Portable な KL1 処理系の構想

関田 大吾
(株) 三菱総合研究所

近山 隆
(財) 新世代コンピュータ技術開発機構

第5世代コンピュータプロジェクトでは並列論理型言語 KL1 により様々な並列プログラムが開発されてきた．現在，これらの KL1 の本格的な処理系としては Multi-PSI, PIM といった専用計算機上のものしかなく，KL1 言語の普及のためには，汎用的な計算機上で動作する処理系が望まれる．本論文では，KL1 を一度 C 言語にコンパイルすることにより，移植性と“それなりの”効率を得る方式の提案を行なう．C 言語を中間言語とすることによりいくつか問題点が予想できたが，この問題は実装上の工夫で回避できる．現在この試験的な処理系が動作しており，さらに実用的にするための検討を行なっている．

Plan for a portable implementation of KL1

Daigo Sekita
Mitsubishi Research Institute, Inc.

Takashi Chikayama
Institute for New Generation Computer Technology

In the FGCS project, many parallel programs have been developed using KL1. However, the practical KL1 implementations have been made only on specially devised hardwares such as Multi-PSI, and PIM. For wide use of KL1 language, a portable implementation is needed. In this paper, the scheme of compilation from KL1 to C for portability and reasonable efficiency is investigated. By choosing C as the intermediate language, some disadvantages had been supposed, but these disadvantages can be avoided by some implementation techniques. Currently, an experimental implementation has been made, and further studies are in progress for the practical use.

1 はじめに

第5世代コンピュータプロジェクトでは並列論理型言語KL1[3, 8]により様々な並列プログラム，例えば並列OSであるPIMOS[5]，その他各種アプリケーションプログラム[1, 6]など，が開発されてきた．これらのプログラムの開発を通じて，並列プログラム言語としてのKL1の有用性は確かめられた．しかしながら，このKL1の本格的な処理系は，現在，Multi-PSI[4]，PIM[7, 2]といった専用ハードウェア上でしか動作しない．専用ハードウェア上での実装は効率が良いという長所がある一方，移植性には決定的に欠ける．KL1プログラムのさらなる広い普及のためには汎用計算機[1]上で利用できる処理系が必要であると判断できる．

汎用計算機上で移植性の良い処理系を作成する方法としてはいわゆる抽象マシンインタプリタを作成するのが容易である．[2]しかしながら，抽象マシンインタプリタでは効率が悪く，本格的なプログラムの実行は困難である．一方，直接ターゲットとなる計算機のnative codeにKL1をコンパイルする方法も考えられるが，移植性を確保することが難しくなる．

そこで，中間言語として比較的移植性の良い低レベル言語を選定し，その言語を介してKL1をnative codeにコンパイルする方法が効率/移植性を両立させる方法として考えることができる．この中間言語としては現状ではC言語が適当であると判断する．

ここでは，KL1を一度C言語にコンパイルすることにより移植性がよく効率も“それなりに”得られる処理系を作るための構想について説明を行なう．

2 中間言語にCを用いることの利点/欠点

KL1をC言語にコンパイルする方式の利点および克服すべき問題点について本章で挙げる．

2.1 利点

以下のような利点が挙げられる．

移植性： C言語という移植性の高い言語を中間言語におくことにより，多くの計算機環境で比較的容易に動作させることができる．

低レベル最適化： マシン依存となるような低レベル最適化についてはCコンパイラの最適化ルーティンに任せることができる．これは移植性の向上のみならず，処理系の作成のしやすさという面から見ても有利である．

他言語とのリンク： UNIXではCをその主言語としており，C言語と，C言語以外の言語とのリンクを行なうためのインタフェースが通常用意されている．

したがって，C言語を中間言語とすることにより，C言語で書かれたユーザ・ルーティンとのリンケージのみならず，C言語以外の言語で書かれたルーティンとのリンケージをとることも容易になることが考えられる．

