ICOT Technical Report: TR-310

TR-310

Reflection概念に基づく
並列論理型言語とその応用

田 中 二 郎

October, 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

Reflection概念に基づく並列論理型言語とその応用

A Reflective Guarded Horn Clauses and its Application

田中　二郎
Jiro TANAKA

（新世代コンピュータ技術開発機構）
ICOT Research Center

Abstract We try to describe the self-description of GHC from the "reflection" viewpoint first. Then we model the distributed computation model of GHC, which consists of GHC-machines, network managers and metacalls. We show how meta level instructions are executed in that system. GHC-machine and metacall can be obtained by extending the self-description of GHC. This paper assumes the basic knowledge of GHC.

Keywords reflection, GHC, metacall, meta system, distributed computation

1. はじめに

第五世代コンピュータプロジェクトとは『論理型言語』を出発点として知識情報処理を行なっていこうとするプロジェクトである。そのためにICOTでは、KL0(核言語第 0版)、KL1(核言語第 1版)、KL2(核言語第 2版)の三種類の核言語を当初より計画している。ここでKL0 とは逐次型、KL1 とは並列型の論理型言語である。既に、逐次型のKL0 についてはESP というユーザ言語が作られ、実用フェイズにはいっている。またKL1 [Furukawa 85] については、上田により提案されていたGHC [Ueda 85] を簡単化したFlat GHCをその言語の中核として採用し、現在、処理系や環境の開発を行っている。(Flat GHC とはGHC プログラムのガード部にシステム述語だけを許すように簡単化したもの。特に断らないかぎり、本稿でのGHC とは、Flat GHCのことである。)

さてKL2 であるが、GHC の延長線上でKL2 を考えることが既に決定されている[ICOT 87] 。どのように延長するかについては、今、様々な可能性を模索中であるが、その可能性の一つとして考えられているのがメタ機能を統一的に取り扱うためのReflection機能の導入である。すなわち、KL1 においては、中核言語であるGHC の枠内に収まらないものは、プラグマ[Shapiro 84]として考えたり、組み込み述語として、いわば、場当たり的に処理してきた。KL2 ではこれらのものをメタ機能として核言語の中になるべく取り込み、統一的に処理したい。こういったメタ機能には、

(1) データをプログラムとして呼び込む機能。高階機能。
(2) 動的な変化を感知する機能。実時間を知る機能。自己の体力を感知する機能。
(3) 動的な実行を指示する機能。実行制御機能。リアルタイム機能。

等があると考えられる。本研究はこうしたKL2 研究の一環としてなされているものである。

2. Reflective GHC

GHC は並列論理型言語であり、言語のレベルでプロセスを動的に生成でき、プロセス間の同期の記述なども容易である。これらのGHC の特徴から、GHC でOSを書いてみたり、並列システムのシミュレーションをやってみたらどうかということを思いつく。

しかしながら実際にこれらの事を試みると、意外と大変である。その原因としては、現在のGHC では、オブジェクトレベルとメタレベルの区別が曖昧で、かつメタ機能が不十分である事があげられよう。

例えば、いまGHC を走らせることができるGHC マシンがあると仮定し、そのOSについて考えよう。OSはゴールを読み込み、そのタスクを実行し、実行結果を出力できることが必要である。しかしながら現在のGHC ではオブジェクトレベルとメタレベルの機能の区分が不明確であるため、こうしたOSが簡単に記述できない。並列システムのシミュレーションにしても状況は同じである。通常、並列な実体に対応するプロセスをGHC のプロセスで記述するが、そのプロセスから発生するメタ・レベルの現象をGHC の枠内で扱えない。

こうしたGHC の欠点を補うため考えられているのがReflective GHC(RGHC)である。RGHCの考え方は3-LISP [Saith 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、Saith は自己記述（メタインタプリタ）を使っている。

我々のRGHCの実現の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Reflection実現のための機構は複雑になる。以下、GHC の自己記述の問題を考察し、例としてGHC の分散計算機の記述を挙げてReflection機能実現の方式についてのべる。

3. GHC の自己記述

プログラム言語の自己記述は、起源的にはLispなどで始められた技法であり、言語の特徴をメタインタプリタの形で表記する。Prologにおいても以下に示す 4行メタインタプリタというのが知られている [Boven 83] 。

```
exec(true) :- !.
exec((P,Q)) :- !, exec(P),exec(Q).
exec(P) :- clause((P:-Q)),exec(Q).
exec(P) :- P.
```

