ICOT Technical Memorandum: TM-0861

TM-0861

GHCによるアドバンスト・リフレクションの試み

田中二郎(富士通)

February, 1990

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 によるアドバンスト・リフレクションの試み

田中二郎

富士通・国際研
(電子メイル: jiro@iias.fujitsu.co.jp)

あらまし　本論文では、まず基本的な概念として、「メタ」と「リフレクション」を定義し、つぎに論理型言語における「メタ計算システム」を定義する．そこにおいて、特に、「メタ・システムにおける表現」と「オブジェクト・システムの実体」の対応について考察を行い、これらの考察に基づき、新たに並列論理型言語GHCによるアドバンストなリフレクティブ・システムを提案する．本システムは、従来までの研究と比較し、(1)並列論理型言語GHC によるquote などを含まないリフレクションの簡潔な定式化であり、(2)体系的な考察に基づいてリフレクティブ・タワーを実現している、などの特徴を持つ．

Advanced Reflection in GHC

Jiro Tanaka

IIAS-SIS, Fujitsu Limited
1-17-25, Shinkamata, Ota-ku, Tokyo 144, Japan
(Email: jiro@iias.fujitsu.co.jp)

Abstract　After defining "meta" and "reflection" as basic terminology, we define "meta-computation system" in logic programming languages. Especially, we consider the correspondence between the representation at the meta-level and the real objects at the object-level. Based on these considerations, we propose "advanced reflective sysytem" in parallel logic language GHC. This system has the following two features comparing to the previous approaches: First, this system is formulated in an elegant manner without using "quote." Secondly, it has the avdanced reflective mechanism in which "reflective tower" can be realized.

1. はじめに

「究極のプログラミング言語」のあるべき要件をひとことで表現するなら、それは「簡潔にして（人間に）分かり易く（記述力が）強力な言語」ということになろう．過去のプログラミング言語発展の歴史をかえりみるとき、プログラミング言語は、与えられたハードウェアなどの制約のもとで、つねにこのような言語を求めて発展してきた．
最近、こうした「簡潔な枠組みで強力な記述力をうるための機構」の一つとして注目を集めている話題に「メタ」や「リフレクション」がある．これらについて、既に筆者らは、RGHCという言語を提案し、幾つかの提案を行ってきた〔Tanaka88-1, 88-2, 90〕．今回、表題で「アドバンスト・リフレクション」という言葉を使用したのは、従来までの、やや応用指向の提案とは異なり、体系的な考察に基づく本格的なリフレクション機構を試みたいという趣旨である．

なお、本研究の動機であるが、並列論理型言語 GHC〔Ueda 85, Furukawa 87〕については、第５世代コンピュータ・プロジェクトの核言語として、ICOTを中心に研究開発が進められきた．GHC はpureな並列言語であり言語レベルで同期やプロセスなどの概念をサポートしているが、反面、並列アーキテクチャ上での実現を想定し、言語仕様を極度に単純化していて、Prologとくらべても貧弱な記述力しか持っていない．このプロジェクトではハードウェアからはじまり、オペレーティング・システム、アプリケーションまでのすべてを核言語をもとに構築する計画であり、このままでは困る．そこで記述力強化の方法としてリフレクションに着目し、3-Lisp〔Smith 84〕のようなことをGHC でも出来ないかと思ったのが研究の発端である．

2. 「メタ」と「リフレクション」

最初に「計算システム」を定義して、それをもとに「メタ」と「リフレクション」を定義したい．ここではMaesの用いた説明〔Maes 86 〕をもとに説明する．

まず計算システムであるが、そのモデル化の仕方にはいろいろな方法が考えられる．例えば、通常のモデル化であれば、「計算システム」は「実行器」や「プログラム」、「データ」などを含む系として考えるのが普通であろう．これを図に示したのが図1 である．ここで実行器は計算機のCPU に対応する．われわれは計算システムの「プログラム」や「データ」に、考えている問題領域のある特性をモデル化し、それを実行器で解くわけである．一般に「プログラム」には問題解決のアルゴリズムをモデル化し、また「データ」には、問題領域の構造が表現される．

