TR-599

リフレクティブGHCとその実現

田中　二郎、的野　文夫（富士通）

October, 1990

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03)3456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

リフレクティブGHC とその実現

田中　二郎[+]　　的野　文夫[++]

[+] 国際情報社会科学研究所，富士通
[++]富士通ソーシアルサイエンスラボラトリ

連絡先：田中二郎　　富士通国際研，〒144 大田区
新蒲田 1-17-25　富士通シスラボ内
☎730-3111　Email: jiro@iias.fujitsu.co.jp

キーワード：GHC，リフレクション，メタ・システム，変数管理

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．最近，こうした『簡潔な枠組みで強力な記述力を得るための機構』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．

並列論理型言語 GHC〔Ueda 85-1, Furukawa 87〕について考えた場合，GHC は純粋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，言語仕様が極度に単純化されていて，貧弱な記述力しか持っていない．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．

これらについては，既に幾つかの提案〔Tanaka88-1, 88-2, 90〕がなされているが，それらはやや応用指向の提案であった．本節ではそれらとは異なった，体系的な考察に基づき本格的なリフレクション機構を持つリフレクティブGHC について報告する．

1.1 計算システムとメタシステム

最初に『計算システム』について考える．計算システム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『プログラム』と『データ』からなり，計算の状態が『継続(Continuation)』として表現される系と考えることができる．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，それを解くわけである．特に，論理型の計算システムでは，計算システムの要素として，『データベース』，『変数環境』，『実行ゴール』，などを考えることができるが，『データベース』は『プログラム』，『変数環境』は『データ』に対応する．また『実行ゴール』は『継続』に対応するもので，これには，最初，実行すべき『初期ゴール』が格納されており，計算の過程では『実行中のゴール列』が格納されている．

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．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（オブジェクト・システムの表現）が入る．

1.2 リフレクティブ・システム

リフレクションとは，一般には『自分自身』について感知し『自分自身』を変更することである．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，プログラム，データ，継続など）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〔Maes 86 〕．

リフレクティブ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が計算システムの中に提供されている必要がある．このような実現

方式にはメタ・システムの枠組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，オブジェクト・システムをメタ・システムの下で稼働させ，二つのシステムの連絡手段として，幾つかの組込述語をオブジェクト・システムの中に用意しておくことである〔Tanaka 88-2〕．オブジェクト・システムは，それらの述語によって感知したい実行情報をメタ・システムから得たり，その情報を修正してメタ・システムに戻すことができる．この方法は比較的インプリメントが容易であるが，メタ・レベルの情報がオブジェクト・レベルで処理されるので，情報のレベルを混同しやすいという短所を持つ．またメタ・レベルから情報を取り出してそれを処理している間にも，メタ・レベルの情報は刻々と変化しているので，この方式は正しいリフレクションのインプリメントではないことにも留意する必要がある．

もう一つの方法は，最初はオブジェクト・システムだけを稼働させ，リフレクションがおこったときだけ，メタ・システムを動的に作り，そこに制御が移るようにすることである．メタ・システムは，リフレクションに必要な計算だけを行い，計算が終了すればメタ・システムを壊してオブジェクト・システムに戻ればよい．さらに，メタ・システムとオブジェクト・システムとを同じ計算システムで実現すれば，メタ・システムを実行中にさらにリフレクションを起こし，メタのメタを呼ぶことも可能になり，原理的にはメタの無限タワー（リフレクティブ・タワー）の構築が可能になる．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．本論文ではこの方式に従ってリフレクティブGHC のインプリメントを行っている．

1.3 本論文の構成

以上，本研究の動機と，計算システム，メタ・システム，リフレクティブ・システムに関して，その概念を簡単に説明した．以下，本論文の構成は次の通りである．まず，2 章では，これらの基礎概念に基づいて，GHC におけるメタシステムの構成について述べる．3 章では GHCにおけるリフレクティブ・システムの構成について述べる．4 章ではリフレクティブGHC の実行例を示し、5 章では関連研究の動向について述べる．また，本研究の特徴及び今後の課題を6 章で述べる．

2. メタ・システムの構成

2.1 簡単なメタプログラム

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）もしくはパニラ・インタプリタとして，たった4行のプログラムが知られてきた〔Bowen 83〕．この4 行プログラムをGHC で記述したGHC in GHCは以下のようになる．

```
exec(true) :- true | true.
```

```
exec((P,Q)):- true | exec(P), exec(Q).
exec(P) :- user-defined(P)   | reduce(P,Q), exec(Q).
exec(P) :- sys(P) | P.
```

