ICOT Technical Memorandum: TM-0771

TM-0771

汎用計算機上のKL1処理系
—PDSS—

平野喜芳，西崎慎一郎，
中越靖行，宮崎芳枝(富士通)，
宮崎敏彦(沖電気)，近山　隆

July, 1989

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan
(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

汎用計算機上のKL1処理系 — PDSS —

The implementation of the KL1 system on the UNIX: PDSS

平野喜芳[1], 中越靖行[1], 西崎慎一郎[1], 宮崎芳枝[1], 宮崎敏彦[2], 近山隆[3]

Kiyoshi Hirano[1], Yasuyuki Nakagoshi[1], Shin'ichirou Nishizaki[1], Yoshie Miyazaki[1], Toshihiko Miyazaki[2] and Takashi Chikayama[3]

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

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

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を制御するオペレーティング・システムPIMOSはKL1で記述されており、その為のUNIX上におけるクロス開発環境として作成したのが、PDSS: PIMOS Development Support Systemである。このような目的で使用する為、PDSSはデバッグ機能を重視して開発した。特に、KL1で最も発生しやすいエラーの1つであるゴールのデッドロックを早期発見する機能を提供しており、デバッグ効率の向上に有効であった。これは、本来、実時間GCの為に用いているMRB(Multiple Reference Bit)によるデータの参照数の管理機構を拡張して利用することにより実現したものである。本報告では、この処理系の実現方式を中心に説明する。

はじめに

本論文では汎用計算機(UNIX)上に開発したKL1の擬似並列処理系 PDSS: PIMOS Development Support System について、その実現方式を中心に説明する。KL1はAND並列の論理型言語 Flat GHC [1] に基づいてICOTで設計された言語であり、OS記述等の為の機能 [2] が拡張されている。

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を制御するオペレーティング・システムPIMOS [2, 3] はKL1で記述されており、その為のUNIX上におけるクロス開発環境として作成したのが、PDSSである。従って、PDSSのKL1の仕様はPIMのものと極力互換性を持つように決めてあり、また、テスト/デバッグ機能を重視して開発した。

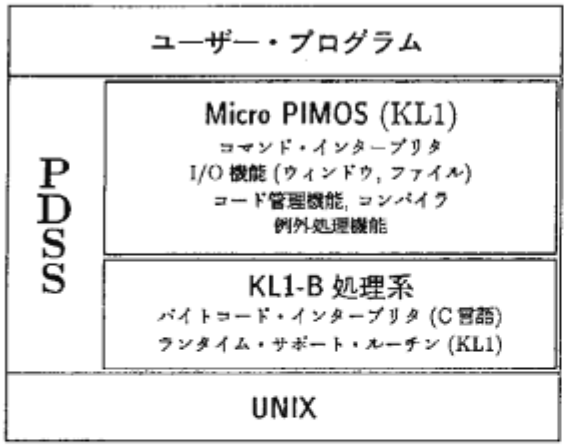

図 1: PDSS の全体構成

1 PDSS の概要

1.1 構成

PDSSは図1に示すように、主に二つの部分から成るシステムである。

一つはコンパイラによりKL1から変換された抽象機械命令KL1-B [4, 5] を実行するKL1-B処理系の部分である。このKL1-BはPrologにおけるWAM [6] に相当するもので、ICOTで開発中の並列マシンはこのKL1-Bを機械語または中間言語として用いている。PDSSではKL1-Bをバイトコードの形でメモリ上に置き、それをC言語で記述したバイトコード・インタープリタで実行する方式を採用した。また、C言語で直接記述するのが難しい部分はKL1自身で記述したランタイム・サポートルーチンをバイトコード・インタープリタから呼び出すようにした。

もう一つは Micro PIMOS [7] と呼ばれるKL1自身で書かれた簡易OSであり、ユーザー・インタフェース、入出力機能の提供等を行う部分である。ここにはKL1コンパイラも含まれている。これはPIMOSのサブセットと言えるもので、OSの保護や資源管理の方法等、基本的な構造は同じになっている。なお、本論文では Micro PIMOS の部分についての説明は省略する。

