プログラミング言語 21 － 1
（１９８９．６．１３）

汎用計算機上の KL1 処理系
— 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]

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

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。並列推論マシンを制御するオペレーティング・システムである PIMOS は並列論理型言語KL1で記述されており、この PIMOS の為のクロス開発環境として、UNIX上で作成されたのが、PDSS — PIMOS Development Support System である。 PDSS は開発用ツールとして、各種のデバッグ・ツールやエラー検出機能を提供しており、特に、MRB(Multiple Reference Bit)によりデータの参照数を管理する事で、ゴールのデッドロックを検出する機能はデバッグ効率の向上に有効であった。本報告では、この処理系の実現方式を中心に説明する。

The implementation of the KL1 system on the UNIX: PDSS

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

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

The parallel inference machine "PIM" is being developed in the FGCS project, and it's operating system "PIMOS" is described with the KL1 language. We developed the KL1 system on the UNIX as the cross environment to develop PIMOS. It is PDSS: PIMOS Development Support System. PDSS include many debugging tools and error detection mechanism. Especially, the eager deadlocked goals detection mechanism is usefull. It is based on the reference management to the data structure by MRB (Multiple Reference Bit). This paper explain the implementation of PDSS.

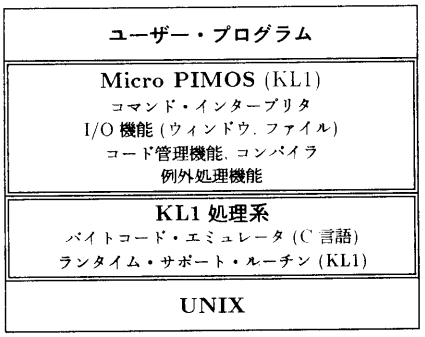

図 1: PDSS の全体構成

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。並列推論マシンを制御するオペレーティング・システムである PIMOS [1] は並列論理型言語KL1で記述されており、この PIMOS の為のクロス開発環境として、UNIX上で作成されたのが、PDSS — PIMOS Development Support System である。従って、PDSS のKL1の仕様は並列推論マシンのものと極力互換性を持つように決めてある。

PDSS は開発用ツールであるので、作成にあたっては、デバッグ機能を最も重視した。その為、各種のデバッグ・ツールやエラー検出機能を提供することにした。これには、スケジューラを拡張する事によって、単一PE処理系でありながら、ゴールのスケジューリングを非決定的にすることにより、実際の並列推論マシンにおける実行順序の非決定性によるバグを発見する機能や、データの参照数を管理する事により、ゴールのデッドロックを早い時期に検出する機能等がある。

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処理系の部分であり、もう一つは Micro PIMOS [6] と呼ばれるKL1自身で書かれたユーザー・インタフェース等を行う部分である。なお、本論文では Micro PIMOS の部分については割愛する。

1 並列論理型言語 KL1

1.1 KL1 の概要

KL1は、AND並列の論理型言語 Flat GHC [8] に基づいてICOTで設計された言語であり、OS記述、アプリケーション記述の為の機能が拡張されている。この拡張にはプログラムのモジュール化機能と荘園、プラグマがある。

KL1プログラムは次のようなシンタックスを持つ節の集合として表される。

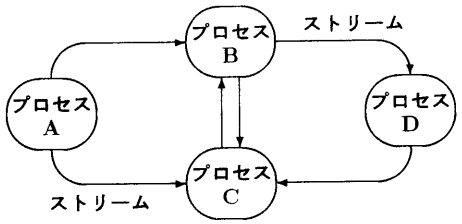

図 2: プロセスとストリーム通信

$$H :- G_1, G_2, \cdots, G_m \mid B_1, B_2, \cdots, B_n.$$

ここで、H, G_i, B_i はそれぞれ、ヘッド部、ガードゴール、ボディゴールと呼ばれ、オペレータ '|' はコミットメントバーと呼ばれる。また、節の中でコミットメントバー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。ガードゴールには組込述語しか書く事ができないが、これは効率的な実現を考慮して採用された制限である。

