TR-189

Multi-PSI における Flat GHC の実現方式

宮崎敏彦、瀧　和男

June. 1986

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

multi-PSI におけるFlatGHC の実現方式

宮崎敏彦　　渡　和男
（財）　新世代コンピュータ技術開発機構

（1）はじめに

multi-PSI は、並列論理型マシンに係わるソフトウェア上の研究を進めるための研究開発用ワークベンチであり、ICOTで開発した PSIマシン複数台を格子状のネットワークで接続したハードウェア構成をとる。［渡86］

multi-PSI 上で進めようとしている開発項目には次のものがある。

(a) 並列論理型言語KL1(Kernel Language version 1)の言語仕様と並列マシン上での処理系の実現方式（このうち言語仕様には、中核部分，下位の機械語及び上位のユーザ言語等の仕様に関する研究が含まれる）。

(b) 並列推論マシン用の実行管理，資源管理方式とそれを取り行うオペレーティング・システムPIMOS(Parallel Inference Machine's Operating System)の実現方式（並列マシン上での負荷の分配，通信の局所性の管理，メモリ管理，ガベージ・コレクション等の方式の研究が含まれる）。

(c) 並列推論マシン用のデバッグ方式とデバッグや計測・評価を行うための支援システムの実現。

(d) ベンチマーク・プログラムとして利用可能な応用プログラムの開発。

これらはいずれも、将来の大規模並列推論マシンの為に、大規模な応用ソフトウェアを、(i) 充分に高い並列性を内在するように記述し，(ii)それをデバッグし，(iii) 実際に並列推論マシン上で効率良く実行する為の諸方式をソフトウェアの面から研究開発しようとするものである。multi-PSI システム上でこれらの研究開発の土台あるいは出発点となるものが、並列マシン環境（分散環境とも呼ぶ）をサポートする並列論理型言語 KL1の処理系である。現時点では、 KL1の言語の中核仕様としてFlatGHC が考えられている。

FlatGHC とは、並列論理型言語 GHC［上田86］に対して、ガード部分でのユーザ定義述語の呼び出しを禁止するなど幾つかの制限を加え、処理系の実現を容易にしたものである。現在は KL1のプロトタイプとして、FlatGHC そのものを試験実装中であり、必要に応じてオペレーティング・システムの記述に必要な機能などを追加していく予定である。実装に当たっては実用性を重視し、ユーザ・インタフェースやデバッグ機能など、使い易さ，処理系の信頼性，性能等に注意を払っている。

以下、本論文では、試験実装中のFlatGHC 処理系について、その基本的な実現方式、特にマルチ・プロセッサ環境に係わる実現方式に関して、プロセッシング・エレメント（以下ＰＥと略す）内の処理、ＰＥ間の処理、ＰＥ内とＰＥ間にわたるユニフィケーション処理等について論じ、最後に検討中の課題についても触れる。

（2）計算機モデル

本処理系の実現方式を設計する上で仮定した並列計算機のモデルは次のようなものである。

(a) 各ＰＥはローカル・メモリを持つ。

(b) 各ＰＥはうえられたFlatGHC のゴールを実行する機能を持つ。

(c) ＰＥ間では共有メモリを持たない。従ってデータ構造を共有する場合には別ＰＥ内部を指すポインタが生じる。

(d) ＰＥ同士はネットワークで接続される。

(e) ＰＥ内のアドレス系とＰＥ間にまたがるアドレス系は異なる。即ち、ＰＥを跨るポインタは、各ＰＥの出入口で系の変換が施される。

このモデルは、将来の並列推論マシンが階層構造を持つ場合の上層部分に注目したモデルの一つである。階層型の並列推論マシンの場合には、ここで言うＰＥは、下層構造の例えば密結合マルチ・プロセッサからなるクラスタと置き換え得るものである。

共有メモリを置かずネットワーク接続を仮定したことは、ＰＥ（クラスタ）同士が実装上密結合しにくい程度に個々のＰＥ（クラスタ）が大規模になる可能性があること、また逆に密結合でないがゆえにＰＥの数の変更が容易であることなどを考慮したものである。またＰＥ内外でのアドレス系の分離は、第一にはＰＥ内とＰＥ間のガベージ・コレクション（以下ＧＣと略す）を分離しＧＣの実現の見通しを良くするためであり、第二に、全系を同一のアドレス空間とした場合にＰＥ台数がアドレスのビット幅で制限されるような制限を緩和する目的からである。

この様なモデルを採用した理由は、multi-PSI 上で研究しようとする負荷分散方式などを、将来の大規模並列推論マシンの上層構造に関する実行制御方式として適用しよう

とする意図からである。

ここで述べるFlatGHC の実現方式では、処理の基本的な単位をFlatGHC におけるゴールのレベルに設定している。即ち、各ＰＥは与えられたゴールを、自身のＰＥ内で実行するに十分な機能を持つ。（ここで言う実行とは、ゴールから呼び出される各候補節のガード部の実行と、その中で選ばれた節のボディ部の展開までを言う。但し、その後のボディ部の全てのゴールの実行も含める場合がある。）ＰＥ間のネットワーク上に流れるメッセージは、ゴールの実行依頼，ガードの実行に必要な変数の値の問い合わせ等である。