2.2 問題点

いっぽう，効率の面より下記のような問題点が予想される．これらの問題点を克服するよう実装方法を考える必要がある．

関数呼び出しのオーバヘッド： naiveなコンパイル方式としては，KL1の1つ述語をCの1つの関数にコンパイルする，という方法が考えられる．しかし，この方法では関数呼び出しオーバヘッドが深刻な問題になり得る．

C言語で書かれたプログラムでは一般的に関数はさほど小さくなく，関数呼び出し回数も比較的少ない．そのため関数呼び出しオーバヘッドが実行時間全体に占める割合はさほど大きくならないため，このオーバヘッドが著しく問題になることはあまりない．

しかしながら，KL1の述語はC言語の関数と比較するとはるかに粒度が小さく，呼び出し回数もはるかに多い．したがってKL1の述語をC言語の関数にコンパイルするとなると関数呼び出しのオーバヘッドが占める割合は多くなることが予想できる．

さらに，Prolog，KL1等で良く行なわれる末尾再帰に対する最適化についても，通常Cコンパイラの関数呼び出しでは行なわれない[3]ため，stackの伸びが著しくなる可能性が高い．

レジスタ割り付け： KL1の実装上，アクセス頻度が非常に高い大域的データが存在する．例えば，heap topを指すポインタはメモリ割り付けを行なう度に参照される．そのため，これらのデータは特定のレジスタ上におくことが効率上非常に有利である．

しかしながらC言語では大域的な変数に対するレジスタ割り付けの指定は一般的には許されず，[4]

---

[1] 通常のUNIX計算機，すなわち，32 bitポインタを持つ，UNIX上の通常のプログラム環境を利用できる計算機を想定する．ここでいうUNIXはBSD,SystemVを共に含む．

[2] すでにPDSSと呼ばれる抽象マシンインタプリタによる疑似並列処理系も製作済みである．

[3] gccなどでは，“自分自身”を呼び出す末尾再帰呼び出しに関しては最適化が行なわれるが，他の関数を呼び出す時に末尾再帰の最適化を行なわない．他の関数を呼び出す時にも末尾再帰の最適化を行なうコンパイラも存在するが，移植性の点で問題になる．

[4] gccなどのコンパイラでは許されているが，この利用は移植時に問題となる．

これらのデータはレジスタ上におくことができない．直接 native code を出力するのであればこのような最適化は可能となるが，移植上不利になる．

割り込み処理対応： 割り込み処理は UNIX-C の枠組では signal を元に行なわれ，通常のコードは常に割り込まれる可能性がある．しかしながら，割り込み処理中にも，メモリの割り付け，データのセットなどを行なう必要があり，これらの処理は割り込まれた通常処理と共通するデータ (例えば，heap top ポインタ) を変更する可能性がある．よって，通常処理中にもこれらのデータをアクセスする度にメモリの lock などによる排他制御が常に必要となり，コード量，実行速度両面のオーバヘッドより好ましくない．

オブジェクトコードサイズ： Multi-PSI, PIM といった専用計算機では命令セットが高レベルであるため，オブジェクトコードのサイズはさほど大きくならない．また，同様の抽象命令セットを持った抽象マシンインタプリタでも同様である．しかしがら，KL1 を C にコンパイルし，さらに汎用計算機のオブジェクトコードに落すと，命令レベルが低いためコードサイズは非常に大きくなる．その結果，ワーキングセットが極めて大きくなり，抽象マシンインタプリタによる実装よりもかえって効率が悪くなる可能性がある．

2.3 解決案

ここまでで挙げた予想される問題については，以下のような方法で解決可能と考える．

関数呼び出しのオーバーヘッド： 関数呼び出しを減らすために，1 つの KL1 のモジュールを 1 つの C 言語の関数としてしまうことが考えられる．

すなわち，モジュール内の述語呼び出しは関数呼び出しではなく全て無条件ジャンプにより実現される．

KL1 のモジュールは意味的 / 機能的に比較的共通した述語が集まったものであるため，述語呼び出しの多くはモジュール内で行なわれる．このモジュール内述語呼び出しの最適化で関数呼び出しオーバヘッドをかなり減らすことが期待できる．

