ICOT Technical Memorandum: TM-0411

TM-0411

GHCによる仮想ハードウェアの構築とリフレクト機能について

田中二郎，太田祐紀子，的野文夫，神田陽治

November, 1987

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

GHC による仮想ハードウェアの構築とリフレクト機能について

田中二郎　　太田祐紀子　　的野文夫　　神田陽治
（ＩＣＯＴ）　　（富士通ＳＳＬ）　　（富士通国際研）

本稿では、並列論理型言語GHC[Ueda 85]に基づく並列環境の構成イメージについて述べ、その多層構造のうち、主として仮想ハードウェアの構築とその機能について述べる。また並列環境において自己の状態を認識する機能、すなわちリフレクト機能の実現についての考察を行う。なお本稿は並列論理型言語GHC についての基礎的知識を前提としている。

1. はじめに

最近、幾つかのマイクロプロセッサを結合して一つの計算機を組み立てる、いわゆるマルチマイクロ計算機が急速に実用段階に入ってきている。例えば、インモスで作られているトランスピュータ、インテルのハイパーキューブ、シークエント社のバランス等がそうである。これらの計算機は実用化され、既に商用機として売り出されている。

こうしたマルチマイクロ計算機は、その価格性能比の良さから今後急速に普及することが考えられる。また、こうした計算機を使っての並列あるいは分散処理が、実用的な手段として普及すると予想される。

しかしながら、これらのマルチマイクロ計算機はそれぞれが様々なアーキテクチャを持っている。これらの上に、いちいち似たようなシステムを作るのは大変である。そこで本稿では分散ハードウェア上に効率的に並列環境を構築する方法を考察する。

2. 並列環境の構築イメージ

ここでは、並列論理型言語GHC [Ueda 85] に基く単一言語システムについて考える。単一言語システムとは、ハードウェア・レベルからアプリケーション・レベルまで全て単一言語で統一的に記述されているようなシステムである。その様なシステムの上にどうやって並列環境を構築するか、その構成イメージを階層的に図に示したのが図１である。

図１　並列環境の構成イメージ

図でも明らかなように、一番下にくるのが分散ハードウェアである。分散ハードウェアは様々なアーキテクチャを持っており、メモリの結合方式（分散メモリ、共有メモリ）やプロセッサの結合トポロジー（バス、リング、メッシュ、ハイパーキューブ）などによってその動作特性は様々である。

分散ハードウェアの上には直ぐ分散GHC の処理系がのる。分散GHC 処理系は、本質的には通常の逐次GHC と異なる点はない。異なっているのは、ゴールが一つのプロセッサだけでなく、多数のプロセッサで同時に実行されることである。したがって分散GHC 処理系のプログラムではゴールをどのプロセッサで実行するかの指定を行う必要がある。

分散環境上での並列プログラミングを考えるうえで、問題になるのは、いかに、負荷を分散させ、高い並列性を得るかである。ゴール実行の指定にはいろいろな方式が考えられるが、ここではその指定をプログラムにプラグマを付加することにより行うと考える。例えば

```
q  :-  g | q1,q2,q3.
```

というプログラムにプラグマを付加すると

```
q  :-  g | q1@p1, q2@p2, q3@p3.
```

のような形となる。これはゴールqが計算されて、ゴールq1,q2,q3 に展開されるとき、q1 は@p1 、q2 は@p2 、q3 は@p3 で指定されるプロセッサで実行することを意味する。このプラグマにも様々な種類が考えられ、代表的なものだけでも、Shapiro により提案されたもの[Shapiro 84]、ICOTで近山らにより提案されたもの、神田により提案されたもの[Kohda 87]などいろいろである。分散GHC 処理系では、それぞれのプロセッサに付けられたIDを指定するという一番原始的なプラグマのみを許し、上に挙げたような複雑なプラグマ方式については、より上位の仮想ハードウェアのレベルで実現されると考える。

この様な分散GHC 処理系を共有メモリの分散ハードウェア上に実装することは比較的容易であろう。分散メモリのハードウェア上に実装する場合、異なったプロセッサ上で共有される変数をメッセージ交換により実現する必要があるが、それについては既に[Miyazaki 86] 等で考察されている。