このメタインタプリタは、実行されるべきゴールがtrueであれば成功して終了する。実行されるべきゴールが複数個あれば、それを分解し、ひとつひとつをそれぞれ実行する。ゴールがtrueでも複数個でもないときは、述語clauseが、与えられたゴールにユニファイ可能な定義節をみつけ、その定義節のボディにゴールを展開する。また、それ以外のときは与えられたゴールはシステム述語であるのでそれを解く。

同様にGHC の一番簡単なメタインタプリタは以下のように記述できよう。

```
exec(true) :- true | true.
exec((P,Q)) :- true  | exec(P),exec(Q).
exec(P) :- not-sys(P)  | reduce(P,Body),exec(Body).
exec(P) :- sys(P)  | P.
```

GHC ではすべての定義節はガードを持っている事を除けば、このGHC のメタインタプリタはPrologのメタインタプリタとほとんど同じである。定義節を見つけるのにPrologではclauseを使っていたが、GHC では述語reduceをつかっている。述語reduceは、与えられたゴールにユニファイ可能な定義節をみつけ、それらの定義節の中でガードが解ける定義節を捜し、その定義節のボディにゴールを展開する。

しかしこのメタインタプリタはこのままではあまり役に立たない。このメタインタプリタを修正あるいは拡張し、有用な情報を取り出せるようにしたい。たとえば一番簡単な拡張として、execを二引数にしてexec(G,R) とすることが考えられる。このexecは、ゴールG を実行し、そのゴールが成功するとR にsuccess 、失敗するとR にfailure をかえす。これによってexecの中のゴールが失敗しても外の世界全体が失敗するのを防ぐことができる。

次に考えるのは、execのゴール実行を外の世界からコントロールしたいということである。すなわち、ゴールの実行を途中で

一時中断(suspend) したり、再開(resume)したり、またその仕事を放棄(abort) したりしたい。そのために考えられているのが、以下の形式の三引数execである[Clark 87]。

exec(G, I, O)

この三引数execのインプット・ストリーム Iからは suspend、resume、abort 等のコントロール情報をいれることができる。例えば、このexecの実行中にゴールの実行を一時中断したくなったらI を[suspend | I'] と具体化する。この時execは休止状態になり、execのインプット・ストリーム I' だけが次の指令を待つ。execを再開したくなった時にはI'を[resume | I'']と具体化する。exec自体が不要となったときには、インプット・ストリームI を[abort | I'] と具体化する。

また、execの実行が成功するとアウトプット・ストリーム Oは[success] 、失敗すると[failure] に具体化される。アウトプット・ストリームからは、この他に実行中に生じた例外やユーザへの出力など各種の情報が上がってくる

この三引数execは以下のように記述できる。

```
exec(true,I,O) :- true | O=[success].
exec(false,I,O) :- true | O=[failure(false)].
exec((A,B),I,O) :- true | exec(A,I,O1),exec(B,I,O2),omerge(O1,O2,O).
exec(A,I,O) :- sys(A),var(I) | sys-exe(A,I,O).
exec(A,I,O) :- not-sys(A),var(I) | reduce(A,I,Body,O,New0),exec(Body,I,New0).
exec(A,[susp | I],O) :- true | wait(A,I,O).
exec(A, [abort | I],O) :- true | O=[aborted].

wait(A,[resume | I],O) :- true | exec(A,I,O).
wait(A, [abort | I],O) :- true | O=[aborted].
```

このexecの特徴は、常に引数でメタ・レベルとの連絡がストリームとして確保されていることである。（すなわち、ゴールの成功や失敗という概念は、絶対的な概念ではなくメタ・レベルへのメッセージと考えられている。）（またこのプログラムの中で使われているvar(I)は、あくまでチェックした時点で情報が来ていない事を確かめているだけであることに留意されたい。）

次にこのGHC のメタインタプリタにスケジューリングキューを陽に導入する。3-LISPにおいてはプログラム全体の継続(continuation)がメタインタプリタ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。スケジューリングキューを陽に導入したexecは四引数になり、exec(H,T,I,O) で最初の二つの引数、H とT がスケジューリングキューの頭部と尾部を示している。（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る。）