レジスタ割り付け： C 言語では大域変数にレジスタを割り付けることはできないが局所変数には 'register' 宣言やコンパイラの解析によりレジスタを割り付けることができる．これを利用し，ここで問題になるような大域的でアクセス頻度の高いデータについては，関数入口でレジスタに割り付けられた局所変数にそのデータを cache する．このため，関数内での大域データへのアクセスは全てレジスタに対して行なえば良いことになる．

先に述べた，1 つの KL1 のモジュールを 1 つの C 言語の関数に割り付ける方法からも関数単位はかなり大きなものであると期待できるため，実際的にはほとんどの大域データに対するアクセスはこのレジスタ上に cache されたものに対するものとなることが期待できる．

割り込み処理対応： 割り込みハンドラでは，適当な flag に割り込みのあったこととその種類だけを記録する，という処理しか行なわないものとする．通常処理の適当なタイミング (例えば，KL1 の reduction の切れ目に当たる処理中など) でこの flag を参照し，割り込みがあったことがわかれば適当な処理を行なう．したがって通常処理では割り込まれることを恐れることなく処理を行なうことができる．

さらにこの割り込み検査は，他の不可欠な検査 (例えば，ヒープ溢れを検出する検査) と同時に行なうことによりさらにオーバヘッドは減らすことが可能である．

オブジェクトコードサイズ： KL1 ではデータの入出力が明確であるため，Prolog と比較すると，大域的な解析等の大規模な最適化を行なうことなくコードサイズを小さくすることができる．また，効率が問題になるような通常処理部については inline 展開するが，例外処理等の効率にさほど影響のない部分は run time library とすることによりコードサイズを大幅に減らすことができる．さらに小さくするためには，ここで挙げている native code による実装とインタプリタ，抽象マシンインタプリタなどの実装を併用することが考えられる．

3 実装の実際

以下のような実装方法を考える．

3.1 データ表現

KL1 の項は 1 ワード 32bit とする．上位 30bit の値部はポインタ又は即値データとして使用する．下位 2bit はタグであり，以下の 4 種類ある．

変数参照： 間接ポインタ，または未定義変数を表す．未定義変数は，自分自身を指す参照として表現する．

Atomic： atomic なデータに対してはさらに下位 2 bit をタグとして付加する．この付加タグにより，

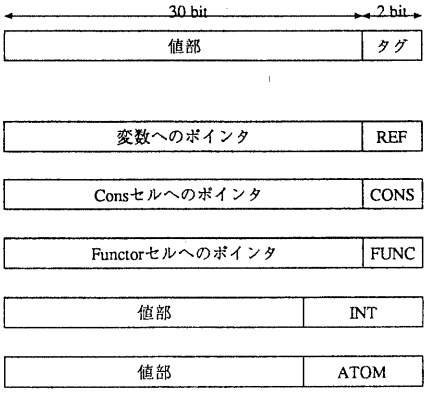

図 1: データ一覧

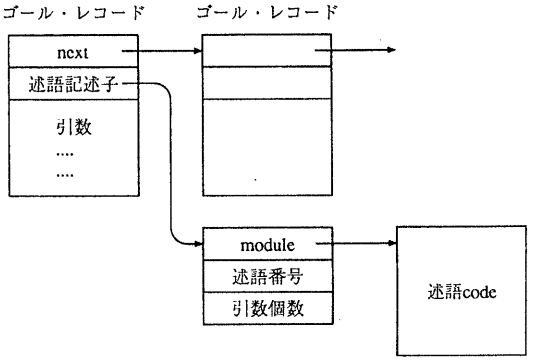

図 2: ゴールスタック

アトム，整数，浮動小数点数等が値部にあることを示す．

Cons： 値部は cons cell すなわち，連続した 2 ワードメモリブロック (各々 car, cdr) を指し示すポインタとなる．

Functor： 値部は連続したメモリブロックを指し示すポインタとなる．指し示されたメモリブロックの先頭には functor ID が入れられ，この ID よりメモリブロックの大きさを知ることができる．残りの部分には，functor 要素がいれられる．