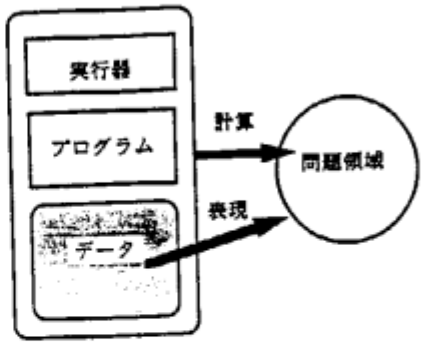

図１　　計算システム

2.1　メタ・システム

メタ・システムとは、その計算システムの問題領域がまた計算システムであるようなシステムである．それを図に示したのが図2 である．

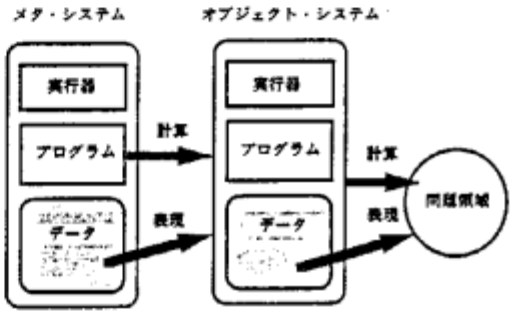

図2　メタ・システム

すなわちメタ・システムのプログラムやデータは、問題領域であるほかの計算システム（オブジェクト・システム）をモデル化したものとなっている．とくにメタ・システムのプログラムは、「メタプログラム」と呼ばれ、オブジェクト・システムの問題解決のアルゴリズムをモデル化したものになっている．またメタ・システムのデータにはオブジェクト・システムの構造をモデル化したもの（すなわちオブジェクト・システムの表現）が入る．

メタ・システムの実現方式としては、オブジェクト・システムの問題解決のアルゴリズムをメタインタプリタの形式で表現するのが一般的である．しかしながら、こういったシステムは必ずしもメタインタプリタで実現されなければならないというわけではない．何らかの形でメタシステムからオブジェクト・システムが定義できればよいのである．歴史的には、メタインタプリタによらないメタ・システムとしてはWeyhrauch のFOL 〔Weyhrauch 80〕、メタインタプリタとしては、Lisp 1.5におけるLispの万能関数の記述などがよく知られている．

2.2　リフレクティブ・システム

リフレクションとは「自分自身」について感知したり、「自分自身」を変更したりすることである．このような能力を計算システムが持つならば、そのような計算システムはプログラムを実行中に、現在の状態（すなわち実行器、プログラム、データなどの様子）を感知（自己感知）し、それに応じた行動（自己変更）を取れるようになる．リフレクティブなシステムを図に示したのが図３である．

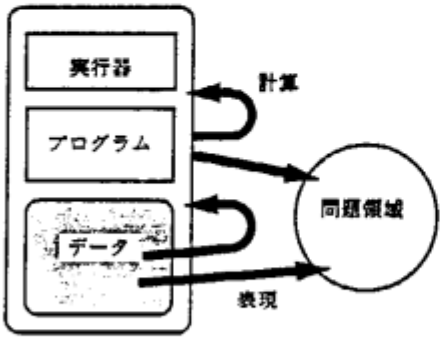

図3　リフレクティブ・システム

この図は、図２のメタ・システムとオブジェクト・システムを重ねたものと考えればよい．すなわち、図２ではメタ・システムがオブジェクト・システムの状態を感知したり変更したりすることができたが、リフレクティブ・システムでは、計算システム自身が自己の状態を感知したり、変更したりできるようになっている．

このようなシステムを実現するには、まず計算システム自身がデータとして表現されている必要がある．そして、そのように表現された計算システムを動的に感知したり、変更したりする仕組みを計算システムの中に提供すればよい．このような実現方式にはメタ・システムを用いるのが簡単である．メタ・システムの中では、オブジェクト・システムがデータとして扱われるので、あとはオブジェクト・システムからメタ・システムへ情報を渡したり、また逆方向に情報を返す手段を提供すれば、オブジェクト・システムは自分自身の状態を感知したり変更できることになり、リフレクティブ・システムとして動く．手段の実現方式としては、大きく二つが考えられる．

一つは、連絡手段として、幾つかの組込述語を用意し、それらによって感知したい情報を直接オブジェクト・システムの中に持ち込むことである．実は筆者らが、従来、RGHCとして提案したシステム〔Tanaka 88-2〕では、このようにしてリフレクションを実現していた．

もう一つの方法は、リフレクティブな述語がオブジェクト・システムで呼ばれると自動的にメタ・システムに制御が移るようにすることである．メタ・システムは、オブジェクト・システムの表現に関して必要な計算を行い、必要な計算が終了すればオブジェクト・システムに戻ればよい．このようなリフレクションの枠組みについて考えたのは、B. C. Smith の3-Lisp〔Smith 84〕が最初である．