受動部にはそのクローズを選択する為の条件が記述されており、各候補節は並列に(任意の順序で)その条件が検査され、最初に成功した節のみが選択される。選択された節の能動部は並列に(任意の順序で)実行される。

なお、ゴールの入力に未定義変数がある為に、クローズ選択ができない場合には、そのゴールは実行を継続する事ができないので、変数が具体化されるまで待たされる。これをサスペンドと呼ぶ。また、変数が具体化された為に、それを待っていたゴールが実行可能な状態になる事をリジュームと呼ぶ。

1.2 プロセスとストリーム通信

これはKL1における代表的なプログラミング・スタイルの1つである。

KL1における「プロセス」[9] 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。従って、これ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。

「ストリーム通信」とは、複数のプロセスの間で共有される変数を用いて行われる通信である。これは通常、変数にリスト構造を具体化することによって実現されている。具体化されるリスト構造のCARにメッセージを書き、CDRには次の通信の為の新しい変数を用意する。これを順次繰り返すことによりプロセス間の一連のメッセージ通信が実現される。

「プロセスとストリーム通信」のスタイルでは、複数のプロセスがお互いに通信を行いながら、処理を進めていく。各プロセスは自分の状態を持っており、処理の進行に従って、状態を変化させていく。これは、オブジェクト指向風のスタイルとみなすこともでき、プログラムのモジュラリティを高め、大規模なプログラム開発に有効なプログラミング・スタイルと言うことができる。

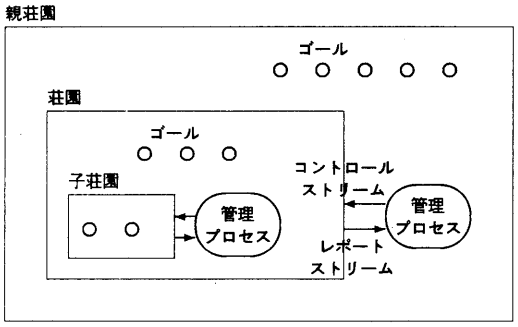

図 3: 荘園のイメージ

1.3 モジュール化機能

モジュール化機能は幾つかの述語定義(現在は1つのファイル)を纏めて扱う機能であり、これを単位として、コンパイルやデバッグが行われる。また、各モジュールはユニークな名前を持つ。

モジュール内で定義された述語は、通常は同一のモジュール内でのみ利用する事ができる。すなわち、述語名の空間はモジュールごとに区別されることになる。そして、他のモジュールから使えるようにする場合には、外部に公開する為のPublic宣言が必要になっている。また、それを利用する側からは、必ずモジュール名と述語名を組にして参照する必要がある。

1.4 荘園

荘園とはKL1プログラムの実行管理や資源の管理を行う為の基本機能であり、そこに接続されたストリームを通じて、その内部のゴール群の実行を制御/観察する事ができるようになっている実行管理単位である。また、荘園をネストさせる事もできる。

荘園は2本のストリームを持っている。コントロール・ストリームは荘園への入力ストリームで、ここにメッセージを流す事により、ゴール実行の中断、再開、放棄を制御したり、資源を割り付けたり、資源の消費状況を調べる事ができる。レポート・ストリームは荘園からの出力ストリームで、ここを監視する事により、実行の終了や例外(リダクションやユニフィケーションの失敗等)の発生を知る事ができる。

PIMOS等のOSは、この荘園の機能を用いてユーザー・タスクの制御を行う。すなわち、OSはユーザー・タスクごとに荘園を生成し、この荘園が持つ2本のストリームをOSのタスク管理部が管理する事によりタスク制御を実現している。

荘園の機能はOSだけでなく、一般のユーザー・プログラムからも利用する事ができる。この場合には、OSのタスク管理部の代わりに、荘園を管理する部分をユーザーが記述する必要がある。