実行すべき初期ゴールP をexec(P) という形で実行すると，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，分解し，一つ一つをそれぞれ実行する．
(3) 実行すべきゴールがユーザ定義述語であるときには，述語reduceが，与えられたゴールにヘッドがユニファイ可能でかつガードの条件を満足する定義節をみつけ，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) 実行すべきゴールが組込述語であるときには，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GHC in GHCになっていることがわかる．しかしながら，この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オブジェクト・レベルの変数を，メタ・レベルの変数で表現している．従って，メタ・レベルで変数の表現についての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』などをメタ・レベルで行うことができない．

(2) 述語reduceの仕様にもよるが，ここではオブジェクト・レベルのデータベースとメタ・レベルのデータベースの区別を行っていない．例えばオブジェクト・レベルの定義節を，特殊な形式でデータベースに登録し，述語reduceがこの形式の述語定義だけを探索するような仕様にして，メタ・レベルとオブジェクト・レベルの定義節を区別することは可能であるが，この場合でもオブジェクト・レベルの定義節はデータベースとして登録されており，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．したがって，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メタ・システムを考えたい．

2.2 メタ・レベルにおけるオブジェクト・レベルの表現

オブジェクト・レベルのメタ・レベルでの表現であるが，オブジェクト・レベルの変数やデータベースがメタ・レベル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．われわれの構築するメタ・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．

① 定数，関数記号，述語記号

オブジェクト・レベルの定数，関数記号，述語記号は，メタ・レベルの中でも同じ記号に対応させる．（これらについて，同じ記号を対応させるのではなく，3-Lispのように'(quote) を使用し，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，論理型言語では『3 』と『'3』を区別しても仕方ないので，ここではそのような方法を採用しない．この方式を採った理由に関しては，6 章で詳述する)．

② 変数，変数束縛

2.1 で述べたように，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・レベルの中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) 』に対応させる．すなわち，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『@数字』の形式で表現する（ここで数字はオブジェクト・レベルの変数に対して一意的に割り当てられる整数である)．

また，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表現する．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．

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二つの変数がユニファイされたときに生ずるものであり，これらに関しては，通常のPrologのインプリメントにおける参照ポインタと同様の扱いをする必要がある．(このため，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，参照ポインタをたぐって値を求めるdereference とよばれる操作が必要となる)

③ 項，定義節

こうした定数，関数記号，述語記号，変数などの記法に合致する記法として，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表現を対応させる．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』で表現される．

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))
```

という基底項におきかわる．

またオブジェクト・レベルのプログラム，すなわち定義節の集合も，メタ・レベルでは，定義節の表現のリストとして，基底項で示す．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( [var(1) | var(2)] ,var(3),var(4)):-true |
     var(4)= [var(1) | var(5)] ,append(var(2),var(3),var(5))),
 (append( [] ,var(1),var(2)):-true | (var(1)=var(2)))]
```

といった定義節の表現のリストで示す．ここでは，定義節の仮引数として表れる変数は，var(数字) の形の基底項で表現されており、実行時に、定義節が捜されて，計算が行われる際には @数字の形式に置き代わる．

2.3 メタ・システムの実現

メタ・システム m-ghc(Goal,Db,Out) のトップレベルの記述は以下の通りである．

```
m-ghc(Goal,Db,Out) :- true |
   transfer(Goal,GoalRep,1,Id,Env),
   exec( [GoalRep ] ,Env,Id,Db,NewEnv,Res),
   make-result(Res,GoalRep,NewEnv,Out).
```

m-ghc は，ゴールとデータベースを入力すると，述語trasnfer，述語exec及び述語make-result を起動する．まず述語transferは，入力されたゴールGoalをメタ・レベルでのオブジェクトの表現であるGoalRep に変換する．GoalRep は，述語execの中で，データベースDbを使って実行され，実行結果は計算結果を表すRes と変数環境NewEnvに反映される．