2.3　3-Lispとリフレクティブ・タワー

3-Lispではリフレクティブな述語がオブジェクト・システムで呼ばれると自動的にメタ・システムに制御が移る．メタ・システムでは、現在実行中のオブジェクト・システムのcontinuation（プログラムの継続）とbinding environment（変数の束縛環境）がデータ化されており、メタ・レベルではそれを自由に取り出したり、変更を加えたりすることができる．こうした機能を使用すると、いままでは組込関数として定義されていた関数もユーザ定義関数として記述することができる．（たとえば、bound 、catch/throw 、lambda、if、quote などもユーザ定義関数として記述できる．）これらの特徴は、小さなコアで強力なシステムを記述することにつながる．

3-Lispではメタ・システムはオブジェクト・システムと同一の計算システムであり、メタ・システムを実行中にもメタを要求する可能性があり、原理的にはメタのメタ、メタのメタのメタという具合にメタの無限タワー（リフレクティブ・タワー）を考える必要がある．

概念的には、最初からメタの無限タワーが存在し、それらが統一性を持って動いていると考えてもよい．しかしながらこのような無限タワーのモデルは明らかにインプリメントが困難である．そこで実際のインプリメントでは、例えば、最初にオブジェクト・システムとともにメタ・システムを動的に構築しうるようなシステム（サポート・システム）を動かし、必要に応じてそこからメタ・システムを構築する技法が使用される．

また、メタ・レベルの実行が終われば、効率の点からも、可能であればメタ・レベルでのインタプリタ実行からオブジェクト・システムの直接実行に戻ることが望ましい．しかしながら実際のインプリメントではそういった機能を省略し、メタ・レベルでのインタプリタ実行を続けるようになっているものが多い．

3.　論理型言語におけるメタ計算システムの記述

論理型計算システムでは、図４に示すよう、「計算システム」の要素として、「実行器」と「データベース」、「実行ゴール」、「変数束縛」などを考えることができる．これは図１の詳細化になっていて、「データベース」の中には「プログラム」が格納される．また「実行ゴール」や「変数束縛」は図１の「データ」に対応するもので、「実行ゴール」には、最初、実行すべき「初期ゴール」が格納され、計算の過程では「実行中のゴール列」が格納される．また「変数束縛」には、これらのゴール列に含まれる変数の束縛環境が格納される．

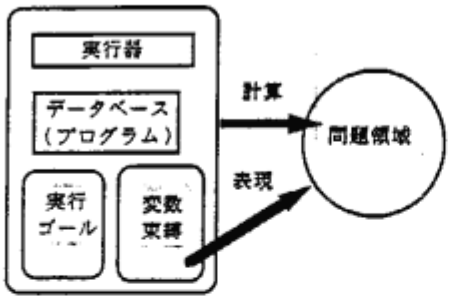

図4　論理型計算システム

3.1　簡単なメタプログラミング

メタプログラムは、Prologにおいては、いわゆるProlog in Prolog（すなわちPrologによるPrologの表現）として知られてきた．これについては、以下の4行プログラムがよく知られている〔Bowen 83〕．

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

ここで、まずオブジェクト・システムのプログラムはmy-clause の中で定義しておく、また、初期ゴールP はexec(P) という形で実行する．すると、このexecは以下のように動作する．

(1) まず、実行すべきゴールがtrueであればゴール実行が成功して終了する．
(2) 実行す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．
(3) 実行すべきゴールがtrueでも複数個でもないときには、述語my-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部にゴールを展開し、そのゴールを実行する．
(4) それ以外のときには、与えられたゴールが組込述語であるかチェックし、そうであればそれを解く．

このように、この4行プログラムは簡単であるが、これでちゃんとProlog in Prologになっていることがわかる．

また、並列論理型言語GHC でも、これとまったく同様に、GHC in GHCを以下のように記述することができる．

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

このプログラムの意味は自明であるので、詳しい記述は省略する．

なお、こうしたProlog in PrologやGHC in GHCにおいて、オブジェクト・レベルとメタ・レベルの間の対応関係は以下のようになっている．

オブジェクト・レベル	メタ・レベル
変数	変数
定数	定数
関数記号	関数記号
述語記号	関数記号
定義節	(特殊な) 定義節

3.2　本格的メタ・システムの構築

しかしながら、このProlog in PrologやGHC in GHCをメタ・システムとして見た場合、以下の点で不十分である．

(1) このプログラムではプログラム実行の表層部がシミュレートされているだけであり、Prologの場合にはユニフィケーションやバックトラックなど、またGHC の場合にはヘッド・ユニフィケーションやガード実行、コミット、ボディ部のユニフィケーションなどのより詳細な実行過程やそれによる変数の束縛環境の変化などの情報がメタレベルで取り出せない．

(2) オブジェクト・レベルの変数にはメタ・レベルでも変数が対応している．従って、メタ・レベルで変数の表現についての（メタ的な）操作を行うことができない．すなわち、ある変数が未定義であるかをチェックする「var 述語のインプリメント」や、ある変数が他の変数と同一のものであるかをチェックする「同一性のチェック」などを行うことができない．