1.2 設計方針

PDSSは開発用ツールとして利用されるので、設計にあたってはテスト/デバッグ機能を最も重視することとし、以下のような目標を設定した。

- 実行順序に非決定性のある環境でプログラムをテストできる。
- プロセスのトレースを行うことができる。
- ゴールのデッドロックをなるべく早く検出する。
- メモリの消費量をなるべく少なくする。

以降、各章でこれらの意義と実現方法を説明する。

2 実行順序が非決定的な環境

2.1 実行順序が非決定的な環境の必要性

並列推論マシンPIMのようなマルチPEのシステムではKL1の各ゴールの実行順序に非決定性がある。これは各PEのメモリの消費状況等の実行環境が違っている為に、実行速度にばらつきが生じるのが主な原因となっている。特に動的な負荷分散機能があると、各PEの負荷がなるべく等しくなるように負荷(ゴール)を融通し合うので実行順序が大きく変わることがある。

正しいKL1のプログラムでは、どのような実行順序で実行しても正しい解(別解の場合もある)が得られるが、プログラムにバグがあると、実行順序によっては正しい結果が出ないという状況が発生する。つまり、実行順序に依存してバグの症状が出たり、出なかったりする。

ところが、PDSSはUNIX上の単一のプロセスとして動作する処理系であり、本来、実行順序に非決定性はない。いつ実行しても必ず同じ実行順序で実行される。その為、いつも同じ結果が得られることになり、プログラムにバグがあっても、たまたま症状が出なかった為にバグが発見できないという事が起こる。

この事から、PIM用のプログラムを開発することを考えると、実行順序に非決定性のある環境でプログラムをテストする必要があると考えられる。そこで、何らかの方法で実行順序に擬似的な非決定性を持たせる事にした。

2.2 擬似乱数によるスケジューリング

PDSSでは擬似乱数を利用する事により、実行順序の非決定性を作り出す事にした。

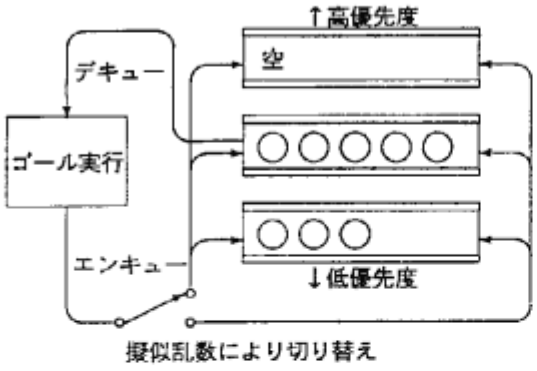

図 2: 擬似乱数によるスケジューリング

PDSSではKL1のゴールをゴール・レコードという構造体で表し、そのうち実行可能なものをレディ・ゴール・プールに入れて管理している。レディ・ゴール・プールは、4096段階の実行優先度に分割してあり、実行するゴールを取り出す場合には、必ず、最も高い優先度の先頭から取り出すようにしている。また、プールにゴールを入れる場合には、通常は親ゴールと同じ優先度のプールの先頭に入れるようにしてあり、実行優先度が指定された場合でも、指定された優先度のプールの先頭に入れるようにした。従って、レディ・ゴール・プールはスタックとして働き、KL1プログラムを深さ優先順で実行する事になる。

実行順序の非決定性を実現する為に、図 2 のように一部のゴールをプールの末尾に入れる機能を追加した。プールの末尾に入れられたゴールは実行が遅れるので、これを擬似乱数で選ぶ事により非決定性を持たせる事ができる。

これは擬似乱数により制御される為、その乱数の種をユーザーが指定することにより乱数の系列が変わり、実行順序を変える事ができる。また、同じ種を指定する事により実行順序を再現する事ができ、バグが発見された時に対処し易くなっている。

3 プロセスのトレース

3.1 プロセスに注目したトレースの必要性

プロセス [8] とはKL1を含む並列論理型言語における最も代表的なプログラミング・スタイルの一つである「プロセスとストリーム通信」のスタイルで用いられる概念である。

このプロセスとは、再帰呼び出しにより実現される、ある程度長いライフタイムを持つ実行過程のことである。複数のプロセスは共有する変数を用いて通信を行う事ができ、それらが集まって目的とする処理を進めていく。各プロセスは自分の状態を引数の形で持っており、処理の進行によって状態を変化させていく。従って、プロセスは動的なオブジェクトと言う事ができる。