（３）ＰＥ内の処理の概要

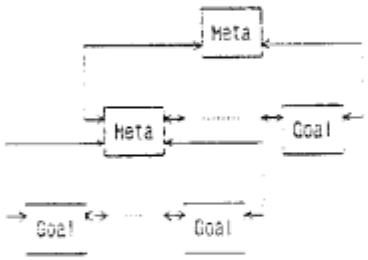
本章では、各ＰＥ内でのゴールの実行と管理に付いて述べる。

（３－１）ゴールの管理

全てのゴールは図．１に示すようなＡＮＤ木と呼ぶ、双方向ポインタで繋がれた木構造で管理される［上田84］，［宮崎85］。この様な管理を行う理由は、

(a) システム・プログラム等をFlatGHC で書こうとした場合、ユーザ・プログラムの失敗がシステムに影響を及ぼさないようにするために、失敗をFlatGHC 自身の枠組みの中で扱える必要がある。このため、ここでは［Clark 84］で提案されている３引数のメタコールを採用した。このメタコールの実現方式として、ゴールを後で説明するような木構造で管理することが考えられる。

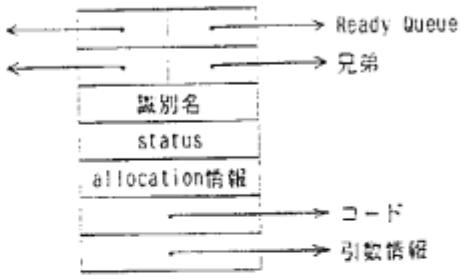
(b) デッドロックを検出しようとした場合，対象となる「仕事」* をつかさどっているゴール群を認識できなければならない。



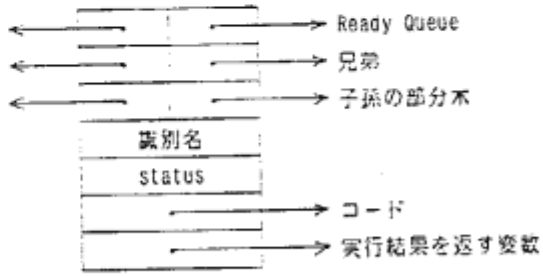
図．１　ＡＮＤ木

---

* 「仕事」：メタコールの元で実行されるゴールの実行過程。狭義ではOSの管理下で直接実行されるメタコールの実行過程。



図．２　ゴール・レコード



図．３　メタコール・レコード

(c) 「仕事」を中断したい（例えば無限ループに陥ってしまった）場合に、どのゴールを中断すべきか認識できなければならない。

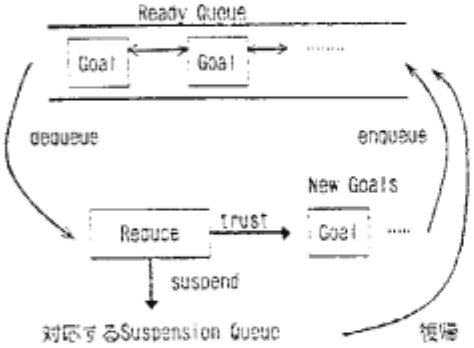
等が挙げられる。

ＡＮＤ木の各部分木の根はメタコールであり、葉はそのメタコールの下で解かれているゴール（あるいはメタコール）に対応する（最も上位に位置するメタコールはシステムが管理しているメタコールである。）。

実際にはＡＮＤ木の各ノードは、それぞれ一つのメタコールあるいはゴールに対応した、メタコール・レコードあるいはゴール・レコードと呼ぶレコードで表される。

メタコール・レコードの形式及びゴール・レコードの形式をそれぞれ図．２と図．３に示す。各フィールドの意味は、

- Ready Queue ：Ready Queue に繋がる時のポインタ。
- 兄弟：ＡＮＤ木における兄弟のレコードを指すポインタ。
- 識別名：ゴール・レコードでは、そのゴールが所属する直接の親のメタコールと同じ。各メタコール・レコードはそれぞれMulti-PSI システム全体で一意に決まる識別名を持つ。
- allocation情報：ゴールを他のＰＥに送出する時に使われる情報を格納。
- status：レコードの状態(ready,suspended,dead)及び属性を格納。属性としては、そのゴールの優先順位，Execution Bound ，デバッグ情報等がある。



図．4　PE内の処理概要

・コード：そのレコードが実行すべきコード（正確には、コードへのポインタが入る間接語）を指す。間接語にはコードを識別するための識別名と、コードがロード中である場合のフラグが付いている。メタコール・レコードのコードはコントロール・メッセージ処理用のコード。

（３－２）ゴールの実行

ＰＥ内における処理の概要を図．４に示す。ＰＥ内にはReady Queue と呼ぶ優先順位付けされた数本のキューがあり実行可能状態にあるゴールが繋がれている。スケジューラは、これらReady Queue のより優先順位の高いキューのゴールを先に実行し、低いキューのゴールはそれよりも高い順位のキューが空になったとき実行する。また、スケジューラはゴール呼び出しの合間に外部割り込みの検出と割り込み処理の振り分けも行う。

