Architecture Abstraction in GHC

5Q-6

田中二郎

新世代コンピュータ技術開発機構

本稿では、並列論理型言語を用いた計算機構造の抽象的記述に関して考察を行う。例として並列論理型言語の分散計算モデルを考える。

1. はじめに

GHC[1]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、オブジェクトのレベルとメタのレベルの機能の区別が曖昧で、かつメタ機能が不十分である事があげられよう。

例えばOSはゴールを読み込みそのタスクを実行し、その実行結果を出力する。しかしながら現在のGHC ではオブジェクトのレベルとメタのレベルの機能の区別が曖昧であるため、こうしたOSが簡潔に記述できない。

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。

そこで本稿では、GHC を変更しオブジェクトのレベルとメタのレベルの機能を統一的に扱えるようにしたReflective GHC(RGHC)で考察を行う（以下本稿で言うGHC とはすべてRGHCのことである）。RGHCの言語仕様の詳しい説明は他の機会に譲るが、考え方としてはS-LISP[2] 等にヒントを得て、ICOTにおけるKL2[3]研究の一環として考察中の言語である。

2. 計算機の抽象的記述

仮想的なghc-machine を考え、その記述を考える。ただしここでは記述と言っても物理的な構造のシミュレーションではなくmachine の機能の記述を考える。まずGHC プログラムの評価をメタ・インタプリタの形式で記述したプログラムを下に示す。（以下本稿で示すプログラムは、説明のため極度に簡単化したものである。より具体的なプログラムは[4] などに示されている。）

```
exec(true.In.Out) :- true  | Out-[success].
exec(false.In.Out) :- true | Out-[failure].
exec((A.B).In.Out) :- true | exec(A.In.O1).
    exec(B.In.O2).omerge(O1.O2.Out).
exec(A.In.Out) :- sys(A).var(In) |
    sys-exe(A.In.Out).
exec(A.In.Out) :- not-sys(A).var(In) |
    reduce(A.In.Body.Out.NewOut).
    exec(Body.In.NewOut).
exec(A.In.Out) :- reflect(A).var(In) |
    Out-[goal(A.In.Out1) | Out1].
exec(A. [abort | In].Out) :- true | Out-[aborted].
        [RGHCのメタ・インタプリタ]
```

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。ここではゴールの成功や失敗という概念は絶対的な概念ではない。むしろそれはメタ・レベルへのメッセージと考えられる。また、入出力や例外もメタ・レベルへのメッセージとして処理される。

このexecは、本来、メタ・インタプリタの形式で用意するのではなくprimitive として処理系の中に用意されていることが望ましい。execを使えばghc-machine は以下のように表現できる。

```
ghc-machine(IdList.[exec(A) | I].O):- true |
    gen-id(Id).
    proc-server(run.Id.Out1.In.Out).
    exec(A.In.Out).
    ghc-machine([(Id.In) | IdList].I.Out2).
    omerge(Out1.Out2.O).
            [GHC マシンの記述]
```

すなわちghc-machine は、入力ストリームからのゴールの入力により、それに対応するIdをgen-idで作ったあと、proc-server とexecを起動する。本稿では細かい記述は省略するが、proc-server は起動されたタスクの管理を行ない、proc-server とexecの対でメタコールの機能を果たす事に留意する必要がある[4]。

3. GHC の分散計算機

次にGHC の分散計算機について考える。分散計算機とはghc-machine 何台かが専用の高速ネットワークで結合され、適当にゴールを他のmachine にメッセージの形で投げ、machine 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システムである。ghc-machine の結合の仕方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（仮想的には）

Architecture Abstraction in GHC
Jiro TANAKA
Institute for New Generation Computer Technology

他のmachine の変数領域にアクセスできるが、自分のmachine の変数領域へのアクセスと比べ低速である場合。

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。本稿ではICOTのMulti-PSI[5]等を念頭におき、③のモデルを想定する。