このプログラミング・スタイルはPIMOSでも多く使われており、プロセスに注目したトレースができるとデバッグの効率を上げる事ができると期待される。

しかし、プロセスは概念的なものであり、KL1処理系は、プロセス・レコードのような構造を持って管理している訳ではない。従って、トレーサー等のデバッグ・ツールがプロセスを捉えるのは難しい。

3.2 コードトレースとゴールトレース

PDSSでは、プロセスに注目したトレースをコードトレースとゴールトレースの機能を組み合わせる事で実現した。

コードトレースとは、トレースしたいプログラムコード(述語)が呼び出された時にトレースを行うもの、つまり、静的なプログラム構造を指定してトレースを行うものである。

ゴールトレースとは、各ゴールごとに、その子孫のゴール(すなわちそのゴールの実行によって生成される子ゴール)をトレースするか、あるいはトレースしないかを指定するもの、つまり、動的な実行構造を指定してトレースを行うものである。例えば、

```
foo :- p.      bar :- p.      p :- q, r.
```

のようなプログラムで、foo がゴール・トレースを行う状態にあり、bar がその状態になかったとすると、foo から呼び出される p も、更にそこから呼ばれる q, r もゴール・トレースを行う状態になるが、同じコード p, q, r でも、bar から呼び出されたゴールはゴール・トレースを行う状態にならない。

以上の2つの機能を用いることでプロセスのトレースを行う。プロセスとは、再帰呼び出しにより実現される、実行過程のことである。再帰呼び出しということは、いつも同じコード(述語)を実行していることを意味するので、まず、そのコードだけをトレースするようにコードトレースの設定を行う。しかし、これだけでは同じコードを実行するプロセスが複数同時に存在する場合にはそれらが混ざって表示されるので目的にプロセスに注目するのが難しい。そこで、さらにゴールトレースを行う。プロセスは再帰呼び出しなので、ある1つのプロセスを構成するゴールは全て親子関係になっており、注目したいプロセスのゴールが実行された時にその子孫だけをトレースするという指定を行えば、コードトレースの指定と組み合わせてプロセスのトレースが実現される。

3.3 問題点

以上の方法によりプロセスに注目したトレースを行う事ができた。しかし、この方法でトレースできるのは数個のプロセスまでと考えられる。現在のトレーサはトレース情報を1つのウィンドウに書き出しているので、トレースするプロセスが多くなると、混ざって表示されるトレース情報がどのプロセスに関するものかを区別するのが難しくなる。

これを解決する為にはプロセスごとにウィンドウを開き、トレース情報を分けて表示する機能が必要になる。また、この為には、トレーサーが個々のプロセスを識別し、出力先を選択する機能が必要となる。しかし、処理系がプロセス・レコードのような構造を持って管理している訳ではないので、これを実現するのは難しいと考えられる。これは今後の大きな課題である。

4 メモリ消費量の低減

4.1 メモリ消費量を減らす必要性

KL1は副作用を許さないので、ユーザーがデータを破壊的に書き換える事により、領域の再利用をする事ができない。その為、急速にメモリを消費し、かなり高い頻度で一括型のGCを起動しなければならない。

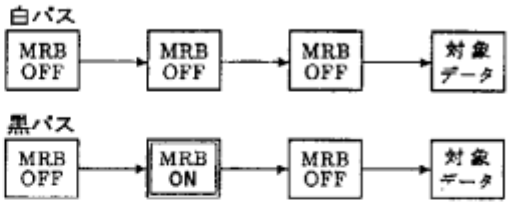

図 3: 白パスと黒パス

そして、GC回数が多くなる事により実行速度が低下すると予想される。

4.2 MRBによる参照数管理

メモリの消費量を減らすには、データの参照数を管理する事により、実行途中で不要となったデータを検出し、回収、再利用することが考えられる。この参照数管理の最も簡単なものがMRB(Multiple Reference Bit) [9] による参照数管理であり、並列推論マシンPIMでも採用されている方式である。これはポインタ側に付けられた1ビットの情報により、それが指しているデータが単一参照であるか、多重参照であるかを区別するものである。