スケジューラから呼び出されたゴールはその候補節のガード部を節の並びに応じて上から順に逐次実行する。そして、ある節のガード部の実行に成功した場合はその節のボディー部のゴールを実行可能状態にしてReady Queue に繋げ、スケジューラに制御を戻す。もちろんこのとき新たに生成されたゴール・レコードは呼び出し元のゴール・レコードの巻わりにＡＮＤ木に登録される。またガード部の実行の途中で中断が発見された場合は、中断の原因となった変数をTrail Stack と呼ぶスタックに積み次の節を実行する。失敗した場合も同様に次の節を実行する。全ての節を試みた結果（つまりどの節も選ばれなかった場合）Trail Stack に変数が積まれていればゴールの中断として、後述するSuspension Queueにそのゴールを繋げる。Trail Stack が空の場合ゴールの実行は失敗であり、失敗処理を行う。

Trail Stack を用いて中断処理を遅らせるのは、それ以後の節がガード部の実行に成功し選ばれた場合、その中断処理が無駄になるのを防ぐためである。

（３－３）ゴールの中断と復帰

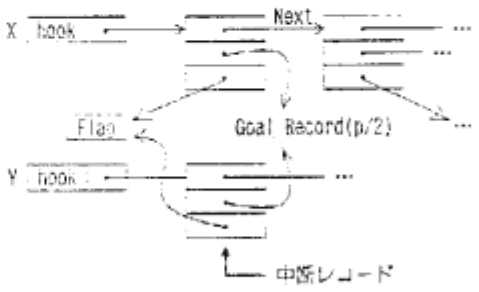
ゴールの実行が中断した場合、そのゴールは中断の原因となった変数が持つSuspension Queueに繋げられる。（この様にゴールをSuspension Queueに繋げることを以下では「変数にフックする」と言うことがある。また、Suspension Queueを持つ変数をhooked variable と呼ぶ。hooked variable は論理的には未定義変数と同じである。）

Suspension Queueは以下の要素から成る中断レコードの列で構成される。

・Next Record
　同一の変数が原因で中断しているゴールを繋げる為、次の中断レコードを指す。
・Goal Record Pointer
　この変数が原因で中断したゴールのゴール・レコードを指す。
・Flag
　中断レコードが指しているゴールが、他の変数が具体化されたことによって、既に復帰（後述）された事を示すフラグ。すなわち、フラグが既にオンの場合そのゴールの復帰は必要ない。

図．５に例を示す。図において、変数 X, Yは共に未定義変数であり、ゴールp(X,Y)はどちらの節のガードも突破出来ず、中断する。この場合、中断の原因となった変数は X と Yの両方であり、ゴール pは両方のSuspension Queueに繋がれ、Flagを共有する。（Suspension Queueの最適化として、中断の原因となった変数が只一つの場合は、ゴール・レコードを直接 Queueに繋げることも考えられる。）

```
Goal:   :- …, p(X, Y), …
Clause: p(a, A) :- true | true.
        p(B, b) :- true | true.
```



図．５　Suspension Queue

hooked variable が具体化されると、そのSuspension Queueに繋がっていた全てのゴールは、そのゴールを指す中断レコードのFlagがオンになっていなければ、Ready Queue に繋ぎ変えられる。このようにフラグを用いて非他制御を行なった場合、不必要になった中断レコードの回収はGCにまかせなければならない。これに対して、同一フラグを共有する中断レコードをループ構造にチェインし、復帰の際に他の中断レコードも回収してしまう方法も考えられる。[Mierowsky85] [江崎86]

（４）ＰＥ間の処理

本章ではアドレス系の変換，ＰＥ間に拡張されたＡＮＤ木，ゴールの送り出し等、ＰＥ間にまたがった処理の概要に付いて述べる。

（４－１）アドレス系の変換

２章でも述べたように、本稿で想定している並列計算機ではＰＥ内のアドレス系とＰＥ間にまたがるアドレス系は異なっている。これは即ち、あるＰＥ内のポインターを他のＰＥに渡す場合にはアドレス系の変換を必要とすることを意味する。

ＰＥ間のアドレス系におけるポインタをここでは外部参照ポインタと呼ぶ。ＰＥ内のアドレス系ではこのポインタは図．６に示すような構造体で表される。図．６において、“代替え変数セル”は参照先の内容が読み込み中であるか否かを示すフラグとして使われる（使用法の詳細は５章で述べる）。“参照先ＰＥ番号”はこの外部参照ポインタが指すＰＥの識別番号であり、“ＰＥ内識別番号”はそのＰＥ内のアドレスを一意に決る識別番号を意味する。各ＰＥの出入口にはこの識別番号をＰＥ内のアドレスに変換する変換テーブルがあり、変換処理はこのテーブルを使って行われる。