1.5 プラグマ

プラグマはユーザー・ゴールの呼び出しに付加する事により、そのゴールの実行を制御する為のものである。ただし、これは補助的なものであり、プログラムの意味を変えるものではない。プラグマは主に実行効率を向上する為に使われる。現在、プラグマとして実行優先度と負荷分散の指定を行う事ができる。

実行優先度の指定は、荘園内割合指定と自己相対指定の2通りが用意されている。荘園内割合指定は、所属荘園で許される上限と下限の間のどのあたりにするかを割合で指定するもので、自己相対指定は、親ゴールの優先度を基準として、上げるか、下げるかを指定するもので、この場合も所属荘園で許される上限と下限を越えないようになっている。

負荷分散の指定は、複数PEの処理系の場合に、どのPE上で実行すべきかを指定するものである。PDSSは単一PEの処理系なのでこの指定は無視している。

2 KL1 処理系

2.1 KL1-B

KL1のプログラムはKL1-Bと呼ばれる抽象機械命令にコンパイルされ実行される。このKL1-BはPrologにおけるWAM [10] 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ではこのKL1-Bをバイトコードの形でメモリ上に置き、C言語で記述されたエミュレータにより実行する方式にした。バイトコード・エミュレータ方式なので、実行速度はあまり速くできないが、色々なデバッグ機能/統計情報収集機能を持たせる事ができ、例えば、KL1-B命令ごとの実行回数をカウントする機能や、メモリの消費/回収状況を調べる機能を持っている。これらにより、処理方式を改良する上で、役に立つ情報が得られた。

2.2 ゴール管理

KL1のゴールはゴール・レコードという構造体で表し、実行可能なものはレディ・ゴール・プールで、待ち状態にあるものはサスペンション・キューで管理する事にした。

レディ・ゴール・プールは、4096段階の実行優先度に分割し、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た。また、プールにゴールを入れる場合には、原則的に、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、プラグマにより優先度が指定された場合には、その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。

PDSSでは、この他に、トレーサーのコマンドにより、ゴールをレディ・ゴール・プールの末尾に入れる事

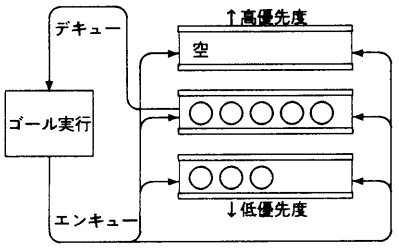

図 4: レディ・ゴール・プール

もできるようにした。これにより、ゴールのトレースを行っている時に、ユーザーからのコマンドで、そのゴールをプールの末尾に入れ、ゴールの実行を意図的に遅らせられる事ができる。

さらに、PDSSでは、擬似乱数により一部のゴールをプールの末尾に入れる事もできるようにした。これは、PIMのようなマルチPEの場合に発生する実行順序の非決定性をシミュレートするものである。実行順序に非決定性がある場合、KL1プログラムに実行順序に依存した部分があると、ある時には動いたプログラムが、別の時には動かないという事になる。PDSSは単一PEの処理系なので、本来、このような非決定性は無い。そこで、このようなバグをPDSS上で発見する為に、乱数による実行順序の制御を導入した。なお、これは擬似乱数を利用した為に、再現性があり、デバッグがし易くなっている。

サスペンションキューは変数にその具体化を待っているゴールを繋いでおくもので、変数が具体化された時に、待っていたゴールを速やかにレディ・ゴール・プールに移動する事ができるようになっている。

2.3 荘園木管理

荘園は処理系内部では荘園レコードという構造体で表し、親子、兄弟関係のポインタにより木構造にしている。KL1処理系は、これらのポインタを利用して、資源の管理等を行っている。また、荘園は、そこに含まれるゴールの数を管理しており、この数が0になることにより、荘園の成功(終了)を検出できるようにしている。