(3) オブジェクト・レベルの述語記号にはメタ・レベルでの関数記号が対応している．またPrologの場合、オブジェクト・レベルの定義節には、述語記号my-clauseで始まる特殊な定義節が対応している．従って、メタ・レベルとオブジェクト・レベルの定義節は区別が可能であるが、メタ・レベルでオブジェクト・レベルの定義節がデータとして表現されているとは言い難い（これはGHC の場合でも述語reduceをさらに記述するとき、同じ問題に直面する）．オブジェクト・レベルの定義節を操作するにはassert、 retractなどの非論理的な組込述語に頼らなくてはならない．

従って上記の欠点を持たないような本格的メタ計算システムを考えたい．本格的メタ・システムの概要を図に示したのが、図5である．この図でわかるように、メタ・システムでは、『実行ゴール』に対応する部分の中にオブジェクト・システムの表現、すなわちオブジェクト・システムの実行器、データベース、実行ゴール、変数束縛などの表現がすべて格納される．またメタ・システムのデータベースはこのオブジェクト・システムの表現の操作法についての情報を格納し、メタ・システムの変数束縛には、これらの表現に含まれるメタ・レベルの変数の束縛環境が格納されている．

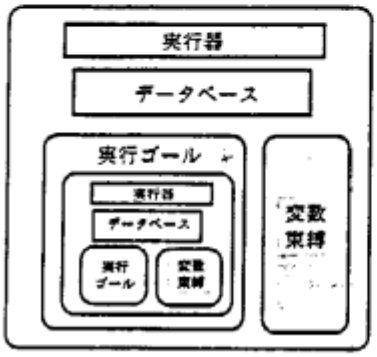

図5　論理型メタ計算システム

4. メタ・システムにおけるオブジェクト・システムの表現

オブジェクト・システムの実体のメタ・システムでの表現であるが、2.1 で述べたように、オブジェクト・システムの実体がメタ・システムで操作できるためには、それらがすべてデータとして表現されていることが必要である．われわれの提案する本格的メタ・システムにおいては、オブジェクト・レベルとメタ・レベルの対応は以下のようになる．

4.1　定数、関数記号、述語記号

オブジェクト・レベルの定数や関数記号は、メタ・レベルでも同じ定数や関数記号に対応させる．また3.1 の4 行プログラムと同様、述語記号は関数記号として扱われる．

これらについて、3-Lispのように、例えば『3 』に『'3』を対応させるといった可能性も考えられるが、われわれはそのような方式を採用しない．理由としては論理型言語はLispなどの言語とは異なり、評価するという概念をもたないからである．Lispでは、この評価という概念を使用して、『式の変形』と『変数の束縛値』を求めることの二つを行っていた．『' 』(quote) は、そうしたことをするかどうかを指示するのに必要であったが、こうしたことは論理型言語では必要ない．

論理型言語では『導出』という概念があるが、これは項の中に含まれる変数の値を具体化していくことに対応し、具体化されたものが『変形』されることはない．また『変数の束縛値』を求めることは論理型言語でも必要であるが、変数は『導出』の過程でつねにその束縛値が問題となるので、quote を用いて束縛値を求めるかどうかを指示する必要がない．

4.2　変数

3.2 でも述べたように、オブジェクト・レベルの変数にメタ・レベルの変数を対応させた場合、メタ・レベルで、変数の表現に関する（メタ的な）操作を行うことができない．こうしたことを実現するためには、『ある変数がどこの変数セルで実現されている』といった、変数の表現についての情報が必要である．そこでオブジェクト・レベルの変数は、メタ・レベルではそれが変数の表現であることがわかるような『特殊な基底項(ground term) 』に1:1 に対応させる．

リフレクティブ・タワーなどを考えるときには、メタ・レベルの上にもメタ・メタ・レベルがあり、そのうえにもメタ・メタ・メタ・レベルがあって…という具合に無限にメタの階層ができる可能性がある．したがって変数の『特殊な基底項』としての表現には、その変数の属するレベルについての情報を示すことが必要である．われわれのシステムにおいては、オブジェクト・レベルの変数は大文字で示されるが、メタ・レベルの中では変数を『@数字』の形式で1:1 対応になるように表現する（ここで数字は変数に対して一意的に割り当てられる整数である）．また、メタ・メタ・レベルでは、オブジェクト・レベルの変数を『!@数字』と表現し、同様にして、レベルが上がるごとに『!!@数字』、『!!! @数字』、『!!!!@数字』…などと表現する．