四引数execのメタインタプリタは以下のようになる。

```
exec(T,T,I,O):-true | O=[success].
exec([true | H],T,I,O):-true | exec(H,T,I,O).
exec([false | H],T,I,O):-true | O=[failure(false)].
exec([A | H],T,I,O):-sys(A),var(I) | sys-exe(A,T,NT,O,NO),exec(H,NT,I,NO).
exec([A | H],T,I,O):-not-sys(A),var(I) | reduce(A,T,NT,O,NO),exec(H,NT,I,NO).
exec(H,T,[susp | I],O):-true | wait(H,T,I,O).
exec(H,T,[abort | I],O):-true | O=[aborted].

wait(H,T,[resume | I],O):-true | exec(H,T,I,O).
wait(H,T,[abort | I],O):-true | O=[aborted].
```

ここではexecはスケジューリングキューを持ち、ゴールを逐次的に処理する。今までのexecでは、reduceがゴールを複数個のゴールに展開したときにはそれを一つ一つのexecに分解し並列に処理していたが、このexecではそれをスケジューリングキュー

に入れて逐次的に処理していく。スケジューリングキューは、残りの仕事(Continuation)を格納していると考えられ、その導入は、プログラムに逐次性を導入すると共に、プログラムの実行の管理を容易にする。

4. GHC の分散計算機

次にこのGHC の自己記述の応用として、GHC の分散計算機について考える。ただしここでは物理的な構造のシミュレーションではなくマシンの機能の記述を考える。GHC の分散計算機とはGHC を高速に実行できるghc-machine 何台かを結合したものである。このようなghc-machine の結合の方式にはいろいろな可能性が考えられる。

①分散メモリ。ghc-machine どうしは独立で、それぞれ異なった変数領域をもち、交換されるメッセージは変数を含まない場合。
②共有メモリ。ghc-machine どうしが同じ変数領域を持ち、異なったmachine 間で変数の共有が可能な場合。
③仮想共有メモリ。ghc-machine どうしはハード的には異なった変数領域をもち、独立であるが、その上にのっているGHC の言語処理系レベルで（メッセージ交換などで仮想的に）共有メモリが実現されている場合。

①はインプリメントが簡単であるが、メッセージの中に変数を含まないようにするのはユーザの責任となりプログラムが難しくなる。②では同一の変数にアクセスの競合がおこるので、実際のインプリメントでは変数のロック機構を導入するなどしてアクセスの競合を防ぐことが必要になる。③では他のマシンの変数領域にアクセスできるが、自分のマシンの変数領域へのアクセスと比べ低速になる。

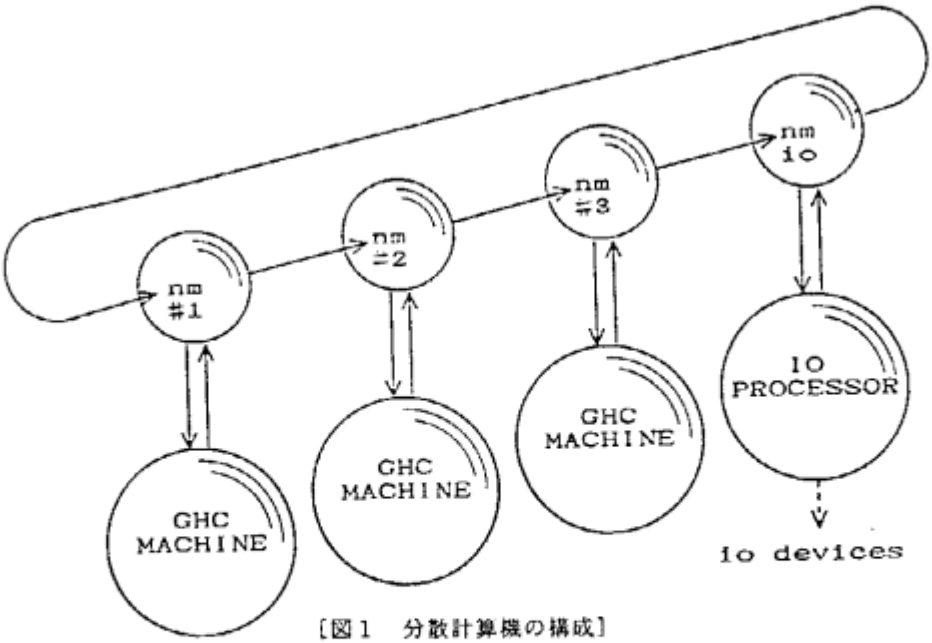
本稿ではICOTのMulti-PSI [Taki 86] 等を念頭におき、③のモデルを想定している。このような分散計算機においては、ghc-machine は、適当にゴールを他のマシンにメッセージの形で投げ、マシンどうしがメッセージをやりとりしながらプログラムを全体として実行していく。こうしたシステムにおける一つのポイントは、いかにして実行されるゴールを分散させるかという点である。これについてはユーザがプログラム中で他のマシンへゴールを投げる事を指定する方式（プラグマ方式）が、すでにShapiro により提案されている [Shapiro 84] 。本稿においてもこのプラグマ方式を採用することとした。