荘園は2本のストリームを持ち、それにより管理プロセスと通信を行うようになっているが、この通信部分をC言語で直接記述するのは難しいので、PDSSでは、この部分をKL1により記述している。従って、荘園の生成は組込述語の形式になっているが、実際には、KL1で記述された荘園ライブラリを呼び出すマクロになっている。そして、このライブラリは、荘園のコントロール・ストリームに送られてくるメッセージを解釈し、それに対応する特別な組込述語により荘園木を操作するようにしている。

2.4 ソフトへの割り出しとエラー報告

PDSSでは、バイトコード・エミュレータ(これはハードウエアに相当する)から、KL1で書かれたソフトウエアを呼び出す場合がある。これを「ソフトへの割り出し」と呼ぶ。このような処理は、ボディの組込述語を実行しようとした時に、入力引数が具体化されていない為に、その組込述語がサスペンドさせる場合等に行われる。

この処理の時には、対応するランタイム・サポート・ルーチンを実行するゴールを生成する。しかし、このようなゴールを普通のユーザー定義述語のゴールと全く同じに扱ってしまうと、この中で例外が発生した時に、「ランタイム・サポート・ルーチンで実行した述語で例外が発生した。」という事しか知る事ができなくなる。

特に、このようなゴールは、親ゴールが実行されてから暫く経ってから実行される事が多く、どのユーザー定義述語から呼び出されたものなのか、分からなくなる場合が多く、デバッグが困難となる。

そこで、PDSSでは、このようなゴールを区別できるようにゴール・レコードに特別な情報を付加し、どのモジュールの、どの述語から呼び出されたものであるかを知る事ができるようにした。これにより、組込述語で例外が発生した場合には、何時でも、それを呼び出したユーザー定義述語が何であるかを表示できるようになり、デバッグ効率を良くする事ができた。

3 トレーサー

PDSSではKL1-Bエミュレータのレベルでトレース機能を提供した。このトレーサーは基本的にゴール単位のトレースであり、ゴールが次のような状態になった時にトレースを行う事にした。

- ゴール呼び出し
- 具体化を待つ為の中断(サスペンド)
- 中断からの復帰(リジューム)
- リダクションの失敗
- スワップ・アウト(割込み、もしくはより高い優先度のゴールがスケジュールされたことによる)

3.1 コード・トレースとゴール・トレース

KL1は、並列言語であるので、トレースの方法としては「コードに注目したトレース」と「ゴールに注目したトレース」の2通りを考える必要がある。

コードに注目したトレースとは、トレースしたいコード(プログラム = モジュール、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。

ゴールに注目したトレースとは、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。

3.2 トレース・ゴールの絞り込み

トレーサでは通常、トレースする範囲を何らかの方法で絞り込む必要があり、PDSSでは、以下のような方法を採用した。

コード・トレースでは、モジュール単位でトレースを行うか行わないかを指定するのを基本とした。また、更に範囲を狭くする為に、トレースを行う状態のモジュールのうち、特定の述語だけをトレースする機能、コード・スパイ機能を持たせた。

ゴール・トレースでは、最初は、全てのゴールがトレースを行う状態とし、途中で指定したゴールだけが、トレースを行わない状態に変更できるようにした。また、更に範囲を狭くする為に、トレースを行う状態のゴールのうち、特に指定したゴールの子孫だけをトレースする機能、ゴール・スパイ機能を持たせた。

PDSSで実際にトレースされるゴールは、これらの組み合わせで決める事にした。基本的には、コード・トレースとゴール・トレースを両方とも行う状態である事が前提であり、スパイ機能を利用する場合には、以下のような4通りの指定ができるようにした。

- コードがスパイされている。
- ゴールがスパイされている。
- コードかゴールの少なくとも一方がスパイされている。
- コードとゴールが両方ともスパイされている。

3.3 プロセスのトレース

KL1における代表的プログラミング・スタイルの1つである「プロセスとストリーム通信」のスタイルで記述されたプログラムのデバッグでは、「プロセス」に注目したトレースができる事が望ましい。