4.3　項

4.1 と4.2 のコンシステントな拡張として、オブジェクト・レベルの項には、メタ・レベルでもそれに対応する項を対応させる．ここでオブジェクト・レベルの項が含む定数や関数記号には、メタ・レベルでも同じ定数や関数記号を対応させる．またオブジェクト・レベルの変数は、メタ・レベルでは変数の表現である『特殊な基底項』に置き換える．

例えば、今、オブジェクト・レベルで、

```
p(a,[H|T],f(T,b))
```

という項があったとすると、メタ・レベルでは

```
p(a,[@1|@2],f(@2,b))
```

メタ・メタ・レベルでは

```
p(a,[!@1|!@2],f(!@2,b))
```

という基底項におきかわる．

4.4　変数束縛

オブジェクト・レベルの変数束縛は、メタ・レベルでは、変数の表現とその値を示す対のリストとして基底項で表現される．以下に変数の表現とその値を示す対の例を示す．

(@1, undf) … 変数@1の値は未束縛である．
(@2, a) … 変数@2の値はa である．
(@3, @2) … 変数@3の値は@2への参照ポインタである．
(@4, f(@1,@2)) … 変数@4の値は、構造体であり、その関数記号はf 、第1 引数は@1、第2 引数は@2への参照ポインタである．

上からも明らかなように、これらの対はオブジェクト・レベルのメモリ・セルの表現とでも言えるものである．また変数から変数への参照ポインタは通常のPrologのインプリメントと同様、二つの変数がユニファイされたときに生ずるものである．このため、一般にユニフィケーションなどで変数の値が要求されるときには、この参照ポインタをたぐってその値を求めるdereference とよばれる操作が必要となる．

4.5 定義節

またオブジェクト・レベルのプログラム、すなわちアサートされた定義節の集合も、メタ・レベルでは、定義節の表現のリストとして、基底項で示される．

例えば、appendのプログラム

```
append( [A | B ],C,D) :- true |
            D = [A | E ], append(B,C,E).
append( [] ,A,B) :- true  | A = B.
```

は、メタ・レベルでは

```
[(append( [@1 | @2] ,@3,@4):-true |
     @4= [@1 | @5] ,append(@2,@3,@5)),
 (append( [] ,@1,@2):-true | (@1 = @2))]
```

といった定義節の表現のリストで示される．

4.6 オブジェクト・システムとメタ・システムの対応

以上のことをまとめると、オブジェクト・レベルとメタ・レベルの対応は以下のようになる．

オブジェクト・レベル	メタ・レベル
定数	定数
関数記号	関数記号
述語記号	関数記号
変数	特殊な基底項
項	基底項 (変数を基底項に置き換え)
変数束縛	基底項 (変数の表現と値の対のリスト)
プログラム (定義節のリスト)	基底項 (定義節の基底表現のリスト)

4.7 メタ・レベルにおけるオブジェクト・レベルのゴールの実行

4 行プログラムではメタ・レベルでオブジェクト・レベルのゴールP を実行するには、ゴール「exec(P) 」をメタ・レベルのゴールとして実行すればよかった．ここで記述した本格的メタ・システムにおいては、これらのかわりにゴール「exec(P $_m$,Env,Db)」をメタ・レベルのゴールとして実行する必要がある．ただし、ここでP $_m$ はオブジェクト・レベルのゴールP のメタ・レベルでの表現，Env はオブジェクト・レベルの変数束縛のメタ・レベルでの表現(通常Env の初期状態は空である)，Db は実行すべきプログラムのメタ・レベルでの表現である．

またこのシステムは変数やプログラムの管理を行うので、メタプログラムは先の4 行プログラムよりかなり複雑になる．本稿では、このメタプログラムの記述は省略するが、変数管理をするGHC のメタインタプリタについては、既に〔Tanaka 88-3 〕にその記述がある．

5. リフレクティブ・システムの構成

これまでの考察に基づき、リフレクティブ・システムの構成について考える．まずリフレクティブ述語の定義を行い、次に、メタ・レベルの表現とオブジェクト・レベルの実体を結びつける操作であるupとdownについて述べる．最後にこれらの操作を実現するリフレクティブ・システムの構成方法について述べる．

5.1 リフレクティブ述語

リフレクティブ述語は、実行されるとリフレクションを起こすようなユーザ定義述語のことであり、このリフレクティブ述語をユーザ・プログラムやユーザ実行ゴールで自

由に使用することができる．

たとえばp(X,Y)というリフレクティブ述語をオブジェクト・レベルで呼び出すと、この述語はメタ・レベルに上がって実行される．メタ・レベルではオブジェクト・レベルの計算システムの構成要素をデータとして見ることができる．すなわち、オブジェクト・レベルで実行中のゴール列の表現G 、オブジェクト・レベルの変数環境の表現Env 、オブジェクト・レベルのプログラムの表現Dbなどを見ることができ、それらをリフレクティブ述語により、変更することができる．