このような分散計算機の構成例を以下に示す。

```
dist-machine(Sys10):- true |
        nm($1,Nm1,Nm2,11,01), ghc-machine(H1,H1,0,11,01),
        nm($2,Nm2,Nm3,12,02), ghc-machine(H2,H2,0,12,02),
        nm($3,Nm3,Nm4,13,03), ghc-machine(H3,H3,0,13,03),
        nm(io,Nm4,Nm1,14,04), io-processor(H4,H4,0,14,04,Sys10).
```

ここでは3 つのghc-machine と一つのio-processorが、それぞれネットワークマネージャnmを経由して結合されている。それぞれのnmは左右のnmと 1方向にリング状に結合され、またghc-machine やio-processorとも双方向に結合されている。この結合の様子を図１に示す。

ここでネットワークマネージャnm(Id,Left,Right,1,0) は5 つの引数を持っている。最初の引数Idはこのネットワークマネージャの識別子、次の二つの引数は左右のネットワークマネージャとのリンク、最後の二つの引数1 と0 はghc-machine やio-processorとの入出力を表している。またghc-machine(H,T,N,1,0)も5 つの引数を持っている。最初の二つの引数H とT はghc-machine の持つスケジューリングキュー、次の引数N はこのghc-machine の中のゴールの総数、最後の二つの引数1 と0 は外の世界との入出力を表している。外の世界との入出力であるが、入力からは実行させたいゴールを入れることができる。また出力からは、他のghc-machine で実行させたいゴールを出す事ができる。またユーザへの入出力やプログラム実行中に生じた各種の情報等も出すことができる。



[図1　分散計算機の構成]

一般に、ghc-machine はメタレベルの命令に出会うとその命令を出力ストリームから放出する。メタレベルの命令はG@P の形をしており、以下のようなものがあると考えられる。

①入出力命令。各ghc-machine は入出力装置を持たないのでio-processorに命令を送ってやる必要がある。入出力命令はG@ioの形をしている。
②例外。プログラム実行中に生じた例外は、[Foster 87] が示唆しているように、Message(G,NewG)@exception という形をしていると想定する。
③プラグマ。本稿では一次元リング構造を想定しているので以下のようなプラグマを想定する。

G@forward	ゴールG をとなりのghc-machine に投げる。
G@id	ゴールG をidで指定されるghc-machine に投げる。
G@auto	ゴールG を一番負荷の低いghc-machine に投げる。

④リフレクション。Reflectiveな命令にはいろいろ考えられるが、ここではload(L)@reflect というghc-machine に自分の負荷状態を聞く命令を想定する。この命令の場合はghc-machine は自分の持つゴールの総数を答える。

5.　分散計算機の記述

前節までで我々の想定する分散計算機の概要について述べた。本節ではこの分散計算機の個々の構成要素、すなわち、ネットワークマネージャ、GHC マシン、メタコールについて述べる。メタコールとはGHC マシン上で実行されるユーザ・プログラムをシステムが制御する単位である。ここでは拡張されたメタインタプリタが一通りに使われる事に留意したい。一つは仮想的なGHC マシンであり、二つ目はメタコールの記述である。

5.1　ネットワークマネージャ

ネットワークマネージャnm(Id,L,R,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L と五番目の引数Out の二個だけである。まず二番目の引数L に情報が来たときは、以下のように振舞う。

```
nm(Id,[G@forward | L],R,In,Out):-Id≠Io |
        R-[G | R1],
        nm(Id,L,R1,In,Out).
nm(Id,[G@Id1 | L],R,In,Out):-id(Id1),Id≠Id1 |
        R-[G@Id1 | R1],
```

```
            nm(Id.L.R1.In.Out).
nm(Id.[G@Id | L].R.In.Out):-true|
            In=[G | In1].
            nm(Id.L.R.In1.Out).
nm(Id.[G | L].R.In.Out):-G ≠Goal@P.Id ≠Io|
            In=[G | In1].
            nm(Id.L.R.In1.Out).
nm(Io.[G | L].R.In.Out):-G ≠Goal@Io |
            R=[G | R1].
            nm(Io.L.R1.In.Out).
```