しかし、KL1のプロセスは概念的なものであり、具体的なプロセス・レコードというようなものは存在しない。そこで、コード・トレースとゴール・トレースを組み合わせることにより、プロセスに注目したトレースができるようになっている。

プロセスは通常、1つまたは数個の述語から成る再帰呼び出しのループで実現されている。従って、それを構成する述語をコード・スパイの対象に、プロセスのゴールをゴール・スパイの対象に設定し、コードとゴールが両方ともスパイされているものだけを見るモードでトレースする。このような指定によりプロセスのトレースを可能としている。

3.4 変数のモニタリング

変数のモニタリング機能はプロセス間を繋ぐ「ストリーム」を監視する為に用意したツールである。これは、変数が具体化されるタイミングでその値を監視する機能であり、変数がストリームの場合はそこを順次流れるメッセージを順次監視できるようにしてある。

これとプロセスのトレース機能を用いる事により、「プロセスとストリーム通信」のスタイルで記述されたプログラムを効率良くデバッグすることができる。

この機能はモニタリングする変数の具体化を待ち、値を表示するゴールを作り、最も高い優先度で実行する事により実現している。具体的には、トレーサーにより変数のモニタリングが指示されると、ユーザーに見えない形で以下のようなプログラムを実行するゴールを作っている。

```
monitor([H|L]):- wait(H), display(H) | monitor(L).
otherwise.
monitor(X):- wait(X), display(X) | true.
```

4 MRBによる参照数管理

KL1は副作用を許さないので、データを破壊的に書き換える事ができない。その為、急速にメモリを消費し、かなり高い頻度でGCを起動しなければならない事が予想される。

そこで、データの参照数を管理する事により、不要となったデータを回収、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) [11] による参照数管理である。これは1ビットの情報により、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。KL1では多くのデータが単一参照であると考えられ、たとえ1ビットのMRBでもメモリ消費を抑える事が期待される。

なお、MRBでは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ので、一括型のGCが必要である。PDSSでは、一括型GCとして、コピー方式のGCを採用した。

4.1 MRBのメンテナンス

MRBはポインタ側に付ける1ビットの情報である。KL1では参照のルート(レジスタ等)からポインタが何段か繋がってデータを指す場合があり、その参照パスのポインタの全てのMRBがOFFの場合を白パス(図では○で表す)、一つでもMRBがONのポインタが含まれる場合を黒パス(図では●で表す)と呼ぶ。

MRBによる参照数管理では、ポインタが指すデータの種類(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5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。

MRBの管理は専用のKL1-B命令により行われる。すなわち、コンパイラがプログラムを解析し、参照数が増

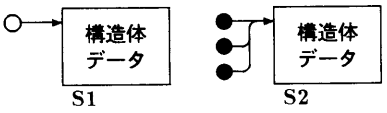

図 5: 構造体参照の MRB 管理

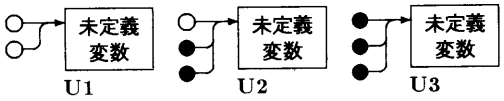

図 6: 未定義変数参照の MRB 管理

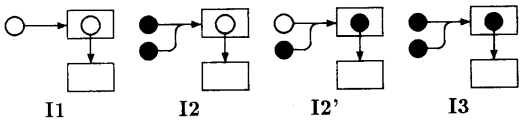

図 7: 変数具体化時の MRB 管理

えるところで、MRB を ON にするような命令を生成している。

ポインタが指している先が未定義変数の場合には、図 6 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。

未定義変数を具体化する場合には MRB のメンテナンスを行う必要がある。これは、「具体化データの参照情報」と「変数参照の参照情報」との OR、すなわち、両方とも白パスならば OFF を、一方が黒パスならば ON を変数の MRB に書き込むことにより行う。こうすることにより、図 6 の状態が、図 5 の状態になる。

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 6 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。