文字列や実装のための特殊な構造 (merger cell など) は全てこの functor として表され，特殊な ID が付けられている．

データ表現一覧を図 1 に挙げる．

3.2 ゴール管理

ゴールは通常データと同様 heap 上におかれ，実行可能なゴール群はスタック状に管理される．ゴール reduction 時には，stack 先頭のゴールが pop され実行される．ゴール実行の結果生成されたサブ・ゴールは stack に push される．

ゴール・スタックは線形リストで実現されている．各ゴール・レコードは固定長で，述語記述子に対するポインタ，述語引数，および次のゴールレコードに対するポインタを持つ．[5]述語記述子はモジュールへのポインタ (すなわち C 言語の関数へのポインタ) とモジュール内での述語番号を持つ．モジュールへのポインタと述語番号により述語を表すコードへのポインタを特定する．使用済みのゴール・レコードはゴール・レコード専用の free list に接続される．ゴール・レコードが新たに必要になった時にはこの free list より供給される．その時 free list が空であるなら，新たに heap より free list を割り付け，ゴール・レコードが供給される．[6] free list を指すポインタは C 関数内ではレジスタ上に cache しているため，高速にゴール割り付けを行なうことができる．

ゴール・スタックの詳細を図 2に記す．

Heap 中には複数のゴール・スタックが存在し，各々のゴール・スタックは優先度順位の違いを表す．各々のゴール・スタックの末尾にはそのスタックより低い優先度のスタックのゴールを実行するような特殊なゴールが繋げられている．[7]

入力引数が未定義であるため実行を中断しているゴールは，その中断の原因となっているセルを変数参照セルとし，そのポインタを通じて参照される．この変数参照セルと中断中のゴールとの間には中断記述子をおく．中断記述子の 1 ワード目には中断記述子であることを表す特殊なアトムを入れておき，2 ワード目はゴールを指すポインタとする．1 つの変数を複数のゴールが待っている場合にはゴールレコードを線形リストとして接続しておく (図 3参照)．他の述語を実行中にこの中断記述子を指している変数セルを具体化したならば，中断状態のゴール (群) はゴール・スタックに push される．

複雑な単一化 (例えば，ゴールの中断を起こしている変数同士の単一化など) は，その単一化を行なう述語を push するという処理を run time library 内で行なっている．この際，この処理を行なわせるため，ゴールを通常の free list から割り付けることは，free list を指すポインタを変更することになり，先に述べたレジスタ上に cache されている free list ポインタを無効

[5]優先度の実装のためには，各ゴール・レコードには優先度フィールドも必要である．

[6]この時，ゴール・レコードは複数個同時に割り付けられる．

[7]最低優先度のゴール・スタックには top level success するゴールが接続されている．

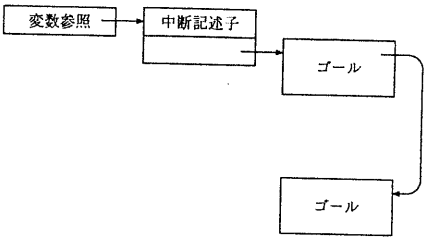

図 3: 中断状態

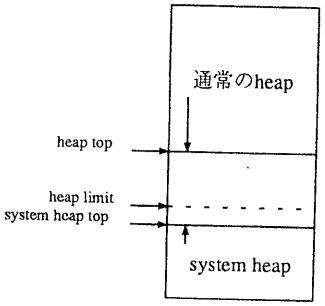

図 4: Heap の構造

化してしまう．これを避けるため，単一化ルーティン内で行なわれるゴール割り付け用には別の free list を用意してあり，通常のゴール割り付け時にはレジスタに cache されている free list ポインタを用いることができる．