なお、MRB方式では1ビットしか情報がないので一旦多重参照になってしまうと、そのデータを回収する事ができない。従って、一括型のGCも必要となる。PDSSでは一括型GCとしてコピー方式のGCを採用した。

4.2.1 参照パス

MRBはポインタ側に付ける1ビットの情報であるが、KL1では参照のルート(レジスタ等)から数段の参照ポインタが繋がってデータを指す場合がある。この経路を参照パスと呼ぶ。MRBの情報を利用する場合には、この参照パスを構成する各ポインタのMRBの論理和により参照数を判断する事になっている。

以降、本論文では図3のように、その参照パスのポインタのMRBが全てOFFの場合を白パスと呼び、○で表す。また、その参照パスに一つでもMRBがONのポインタが含まれる場合を黒パスと呼び、●で表す。

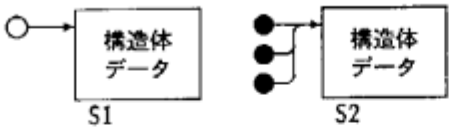

図 4: 構造体参照の MRB 管理

図 5: 未定義変数参照の MRB 管理

4.2.2 MRB 管理の基本

MRBによる参照数管理では、参照パスの各ポインタのMRBを後述する方法によりメンテナンスを行い、常に次のような状態になるように管理する。なお、ポインタが指すデータの種類が構造体なのか、未定義変数なのかにより、管理のしかたが異なる。

ポインタが指している先が構造体の場合には、図4のように参照が1本だけの場合(S1)には白パスにすることが許され、それ以外では全て黒パスとなるように管理している。従って、参照が白パスならば単一参照である事を、黒パスならば他にも同じ構造体を見ている参照があるかも知れない事を意味する。

ポインタが指している先が未定義変数の場合には、図5のように参照が2本以下の場合(U1)には両方を白パスにすることが許され、それ以外では高々1本の白パスと他は黒パスとなるように管理している。この違いは、未定義変数の場合には、変数を具体化するパスと、その値を読み出すパスがあるのが普通であり、特にこれが1本ずつのことが多いことによる。

4.2.3 MRB のメンテナンス

各ポインタのMRBは以下の場合に操作される。

① MRB を ON にする命令

構造体や変数は、最初に作られた時は単一参照であり、白パスで指されている。そして、実行途中で参照数が増えた場合には MRB を ON に変更し、黒パスに変える操作を行う必要がある。このような場面はコンパイラがプログラムを解析する事で検出できるので、MRBをONにするような命令を生成している。

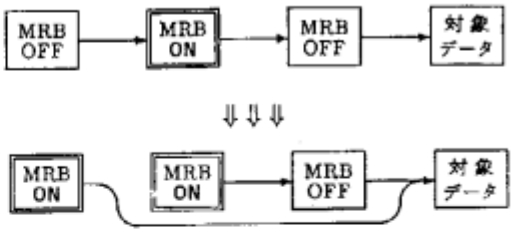

図 6: デレファレンス時の MRB 管理

例えば、

```
foobar(X) :- true | foo(X), bar(X).
```

のような場合、foobar が実行されると、引数で与えられた変数 X は foo と bar の2つのゴールに渡されるので参照数が増える事がわかる。従って、コンパイラは変数 X へのポインタのMRBをONにする命令を生成する。

この命令により、図4のS1の状態の構造体はS2の状態に変化する。また、図5のU1の状態の未定義変数はU2の状態に変化する。

② デレファレンス

デレファレンスとは、複数段のポインタが繋がってデータを指している場合に、そのデータを直接指すようにポインタを繋ぎ替える事である。このとき、参照パスの白黒を変えることはできない。そこで、もとが黒パスであれば、図6のように、データを直接指すようになったポインタのMRBをONにするようにしている。

③ 構造体要素を読む場合

最初、構造体は白パスで作られ、要素のMRBもOFFとなっているが、① の操作により、構造体が黒パス参照に変わる場合がある。しかし、この時には要素のMRBはOFFのままである。ところが、構造体の参照パスが複数あるということは、その要素への参照パスも複数できる可能性があり、要素のMRBをOFFのままにしておく事はできない。