4.2 実時間 GC

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分かるので、以下のような回収命令を生成し、それを実行する事により回収を行っている。

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る場合には、一旦回収してから割り付けるのではなく、領域を再利用する命令を用意した。

reuse_list	Ri,Rold
reuse_vector	Ri,Rold

また、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は、そのデータを直接指すようにポインタを繋ぎ替える (デレファレンスする) ときに、途中のポインタが使っていた領域を回収している。

4.3 構造体の書き換え

構造体の書き換えは、

set_vector_element(V,P,E,NewV)

の様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、他からの参照が無いので、**V** が破壊されても、論理的には正しく見える。

4.4 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、図 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 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 (500KW) 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多重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少なくなる。実

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比べた割合。

表 1: MRB による参照数管理の効果

用プログラムであるKL1/KL1 コンパイラでは、この値がかなり小さく、MRBの効果が大きかったと言える。

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。これは実用プログラムにおける一般的な傾向と考えられる。

以上の事から、MRBによる参照数管理は実用プログラムにおいて、かなりの効果が期待できると考えられる。特に、構造体の書き換えによる効果が大きいことは、手間の掛かるフリーリストによるメモリ管理をしなくても、十分な効果がある事を意味し、処理系の簡略化と速度向上が期待できる。

5 MRB によるデッドロック検出

KL1 における「デッドロック」とは、あるゴールが未定義変数が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。

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ている。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 [12] 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。MRB方式のデッドロック検出では、最初のデッドロックが発生した時点で検出できる可能性が大きく、デバッグを行い易くできる。

5.1 変数を具体化する参照の管理

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。

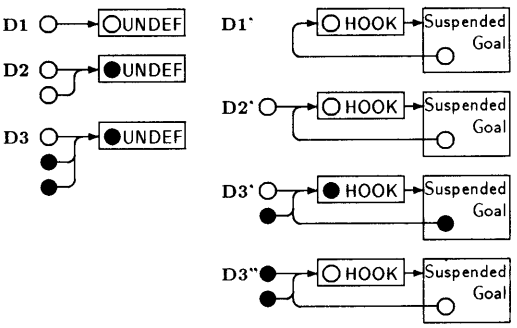

図 8: デッドロック検出の為の MRB の拡張

この為に従来使われていなかった未定義変数セルのMRBを利用する事にした。具体的には、図 8 のように未定義変数セルの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、HOOK はある事を意味する。

未定義変数セルの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○→○(白パスでMRB=OFF の変数セルを指している) 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=ON の変数セルを指している) 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 (自分以外に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 未定義変数セルの MRB メンテナンス

未定義変数セルは最初、図 8 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セルのMRBを操作している。

サスペンド時には参照パス側のMRB(従来と同じルールでメンテナンスされている) を変数側のMRBにコピー

する。従って、MRBは図8の右側で示すような状態に変更している。

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 O→O となっているので **D1** のVOID変数と同じに扱われる。

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 ●→O 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。すなわち、リジューム時には変数セルであったものは既に具体化されているので、従来のMRBルールによりON/OFFを設定している。

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 O→O. 図8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、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、もしその変数の具体化を待っているゴールが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、その変数の具体化を待っているゴールが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てあり、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。

また、PDSSの処理系はほとんどがC言語で記述されている為に、改造が簡単であり、現在は、KL1の新しい処理方式の評価や改良の為のツールとしても利用されている。

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] 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.

[2] 佐藤, 近山, 杉野, 瀧: PIMOSの概要 —並列推論マシン用オペレーティング・システムの構築—, 情報処理学会第34回全国大会, 2P-8, 1987-3.

[3] 佐藤, 越村, 近山, 藤瀬, 松尾, 和田: PIMOSの資源管理方式, 並列処理シンポジウム '89, pp267-274, 1989-2.

[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 '89, pp187-194, 1989-2.

[6] 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.

[7] R. Stallman, GNU Emacs マニュアル, 共立出版, 1988.

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

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

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

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

[12] 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.

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