述語p の場合であれば、われわれはリフレクティブ述語を

```
reflect(p((X,Y),(G,Env,Db),(NG,NEnv,NDb,R)))
                        :- guard | body.
```

という形式で定義することができる．(これは3-Lispのreflective procedureの定義にヒントを得ている)．

リフレクティブ述語はメタ・レベルに上がって実行される述語と考えることができる．したがって、リフレクティブ述語の定義の中では、p の引数は、上に示したように、メタ・レベルのものに変換されている．すなわち、リフレクティブ述語においては、引数は三つのグループからなる．最初のグループは、オブジェクト・レベルの引数に対応するものであり、ここでは、X とY が、オブジェクト・レベルでのp の引数のメタ・レベルでの表現に対応する (pの引数そのままではなく、それのメタ・レベルでの表現であることに注意)． 二つ目のグループは、オブジェクト・レベルの計算システムの内部状態を示すものであり、G,Env,Dbは、それぞれ、ゴール列、変数環境、プログラムの表現である．三つ目のグループは、この述語の計算が終了後、オブジェクト・レベルの計算システムが持つべき新しい内部状態に対応するもので、NG,NEnv,NDb とR の四つからなる．ここで、R はオブジェクト・レベルのゴール実行の結果(成功、失敗など) の表現である．

このリフレクティブ述語がユーザ・プログラムで実行されると、この述語は自動的にメタ・レベルに上がって実行される．この段階でX,Y,G,Env,Dbにはオブジェクト・レベルの表現が、それぞれ代入される．この述語の実行が終わると制御は再び、オブジェクト・レベルに戻るが、そのさいNG,NEnv,NDb,R の値から、オブジェクト・レベルのゴール、変数環境、プログラム，ゴール実行の結果がそれぞれ作られる．

例えば、ある変数X が未束縛であるかどうかを調べるvar(X)という述語であれば、

```
reflect(var(X,(G,Env,Db),(NG,NEnv,NDb,R)))
      :- unbound(X,Env) |
         (NG,NEnv,NDb,R)=(G,Env,Db,success).

reflect(var(X,(G,Env,Db),(NG,NEnv,NDb,R)))
      :- bound(X,Env) |
         (NG,NEnv,NDb,R)=(G,Env,Db,failure).
```

という形式で定義できる．すなわちオブジェクト・レベルの変数X は、リフレクティブ述語が実行されるとメタ・レベルに上がるので、ここでは単に、メタ・レベルの表現X が変数環境の表現Env の中で束縛されているかを調べればよい．

オブジェクト・レベルで実行中のゴールの数(すなわちメタ・レベルにおけるゴール列の表現の長さ) を求めるcurrent-load(N) という述語であれば、

```
reflect(current-load(N,(G,Env,Db),
  (NG,NEnv,NDb,R)))
     :- true |
     length(G,X),
     NEnv= [(N,X) | Env ],
     (NG,NDb,R)=(G,Db,success).
```

と定義できる．

また、オブジェクト・レベルからある定義節をプログラムに加えるadd-clause(CL)という述語は、

```
reflect(add-clause(CL,(G,Env,Db),
  (NG,NEnv,NDb,R)))
  :- true |
     dereference(CL,Env,NCL),
     add-db(NCL,Db,NDb),
     (NG,NEnv,R)=(G,Env,success).
```

と記述できる．すなわち、ここではメタ・レベルの表現CLを変数環境の表現Env でdereference したNCL をまず作り、それをプログラムの表現の中に加えている．

5.2 upとdown

変数や変数の表現を含まない項ではオブジェクト・レベルとメタ・レベルの表現は同じであるので、オブジェクト・レベルとメタ・レベルが情報のやりとりをするときに困る事はない．

オブジェクト・レベルとメタ・レベルが、常にこうした項のみで情報のやりとりをするならば、それはそれなりに充分である．しかしながら、ときどき、われわれはオブジェクト・レベルとメタ・レベルの間で変数や変数の表現を含む項のやりとりをしたい．このために使われるのがup(↑) とdown(↓) である．upは与えられた情報をもう一段上のメタ・レベルの記法に変換する．downは、逆に、与えられた情報を一段下のオブジェクト・レベルの記法に変換する．(したがって常に↑↓X や↓↑X は、存在すれ