述語transferは５個の引数を持ち，第１引数のGoalをメタ・レベルのでのオブジェクトの表現に変換して，第２引数GoalRep に出力する．2.2 で述べたようにGoalRep ではGoalの変数がすべて“@数字”を割り当てた特殊な形に変形されている．述語transferの第３引数は“@ 数字”の形で割り当てる変数番号の初期値を示しており，第４引数のIdには，次の変数番号が返される．(すなわち，transferでGoalを変換するのに1 からn までの変数番号が使われたとすると，Idにはn+1 が返される．) また第５引数のEnv にはここで作られた変数“@数字”とその値の対応のリストが変数環境としてしまわれる．

例えば，いまGoalとして，“exam([H | T] ,T)”が入力されたと仮定すると，
transfer(exam([H | T] ,T),NGoal,1,Id,Env) が実行され，

```
NGoal =  exam( [@1 | @2] ,@2)
Env   =  [(@1,undf),(@2,undf) ]
```

になる．
またexecは以下のように記述できる．

```
exec( [] ,Env,Id,Db,NewEnv,Res)
    :-true |
    (NewEnv,Res)=(Env,success).
exec( [true | Rest] ,Env,Id,Db,NewEnv,Res)
    :-true |
    exec(Rest,Env,Id,Db,NewEnv,Res).
exec( [false | Rest] ,Env,Id,Db,NewEnv,Res)
    :-true |
    (NewEnv,Res)=(Env,failure).
exec( [P | Rest] ,Env,Id,Db,NewEnv,Res)
    :-user-defined(P)  |
    reduce(P,Rest,Env,Id,Db,NRest,Env1,Id1),
    exec(NRest,Env1,Id1,Db,NewEnv,Res).
exec( [P | Rest] ,Env,Id,Db,NewEnv,Res)
    :-sys(P) |
    sys-exe(P,Rest,Env,NRest,Env1),
    exec(NRest,Env1,Id,Db,NewEnv,Res).
```

2.1 で簡単なGHC のメタプログラムについて述べたが，このプログラムは2.1 の4 行プログラムをエンハンス（富裕化）したものである．このexecは6引数からなり，第一引数は実行ゴール列，第二引数は初期の変数環境，第三引数は次に割り当てられる変数番号の初期値であり，第四引数はデータベース，第五引数は実行が終わったときの変数環境，第六引数はexecの実行結果を表す．また述語reduceは，ゴールP にユニファイ可能な候補節のリストを獲得し，候補節の中からコミット出来る節をひとつ選んで，その節のボディ部のゴールを実行ゴール列Restに加えNRest を作る．もしもゴールP がサスペンドしたときは，P を実行ゴール列の最後に加える．また，コミット出来る節がないときにはゴールfalse を実行ゴール列に加える．ここではreduceの詳しい記述は省略するが，くわしくは〔Tanaka 88-2 〕を参照されたい．
またmake-result は，以下のように記述できる．

```
make-result(success,GoalRep,Env,OutGoal)
```

```
        :-true |
        deref-variable(GoalRep,Env,Goal),
        OutGoal=success(Goal).
    make-result(failure,GoalRep,Env,OutGoal)
        :-true |
        deref-variable(GoalRep,Env,Goal),
        OutGoal=failure(Goal).
```

すなわち，execからRes としてsuccess やfailure が上がってきたとき，述語deref-variableはGoalRep の中の変数を，変数環境Env でdereference してOutGoal に出力する．述語deref-variableはちょうどtransfer述語と逆の機能を持っている．

3　リフレクティブGHC の構成

3.1　リフレクィブ・システムにおけるメタの階層

すでに1.2 で述べたように，リフレクティブGHC では，最初にオブジェクト・システムだけを稼働させ，リフレクションがおこったときだけ，メタ・システムを動的に作り，そこに制御が移るようにしてリフレクションを実現する．メタ・システムは，リフレクションに必要な計算だけを行い，計算が終了すればメタ・システムを壊してオブジェクト・システムに戻る．また，リフレクティブGHC では，メタ・システムとオブジェクト・システムを同一の計算システムで実現しているので，メタ・システムを実行中にさらにリフレクションを起こし，メタのメタを呼ぶことも可能であり，原理的にはメタの無限タワー（リフレクティブ・タワー）の構築が可能である．