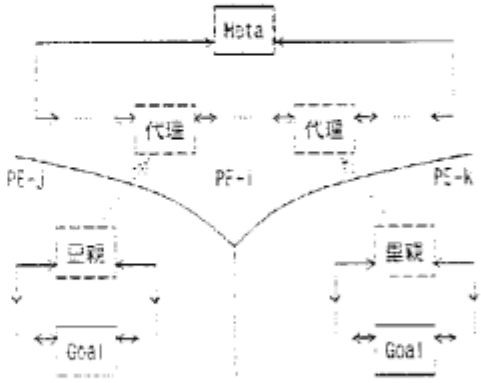
変換テーブルはその用途に応じて、輸出テーブルと輸入テーブルが用意されている。輸出テーブルはそのＰＥ内へのポインタを変数の値、あるいはゴールの引き数情報として送る必要がある場合に用いられる。具体的には、ＰＥ外に輸出するデータ内に未定義変数が含まれていた場合、その未定義変数のセルへのポインタとそのポインタに与えられる識別番号の組が輸出テーブルに登録され、輸出先ＰＥには識別番号と輸出元ＰＥ番号からなる外部参照ポインタが渡される。輸入テーブルには返信（例えばゴールの実行結果，変数の値の問い合わせに対する答え等）が必要なメッセージを他のＰＥに送る際にその返信先アドレスが登録される。このことからも解るように、輸入テーブルの各エントリーは対応する返信があった時点でテーブルから削除することができる。一方、ＰＥ間ＧＣの考察の際にも述べるが、一般に輸出テーブルのエントリーの削除のタイミングを知るのは非常に困難である。

図．６　外部参照ポインタ

（４－２）ＰＥ間のゴールの管理

３章で述べたように、同一の「仕事」を行っているゴール群は同じＡＮＤ木に属している。並列計算機上で協調しながら「仕事」を進めている場合には、このＡＮＤ木が複数のＰＥにまたがって作られていると考えられる。一方、ゴールの実行によってＡＮＤ木の各部分ではゴール・レコードの消去あるいは登録が頻繁に起っている。この場合、木を単純に複数のＰＥに展開すると、木の切れ目でのＰＥ間の通信量及び、根が存在するＰＥの負荷が非常に高くなってしまう恐れがある。そこで、ここでは後述するような［近山 85a］で提案されている代理／同報を使ったＡＮＤ木の管理方式を用い、木の管理に関するＰＥ間通信を少なくすることを考える。

ＰＥ間にまたがる木の場合、子孫のゴールのうちあるものは別のＰＥに存在することになる。この場合に、根であるメタコール（もしくは後述する里親）の存在するＰＥでは、別のＰＥにあるゴール群の代わりにそれらを代表するノードを一つ登録することにする（このノードのことをここでは代理レコードと呼ぶ）。一方、子孫のゴール群を持つ別ＰＥでは里親レコードと呼ぶノードを作り、その里親の元でゴール群を管理する。図．７は代理／里親を使ったＡＮＤ木の例である。代理／里親レコードの作成及び登録



図．７　代理／里親を使ったゴールの管理

の手順を以下に示す。

［代理／里親レコードの作成及び登録］

まず次のことが仮定されているものとする。

・各ＰＥには、そのＰＥ内に存在するメタコール・レコードとその識別名の対からなる表が管理されている。

・同様に里親レコードとその里親が所属しているメタコールの識別名の対からなる表も管理されている。

上記の条件の下で、ゴールを他のＰＥに依頼する場合、

(1) まず、送り側で、代理レコードを無条件に生成し木に登録。このとき、ゴールの送り先ＰＥ番号及び所属するメタコールの識別名を代理レコードにセットし（すなわち、代理レコードは対応する里親レコードを、外部参照ポインタで直接指すのではなく、ＰＥ番号とメタコールの識別名を用いて間接的に指す）、

(2) 依頼するゴールと共に、生成した代理レコードへの外部参照ポインタと所属するメタコールの識別名を付けて送る。

(3) 受け側では、対応するメタコールの識別名を持つ里親（あるいはメタコール自身）が既に存在していれば、ゴールをその部分木に登録し、先ほど生成した代理レコードが不必要である旨送り側に伝える。

(4) 無ければ、新たに里親レコードを生成し、代理レコードへの外部参照ポインタ及び識別名をセットすると共に、依頼されたゴールをその子供として登録する。

この様な手順を取る利点は、

・里親を作ったＰＥがそのアドレスを依頼主に知らせる必要が無くなる。

・複数のＰＥにまたがった代理／里親のチェインが出来ない。

・ゴールの送出時に、送り先に既に里親が存在するか否かを知る必要が無い。たとえ知ることが出来たとしても、例えば次のような場合に、正しく管理されるよう実現することは非常に困難である。すなわち、送り側がゴールを送ろうとする時と、受け側で対応する里親の全ての子孫が終了し、終了メッセージを代理側に知らせようとすることがほぼ同時に起った場合等である。

