウエアで用意されたレジスタ数とは関係なく実行することができ、マシンに依存した制限を減らすことができた。

3.2 KL1-B レベルの最適化

メモリベース方式を採用することにより、従来の問題点を解決することができたが、さらに処理効率を上げるために、いくつかの改良をおこなった。これらは、主にコンパイル時の最適化の強化であり、従来のKL1-Bに適用することも可能である。

3.2.1 不要構造体の要素の利用

3.1.3 で述べたゴールレコード上の引数の利用と同じように、構造体データについても同様の解析を行ない、不要構造体の要素を積極的に利用するようにした。即ち、不要となった構造体の要素データを新たに割り付ける構造体の同じ要素位置

にそのまま渡している場合には、もし、その組合せで構造体を再利用することができれば、構造体要素上のデータをそのまま利用することができ、要素をコピーするためのRead/Writeの操作が不要になる。

ただし、構造体ではMRBがONの場合があるので、その時には、不要構造体の再利用が不可能、即ち、このゴールでは不要であるが、ほかのゴールで使用しているかも知れないので、新しい構造体を割り付ける必要がある。そこで、このような時には、そのまま利用するつもりだった不要構造体上の要素を新たに割り付けた構造体にコピーする操作が必要になる。このために、MRBがONのときに、どの要素をコピーすればよいかという情報を持たせた **reuse_with_elements** 命令を用意した。

この最適化は、次のような場合に有効である。

```
foo(abc(A,B)) :- true | bar(xyz(A,B)).
```

もし、この最適化をしない場合には、KL1-Bは、

```
    ...
    load_wait    r0,0,r1,susp
    is_vector    r1,susp
    test_arity   3,r1,susp
    read_wait    r1,0,r2,susp
    is_atom      r2,susp
    test_atom    abc,r2,susp
    commit
    read         r1,1,r3          ★
    read         r1,2,r4          ★
    reuse        r1,vector(3),vector(3)
    put_atom     xyz,r5
    write        r5,r1,0
    write        r3,r1,1          ★
    write        r4,r1,2          ★
    store        r1,r0,0
    ...
```

のようになるが、この最適化により、

```
    ...
    load_wait    r0,0,r1,susp
    is_vector    r1,susp
    test_arity   3,r1,susp
    read_wait    r1,0,r2,susp
    is_atom      r2,susp
    test_atom    abc,r2,susp
    commit
    reuse_with_elements
        r1,vector(3),vector(3),{0,1,1}
    put_atom     xyz,r5
    write        r5,r1,0
    store        r1,r0,0
    ...
```

のように、★印の付いていたRead/Write命令を削除することができた。

新命令 **reuse_with_elements** は、不要構造体のMRBがOFFの時には、reuse命令と全く同じなので、この場合には削除したRead/Write命令の分が得になる。MRBがONの時には、要素をコピーする必要があるが、このコストは、削除した命令と同じであり、従来のKL1-Bに比べて同等であるといえる。

3.2.2 単一待ちの最適化

通常のKL1プログラムは、ほとんどの場合、単一待ちになっている。そこで、この場合には、メモリ上にあるサスペンションスタックを使用しないで、具体化チェックのために変数を乗せたレジスタを指定することで、サスペンド処理を行なうことにした。このために、次のような命令を用意した。

機能	通常用	単一待ち最適化用
read+wait	read_wait	**read_wait_single**
load+wait	load_wait	**load_wait_single**
suspend	suspend	**suspend_single**

この命令を使うことができた場合には、サスペンションスタックを読み書きする操作や、単一待ちか、多重待ちかの判断が不要になり、サスペンドの処理を軽くできた。また、サスペンションスタックポインタを使わない場合が多くなるので、これを汎用レジスタと兼用にして、作業用にも使うことができるようにした。

3.2.3 otherwise/alternatively の最適化

コンパイル時に、実行の流れを解析することで、無駄なotherwise/alternatively命令を実行しなくて済むようにした。これらの命令は、もし待つべき変数がない場合にはNOPと見做せるので、実行時の流れに沿って、otherwise/alternatively命令に到達する迄の間に、待つべき変数が有ったか、無かったかを調べることにした。この結果、もし待つべき変数がな

い場合には、これらの命令の次にラベルを作り、そこへ分岐してくるようにした。

例えば、下のような場合には、★印の命令の分岐先は、元々otherwise命令のところであるが、最適化により、otherwise命令の下に作ったラベルに変更している。

```
foo(bar) :- …
otherwise.
次の節

foo/1:
  load_wait   r0,0,r1,susp
  is_atom     r1,fail        ★
  test_atom   bar,r1,fail    ★
    …
susp:
  otherwise   foo,1
fail:
  次の節
```

また、otherwiseの場合には、単一待ちの最適化と併用することができ、その場合には、以下のように、otherwise命令を、suspend_single命令に変更する。

```
foo/1:
  load_wait_single  r0,0,r1,susp
  is_atom           r1,fail
  test_atom         bar,r1,fail
    …
susp:
  suspend_single    r1,foo,1
fail:
  次の節
```