リフレクティブ・タワーにおいては，オブジェクト・レベルの上にメタ・レベルがあり，メタ・レベルの上にもメタ・メタ・レベルがあり，その上にもメタ・メタ・メタ・レベルがあって…という具合にメタの階層ができている．したがって変数の「特殊な基底項」としての表現についても，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になる．そこで，リフレクティブGHC では，メタ・レベルの中では，オブジェクト・レベルの変数を「@数字」の形式で表現する他に，メタ・メタ・レベルの中では，「@！数字」と表現し，レベルが上がるごとに「@!!数字」，「@!!! 数字」，「@!!!!数字」…などと表現する．

3.2　リフレクティブ述語

リフレクティブGHC はリフレクティブ述語を用いて，オブジェクト環境の内部状態を見たり，変更したりする機能を持つ．

リフレクティブ述語は，メタ・レベルで実行される述語であり，その引数に，一時凍結されたオブジェクト環境を入力にもち，メタ・レベルで何らかの処理の後，再開始時点のオブジェクト環境を出力する述語である．それゆえ，リフレクティブ述語はユーザが自由に定義するものであるが，以下の形式で定義されなければならない．

```
reflect( < ゴール> ,(G,Env,Db),(NG,NEnv,NDb)) :-
                <ガード・ゴール列> | <ボディ・ゴール列>.
```

リフレクティブ述語の定義は，三引数の述語reflect を用いて定義される．reflect は第一引数にリフレクションを起こしたオブジェクト・レベルのゴールを持つ．第二引数には，凍結されたオブジェクト・レベルの環境，すなわち，実行中のゴール列 G，変数環境 Env，データベースDbの三つ組を持つ．第三引数には，再開始するオブジェクトの環境を格納する変数の三つ組NG,NEnv,NDb を取る．

また，< ガード・ゴール列> にはこの定義節がコミットされる条件，< ボディ・ゴール列> にはメタ・レベルで行うべき処理を記述する．このガードやボディでは，凍結されたオブジェクト・レベルの状態を参照したり，変更したりすることができる．またこの述語定義では，ボディ部で必ず再開始時点のオブジェクト環境を出力する手続きを行わなければならない．（これは3-Lispのreflective procedureの定義にヒントを得ている）．

3.3　リフレクティブ述語の実行

リフレクティブ述語は，予めユーザがリフレクションを起こすゴールとして定義したゴールが呼ばれたとき起動する．リフレクションを起こすゴールとは，3.2章で記したreflect 述語の第一引数に書かれるゴールである．例えば，リフレクティブ述語が次のように定義されているとき，

```
reflect(p(A),(G,Env,Db),(NG,NEnv,NDb)):- true |
     q(...),...
```

p がリフレクションを起こすゴールである．

リフレクションを起こすゴールが呼ばれると，メタ・システムを動的に作り，リフレクションを起こしたゴールを変形し，オブジェクト・レベルの環境を表現する変数列を加えたリフレクティブ・ゴールを作り，それをメタ・システムで実行する．この様子を図1に示す．リフレクティブ・ゴールは，メタ・システムでなんらかの処理をした後，再開始するオブジェクト・レベルの環境に対応する変数NG,NEnv,NDb を設定する．メタ・システムの実行が終わるとNG,NEnv,NDb からオブジェクト・レベルの環境が作られて、再びオブジェクト・レベルのゴール実行が再開される．