以下にＡＮＤ木の基本的な操作の概略を示す。

(a) あるノードが成功した場合。

(1) そのノードを木から削除する。

(2) 削除することによってその部分木が根だけになった場合、

(2-1) 根ノードが里親の場合はその代理を成功させる。

(2-2) メタコールの場合は、結果を返す変数を'success'とユニファイし、成功したらこの根ノードを成功させる。失敗したら根ノードを失敗させる。

(b) あるノードが失敗した場合。

(1) そのノードが所属している部分木の全てのノードを削除する（失敗させる）。部分木中に代理があった場合は対応する里親を失敗させる。

(2) その部分木の根がメタコールの場合、結果を返す変数を'fail'とユニファイし、成功したらこの根ノードを成功させる。失敗したら根ノードを失敗させる。

(3) 里親の場合その代理ノードを失敗させる。

(c) あるノードが新たにノードを生成する場合。

(1) ノードがメタコールのときは、生成されたノードはこのメタコールの子孫として新たに部分木を構成する。

(2) ゴールの場合は、

(2-1) 新ノードが同じＰＥであれば、生成するノードの代わりにＡＮＤ木に登録する。

(2-2) 別のＰＥに依頼するものであれば、先に述べた手順に従って木に登録する。

（５）ユニフィケーション

分散環境においてユニフィケーション処理を実現する場合、主に次の点が重要である。

(a) デッドロックを招かずに、別々のＰＥで起こる同一変数に対するユニフィケーションが正しく行われる。

(b) ユニフィケーションの為の通信量をなるべく少なくする。

(c) 無限の探索空間同士のユニフィケーションのように暴走した場合にも、少なくとも処理の中断くらいは可能。

(a) は具体的には、ＰＥにまたがって変数の参照ポインタがループを作ってしまった場合である。この様な場合ループを構成している２つ以上の変数に異なったＰＥで同時に値を書き込もうとした時、デッドロックを招かずに正しく値をいれる為には非常にコストが掛かってしまう。この様な事態を防ぐ為にここでは、ＰＥ番号を用いてＰＥの順序付けを行い、その順序に従って参照ポインタの指す方向が一定になるようにする。

(b) については種々の最適化が考えられるが、ここでは

特にガード部とボディ部のＰＥ間にまたがるユニフィケーションを、異なった方式で実現することを考える。すなわち、ガード部のユニフィケーションではそのとき必要な値を自身のＰＥに読み込んで実行し、ボディ部ではユニフィケーション処理そのものを参照先のＰＥに依頼してしまう。これは、前者の場合、

(i) その後でやるべき仕事（ボディのユニフィケーション及びボディのゴールの展開等）が多い。

(ii)他の候補節が同じ変数に対してユニファイすることが多く、又ボディのゴールでその値を使うかもしれない。

等の理由からである。又後者は、

(i) ボディのユニフィケーションは基本的に代入であり、処理そのものが軽い。

(ii)参照先のＰＥでその値を使うかも知れない。

からである。

(c) については、ユニフィケーションも基本的に一般のゴールと同様、ＡＮＤ木を用いて管理する事により対処する。

（５－１）ガード部のユニフィケーション

表．１はガード部のユニフィケーションの概要を示したものである。

ガード部のユニフィケーションを考える場合、論理変数は、その同期機構に着目して２種類に分類することができる。すなわち、呼び出すゴールの環境に所属するものと、呼び出される節のガード部の環境に属すものである。表．１では前者の未定義変数をg-undef 、後者の未定義変数をl-undef で表している。ここで注意しておきたいことは、FlatGHC の言語仕様を、変数の分類（言い替えればどのタイプのユニフィケーションか）がコンパイル時に決定できるよう決められていると言うことである。

表．１においてsuspend はそのユニフィケーションが中断することを意味する。ここで注意すべき事は、我々が実現するFlatGHC ではg-undef 同士のユニフィケーションは、たとえ２つの変数が同一のものであってもsuspend する、ということである。次のようなプログラムを考えて見よう。

図．８　read要求の場合の
輸入テーブルへの登録

```
goal:   :- p(X, Y), X=Y.
clause: p(A, A) :- true  true.
```

上の例で X=Yがp(X,Y)よりも後に実行された場合、X=Y の実行によってゴールp(X,Y)を復帰させる為には、「 XとＹがかつてユニファイされようとした」事が解らなければならない（具体的には双方のSuspension Queueの内容を比較し、同じゴールがフックされていたらそのゴールを復帰させる）が、これは非常にコストがかかる。

read-valueは他のＰＥに対して変数の値の問い合わせ要求が必要であることを意味する。この場合、Trail Stack には、どの変数に対して問い合せが必要か等、外部参照ポインタを元に要求メッセージを作るための情報が積まれる。実際の要求メッセージ（以下readメッセージとも言う）が対応するＰＥに送られるのは、どの候補節も選ばれなかった場合であり、呼び出されたゴールは中断状態になる（もちろん、いずれかの節が選ばれればreadメッセージを送る必要はない）。中断されるゴールはその原因となった外部参照ポインタの代替え変数セルにフックされる。若し、既にこの代替え変数セルに他のゴールがフックされていた場合、それはすなわちその変数に対する値の問い合わせ要求が既に出されていることを意味し、この場合もreadメッセージを送る必要はない。exref 同士の場合はどちらか一方のＰＥに対してreadメッセージを出し、他方はその返信があった後まだ必要であればreadメッセージを送る。

図．８に示すように、readメッセージの返信先識別番号は輸入テーブルに登録される。識別番号の対として登録されるアドレスは外部参照ポインタのセルである。返信があった場合は、まず輸入テーブルからそのエントリを削除し、

表．１　ガード部のユニフィケーション

X ＼ Y	l-undef	g-undef	exref	constant	structure
l-undef	X := ref(Y)	X := ref(Y)	X := Y	X := Y	X := Y
g-undef	Y := ref(X)	suspend	suspend	suspend	suspend
exref	Y := X	suspend	read-value*	read-value	read-value
constant	Y := X	suspend	read-value	X = Y	fail
structure	Y := X	suspend	read-value	fail	X = Y