3.2.4 diffの実現方式の変更

従来のKL1-Bでは、diff \= は組込述語として実装していた。そのため、\= の両辺はガード組込述語の入力引数になるので、構造体を書くことができなかった。そこで、クローズインデキシング機能の拡張により \= を実現するようにした。

新しい方式では、\= の成功は、それに対応する = の失敗と定義し、インデキシング木で、対応する = を表現するノードの失敗の場合の行き先を全て一箇所に集め、そこを \= の成功とした。このためには、失敗ラベルとサスペンドラベルを完全に分ける必要があり、ガードのジェネラルユニフィケーションの命令equalには2つのラベルを付けるようにした。

この変更により、= と同じように、\= の両辺に構造体を書くことが可能になった。また、\= と = が排他になるので、次の例のように、チェックを1つにまとめることができた。

```
foo(X) :- X  = {bar} | ☆.
foo(X) :- X \= {bar} | ★.

foo/1:
  load_wait_single  r0,0,r1,susp1
  is_vector         r1,fail
  test_arity        1,r1,fail
  read_wait_single  r1,0,r2,susp2
  is_atom           r2,fail
  test_atom         bar,r2,fail
  commit
   ☆
fail:
  commit
   ★
susp1:
  suspend_single    r1,foo,1
susp2:
  suspend_single    r2,foo,1
```

4 おわりに

現在は、まだ評価を始めた段階であるが、改良された抽象機械語KL1-Bを採用することにより、レジスタの使用効率が良くなり、コンテキストスイッチが軽くなるので、並列プログラムをより高速に実行できると期待している。しかし、新KL1-Bにも問題点が残されている。

この問題点は、appendのような最下位のサブルーチンを、途中でサスペンドが起きないように実行した場合に発生する。このような場合には、従来のKL1-Bでは再帰呼び出しによる実行を全てレジスタ上で行なうことができた。しかし、新KL1-Bでは、再帰呼び出しの場合にも一旦メモリ上に引数を書き出すので実行速度が遅くなる。このようなサブルーチンは、何回も呼び出される場合が多く、これが遅いと、性能への影響が大きいと考えられる。

この対策として、このようなサブルーチンをコンパイラが検出するか、ユーザーがプラグマを付けることにより、特別なコードを生成することを

考えている。即ち、このようなサブルーチンは、通常、引数が少ないので、レジスタに引数を置いてもレジスタ不足にはならない。そこで、このような場合に限り、従来のようにレジスタに引数を置いて実行するようなコードを生成するわけである。

例えば、appendの場合には以下のようなコードを生成する。

```
ap([]    ,Y,Z):-true|Z=Y.
ap([H|X],Y,Z):-true|Z=[H|L],ap(X,Y,L).

ap/3:
  load                  r0,0,r1       ★
  load                  r0,2,r2       ★
loop:
  wait_single           r1,susp <---+
  is_list               r1,next     |
  commit                            |
  read                  r1,cdr,r3   |
  reuse_with_elements   r1,list,list,[1|0]
  alloc_variable        r4          |
  write                 r4,r1,cdr   |
  unify_bounded         r1,r2       |
  move                  r3,r1       |
  move                  r4,r2       |
  execute_quick         loop    ----+
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  enqueue               r0,ap,3
  proceed
next:
  is_atom               r1,fail
  test_atom             [],r1,fail
  commit
  load                  r0,1,r1
  collect_goal          r0
  unify                 r1,r2
  proceed
fail:
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  fail                  ap,3
susp:
  store                 r1,r0,0       ☆
  store                 r2,r0,2       ☆
  suspend_single        r1,ap,3
```

ここでは、実行に先立ち、★印の命令により、当面必要となる第1、第3引数をレジスタ上に読み込んでいる。また、コンテキストスイッチの部分では、☆印の命令によりレジスタからメモリ上に書き戻している。

実行のメインの部分は全て、レジスタ上で行なわれており、子ゴールを呼び出す部分では、move命令によりレジスタ上で引数を準備している。execute_quick命令では、レジスタ上の環境で子ゴールを呼び出すが、スリットチェック処理が必要な場合には、この命令はなにも行なわずに、次命令からの store と enqueue が実行される。

このようなコード生成の最適化は、次の版のコンパイラで採用する予定であり、どのようなプログラムであっても、かなり高速に実行できるようになると期待している。

謝辞

日頃ご指導頂いている内田俊一室長をはじめとするICOT第4研究室の研究員の方々に感謝します。また、数々の貴重な助言をいただいた、ICOTならびに各社PIMグループの方々に感謝します。

参考文献

[1] Y.Kimura, T.Chikayama: An Abstract KL1 Machine and its Instruction Set. In Proceedings of 1987 Symposium on Logic Programming, Sep. 1987.

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

[3] 木村康則, 西崎慎一郎, 中越靖行, 平野喜芳: KL1のクローズインデキシング方式, JSPP '89, pp187-194, Feb. 1989

[4] 木村康則, 近山隆: 並列論理型言語KL1の多重参照管理によるガーベジコレクション, 第22回 プログラミング言語研究会, Oct. 1989