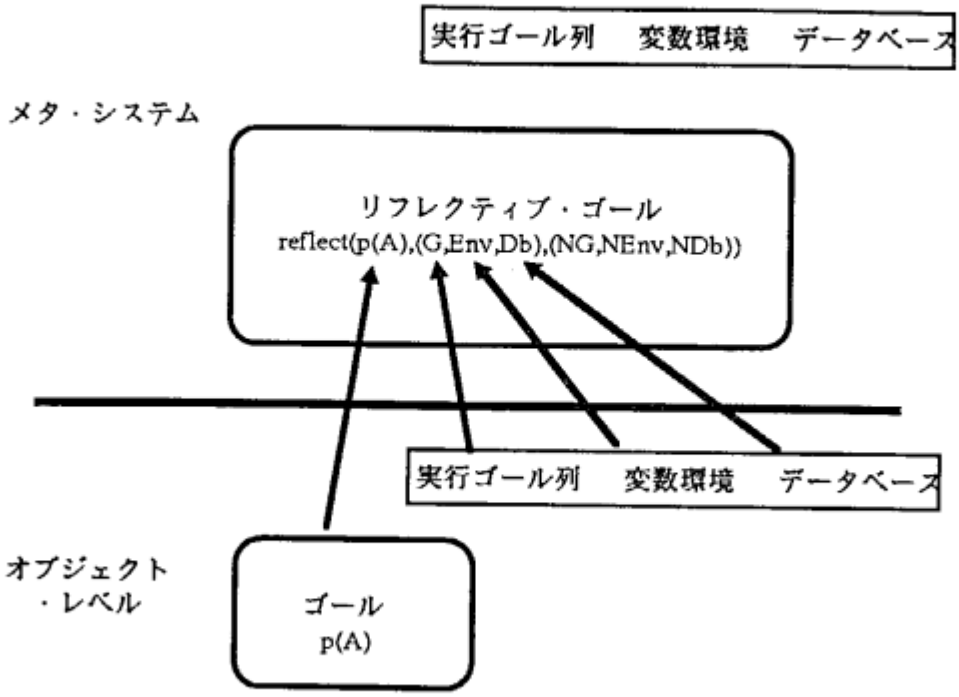

図1　リフレクティブ・ゴールの組み立て

リフレクティブ述語の定義の例を以下に示す．まず，ある変数が未束縛であるかどうかを調べるvar(X,Res)という述語であれば，

```
reflect(var(X,Res),(G,Env,Db),(NG,NEnv,NDb))
      :- unbound(X,Env) |
         NEnv= [(Res,unbound) | Env ],
         (NG,NDb)=(G,Db).
reflect(var(X,Res),(G,Env,Db),(NG,NEnv,NDb))
      :- bound(X,Env) |
         NEnv= [(Res,bound) | Env ],
         (NG,NDb)=(G,Db).
```

という形式で定義できる．すなわちオブジェクト・レベルの変数は，メタ・レベルでは，変数の表現（名前）として扱われ，その値は変数環境の中にしまわれているので，ここでは単に，変数環境Env の中を見て，その変数が束縛されているかどうかを調べればよい．

また，オブジェクト・レベルで実行中のゴールの数を求めるcurrent-load(N) という述語であれば，

```
reflect(current-load(N),(G,Env,Db), (NG,NEnv,NDb))
```

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

と定義できる．

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

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．

3.4 メタ述語定義

リフレクティブ述語はメタ・レベルで実行される述語と考えられるので，リフレクティブ述語定義のボディ部でユーザ定義述語を用いるときには，それらはメタ・レベルのデータベースに登録されている必要がある．

そのために使われるのがメタ述語定義である．メタ述語定義は，リフレクティブ述語同様，ユーザが自由に行うことができる．

メタ述語の定義は，述語定義のヘッド部をmetaで括って行う．例えば，

```
    H :- G  | B.
```

をメタ・レベルのデータベースに登録したい場合，

```
    meta(H) :- G  | B.
```

と定義する．また，メタ・メタ・レベルのデータベースに登録したい場合，metaをさらに重ねて，

```
    meta(meta(H)) :- G  | B.
```

と定義する．

ここで定義したメタ述語は，最初はこのままの形でオブジェクト・レベルのデータベースに登録される．リフレクティブ述語が呼び出されてレベルが一段上がる毎にmetaが一皮

取り除かれ，メタ・レベルのデータベースに登録される．またリフレクティブ述語定義についてはそのままの形式でメタ・レベルのデータベースに登録される．すなわち，メタ述語定義をリフレクティブ述語定義と組み合わせることにより，リフレクティブGHC では多様な機能や柔軟なメタ概念の表現が可能である．

このメタ述語定義の使用例としては，2.3 で述べた『exec』定義を変形し，meta述語として定義しておくことが考えられる．そうすると，このexecを使い，ゴールをインタプリティブに実行することができる．これは述語interpretiveを以下のようにリフレクティブ述語として定義することにより可能となる．

```
reflect(interpretive(P),(G,Env,Db),(NG,NEnv,NDb))
        :-true |
        exec( [P ] ,Env,1,Db,NEnv, _),
        (NG,NDb)=(G,Db).
```

このインタプリティブ実行を使えば，ある特定のゴールP だけをインタプリティブ実行して，残りのゴールを直接実行することも可能となる．またこの『exec』をエンハンスして『debbugger 』を作り，このシステムをデバッガとして使用するようなことも考えられる．