また、あるmachine から他のmachine へゴールを投げるときの指定の仕方であるが、これについてはユーザの責任でプログラマがプログラム中で指定する（例えば、G@P と指定すると、ゴールG をP で指定されるmachine に送れという意味）と仮定する（この指定の事をプラグマと呼ぶ）。プラグマのexecでの処理は簡単である。@ はreflectiveなoperatorであるので、前述のメタ・インタプリタでも明らかなように、G@P は実行の途中でそれの入出力ストリームとともに出力ストリームに割り出される。後はghc-machine どうしを結ぶnetwork が、G を適切なmachine に送る。転送されたゴールが変数を含む場合、そのゴールの評価にあたり元のmachine の変数をアクセスすることになる。その場合には、仮定により自分のmachine の変数領域へのアクセスと比べ低速となる。（そのためそれらのトレードオフを考慮しプラグマを付ける必要があるが、それらはプラグマを付けるプログラマの責任になる。）

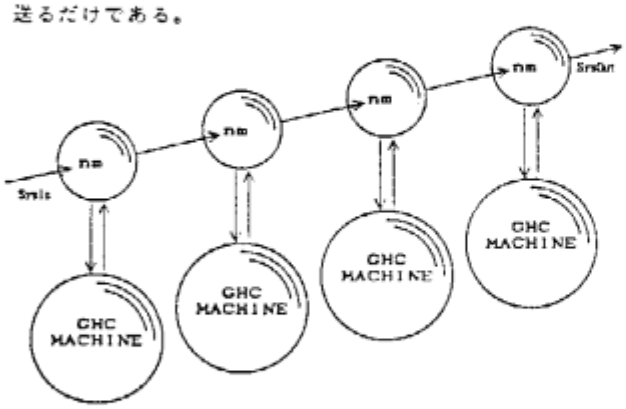
4．分散計算機の記述

前章に示したような分散計算機の構成例を以下に示す。

```
dist-machine(SysIn, SysOut):- true |
    nw(SysIn,Nw1,11,01), ghc-machine(11,01),
    nw(Nw1,Nw2,12,02), ghc-machine(12,02),
    nw(Nw2,Nw3,13,03), ghc-machine(13,03),
    nw(Nw3,SysOut,14,04), ghc-machine(14,04).
```

［分散計算機の構成例］

この例では4 つのghc-machine が一次元状に結合されている（以下の図を参照）。network-manageの記述は省略するが、nwは単にSysIn やghc-machine 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出力ストリームに送るだけである。



［一次元状に結合されたghc-machine］

ghc-machine はSysIn にゴールが入力されるとプロセスを起動する。これはまったく前と同じである。違うのは、ghc-machine がこのほかに他のmachine から投げられたゴールを処理しなければならない点である。

それにはghc-machine に以下のような定義を付け加えてやれば良い。

```
ghc-machine(IdList,[goal(A,In,Out)|I],O)
    :- true |
        exec(A,In,Out),
        ghc-machine(I,O).
```

しかしながら、この場合せっかく起動されたexecも、入出力は元のproc-server に繋がっているInとOut を使うことになり、これでは実行効率が低下する。そこでproc-server のlocal な代理として、各ghc-machine に一つlocal-serverを作ることにすれば効率の問題は解決する。すなわち投げられたゴールが来た場合、そのプロセスについてすでにlocal-serverが出来ているかを調べる。出来ていればそのlocal-serverの管理下でゴールを起動する。そうでなければ新しくlocal-serverを作り、その下でゴールを起動すれば良い。

この方式は実際のMulti-PSI においては、分散メタコールの実現として、処理系に組み込まれ実現されている方式である[6]。本稿ではlocal-serverとexecの対でソフト的に実現されていることに留意する必要がある。

5．まとめ

以上、RGHCを用いた計算機構造の抽象的記述に関する考察を行った。結論として言えることは、このRGHCの枠組みは非常に簡潔かつ強力であると言うことである。

このRGHC処理系の上には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。本稿で挙げた仮想分散計算機もまたその一例であろう。こういった特色はGHC 処理系の軽量化及び効率化に繋がると思われる。

なお、本研究は第五世代コンピュータ・プロジェクトの一環として実施されたものである。

［参考文献］

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

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

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

[4] 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.

[5] 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.

[6] 宮崎　他: Multi-PSI におけるFlat GHCの実現方式. The Logic Programming Conference '86.ICOT. pp.83-92. June 1986.