そこで、要素への参照パスは要素を読むことによりできるので、この時に要素のMRBを調整することにした。つまり、黒パスで指された構造体の要素を読み

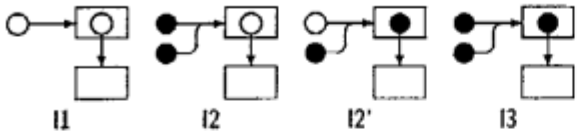

図 7: 変数具体化時の MRB 管理

出す場合には、要素のMRBをONにしてから読み出すことにしてある。

④ 未定義変数の具体化

一般の構造体と未定義変数ではMRBの管理のしかたが違っていたので、未定義変数を具体化する場合にも、MRBのメンテナンスを行う必要がある。

具体化時には未定義変数に具体化データへのポインタを書き込むが、このポインタのMRBに適当な値をセットする必要がある。これを決めるには、「具体化データの参照情報」と「変数参照の参照情報」を用いる。これらが両方とも白パスであれば、MRBをOFFに、一方が黒パスならばMRBをONにする。こうすることにより、未定義変数の場合にだけ許された、白パス2本の状態や複数の参照の中に1本だけ白パスがあるという状態は、白パス1本の状態か複数の黒パスだけの状態に変化する。

未定義変数を白パスで指されるデータで具体化した場合には、図 7のようにしている。I1 は 図 5 の U1 の変数を具体化した場合、I2 は U2 を白パス側から具体化した場合、I2' は U2 を黒パス側から具体化した場合、I3 は U3 を具体化した場合である。未定義変数を黒パスで指されるデータで具体化した場合には、変数のMRBは必ずONとなり、必ず黒パスでデータを指すようにしている。

4.3 メモリの回収と再利用

MRBにより参照数を管理する事により、実時間GCを行う事ができる。白パスで指されている構造体の参照数は1であるので、あるゴールがこのような構造体を参照するのを止めた時には、参照数が0となり、構造体が占めていた領域を回収する事ができる。

この、「参照するのを止める」というのは、コンパイラがプログラムを解析する事により検出できる。例えば、

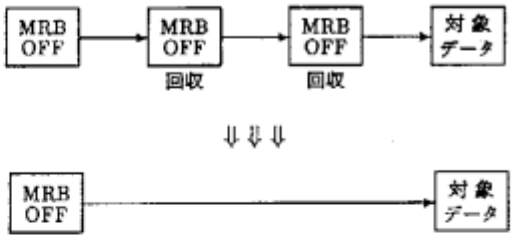

図 8: デレファレンス時の回収

```
foo([H|L],{A,B},_) :- true | foo(H,L,A,B).
```

の場合、第1引数のリスト、第2引数のベクタ、および第3引数はボディ部では使われていないので、参照するのを止めたといえる。

このような状況では、以下のような回収命令を生成し、回収を行っている。これらの命令では、構造体が白パスで指されていた場合には、その構造体が占めていた領域を回収する。なお、未定義変数の場合は白パスでも複数の参照があるので、回収することはできない。

```
collect_list    Ri
collect_vector  Ri
collect_value   Ri
```

なお、これらの命令により回収されたメモリは後で、再利用する為にフリーリストにより管理している。しかし、フリー・リストの操作は手間が掛かるので、同じ述語内で不要になったものと同じ大きさの構造体を割り付けている場合の為に、領域を再利用する命令を用意した。これにより、一旦回収してから割り付ける処理が不要となる。これらの命令では、構造体が白パスで指されていた場合には、そのデータが占めていた領域を新しい構造体を格納する為に利用する。黒パスで指されていた場合には新しい領域を割り付ける。

```
reuse_list      Ri,Rold
reuse_vector    Ri,Rold
```

また、白パスの参照をデレファレンスした場合にも、図 8 のように、途中のポインタが使っていた領域を回収できる。

プログラム	prime10000	queen8	qlay8	bup	KL1/KL1コンパイラ
リダクション数	789093	38878	19419	34857	15263444
デレファレンス時の回収	779091 (0.33)	2899 (0.06)	10109 (0.22)	28549 (0.39)	11102879 (0.03)
KL1-B命令による回収	17692 (0.01)	17836 (0.39)	0 (0.00)	5680 (0.08)	2995298 (0.01)
KL1-B命令による再利用	1538180 (0.66)	2478 (0.05)	0 (0.00)	19286 (0.26)	6452458 (0.02)
構造体の書き換え	0 (0.00)	0 (0.00)	0 (0.00)	0 (0.00)	290672781 (0.90)
最大使用量	30005 (0.01)	22766 (0.50)	35245 (0.78)	20078 (0.27)	500000 * 25 (0.04)
総使用量(累積)	2338506 (1.00)	45969 (1.00)	45353 (1.00)	73503 (1.00)	323476023 (1.00)