次にそこから指される外部参照ポインタのセルに返信された値を書き込み、その先の代替え変数セルに繋がれたゴールを復帰させる。

readメッセージに関する最適化として、readすべき値のタイプあるいは構造体の複雑度（例えばレベル２以上のリスト）を付けることが考えられる。

（５－２）ボディ部のユニフィケーション

先にも述べたように、ＰＥ間にまたがる未定義変数同士のユニフィケーションの際ポインタがループするのを防ぐ為に、本稿で述べる方式では、外部参照ポインタの張られる方向を、ＰＥ番号の小さい方から大きい方に張るように決める。ボディ部のユニフィケーションの概要を図．９に示す。

図．９において、ref(X)は変数 Xを指すポインタを意味し、X := Yは Yの値を Xに代入することを意味する。また、sendは第一引数で指定されるＰＥに第二引数で示されるメッセージを、第三引数で与えられた返信先を付けて送ることを意味する。メッセージ中の未定義変数などに関するアドレス系の変換はこの時に行われる。個々のユニファイ・メッセージは基本的には「ユニファイするゴール」を当該ＰＥに依頼するものと考えることが出来るが、値の読み出し，変数への代入など簡単な処理については最適化も考えられる。（例えば、代理／奥義を作らず、メッセージを送出するゴールの直接の親（すなわちメタコールもしくは豊穣）が送出されたメッセージの総数を管理し、返信の度にdecrement する。メッセージの受け側ではメッセージ・インタプリタがメッセージレベルで直接解釈してしまう。）構造体のユニファイを依頼するメッセージは、その構造体の幾つかの要素が再び外部参照ポインタであった場合、複数のユニファイ・メッセージを生成することがある。これはあるゴールが適当な節を使って複数のゴールに書き替えられた場合と同様の解釈ができる。

以下では図の①～④のメッセージについて簡単に説明を加える。

①は外部参照ポインタ Yの指す変数と未定義変数 Xをユニファイするメッセージを、 Yが指すＰＥに対して送ることを意味する。受け側では与えられたＰＥ内識別番号を使って輸出テーブルを引き、対応する変数と Xを指す外部参照ポインタについてunify を適用する。結果として成功が返される。

②は少なくとも一方がhooked variable の場合である。メッセージの基本的な意味は、変数 Xが外部参照によって Y（正確には Yによって指されている変数）を指し、 Yが Xの持つSuspension Queueを指すようにすることである。受け側では、 Yが既に値を持っていた場合、 Xに対しSuspension Queueに繋がるゴールを復帰させるようメッセージを送る。（もちろん Yの値も同時に送る）未定義の場合は Suspension Queueを指すポインタを代入し、成功した旨返信する。 Yもhooked variable であった場合、 Yの持つSuspension Queueの末尾に XのSuspension Queueへのポインタを繋げ、成功した旨返信する。 Yが再び外部参照ポインタであった場合は、同じunify-with-hook メッセージをその参照先に送る。このときの返信先はそのまま。

③は②の場合と異なり、 Yに対して Xへのポインタを張れ、という場合である。このとき Yが未定義であれば単純に Xへの外部参照ポインタを Yに代入する。 Yが既に値を持っていた場合は、②と同様、 Xに対してSuspension Queueのゴールを復帰するよう、 Yの値付きでメッセージを送る。 Yがhooked variable の場合、 Yには Xへの外部参照ポインタを代入し、 Yの持つSuspension Queueを XのSuspension Queueに繋げるためのメッセージを Xに対して送る。

④の場合、送られた Yの値（constant）と Xの値をユニファイし、結果を返信する。

⑤は基本的には④と同じである。このとき Xの指すＰＥには Yが指すstructure の１レベルのみがコピーされる。（先の③，⑤についても同じ）これは、(a) 全てコピーしても結局使われないかもしれないのに対し、ネットワークを介してコピーするのは非常にコストがかかる。(b) 無限の構造体があった場合、コピーが止まらなくなってしまう。などの理由からである。もちろんこれについては、ユーザがあらかじめコンパイラに、値の流れる方向に関してなんらかの情報を与える事によって最適化出来るであろうが、未だ検討が充分ではない。

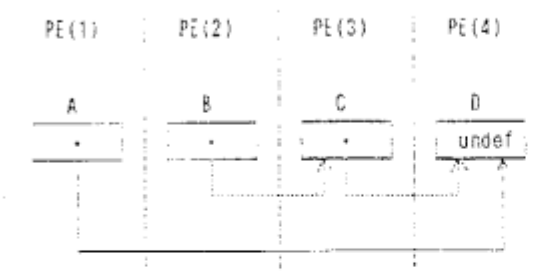
ＰＥ間のユニフィケーションの簡単な例を以下に示す。

【例】 ＰＥ番号がそれぞれ１，２，３，４であるような４台のプロセッサを考える。ここでPE(1) に変数 A，PE(2) に B，PE(3) に C，PE(4) に Dがあるものとする。また Aは既に Dとユニファイされており、 B，C，Dは未定義変数とする。