この分散GHC 処理系の上に乗るのが仮想ハードウェアである。分散GHC 処理系のレベルでは、メモリは抽象化されているが、プロセッサの数や結合については分散ハードウェアのアーキテクチャを引き継いでおり、汎用性に欠けている。我々は、分散GHC 環境上に、仮想ハードウエアを構築し[Taylor 87] 、その上でシステムを走らせることにした。仮想ハードウエアは、複数のプロセッサを持ち、プロセッサの数、プロセッサの結合ネットワークの形を自由に指定して構築できる。

この仮想ハードウエアの上にはプログラミングシステムが構築される。プログラミングシステムとは、その上でユーザがプログラムを入力したり実行したりできる環境のことであり、オペレーティング・システムのユーザ・インタフェースにあたるものである。プログラミングシステムの基本的な機能には、失敗の処理、ユーザ・プロセスのコントロール、例外処理などが考えられる[Tanaka 87a, Tanaka 87b]。そしてこのプログラミングシステムの管理の下でユーザプログラムが走ることになる。

本稿ではこの並列環境の多層構造のうち、主として仮想ハードウェアとその機能についてのべる。

3. 仮想ハードウェアの構築

仮想ハードウエアでは、ユーザが任意の数の（仮想）プロセッサを指定でき、また、そのプロセッサの結合方式も実装されている物理ハードウェアによらず任意に指定できることが望ましい。ユーザが、動かすプログラムに適した抽象ハードウエアの構築を可能にするためである。

我々のシステムでは、（仮想）プロセッサを任意の個数指定出来、またネットワークの形も 1次元配列、輪状、格子状、n次元ハイパーキューブなど自由に指定できる。本稿ではその仮想ハードウエアを自動的に構築することを考える。

3.1 仮想ハードウエア

仮想ハードウエアの構築にあたり、仮想プロセッサの生成部とネットワークの構築部は、分けるほうがよい。仮想ハードウエア構築プログラムに、ネットワークの形によらない汎用性を持たせるためである。

```
boot (Form,N):- true |
        net(Form,N,I-chs,O-chs)
        make-processors(I-chs,O-chs).
```

bootは、第１引数に、ネットワークの形、第２引数にネットワークの大きさを入力する。このプログラムは、仮想ハードウエアのネットワークをnet で構築し、make-processors で、仮想プロセッサをネットワークにつなぐ。例えば、１次元上に仮想プロセッサが５個つながる場合であればboot(line,[5])、仮想プロセッサを４ｘ５ｘ３の立方体状につなげたいならば、boot(cube,[4,5,3])のようにする。

net は、第１引数（形）、第２引数（大きさ）で指定するネットワークを構築し、第３、第４引数に、ネットワークの各ノードの入出力チャネルを、入力、出力に分けてリストの形で出力する。

make-processors は、calculate-for-map(N,PE) でまずネットワークマネージャnmと抽象GHC マシンghc-machine の分散ハードウェアへの割振りを計算し、net で作ったネットワークの各ノードへ、ネットワークマネージャnmと抽象GHC マシンghc-machine をはめ込む。

```
make-processors(I-chs,O-chs):-true |
        make-processors(I-chs,O-chs,1).
make-processors([I-ch |It],[O-ch |Ot],N):-
        true |
        calculate-for-map(N,PE),
        nm(N,I-ch,O-ch,I,O)@PE,
        ghc-machine(H,H,I,O,[])@PE,
        N1 :=N+1,
        make-processors(It,Ot,N1).
make-processors([],[],_):-true | true.
```

ここでネットワークマネージャnm(N,I-ch,O-ch,I,O) は5 つの引数を持っている。最初の引数N はこのネットワークマネージャの識別子、次の二つの引数は他のネットワークマネージャとの入力チャネルおよび出力チャネル、引数I とO は抽象GHC マシンからの入出力である。

またghc-machine(H,T,I,O,Plist)も5 つの引数を持っている。最初の二つの引数H とT は、重リストの形で表された、抽象GHC マシンのスケジューリングキューであり、引数H とT はスケジューリングキューの頭部と尾部を示し、初期値は空に設定する。（この重リストによるスケジューリングキューの実現は、prologによるConcurrent Prolog インタプリタの実現[Shapiro 83]で使われた方法をまねたものであり、抽象GHC マシンに逐次性を持ち込むため導入したものである。）また引数I とO はネットワークマネージャへの入出力である。入力からはその抽象GHC マシンで実行されるべきゴールが入ってくる。出力からは、他の抽象GHC マシンで実行させたいゴールやプログラム実行中に生じた各種の情報等を出力する。最後の引数Plist は、その抽象GHC マシン内で実行中のユーザの仕事への入力ストリームのテーブルであり、初期値は[]を設定する（詳しくは4.2 節を参照のこと）。