表の中の数値の単位はワード数。括弧内は総使用量に対する割合。

表 1: MRB による参照数管理の効果

4.4 構造体の書き換え

KL1 は副作用を許さないので、ユーザーがデータを破壊的に書き換えることはできない。しかし、MRB による参照数管理を用いることにより、ユーザーに見えないレベルで処理系が書き換えを行い、メモリを節約することができる。

この構造体の書き換えは、

set_vector_element(V,P,E,NewV)

のような組込述語で行っているものである。この組込述語は構造体 V の P 番目の要素だけを E に変更した新しい構造体 NewV を作るものであるが、構造体 V が白パスで指されている場合には、それを直接書き換えて、NewV に出力するようにしている。

このようにしても、白パスで指された V に対しては、他のゴールからの参照が無いことが保証されているので V が破壊されても、論理的には正しく見える。

4.5 MRB による参照数管理の効果

実際にプログラムを実行する事で、MRB による参照数管理の効果を調べた。この為に 4 種類のベンチマーク・プログラムと KL1/KL1 コンパイラを使用した。prime10000 は 10000 までの素数生成、queen8 はエイト・クイーン問題、qlay8 はエイト・クイーン問題の別の解法、bup はボトムアップ・パーザーである。また、KL1/KL1 コンパイラは KL1 で書かれた KL1 コンパイラで、自分自身をコンパイルした場合である。これはファイルの入出力、パーザー等の処理を含むもので、実用プログラムと言う事が出来る。

実行結果は、表 1 のようになっている。この表では、最大使用量と総使用量の比が MRB による効果を表している。総使用量は、もし MRB による管理を行っていなかった場合に消費すると考えられる量であり、これが、MRB による管理を行った事により、最大使用量で示される量のメモリだけで実行できた訳である。従って、この値が小さいほど効果が大きい事を示す。なお、KL1/KL1 コンパイラでは処理系の持つメモリ領域(500KWord)を途中で使い切ってしまい、一括型 GC を 25 回行っている。

これを見ると、プログラムによる差が非常に大きい事が分かる。これは各プログラムで単一参照の割合が大きく違う為で、この割合が多いほど効果は大きくなる。単一参照の割合はプログラムの記述のしかたにより、かなり変化するので、プログラム作成時に MRB による管理のことを意識することにより、MRB の効果をより大きくすることができると考えられる。従って、KL1/KL1 コンパイラでは、MRB による管理のことを意識して作成されている為、MRB による管理の効果を十分に引き出すことができたといえる。

回収要因の傾向を見てみると、KL1/KL1 コンパイラは他のベンチマーク・プログラムと違い、ほとんどが構造体の書き換えによるものである。これは、コンパイラがテーブル(構造体)を作業用に使っており、これを頻繁に書き換えている為である。このようなテーブル操作は実用プログラムでは良く行われる操作であり、MRB による参照数管理は実用プログラムにおいて、かなりの効果があると期待される。