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。G@Idの形をしていればnmの持つ自分のIdと比べ同じであればghc-machine に送る、違えば右のリンクに転送する。それ以外は自分のghc-machine に送る。

また五番目の引数Out には必ずプラグマ付きメッセージが送られる。ghc-machine からOut に情報が来たときは、以下のように処理される。

```
nm(Id.L.R.In.[G@P | Out]):-P≠auto|
            send(G@P.Id.R.R1.In.In1).
            nm(Id.L.R1.In1.Out).
nm(Id.L.R.In.[G@auto | Out]):-true |
            In=[load(Len)@reflect | In1].
            R=[send-lowest(Id.Len.Id.G)@all | R1].
            nm(Id.L.R1.In1.Out).
```

メッセージがG@autoという形をしていなければ、メッセージに応じ、右のリンクに(@forward ならforward を一個はがし、@Id ならそのまま) 転送したり、自分の入力に送ったりする。G@autoと言う形であれば、まず自分のghc-machine に負荷を聞き、右のリンクに他のすべてのghc-machine の負荷を聞くsend-lowest メッセージを送る。送られたメッセージは以下のように処理される。

```
nm(Id.[send-lowest(Id1.LL.LId.G)@all | L].R.In.Out):-Id≠Id1 |
            In=[load(Len)@reflect | In1].
            which-lower(LL.Len.Lower.LId.Id.LowerId).
            R=[send-lowest(Id1.Lower.LowerId.G)@all | R1].
            nm(Id.L.R1.In1.Out).
nm(Id.[send-lowest(Id.LL.LId.G)@all | L].R.In.Out):-LId ≠Id|
            R=[G@LId | R1].
            nm(Id.L.R1.In.Out).
nm(Id.[send-lowest(Id.LL.Id.G)@all | L].R.In.Out):-true|
            In=[G | In1].
            nm(Id.L.R.In1.Out).
```

send-lowest メッセージは、最初にメッセージが発信されたId、それまでの load の最少値LL、そのId、処理されるゴールGを抱え、リングを一周する。リングを一周した時点で、Loadが最小値のIdにG を投げる。

5．2　GHC マシンの記述

次にghc-machine であるが、ghc-machine(Head.Tail.N.In.Out) はInにゴールが入力されると起動する。Inには、①処理系や

OSの仕事が入力される場合、②ユーザの仕事が入力される場合の二通りが考えられる。ここでは①ではゴールは裸のまま入力され、②の場合にはgoal(G,I,O,MaxRC) の形で入力されると仮定する。

```
ghc-machine(Head.Tail.N.[G | In].Out):-
        G =goal( _, _, _, _).G =load( _)@reflect |
        schedule(G.Tail.NewTail).
        NN:=N+1.
        ghc-machine(Head.NewTail.NN.In.Out).
ghc-machine(Head.Tail.N.[goal(G.I.O.MaxRC) | In].Out):-true |
        goal-server(C1.CO.I.O.MaxRC.Out1).
        merge(Out1.Out2.Out).
        schedule(G.H.T).
        schedule(call(C1.CO.H.T.O.MaxRC).Tail.NewTail).
        NN:=N+1.
        ghc-machine(Head.NewTail.NN.In.Out2).
ghc-machine(Head.Tail.N.[load(L)@reflect | In].Out):-true |
        L=N.
ghc-machine(Head.Tail.N.In.Out).
```

処理としては、①の裸のゴールの場合は、そのままghc-machine のスケジューリングキューに入り、②のgoal(G,I,O,MaxRC) の場合には、goal-server が作られて、同時にcall(C1,CO,H,T,O,MaxRC) がスケジューリングキューに入る。個々ではgoal-server はcallと組になって働き、callからの出力を適切な出力チャネルに振り分ける役目をする。またload(L)@reflect というReflectiveなメッセージの時には自分のゴール総数N をL に代入する。