3.2　ネットワークの構築

ネットワークは、その形や大きさにかかわりなく、「n次元のネットワークは、n－1次元のネットワークに、1次元を加えたものである」ということができる。それならば、ネットワークの構築は、どんなものでも、n－1次元のネットワークを複数個作り、その各ノードを1次元のネットワークでつなぐことで実現できる。例えば、4×5×3の立方体ネットワークは、4×5の格子状ネットワークを3枚作り、その3枚の格子状ネットワークの各ノードをつなぐことで完成する。別の見かたをすると、ネットワークの構築は、1次元ネットワークを基礎とし、これに1次元1次元組み上げていくと考えることができる。

次のプログラムは、格子状ネットワーク、立方体ネットワークをそれぞれ作るプログラムである。net は、第1引数に、ネットワークの形、第2引数に、ネットワークの大きさを入力して呼びだす。そうすると、格子状ネットワーク、立方体ネットワークが、それぞれ構築され、各ノードの入出力チャネルが、生成された順に、入力チャネルと出力チャネルに分かれてリストに入って出力される。

```
net(cube,[N,M,H],I-chs,O-chs):- true |
        NN:=N * M,
        n-net(mesh,[N,M],H,I-ch-m,O-ch-m),
        n-net(linear,[H],NN,I-ch-c,O-ch-c),
        connect-net(I-ch-c,I-ch-m,I-chs),
        connect-net(O-ch-c,O-ch-m,O-chs).
net(mesh,[N,M],I-chs,O-chs):- true |
        n-net(linear,[N],M,I-ch-m,O-ch-m),
        n-net(linear,[M],N,I-ch-c,O-ch-c),
        connect-net(I-ch-c,I-ch-m,I-chs),
        connect-net(O-ch-c,O-ch-m,O-chs).

n-net(Form,Net,N,I-chs,O-chs):-N≠0 |
        net(Form,Net,I-ch,O-ch),
        I-chs=[I-ch |It],
        O-chs=[O-ch |Ot],
        N1 := N-1,
        n-net(Form,Net,N1,It,Ot).
n-net(Form,Net,0,I-chs,O-chs):- true |
        I-chs=[],
        O-chs=[].
```

このnet は、最初にn-net で、n－1次元のネットワークを複数個作り、次のn-net で、1次元ネットワークをn－1次元ネットのノードの数だけ作り、最後にconnect-net で、複数のn－1次元ネットの各ノードを1次元ネットワークでつなぐ。

n-net(Form,Net,N,I-chs,O-chs) は、単にnet(Form,Net,I-ch,O-ch) をN 回呼び、ネットワークごとにノードのチャネルがまとめ、入力、出力に分けて出力する。

またconnect-net は、複数のネットワークを1次元ネットワークのワイヤでつなぐ処理をする。ワイヤでつながれると、各ノードはネットワークにおけるチャネルと、ワイヤにおけるチャネルの両方を結合してもつことになる。

なお、ここでは、格子状ネットワーク、立方体ネットワークの構築について示したが、同様の方法でn次元hyper-cubeネットワークも構築できる。

次にn次元ネットの基礎となる1次元ネットワークの構築について考える。1次元状ネットワークにおいては、あるノードの右方向への出力チャネルは、そのノードの右隣のノードの左方向からの入力チャネルと等しい。