4 Heap 領域管理

Heap は図4で示すように以下の2つの領域に分けられる．

通常の heap： 通常の処理で割り付けられるデータを置く領域．図の上方より下方に割り付けられる．

System heap： 例外的なデータ割り付け (例えばゴール・レコード不足時) を行なうための領域．図の下方より上方に割り付けられる．

2つの領域の間に heap limit がおかれ，heap limit pointer がおかれる．例外的なデータ割り付け用の領域は別にあるため，ゴール・レコードなどの例外的なデータの割り付けが起きたとしても通常のデータ割り付け用の heap ポインタは C の関数で持つ cache された局所変数を利用して構わない．

通常の heap が溢れを起こすとごみ集めの処理が行なわれる．ごみ集めは copying 方式で行なう予定である．System heap が溢れを起こした場合には heap limit を上方に移動させる．System heap ポインタと heap limit，および通常の heap ポインタと heap limit の間は常に 1 reduction で必要となるメモリに対して十分の余裕分を持たせてあり，reduction の切れ目でのみ heap 溢れの検査を行なえば良い．

5 割り込み処理

他プロセッサからの通信などによる割り込み (UNIX の signal) が起きると，割り込みハンドラは大域変数である特定の flag に割り込み種類を保存し，heap limit を指すポインタを変更する．このため，次回の heap 溢れの検査時 (つまり reduction の切れ目) に (疑似的に)heap 溢れが検出されることになり，heap 溢れ時の処理ルーティン内で，本当に heap 溢れが起きたのか，それとも割り込みがあったのか (又はその両方か) を検査し，適当な処理を行なう．このため，通常処理中は，割り込みがかかることにより heap ポインタなどの変更が行なわれることを考慮することなしに処理を行なうことができる．さらに例外処理の検査も heap 溢れの検査と同時に行なうことができる．

同様の処理は，現在 reduction 中のゴールよりもより高い優先度のゴールが新たに実行可能状態になったことを示すためにも利用することができる．

6 コンパイル

現在はコンパイル時には大域的な解析は行なわず，とりあえず述語内のみの解析によってコードを出すことを考えている．節の選択については，clause indexing による最適化を行なう．

KL1 の1つのモジュールは1つの C の関数にコンパイルされ，以下のような処理を行なう．

1. モジュールの先頭にある，top level の関数に jump する．
2. 述語内では以下のようなことを行なう．
   (a) ゴール・スタックよりゴール・レコードを pop し，ゴール・レコードに保持されている引数を局所変数にコピーする．
   (b) 引数を調べ，KL1 のガード部に書かれていた条件により，どの節部を適用すべきか決定する．
   (c) KL1 のボディ部に書かれていた処理を行なう．すなわち，以下のようなことを行なう．
   - メモリ割り付け
   - 変数単一化
   - ゴールの生成，ゴール stack への push
   - (可能であるなら) 再帰呼び出しによる次のゴールへの jump

```
nrev([], Y) :- true | Y=[].
nrev([A|X], Y) :- true |
    nrev(X, Y1), append(Y1, [A], Y).

append([], Y, Z) :- true | Y=Z.
append([A|X], Y, Z) :- true |
    Z=[A|Z1], append(X, Y, Z1).
```

図 5: naive reverse プログラム

再帰呼び出しされる述語が，同一モジュール内であれば，対応するC側の述語部にある引数部分に呼び出されるゴールの引数が渡され，heap 溢れの検査を行なったのちに jump する．呼出しされるゴールが外部モジュールにあったり，呼出されるゴールがない場合には，3に飛ぶ．