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。①の場合にはそのまま処理がなされ、②の場合にはcall(I,O,H,T,N,NN,RC,MaxRC,B) やcall-susp(I,O,H,T,RC,MaxRC,B) が実行される。（これらについては、次の節で触れる。）

```
ghc-machine([G@P | Head].Tail.N.In.Out):-var(In) |
        Out=[G@P | Out1].
        ghc-machine(Head.Tail.N.In.Out1).
ghc-machine([Goal | Head].Tail.N.In.Out):-var(In).notsys(Goal) |
        reduce(Goal.Tail.NewTail.N.NN).
        ghc-machine(Head.NewTail.NN.In.Out).
ghc-machine([Goal | Head].Tail.N.In.Out):-var(In).sys(Goal) |
        solve-system(Goal.Tail.NewTail.N.NN).
        ghc-machine(Head.Tail.NN.In.Out).
ghc-machine([call(I.O.H.T.RC.MaxRC) | Head].Tail.N.In.Out):-var(In) |
        call(I.O.H.T.N.NN.RC.MaxRC.B).
        schedule(B.Tail.NewTail).
        ghc-machine(Head.NewTail.NN.In.Out).
ghc-machine([call-susp(I.O.H.T.RC.MaxRC) | Head].Tail.N.In.Out):-var(In) |
        call-susp(I.O.H.T.RC.MaxRC.B).
        schedule(B.Tail.NewTail).
        ghc-machine(Head.NewTail.N.In.Out).
```

ここで重要なことは、入力には①処理系やOSの仕事、②ユーザの仕事の二通りがあり、①では直接に処理がなされるが、②の場合にはcallやcall-susp が起動され、処理はそれらのメタコール述語に任されることである。

5．3　メタコールの記述

既に述べたように、メタコールとはGHC マシン上でユーザプログラムをシステムが制御する単位の事である。メタコール述語には九引数callと、そのcallがsuspend された状態であるのを示すのに使う七引数call-susp の二種類がある。本方式においてはメタコール自身スケジューリングキューを持っている。

九引数call(In,Out,Head,Tail,N,NN,RC,MaxRC,B)で、InとOut は外の世界との入出力、HeadとTailはメタコールの持つスケジューリングキューである。N とNNはghc-machine の持つゴールの数を数えるのに使われる引数で、RCはメタコールの現在までのリダクションの数、MaxRC はそのメタコールに許されるリダクションの数の最大値、B はghc-machine のキューにつめるメタコールを格納する変数である。

九引数callであるが、入力Inからメッセージが来たときにはcall-control述語を起動する。メッセージがsuspend であればBにcall-susp をつめる、またメッセージがabort であれば、B にはtrueをつめ終了する。

```
call([Control | In],Out,Head,Tail,N,NN,RC,MaxRC,B):-true |
        call-control(Control,In,Out,Head,Tail,N,NN,RC,MaxRC,B).
call-control(suspend,In,Out,Head,Tail,N,NN,RC,MaxRC,B):-true |
        Out=[suspended | Out1],
        NN=N,
        B=call-susp(In,Out1,Head,Tail,RC,MaxRC).
call-control(abort,In,Out,Head,Tail,N,NN,RC,MaxRC,B):-true |
        Out=[aborted],
        length(Head,Tail,N1),
        NN=N-N1,
        B=true.
```

入力Inからメッセージが来ないときには、スケジューリングキューからゴールを取り出し、通常の処理を行う。

```
call(In,Out,[G@P | Head],Tail,N,NN,RC,MaxRC,B):-
        var(In),MaxRC>RC |
        Out=[G@P | Out1],
        NN:=N-1,
        B=call(In,Out1,Head,Tail,RC,MaxRC).
call(In,Out,[Goal | Head],Tail,N,NN,RC,MaxRC,B):-
        var(In),MaxRC>RC,not-sys(Goal) |
        reduce(Goal,Tail,NewTail,RC,RC1,Out,Out1,BodyN),
        NN:=N+BodyN-1,
        B=call(In,Out1,Head,NewTail,RC1,MaxRC).
call(In,Out,[Goal | Head],Tail,N,NN,RC,MaxRC,B):-
        var(In),sys(Goal),MaxRC>RC |
        solve-system(Goal,Tail,NewTail,RC,RC1,Out,Out1,NGN),
        NN:=N+NGN-1,
        B=call(In,Out1,Head,NewTail,RC1,MaxRC).
call(In,Out,X,X,N,NN,RC,MaxRC,B):-var(In) |
        Out=[success(RC)].
```

```
                NN-N,
                B=true.
call(In,Out,H,T,N,NN,MaxRC,MaxRC,B):-var(In) |
                Out=[count-over],
                NN=N,
                B=true.
```