ばX と同じである．またこの記法は本質的には3-Lispと同じであるが、常にX の評価を行わないところが相違点である．）

例えば、オブジェクト・レベルで実行中のゴール列について、あえてそのメタ・レベルの表現を取り出すget-q(Q)という述語を考えると、get-q は、

```
reflect(get-q(Q,(G,Env,Db),(NG,NEnv,NDb,R)))
  :- true |
     NEnv=[(Q,↑G)|Env],
     (NG,NDb,R)=(G,Db,success).
```

と定義できる．ここでget-q で取り出されるQ はオブジェクト・レベルで実行中のゴール列のメタ・レベルでの表現であり、get-q のリフレクティブな定義の中では↑G はメタ・メタ・レベルの表現になっていることに留意したい．逆に、実行中のゴールを、Q で表現されるゴール列に置き換えるput-q(Q)という述語であれば、

```
reflect(put-q(Q,(G,Env,Db),(NG,NEnv,NDb,R)))
  :- true |
     NG=↓Q,
     (NEnv,NDb,R)=(Env,Db,success).
```

と定義できる．

5.3 リフレクティブ・システムの実現

このようなリフレクティブ述語とup、downを実現するための構成として、図6 のような構造を考える．すなわち、最初にオブジェクト・システムをサポート・システムの下で走らせる．（このサポート・システムは、実際には、オブジェクト・システムの特殊なメタ・システムとして実現する）オブジェクト・システムの中でリフレクティブな述語が呼ばれると、メタ・システムを作る必要がある．そこでサポート・システムは、オブジェクト・システムをデータ化し、新しく作ったメタ・システムのデータとして格納する（この操作をわれわれはprojectionと呼んでいる）．

また、メタ・システムでのゴールの実行が終了したとき、メタ・システムを消去し、オブジェクト・システムの実行に戻ることが可能である（この操作をわれわれはamendment と呼んでいる）．

こうしたprojectionやamendment を行うことにより、われわれの実現したリフレクティブ・システムでは、自由にメタの階層を積み重ねたり、取り崩したりできるようになっている．

6. 関連研究の動向

関連研究の動向であるが、リフレクションの一般的なサーベイに関しては、著者らによる文献〔Sugano 89-1〕を参照されたい．現在ここで使っているような意味でのリフレクションは、Smith の3-Lispで使われた意味に近く、既に述べたように、本研究の動機も、3-Lispのようなことをを GHC でも行いたいというところから出発している．

並列論理型言語のリフレクションに関しては、筆者らによるRGHCに関連した研究〔Tanaka 88-2〕が古いと思われる．しかしながら、このRGHCにおいては、組み込み述語により、感知したい情報をオブジェクト・レベルに持ち込むことによりリフレクションを実現しており、やや応用指向の研究に止まり、リフレクションの本格的な枠組みを言語仕様、インプリメント仕様として提案するには

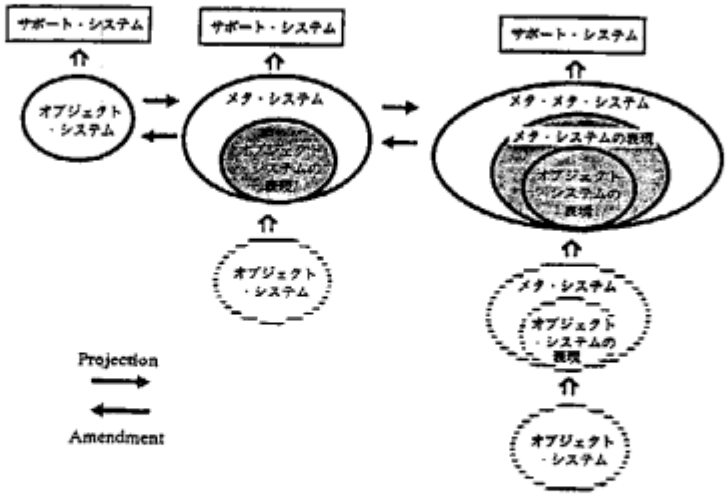

図6 リフレクティブ・システムの実現

到らなかった．

また、論理型言語に関して、リフレクションを用い、その意味論を論じた研究として〔Sugano 89-2 〕がある．そのほか、リフレクションではないが、Lloyd は論理型言語のメタ・システムを提案しており〔Lloyd 88〕、そのアプローチは菅野の形式化と近い．なお、これらの研究の動機は、主として論理型言語における非論理的述語（すなわち、assert、retract 、入出力、var 述語、ゴール実行の成功や失敗、否定の実現など）を論理的に扱いたいというところにあり、筆者らの動機とは本質的には相違はないものの、多少ニュアンスを異にしている．

7. まとめ

以上、GHC におけるアドバンスト・リフレクションの試みについて、その概略を述べた．本研究を6 章に述べた関連研究と比較するとき、本研究の特徴は以下のようにまとめられよう．