(d) ガード部に書かれていた条件により実行できる節がない場合には失敗 / 中断処理を行なう．

3. 選択された節に記述されていた呼び出し処理が，外部モジュールの呼び出しであったり，呼び出しがなかったりする場合には，以下の処理を行なう．

   (a) heap 溢れの検査．検査結果により，割り込み処理や，ごみ集めが行なわれる．

   (b) ゴール・スタック先頭のゴールが選ばれる．このゴールに保持されている述語記述子を調べ，現在処理を行なったモジュールと同じモジュールを示している場合には，引続き，現在のモジュールの述語選択部に制御を移す．述語選択部では，述語記述子に書かれているモジュール内述語番号により対応する述語に制御を移す．
   述語記述子に示されるモジュールが現在のものと異なる場合には，そのモジュールの top level へ制御を移す．

単一化については，一方の引数がゴールの中断要因になっていない未定義変数である場合のみ inline に展開したコードを出力する．それ以外の場合には汎用単一化ルーティンを呼び出す．

7 予備的な性能評価

現在，ここまで記述してきた内容の試験的な処理系がコンパイラも含めて既に動作している．表 1にその試験的処理系の評価結果を示す．

評価プログラムは 30 要素の naive reverse(496 reductions, 図 5参照) で，SUN SparcStation2, 1+, および，Sequent Symmetry で評価を行なった (コンパイラはいずれも gcc 2.2.2, 最適化オプションは -O2). 処理時間にはシステム時間および，ごみ集めの時間を含む．

表 1: 評価結果

	Sparc 2	Sparc 1+	Symmetry
繰り返し回数	10,000	10,000	2,000
処理時間 (mscc)	4,730	8,190	7,270
LIPS	1,048,625	605,616	136,451
text size(byte)	688	688	696
data size(byte)	24	24	24

8 将来的な見通し

以上で述べてきたような方法により，C を中間言語とすることにより移植性の良い高効率な KL1 の処理系を作成することが可能であると考える．しかしながら，Multi-PSI, PIM 上の処理系で現在動作している PIMOS などのプログラムの変更を最小限に押えるためには，以下で挙げるような項目についてさらに検討が必要である．

8.1 多重待ち合わせ

述語が，複数の要因により中断した場合，その要因となった未定義変数のうち 1 つの具体化により中断状態が解除され実行される場合がある (図 6で示す merger プログラムを参照)．この場合，中断状態を直接解除した以外の変数についての中断状態も解除しないと，複数回数このゴールは実行されることになるが，上記までの処理系ではゴールからその中断要因となっている変数への逆ポインタがないため，この解除を行なうことはできない．

現在考えている対応は以下の通りである．

- Multi-PSI, PIM 等で行なわれている，多重待ち合わせの方法をとる．すなわち，多重待ち合わせ時には，単一の要因による中断とは異なるゴール管理を行ない，中断要因となっている変数のうち 1 つでも単一化された場合には他の要因からもこのゴールは中断解除される．

- プログラム変換，コンパイル技法により多重待ち合わせは避けるようにする．多重待ち合わせの代表的な例である merger についてはシステム組み込みのものを提供し，効率を保持する．

```
merge([], Y, Z) :- true | Y=Z.
merge(X, [], Z) :- true | X=Z.
merge([A|X], Y, Z) :- true |
    Z=[A|Z1], merge(X, Y, Z1).
merge(X, [A|Y], Z) :- true |
    Z=[A|Z1], merge(X, Y, Z1).
```

図 6: merger プログラム

8.2 マルチ・プロセッサ対応

現在，マルチ・プロセッサに対する対応としては，各プロセッサで UNIX のプロセスを走行させ，その間でプロセス間通信を行なわせる，という方法を考えている．プロセス間通信プリミティブには UNIX の internet socket を用いる．[8]

KL1 レベルではこのプロセッサ (プロセス) 間の通信は，共有変数の単一化により行なわる．この共有変数の中に伝えられるデータは変数であっても良いことになっており，いわゆる不完全メッセージによる双方向の通信を行なうことができる．