ここではスケジューリングキューから取り出されたゴールがプラグマ付きゴールであれば出力チャネルに送り出す。また通常のゴールであればそれを解く。callの引数N とNNはghc-machine のゴールの総数を計算するのに使われ、古い総数N を受けとっては新しい総数NNを計算して返す。またリダクションのRCは述語reduceの中で計算されている。

またcall-susp 述語は以下のように記述できる。

```
call-susp([resume | In],Out,Head,Tail,RC,MaxRC,B):-true |
                Out=[resumed | Out1],
                B=call(In,Out1,Head,Tail,RC,MaxRC).
call-susp([abort | In],Out,Head,Tail,RC,MaxRC,B):-true |
                Out=[aborted],
                B=true.
call-susp(In,Out,Head,Tail,RC,MaxRC,B):- var(In) |
                B=call-susp(In,Out,Head,Tail,RC,MaxRC).
```

call-susp はメッセージがresumeであればB にcallをつめる、またメッセージがabort であれば Bにtrueをつめ、終了する、また入力に何も来ていない時には、そのままB につめる。

6. まとめ

本報告では、まずGHC の自己記述の問題を考察し、次にGHC の分散計算機のモデルを示し、そこに必要とされるメタレベルの命令について考察を行った。さらに分散計算機の記述を、ネットワークマネージャ、GHC マシン、メタコールの三点から行い、そこでメタレベルの命令がいかに実行されるかを示した。特にGHC マシン、メタコールについてはGHC の自己記述を拡張することにより得られることを示した。

なお、本報告であげたGHC マシンは非常に簡単化したもので、プログラミングシステムとしては最小限の機能しか含んでいない。しかしながらこの計算機の記述を拡張して[Tanaka87a, Tanaka 87b] のような一般的なプログラミングシステムとすることは比較的容易である。また、本研究は、Shapiro やFosterらの研究成果[Shapiro 83, Shapiro 84, Silverman 86, Foster 86, Foster 87, Clark87] を踏まえている事を指摘しておく必要があろう。

この考察でもわかるように、Reflection概念に基づいたGHC の枠組みは非常に簡潔かつ強力である。従って、この枠組みの上にソフト的に柔軟かつ強力なシステムを構築する事が可能である。

なお、最初に述べたように、本研究はKL2 研究の一環としてなされているものである。所期の目標を達成するために、今後いっそうの考察および研究が必要である。

7. 謝辞

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。日頃、本研究に関連してディスカションの相手になってくれるICOT第一研究室の諸氏に感謝する。なお、本研究の一部は、富士通SSL の太田祐紀子、的野文夫の両氏に負っている。

［参考文献］

[Boven 83] D.L.Boven et al.: DECsystem-10 Prolog User's Manual.University of Edingburgh. 1983.

[Furukawa 85] 古川、竹内、宮崎、上田、田中：核言語第一版説明資料、ICOT、1985年 5月.

[Clark 87] K.Clark and I.Foster: A Declartive Environment for Concurrent Logic Programming. Lecture Notes in Computer Science 250. TAPSOFT'87. pp.212-242. 1987.

[Foster 86] I.Foster: The Parlog Programming System (PPS). Vesion 0.2. Imperial College of Science and Technology. 1986.

[Foster 87] I.Foster: Logic Operating Systems: Design Issues. In Proceedings of the Fourth International Conference on Logic Programming. Vol.2. pp.910-926. MIT Press. May 1987.

[ICOT 87] ICOT: 核言語第二版(KL2) の検討. 16pp.. 1987.

[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 International Conference on Fifth Generation Computer Systems 1984. pp.458-470. ICOT. 1984.

[Silverman 86] W.Silverman et al.: The Logix System User Manual. Version 1.21. Weizmann Institute. Israel. July 1986.

[Smith 84] B.C.Smith: Reflection and Semantics in Lisp. In. Proc. of 11th POPL. Salt Lake City. Utah. pp.23-35. 1984.

[Taki 86] K.Taki: The Parallel Software Research and Development Tool: Multi-PSI System.日仏AIシンポジウム 86. ICOT. pp.365-381. Oct. 1986.

[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. 22-3. 1987年10月.

[Ueda 85] K.Ueda: Guarded Horn Clauses. ICOT Technical Report. TR-103. 1985.