この様な状況の下で、時刻 tに、PE(2) で $B=A^{exref}$，PE(3) で $C=B^{exref}$ なるユニフィケーションが実行された場合の一例を以下に示す。なお X^{exref} は Xを指す外部参照ポインタを意味する。

```
%    undef - 未定義変数
%    hook  - Suspension Queueを持つ未定義変数
%    exref - 外部参照ポインタ
%    A のPE番号 - Aがundef かhookならばA の変数セルのあるＰＥ番号、exref ならそれが指すＰＥ番号
%    id    - 輸出テーブルに登録されたＰＥ内識別番号
procedure  unify(X, Y)
 case(tag(X)) {
   undef:
      case(tag(Y)) {
         undef:      X := ref(Y) ;
         hook:       X := ref(Y) ;
         constant:   X := Y ;
         structure:  X := Y ;
         exref:      if (X のPE番号 < YのPE番号) then  X := Y ;
                     else  send(YのPE番号, unify-with-undef(Yのid, X), 返信先) ;          …… ①
         } ;
   hook:
      case(tag(Y)) {
         undef:      Y := ref(X) ;
         hook:       X のSuspension QueueをY のQueue に繋げ ;
                     X := ref(Y) ;
         constant:   X のSuspension Queue内のゴールを復帰し ;
                     X := Y ;
         structure:  X のSuspension Queue内のゴールを復帰し ;
                     X := Y ;
         exref:      if (X のPE番号 < YのPE番号)
                        then send(Y のPE番号, unify-with-hook(Y のid,Xが指すSuspension Queue),返信先) ;  … ②
                             X := Y ;
                        else send(Y のPE番号, unify-with-hook2(Yのid, X), 返信先) ;       …… ③
         } ;
   exref:
      case(tag(Y)) {
         undef:      if (X のPE番号 > YのPE番号) then  Y := X ;
                     else  send(XのPE番号, unify-with-undef(Xのid, Y), 返信先) ;
         hook:       if (X のPE番号 > YのPE番号)
                        then send(X のPE番号, unify-with-hook(X のid,Yが指すSuspension Queue),返信先) ;
                             Y := X ;
                        else send(X のPE番号, unify-with-hook2(Xのid, Y), 返信先) ;
         constant:   send(XのPE番号, unify(X のid, Y), 返信先) ;                        …… ④
         structure:  send(XのPE番号, unify(X のid, Y), 返信先) ;                        …… ⑤
         exref:      if (X のPE番号 < YのPE番号)
                        then  send(X のPE番号, unify(X のid, Y), 返信先)
                        else  send(Y のPE番号, unify(Y のid, X), 返信先) ;
         } ;
   以下省略
   } ;
```

図. 9　ボディ部のユニフィケーション　概要



図．１０　ＰＥ間ユニフィケーション

－時刻 t　－

[PE(2)]

B=A^exref を実行。これによりPE(1) に対してメッセージunify-with-undefを送る。

[PE(3)]

C=B^exref を実行。PE(2) に対してunify-with-undefを送る。

－時刻 t+1　－

[PE(1)]

PE(2) からのメッセージ処理。 Aは既に Dへの外部参照ポインタを持っており、PE(2)<PE(4) なので、PE(2) に対し Dとユニファイする旨メッセージを送る。

[PE(2)]

PE(3) からのメッセージ処理。 Bは未定義変数なので Cへの外部参照ポインタを Bに代入。

－時刻 t-2　－

[PE(2)]

PE(1) からのメッセージ処理。 Bは既に Cとユーファイされており、PE(1)<PE(3) なので、PE(3) に Dとユーファイする旨メッセージを送る。

－時刻 t-3　－

[PE(3)]

PE(2) からのメッセージ処理。 Cは未定義なので Dへのポインタを Cに代入。

注）この例ではＰＥ間で時刻が完全に同期しているように書かれているが、これは単に説明を簡単にするためであり、実際のmulti-PSI では非同期である。

（６）今後の課題

以上、multi-PSI におけるFlatGHC の実現方式について述べて来たが、まだ幾つかの点が検討課題として残されている。以下ではそれらの内で特に重要と思われるものに付いて簡単に述べることにする。

(a) ガベージ・コレクション

ＰＥ内とＰＥ間のアドレス系を区別することによって、ＰＥ内のＧＣはそれぞれ独立に行うことが出来る。本稿で述べた実現方式においては、ＰＥ間のＧＣはすなわち輸出テーブルの管理であると言えよう。輸出テーブルのエントリの管理方式としては、参照カウント法による実行時の管理や、並列計算機全体で一斉にチェックする方式などが考えられる。しかし前者は、カウントのupとdownの順序性を保証するために、FlatGHC の実行には不必要な逐次性を導入しなければならないなど、我々が目的としている高並列システムには馴染みが悪いと思われる。後者は、完全なゴミ集めは（実用的速度の点で）非常に困難であると思われるが、当面の実験システムを考えた場合適当であるかも知れない。

(b) プログラム・コードの管理

近年のデバイス技術の発達を考慮にいれても、我々が目標としている大規模並列推論マシンのＰＥの実装を考えた場合、実行しようとしているプログラム・コード全体を各ＰＥが持つことは不可能であるかもしれない。この場合は、コードのダイナミック・ローディング等を考える必要があるが、ゴールの割り付けに関連して、何処からコードをロードするのが最適か、何時，どのコードを捨てるか等、まだ未検討な部分が多い。また、巨大なテーブルのように動かさないコードも考える必要があろう。