以上の事から、MRB による参照数管理を行うことにより、メモリ消費をかなり抑えることができたと考えられる。しかし、これにより、一括型 GC の回数は

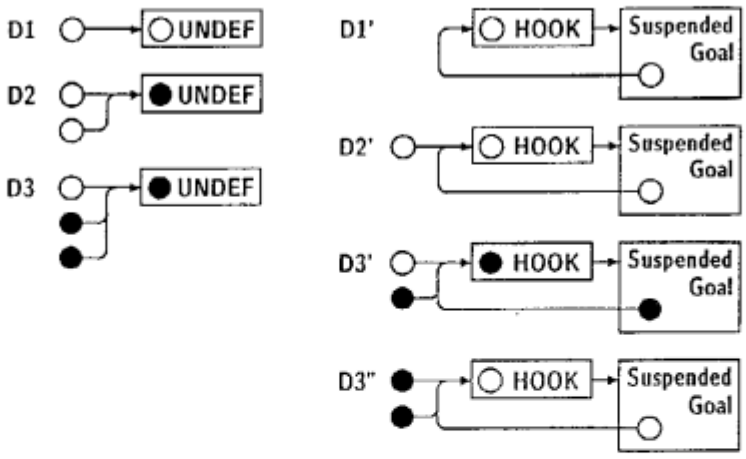

図 9: デッドロック検出の為の MRB の拡張

減少したが、実行速度を早くする効果は少ないと考えられる。PDSS は並列推論マシン PIM と違い、マルチ PE のシステムではないので、一括型の GC をかなり短い時間で行うことができるが、ハードウエアサポートが無いので、回収した領域を管理するためのフリーリスト操作には時間がかかる。

回収時の操作を見てみると、回収要因のなかで、KL1-B命令による再利用と構造体の書き換えはフリーリスト管理を行う必要がないことが分かる。そこで、MRB による参照数管理は行うが、フリーリスト管理を行わない処理系を考えてみる。実用プログラムにおいては、構造体の書き換えの割合が多いと考えられるので、このような処理系でも十分な効果が期待でき、また、手間の掛かるフリーリスト管理をしないので、処理系の簡略化と速度向上が期待できる。

5 デッドロックの早期検出

5.1 デッドロックの早期検出の必要性

KL1 における「デッドロック」とは、あるゴールが未定義変数の具体化されるのを待っているにもかかわらず、その変数が永久に具体化されない為に、実行を進めることができない状態を言う。

ゴールのデッドロックは一括型 GC 時に検出できる [10] が、もしゴールのデッドロックが起きると、そのゴールの実行結果を待っているゴールも芋蔓式に次々とデッドロックしていく性質がある。この為、一括型 GC時の検出を待っていると、非常に多くのデッドロックが一度に報告される事になり、原因の解析が難しくなる。デバッグ効率を上げる為には、なるべく早くデッドロックを検出する必要がある。

5.2 MRB による参照数管理の拡張

PDSS では MRB による参照数管理を拡張することによりゴールのデッドロックを積極的に検出する機構を取り入れることにした。

通常の MRB では参照数だけを管理していたが、ゴールのデッドロックを検出する為には、「変数を具体化する可能性のある参照」の数を管理するように拡張する必要がある。すなわち、ある変数の具体化を待ってサスペンドしているゴールからは、変数への参照は存在するが、この参照を通して変数を具体化することはないので、このような参照を区別するわけである。

5.2.1 未定義変数の MRB の基本

この為に従来使われていなかった未定義変数の MRB を利用する事にした。具体的には、図 9 の左側ように未定義変数の MRB を設定する。ここで、未定義変数のタグ UNDEF はその変数の具体化を待っているゴールがない事を意味する。

未定義変数の MRB は参照パスの白黒と組み合わせて使い、それぞれ次のような意味を持つ。

D1: 所謂 VOID 変数。MRB が O→O (白パスで MRB の OFF の変数を指している) となっており、「変

数を具体化する可能性のある参照」が1本だけである事を表す。

D2: 2個所から参照されている変数。MRBが○→●(白パスでMRBのONの変数を指している)となっており、「変数を具体化する可能性のある参照」が2本ある(自ゴール以外に1本ある)かも知れない事を表す。

D3: 3個所以上から参照されているかもしれない変数。この場合高々1本の白パスの参照があり、MRBは○→● または ●→● となる。

5.2.2 未定義変数のMRBのメンテナンス

未定義変数は最初、図9の左側の何れかの形で生成されるが、サスペンド/リジューム等の処理の進行により変数のMRBも操作する必要がある。

サスペンド時には、具体化を待つ未定義変数のタグをHOOKに変更し、そこから、ゴールレコードを指すようにしている。このとき、未定義変数のMRBに適当な値をセットする必要がある。これは、参照パスの白黒により決定する。もし白パスで指されていた場合には、未定義変数にMRBをOFFに、黒パスで指されていた場合には、未定義変数にMRBをONにする。従って、MRBは図9の左側から右側で示すような状態に変更される。