したがって、net は、まず入力チャネルI-ch、出力チャネルO-chを確保し、linearを呼ぶ。linearは、第1引数に、いくつノードを作るかのカウンタ、第2引数に、ひとつ前のノードの入力チャネル、第3引数に、ひとつ前のノードの出力チャネルを持ち再帰的に呼ばれる。linearは一つずつチャネルをつないでいくので結果的に1次元ネットワークが張られることになる。

```
net(linear,[N],I-chs,O-chs):-true |
        N1:=N-1,
        I-chs=[[I-ch] |It],
        O-chs=[[O-ch] |Ot],
        linear(N1,I-ch,O-ch,It,Ot).
linear(N,I-ch,O-ch,It,Ot):-N≠1 |
        It=[[O-ch,I-ch-next] |Itt],
        Ot=[[I-ch,O-ch-next] |Ott],
        N1:=N-1,
        linear(N1,I-ch-next,O-ch-next,Itt,Ott).
linear(N,I-ch,O-ch,It,Ot):-N=1 |
        It=[[O-ch]],
        Ot=[[I-ch]].
```

こういった仮想ハードウエアの構築については[Taylor 87] などにも示されているが、入出力チャネルのネット張りをすべてヘッド・ユニフィケーションで記述するため、プログラムは複雑かつ特殊なものとなり、汎用性に乏しいものであった。

4.　仮想ハードウェアの機能

前節までで仮想ハードウェアの構築について記述した。本節では仮想ハードウェアの機能について考察する。図2 に、一次元ネットワークに構築された仮想ハードウェアの一例を示す。

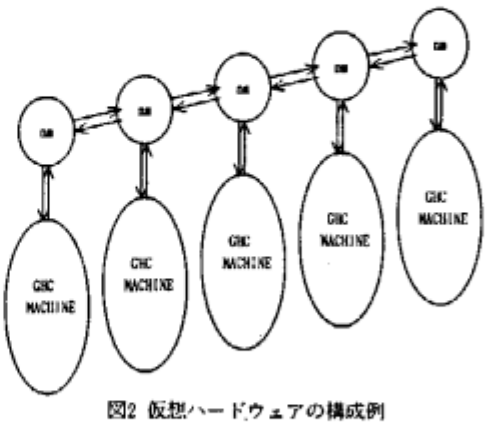

図2 仮想ハードウェアの構成例

この例では5 つの抽象GHC マシンが、それぞれネットワークマネージャnmを経由して一次元状に結合されている。ネットワークマネージャは抽象GHC マシンからきたプラグマつきメッセージを、プラグマに従い適切な抽象GHC マシンに送る。

4.1　ネットワークマネージャの記述

ネットワークマネージャnm(N,I-ch,O-ch,In,Out)は五個の引数を持つが、入力ストリームは二番目の引数I-chと五番目の引数Out の二個だけである。まず二番目の引数I-chに情報が来たときは、以下のように振舞う。

```
nm(Id,[[G@forword |L],R],[LO,RO],In,Out):-true |
      RO=[G |RO1],
      nm(Id,[L,R],[LO,RO1],In,Out).
nm(Id,[L,[G@backword|R]],[LO,RO],In,Out):-true |
      LO=[G |LO1],
      nm(Id,[L,R],[LO1,RO],In,Out).
nm(Id,[[G |L],R],O-ch,In,Out):-G ≠Goal@P |
      In=[G |In1],
      nm(Id,[L,R],O-ch,In1,Out).
nm(Id,[L,[G |R]],O-ch,In,Out):-G ≠Goal@P |
      In=[G |In1],
      nm(Id,[L,R],O-ch,In1,Out).
```

左のチャネルから来たゴールがG@forward の形をしていれば、forward を一個はがして右のリンクに転送する。右のチャネルからきたゴールがG@backwardの形をしていれば，backwardを一個はがして左のリンクに転送する。プラグマが付いていなければ，nmに繋がった自分のghc-machine に送る。

また五番目の引数Out にghc-machine から情報が来たときも、同様に振舞う。

```
nm(Id,I-ch,[LO,RO],In,[G@forward |Out]):-true |
      RO=[G |RO1],
      nm(Id,I-ch,[LO,RO1],In,Out).
nm(Id,I-ch,[LO,RO],In,[G@backward |Out]):-true |
      LO=[G |LO1],
      nm(Id,I-ch,[LO1,RO],In,Out).
```

4.2　抽象GHC マシンの記述

次にghc-machine であるが、ghc-machine(Head,Tail,In,Out,Plist) はInにゴールが入力されると起動する。Inには(1) 抽象GHC マシンの上に構築されるプログラミングシステムの仕事が入力される場合と、(2) ユーザの仕事が起動される場合、の二通りが考えられる。