また，socket によるプロセス間通信のオーバヘッド軽減のため，多少の通信遅延を覚悟して，スループット増大のためにバファリングすることも考えられる．

8.3 デバガ

デバガの実現のためには以下の 2 通りが考えられる．

UNIX のデバグ環境の流用： コンパイルしたコードを dbx，gdb などの UNIX のデバガで直接デバグする．この場合，ソースコード情報をオブジェクトコード内に埋め込むにしても，KL1 のソース情報を埋め込むことはかなり困難であることが考えられ，一般アプリケーション開発者には極めて不十分であることが予想できる．しかしながら，C 言語等の他言語で記述したプログラムをリンクしている場合にはこの方法は有利である．

PIMOS のデバグ環境の移植： 現在 PIMOS で利用されているデバグ環境を移植することは考えられるが，UNIX プロセス外部から KL1 プロセスの実行を制御することになるため，その仕組みを UNIX に持ち込むためには若干の手直しが必要である．

[8] 共有メモリのマルチ・プロセッサ計算機では共有メモリを介した通信方式もサポートする予定である．

8.4 コンパイラ

現状作成されているコンパイラは Prolog で書かれており，いわゆるクロス開発環境となっているが，最終的には KL1 で書かれたセルフ・コンパイラがあることが望ましい．

しかしながらセルフ・コンパイラを実現するためには UNIX の object コードを KL1 のデータ (モジュール型，コード型) として扱うための枠組が必要である．移植性まで考慮に入れると，UNIX のオブジェクトコードの仕様が多様であることより，さらに難しくなる．incremental な loading も必要とされ，移植性の確保が困難になる．

そこで，セルフ・コンパイラでコンパイルしたコードは別プロセス (群) で動作させるという方法が考えられる．これは KL1 の荘園機能の仕様を若干手直しすれば可能である．

9 まとめ

汎用的な計算機で利用できる “それなりに” 高効率な KL1 処理系を実現するため，C 言語を中間言語とする方式を提案した．この方法により，移植性およびそれ以外の幾つかの利点が得られた．効率面での欠点が予想できるが，この欠点も実装上の工夫により問題を軽減することができると考える．

さらに，KL1 の本格的な処理系とするためには，言語上の幾つかの仕様についての実現方法に課題が残されており，今後検討を行なう．

参考文献

[1] K.Nitta, Y.Otake, S.Maeda, M.Ono, H.Osaki, and K.Sakane. HELIC-II:A Legal Reasoning System on the Parallel Inference Machine. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1992*, pp. 1115–1124, 1992.

[2] K.Taki. Parallel Inference Machine PIM. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1992*, pp. 50–72, 1992.

[3] K.Ueda and T.Chikayama. Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. (33)6, pp. 494–500, 1990.

[4] S.Uchida, K.Taki, K.Nakajima, A.Goto, and T.Chikayama. Research and Development of the Parallel Inference System in the Intermediate Stage of the FGCS Project. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 16–36, 1988.

[5] T.Chikayama, H.Sato, and T.Miyazaki. Overview of the Parallel Inference Machine Operating System (PIMOS). *Proc. Intl. Conf. on Fifth Generation Computer Systems 1988*, pp. 230–251, 1988.

[6] Y.Matsumoto and K.Taki. Parallel Logic Simulator based on Time Warp and its Evaluation. *Proc. Intl. Conf. on Fifth Generation Computer Systems 1992*, pp. 1198–1206, 1992.

[7] 後藤厚宏. 並列型推論マシンのアーキテクチャ. 情報処理, Vol. 32(4), pp. 458–467, 1991.

[8] 宮崎敏彦. 並列論理型言語 KL1 の実現方式と並列 OS の記述. 電子情報通信学会論文誌, Vol. J71-D, pp. 1423–1432, 1988.