D1': D1の状態の参照を持っているゴールがサスペンドした状態の変数。この状態になった変数を具体化するゴールは存在しない。

D2': D2の状態の参照の一方を持っているゴールがサスペンドした状態の変数。サスペンドしてない側からは○→○となっているのでD1のVOID変数と同じに見えるようになる。

D3': D3の状態の黒パス参照の1つを持っているゴールがサスペンドした状態の変数。サスペンドしてない側からはD3の状態と変わらない。

D3": D3の状態の白パス参照を持っているゴールがサスペンドした状態の変数。サスペンドしてない側からはMRBが●→○となる。これも「変数を具体化する可能性のある参照」が他にもあるかも知れない事を表す。

リジューム時には従来のMRBのルールでメンテナンスする。

5.3 ゴールのデッドロックの検出

ゴールのデッドロックは「変数を具体化する可能性のある参照」が1本だけ(MRBが○→○, 図9のD1とD2'に相当)の変数について、以下のような操作を行った場合に検出している。

- サスペンド時

  このような変数の具体化を待とうとした場合、他に「変数を具体化する可能性のある参照」が無くなるので、この変数は永久に具体化されなくなる。従って、今サスペンドしたゴールと、もしその変数の具体化を待っている別のゴールが既にあればそれがデッドロックする。

- アクティブ・ユニフィケーション時

  2つのこのような変数どうしをアクティブ・ユニフィケーションした場合、このユニフィケーションの為に双方の変数とも残された参照パスが消費されてしまうので、もしその変数の具体化を待っているゴールが既にあればそれらがデッドロックする。

- collect_value 命令実行時

  このような変数を含むデータを回収しようとした場合、「変数を具体化する可能性のある参照」が無くなるので、その変数の具体化を待っているゴールが既にあればそれがデッドロックする。

まとめ

以上のように、PDSSはデバッグ環境の向上を優先的に考えた、実用的な処理系であり、PIMOSのクロス開発環境として、十分な成果をあげる事ができた。また、汎用計算機上の手軽なKL1処理系として、PIMOS以外のKL1プログラムの開発やKL1の入門用として多く利用されている。

PDSSで提供しているデバッグ機能の中では、ゴールのデッドロックを早期発見する機能が特に有効であった。ゴールのデッドロックはKL1で最も発生しやすいエラーの1つであり、デバッグ効率がかなり向上したと思われる。逆に、プロセスに注目したトレース機能

は不十分であり、この改良が今後の課題として残されている。

謝辞

日頃ご指導頂くICOT第4研究室の内田俊一室長ならびに研究員各位に感謝します。また、富士通研究所の木村康則氏に感謝します。

参考文献

[1] K. Ueda, Guarded Horn Clauses: A parallel logic programming language with the concept of a guard. TR-208, ICOT, 1986.

[2] ICOT第4研究室編 PIMOS機能設計書, TM-503, ICOT, 1988.

[3] T. Chikayama, H. Sato and T. Miyazaki, Overview of the Parallel Inference Machine Opereting System (PIMOS), In Proc. of FGCS'88, Vol 1, pp230-251, 1988.

[4] Y. Kimura and T. Chikayama, An Abstract KL1 Machine and its Instruction Set. In Symposium on Logic Programming '87, pp468-477, 1987.

[5] 木村, 西崎, 中越, 平野, 近山: KL1のクローズ・インデキシング方式, 並列処理シンポジウム 1989, pp187-194, 1989-2.

[6] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 209, SRI International, 1983.

[7] 中越, 平野, 宮崎, 西崎, 宮崎: KL1による簡易OS — Micro PIMOS, 情報処理学会第36回全国大会, 7H-6, 1988-3.

[8] E. Shapiro and A. Takeuchi, Object Oriented Programming in Concurrent Prolog, New Generation Computing, Springer Verlag Vol.1, No.1 pp25-48, 1983

[9] T. Chikayama and Y. Kimura, Multiple Reference Management in Flat GHC. In Proc. of the Fourth International Conference on Logic Programming, 1987.

[10] 平野, 中越, 宮崎, 西崎, 宮崎: 汎用計算機上のKL1処理系におけるメモリ管理とデッドロック検出, 情報処理学会第36回全国大会, 7H-5, 1988-3.

[11] ICOT第4研究室: PDSS — 言語仕様と使用手引 — , ICOT TM-437, 1988.