(1) の場合はゴールが裸でじかに、(2) の場合にはゴールG はcallにくるまれてcall(Id,G,I,O,MaxRC)という形式で入力される。ここでIdは、ユーザ・プロセスの識別子、G はユーザゴール、I,O は外の世界への入出力、MaxRC はこのユーザゴールに許されるリダクションの数の最大値である。

(1) プログラミングシステムの仕事

Inにゴールが到着すると、まずghc-machine のスケジューリングキューに入れる。

```
ghc-machine(Head,Tail,[G |In],Out,Plist):-
      ps(G) |
      schedule(G,Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,Plist).
```

ここでガードのps(G) は入力されたゴールがユーザゴールでないことを確かめる組込述語である。一方、Inに入力がない場合には、スケジューリングキューからゴールを取り出して処理を行う。（ガード部のvar(In) はInに入力がないことを確かめるのに使われる。）

```
ghc-machine([G@P |Head],Tail,In,Out,Plist):-var(In) |
      Out=[G@P |Out1],
      ghc-machine(Head,Tail,In,Out1,Plist).
ghc-machine([Goal |Head],Tail,In,Out,Plist):-
      var(In),sys(Goal) |
      solve-system(Goal,Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,Plist).
ghc-machine([Goal |Head],Tail,In,Out,Plist):-
      var(In),notsys(Goal) |
      reduce(Goal,Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,Plist).
```

すなわち、ゴールがプラグマ付きだったらnmに送る。組込述語だったら、solve-systemで解く。すでにシステム中で定義されている述語であればreduceでそれを解く。

(2) ユーザゴール

前述したようにユーザゴールはcallにくるまれてcall(Id,G,I,O,MaxRC)という形式で入力されてくる。抽象GHC マシンはこのユーザゴールに対して、ゴールを実行するexecと、その実行管理を行うproc-server を二つの対で抽象GHC マシンにスケジュールする。

```
ghc-machine(Head,Tail,[call(Id,G,I,O,MaxRC) |In],
      Out,Plist):-true |
      schedule((merge(I,II,IM),
            proc-server(Id,G,CI,CO,IM,0),
            exec(CI,CO,[G |T],T,0,MaxRC)),Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,[(Id,II)|Plist]).
```

作られたexecのスケジューリングキューには、ユーザゴールが格納されている。またexecのリダクション数RCは0 に初期化されている。抽象GHC マシンの5 番目の引数であるPlist には、proc-server への入力ストリームがIdと共に格納される。（ユーザゴールの入力I とPlist の入力IIはmerge されてIMになっている。）

一方、Inに入力がない場合には、スケジューリングキューからゴールを取り出して処理を行う。スケジューリングキューの先頭にexecが現れたとき、抽象GHC マシンは、変数B を引数に付加したexecを呼び出す。（変数B には、execを1 リダクション後再び抽象GHC マシンのスケジューリングキューに格納するゴールが返る。）

```
ghc-machine([exec(I,O,H,T,RC,MaxRC) |Head],Tail,
      In,Out,Plist):-var(In) |
      exec(I,O,H,T,RC,MaxRC,B),
      schedule(B,Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,Plist).
```

ユーザゴールの実行中にプラグマの付いたゴールに当たると、execはproc-server を通じて他の抽象GHC マシンにそのゴールG をgoal(Id,G,I,O,MaxRC)という形式で投げる。投げられた抽象GHC マシンは、そのプロセスについて、すでにlocal-serverが出来ているかをPlist で調べる。出来ていれば、そのlocal-serverの管理下でexecを実行する。そうでなければ新しくlocal-serverを作り、その下でexecを起動する。(local-server はproc-server への連絡をその抽象GHC マシン内でとりまとめる役目をする。local-serverを作るのは、作られるexecが元のproc-server に直接つながっていると、実行効率が低下するからである。) この処理は以下のように記述できる。

```
ghc-machine(Head,Tail,[goal(Id,G,I,O,MaxRC) |In],
      Out,Plist):-member(Id,Plist)|
      send(goal(G),Id,Plist,NewPlist),
      0=[],
      ghc-machine(Head,Tail,In,Out,NewPlist).
ghc-machine(Head,Tail,[goal(Id,G,I,O,MaxRC) |In],
      Out,Plist):-not-member(Id,Plist)|
      schedule((merge(I,II,IM),
            local-server(Id,CI,CO,IM,0),
            exec(CI,CO,[G |T],T,0,MaxRC),Tail,NewTail),
      ghc-machine(Head,NewTail,In,Out,[(Id,II)|Plist]).
```