(c) ゴール割り付けの制御

並列計算機において、プログラムの並列性を最大限に引き出すためには、ゴールをどの様にＰＥに割り付けるかが非常に重要な問題である。これについては、主に次の２つの方法を併用することが考えられる。

・ユーザによる割り付け情報の提示。

あらかじめプログラムに割り付け用の制御情報を付加する。これは一種のプラグマと考えられ、このための言語を設計することが必要である。言語の機能としては、①物理的なＰＥ番号が指定できる，②実行時に、親ゴールの割り付け情報あるいはデータ等により指定できる，等の機能が望まれよう。割り付けの方式については、［Shapiro 84］，［近山 85b］が参考になろう。

・システム（ＯＳ）による自動的なロード・バランス

なるべくシステム全体の情報が必要ない方式が望ましい。現在上記②に関連して検討中である。

(d) デッドロックの検出

FlatGHC のような並列型の言語の場合、プログラムのデバッグ（言い換えれば実用性）を考えると、デッドロックを検出できることは不可欠であると言えよう。しかし、現

実的には検出の対象としている「仕事」がそれだけで閉じていないような場合があり（例えばＯＳとのＩ／Ｏ），非常に困難であることが予想される。そこで、ここではまず問題を簡単にするために、「仕事」はそれだけで閉じており、中断の原因はその「仕事」内にしか存在しないと仮定する。この場合、例えば次のような手順でデッドロックを検出することは出来よう。

(1) 疑いのある「仕事」内の全てのゴールをＡＮＤ木を辿ることによってチェックする。このとき、中断状態にあるゴールを見つけると、『他のゴールの実行によって実行可能になっても動くな』と命令しておく。実行可能状態にあるゴールが一つでもあれば、先ほどの命令を解除し実行を再開する。もしこの一回めのチェックで全てのゴールが中断状態にあった場合は、

(2) ＡＮＤ木を構成しているＰＥを繋いでいるネットワーク上の全てのリンクにダミーのメッセージを流し（これが終了するとネットワーク上には現在チェックしている「仕事」に関するメッセージが流れていない事が保障される），

(3) 再びＡＮＤ木を辿り、先ほどの命令によって『実行可能になったが中断している』ゴールが無いかチェックし、無ければデッドロックと見なす。一つでもあった場合は、(1) と同様に命令を解除し実行を再開する。

「仕事」が閉じていない場合でも、少なくとも相手をＯＳだけに限定し、その出入口では厳密に値の流れる方向を規定することによって、ある程度は上記の方法を適用できるかもしれないが、まだ検討が充分ではない。

（７）おわりに

本論文では、現在ICOTにおいて開発中であるmulti-PSI上のFlatGHC の実現方式について述べた。本方式の特徴は (a) ＰＥ内とＰＥ間のアドレス系を分離しＧＣの見通しを良くすると共に、将来の並列計算機の構築をＰＥの台数の点で自由度の高いものにする，(b) 参照ポインタのループを防ぐためにＰＥ番号を用いてポインタの張られる方向を一定に決めた、等が上げられる。現在multi-PSI は、その接続ハードウェアの製造及びFlatGHC の処理系の試作を行っており、今年度中にそのプロトタイプが稼動する予定である。

最後に、多くの助言あるいは示唆をいただきましたmulti-PSI 検討会の方々及び第一，第四研究室の方々に感謝の意を表します。

参考文献

［上田84］ 上田和紀，近山隆． Efficient Stream/Array processing in logic programming languages, Proceedings of the international conference on fifth generation computer systems '84, pp.317-326 (1984)

［上田86］ 上田和紀． Guarded Horn Clauses, 東京大学 学位論文 (1986)

［江崎86］ 江崎令子，その他． GHC サブセット逐次型処理系の作成． 情報処理学会第３２回全国大会 2G-4 (1986)

［齋86］ 齋和男，その他． Multi-PSI システムの概要，情報処理学会第３２回全国大会 5Q-8 (1986)

［近山 85a］ 近山隆． AND/OR木の管理方式案． ICOT (multi-PSI検討会内部資料) (1985)

［近山 85b］ 近山隆． computing power の均一分布model ． ICOT(multi-PSI検討会内部資料) (1985)

［宮崎85］ 宮崎敏彦，その他． A sequential implementation of Concurrent Prolog based on tne shallow binding, proceedings of tne symposium on logic programming, pp.110-118 (1985)

［村上86］ 村上健一郎，その他． 並列推論マシーンにおける分散型ユニファイア構成方法の検討． 通信学会コンピュータ・システム研究会 発表予定

[Clark 84] Clark K.L. and Gregory S., PARLOG: parallel programming in logic, Research report DOC 84/4, Dept. of Computing, Imperial College, London (1984)

[Mierowsky85] C.Mierowsky,S.Taylor,E.Shapiro,J.Levy and M.Safra, The design and implementation of Flat Concurrent Prolog, Weizman Institute Technical Report CS85-09, 1985.

[Shapiro 84] Ehud Shapiro, SYSTOLIC PROGRAMMING: A Paradigm of Parallel Processing, Proceedings of the international conference on fifth generation computer systems '84, pp.458-470 (1984)