(1) 並列論理型言語によるシンプルな定式化．
Lispなどと比較して、論理型言語は「評価する」という概念を持たないことを前述したが、それを反映して本論文の定式化は、構文要素としてquote を含まない定式化となっている．すなわち、従来quote で区別されていた、オブジェクト・レベルの実体とメタ・レベルの表現との違いは、本システムで提供する変数の基底項による表現や変数束縛によって、より簡潔に表現されている．上述のLloyd、 菅野の定式化はすべてquote を含む定式化であり、結果として、本定式化は、より論理型言語の特徴を生かした定式化となっている．

(2) リフレクティブ・タワーの実現．
本システムでは、リフレクティブ・タワーが構築可能であり、筆者らの以前のRGHC〔Tanaka 88-2 〕と比較しても、メタ・レベルとオブジェクト・レベルを完全に切り分けた、より本格的な枠組みとなっている．

なお、本システムのインプリメントについては、上田らによるPSI-II上のGHC システムに多少の機能追加を行い、その上にメタインタプリタの形式でリフレクティブなGHC システムをのせる方向で試作を行った．試作版は、本稿で述べた言語仕様よりすこし古い言語仕様で開発を行ったが、変数の基底表現やリフレクティブ・タワーなどの作動は確認ずみである．本稿の言語仕様への修正は比較的容易であると考えられる．

また、紙面の関係で述べなかったが、本研究では応用として、リフレクティブ・システムの枠組みを用いたシステム・プログラミングなどへの適用を考えている．こうした分野への応用は、今後並列、分散ハードウェアの急速な普及につれて急速に重要となってくることが予想される．

本稿の冒頭で、『メタ』や『リフレクション』を、『簡潔にして強力な記述力を求めるための機構』として位置づけたが、本稿で提案したアドバンスト・リフレクションにより、GHC は『究極のプログラミング言語』にまた一歩近づいたと見ることもできよう．

なお、本研究はまだ進行中のものであり、今後の課題としては言語仕様の更なる洗練化と共に、実際の分散ハードウェア上でのシステム・プログラミングへの適用、インプリメントの高速化などが考えられよう．

［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われたものである．本研究に関連して日頃ディスカッションの相手になってくれる国際研の同僚である菅野博靖、神田陽治、村上昌己の諸氏に感謝する．また本研究の一部および本システムのインプリメントは、富士通SSL の的野文夫、太田祐紀子の両氏に負っている．

［参考文献］

〔Bowen 83〕
D.L. Bowen et al.: DECsystem-10 Prolog User's Manual, University of Edingburgh, 1983.

〔Furukawa 87 〕
古川，溝口共編：並列論理型言語GHC とその応用，共立出版，1987.

〔Lloyd 88〕
J. W. Lloyd: Directions for Meta-Programming, Proc. of International Conference on FGCS 1988, Vol.2, ICOT, pp.609-617, 1988.

〔Maes 86 〕
P. Maes: Reflection in an Object-Oriented Language, Preprints of the Workshop on Metalevel Architectures and Reflection, Alghero-Sardinia, October 1986.

〔Smith 84〕
B.C. Smith: Reflection and Semantics in Lisp, 11th. POPL, Salt Lake City, Utah, pp.23-35, 1984.

〔Sugano 89-1 〕
菅野、田中：メタ推論とリフレクション、情報処理、Vol.30、No.6、pp.694-705、1989.

〔Sugano 89-2 〕
H. Sugano: A Formalization of Reflection in Logic Programming 、富士通国際研リサーチレポート、No.98 、1989.

〔Tanaka 88-1 〕
J. Tanaka: A Simple Programming System Written in GHC and Its Reflective Operations, The Logic Programming Conference '88, ICOT, pp.143-149, 1988.

〔Tanaka 88-2 〕
J. Tanaka: Meta-interpreters and Reflective Operations in GHC, Proc. of International Conference on FGCS 1988, Vol.2, ICOT, pp.774-783, 1988.

〔Tanaka 88-3 〕
田中, 的野: 変数管理をするGHC の自己記述, 電子情報通信学会コンピュテーション研究会, pp.41-50, 1988年5 月26日.

〔Tanaka 90 〕
J. Tanaka, Y. Ohta, F. Matono: An Overview of ExReps System, Fujitsu Scientific and Technical Journal, Vol.25, No.1, 1990 (to appear).

〔Ueda 85 〕
K. Ueda: Guarded Horn Clauses, ICOT Technical Report TR-103, 1985.

〔Weyhrauch 80〕
R. Weyhrauch: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, Vol.13, pp.133-170, 1980.