5.　並列環境におけるリフレクト機能

本稿では並列環境の構築および機能について、主に仮想ハードウェアの視点から考察した。本節では、並列環境において自己の状態を認識する機能、すなわちリフレクト機能、について考察する。

ここではリフレクト機能を，オブジェクトが，メタ部分における自己に関するある情報・状態を手に入れ，なんらかの自己記述の改造を行った後，その状態をメタに戻すと言う観点からとらえる。リフレクト機能の考え方は3-LISP [Smith 84] にヒントを得たものである。3-LISPでは、オブジェクトレベルのプログラムからメタな情報を自由に取り出し、変更を加える事が出来るようになっている。3-LISPにおけるメタな情報としては、プログラム全体の継続(continuation)と変数の束縛環境の 2種類が考えられている。オブジェクトレベルからこれらのメタな情報を取り出す仕組みとして、Smithは自己記述（メタインタプリタ）を使っている。

我々のリフレクト機能の実現方法も原則的にはこれと同様である。ただしGHC は並列性を持ち、またゴール実行の成功や失敗などのメタレベルの情報がプログラム実行中に発生するので、若干実現のための機構は複雑になる。GHC の自己記述としてはghc-machine とex

ecの二つが考えられるが、それらに我々はスケジューリングキューを陽に導入した。8-LISPにおいてはプログラム全体の継続(continuation)が自己記述の中に陽に導入されていたが、GHC ではスケジューリングキューがcontinuationの代りをすると考えたわけである。

以下GHC で並列環境を構築した場合に利用できるリフレクト機能について、二つの簡単な例とその記述を挙げる。

5.1 execにおけるリフレクト機能

抽象GHC マシンの中で実行されるユーザプロセスは、proc-server とexecの対で実行され、execは最大リダクション数の制限を持っている。ここで、ユーザ・プログラムの現在のリダクション数が、制限まであと100 以内となった時、ユーザが入力した数だけ制限を上に延ばすプログラムを記述してみる。

```
check-rc:-true |
      get-rc(MaxRC,RC),
      RestRC:=MaxRC-RC,
      check(MaxRC,RestRC).

check(MaxRC,RestRC):-100>RestRC |
      get-q(H,T),
      input(AddRC)@io,
      NRC:=MaxRC+AddRC,
      put-rc(NRC),
      T=[check-rc@exp | NT],
      put-q(H,NT).
check(MaxRC,RestRC):-100=<RestRC |
      get-q(H,T),
      T=[check-rc@exp | NT],
      put-q(H,NT).
```

check-rcに@expのプラグマを付け、他の実際に動かしたいユーザゴールと共に動かす事により、上に述べた効果が得られる。このプログラムの中で、get-rc(MaxRC,RC),put-rc(RC),get-q(H,T),put-q(NH,NT) はリフレクト述語である。get-rcは、リダクションの最大値と現在の値を与え、put-rcは、リダクションの最大値を更新する述語である。またget-q(H,T)はexecのスケジューリングキューの値を与え、put-q(NH,NT)はスケジューリングキューの値を更新する述語である。

@expというプラグマは、それを付加したゴールが、メタコールのスケジューリングキューの先頭に現れたとき、このゴールを最後まで実行してから、他のスケジューリングキュー内の仕事を行えという急行ゴールを示す。express ゴールを実行するときexec(7引数)は急行ゴールを実行するキューを別に持つ以下の9 引数execを呼び出す。また急行ゴールを実行するキューが空になれば9 引数execは7 引数execになる。

```
exec(In,Out,[G@exp | H],T,RC,MaxRC,B):-var(In) |
      exec([G | ET],ET,In,Out,H,T,RC,MaxRC,B).
exec(EH,EH,In,Out,H,T,RC,MaxRC,B):-var(In) |
      exec(In,Out,H,T,RC,MaxRC,B).
```

この9 引数execで、リフレクト述語get-rc(MaxRC,RC),put-rc(RC),get-q(H,T),put-q(NH,NT) の処理は以下のように記述できる。

```
exec([get-rc(Max,C) | EH],ET,In,Out,H,T,RC,MaxRC,B):-
      true |
      Max:=MaxRC,
      C:=RC,
      RC1:=RC+1,
      B=exec(EH,ET,In,Out,H,T,RC1,MaxRC).
exec([put-rc(C) | EH],ET,In,Out,H,T,RC,MaxRC,B):-
      true |
      RC1:=RC+1,
      B=exec(EH,ET,In,Out,H,T,RC1,C).

exec([get-q(NH,NT) | EH],ET,In,Out,H,T,RC,MaxRC,B):-true |
      RC1:=RC+1,
      NH:=H,
      NT:=T,
      B=exec(EH,ET,In,Out,H,T,RC1,MaxRC).
exec([put-q(NH,NT) | EH],ET,In,Out,H,T,RC,MaxRC,B):-true |
      RC1:=RC+1,
      B=exec(EH,ET,In,Out,NH,NT,RC1,MaxRC).
```

5.2 負荷分散におけるリフレクト機能

分散ハードウェア上のプログラミングシステムは負荷分散を行う必要がある。ここで、get-q,put-q を利用し、GHC マシンのloadがある度数以上、ここではスケジューリングキューの長さが100 以上の場合には、そのマシン中で実行中の仕事の一部を外の抽象GHC マシンに投げるという述語load-balanceを示す。

```
load-balance:-true |
   get-q(H,T),
   q-length(H,T,L),
   load-balancing(L,H,T).

load-balancing(L,H,T):-
   L=<100 |
   T=[load-balance@exp | NT],
   put-q(H,NT).
load-balancing(L,H,T):-
   L>100 |
   separate-q(L,H,T,NH,NT,Out),
   Out@out
   NT=[load-balance@exp | NNT],
   put-q(NH,NNT).
```

load-balance@exp を抽象GHC マシンで一つずつ実行する事によって、負荷分散が行われる。(@exp がGHC マシンのスケジューリングキューの先頭に現れた時は、execにおける@expの処理と同様に、急行ゴールを別に持つ7 引数ghc-machine を呼び出す。)

get-q により抽象GHC マシンのスケジューリングキューが取り出され、それの長さL をq-lengthで求める。L が100 以上であればseparate-qでゴールの一部をOut に取り出し、Out@out で抽象GHC マシンの外に放出する。ここではput-q, get-qが抽象GHC マシンのスケジューリングキューを操作するのに使われている。put-q, get-qの記述は、前述のexecにおける処理と同様である。

6. まとめ

本稿では最初に並列環境の構成イメージについて述べ、その多層構造のうち、主として仮想ハードウェアの構築とその機能について述べた。また並列環境において自己の状態を認識する機能、すなわちリフレクト機能についての考察を行った。今後さらに研究を進めていく予定である。

なお本研究は、第5世代コンピュータ・プロジェクトの一環として行なわれたものである。

[参考文献]

[Kohda 87] 神田：並列論理型プログラムにおける新プラグマ方式とその応用、情報処理第35回全国大会, 4Q-5, 1987年 9月.
[Miyazaki 86] 宮崎、滝: Multi-PSI におけるFlatGHC の実現方式、The Logic Programming Conference '86、ICOT、1986、pp.83-92.
[Shapiro 83] E.Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report.TR-003, 1983.
[Shapiro 84] E.Shapiro: Systolic Programming: A Paradigm of Parallel Processing, in Proc. of the Int. Conf. on FGCS 1984, pp.458-470. ICOT, 1984.
[Smith 84] B.C.Smith: Reflection and Semantics in Lisp, In Proc. of 11th POPL, Salt Lake City, Utah,pp.28-35, 1984.
[Tanaka 87a] 田中 他：GHC 応用プログラム(4) －簡単なOS－, 並列論理型言語GHC とその応用, 第9 章, 共立出版, 1987.
[Tanaka 87b] 田中 他：GHC による簡単なプログラミングシステムの記述、ソフトウェア基礎論研究会報告 No.22-3,1987 年10月.
[Taylor 87] S.Taylor et al.: A Layered Method for Process and Code Mapping, New Generation Computing, Vol.5, No.2, pp.185-205, OHMSHA, 1987.
[Ueda 85] K.Ueda: Guarded Horn Clauses, ICOT Technical Report, TR-103, 1985.