3.5　シフト・アップとシフト・ダウン

リフレクティブ述語は自動的に一段上のメタ・レベルで実行される．そのさいに，リフレクティブ述語が擁むオブジェクト環境の表現は，自動的に（メタ・レベルから見て）一段下の表現になる．また，またリフレクティブ述語の実行が終わると制御は再びオブジェクト・レベルの環境に戻り，そのとき一段下の表現になっていた情報は自動的に元の表現に戻る．このようにリフレクティブ述語の実行においては表現のレベルのアップとダウンが自動的におきるので，ユーザは表現のレベルのアップ，ダウンをプログラムの中で指定する必要はない．

しかしながら，ときどき，メタ・レベルの情報を強制的にオブジェクト・レベルに移動したりそれをまたメタ・レベルに戻したりしたい場合がある．これらのために必要なのがシフト・アップとシフト・ダウンである．シフト・アップ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．シフト・ダウンは，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．例えば，オブジェクト・レベルで実行中のゴール列を，その表現をオブジェクトの変数より，一段下のレベルの表現で取り出す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))
  :- true |
     shift-down(G,Down-G),
     NEnv= [ ( Q,Down-G) | Env ] ,
     (NG,NDb)=(G,Db).
```

と定義できる．ここでget-q で取り出されるQ に束縛されているのはオブジェクト・レベルの変数であり，また，G に束縛されている実行中のゴール列に含まれる変数もオブジェクト・レベルの変数である．ここで，『変数にオブジェクトの実行中のゴール列を束縛する』ことをしたい場合，束縛される変数とオブジェクトのゴール列ではレベルの区別をつける必要がある．そこで，G に束縛されている実行中のゴール列のレベルを一段下げて，Q に束縛されているオブジェクトの変数の値にする．

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))
  :- true |
     shift-up(Q,NG),
     (NEnv,NDb)=(Env,Db).
```

と定義できる．

3.6 リフレクティブGHC の実現

次にリフレクティブGHC の実現について述べる．3.1 ～3.5 に示したようなリフレクティブGHC の実現方法としては，いろいろな可能性が考えられるが，最も効果的な実現方法は，Warrenコードに対応する抽象機械語から設計をしなおし，それを直接にインプリメントすることである．この場合，リフレクションの実現のために，自己の環境をデータとして扱ったり，また逆にデータを環境に変換するような機構を抽象機械語レベルで持つことが必要になる．

また，リフレクティブGHC のインプリメントのもうひとつの可能性として，既存のGHC の上でインタプリタとして実現することが考えられる．この場合あまり実行速度などでは期待できないものの，比較的簡単にインプリメントできるという利点がある．われわれはこの方法を用いてリフレクティブGHC のインプリメントを行った．この場合，リフレクティブ・システム r-ghc(Goal,Db,Out) のトップレベルの記述は以下の通りである．

```
r-ghc(Goal,Db,Out) :- true |
   transfer(Goal,GoalRep,1,Id,Env),
   exec( [GoalRep ] ,Env,Id,Db,NewEnv,_,Res),
   make-result(Res,GoalRep,NewEnv,Out).
```

これは，2.3 のメタ・システムの実現と全く同じであり，この記述の意味は自明であろう．しかしながらリフレクションを実現するため，execの定義に，以下の定義節を一つ追加してやる必要がある．

```
exec( [G | Rest] ,Env,Id,Db,NewEnv,Res):-
```

```
reflective(G,Db) |
create-meta-db(Db,Meta-Db),
shift-down((G,Rest,Env,Db),(Down-G,Down-Rest,Down-Env,Down-Db)),
exec( [reflect(Down-G,(Down-Rest,Down-Env,Down-Db),(@1,@2,@3)) ] ,
      [(@1,undf),(@2,undf),(@3,undf) ] ,4,Meta-Db,New-Meta-Env,_),
deref-variable((@1,@2,@3),New-Meta-Env,(D-Rest2,D-Env2,D-Db2)),
shift-up((D-Rest2,D-Env2,D-Db2),(NRest,NEnv,NDb)),
exec(NRest,NEnv,Id,NDb,NewEnv,Res).
```

この定義節はリフレクティブ・タワーの構築を担当する．まずcreate-meta-dbによってオブジェクト・レベルのデータベースからメタ・レベルのデータベースが作られる．ここでは，3.4で述べたように，リフレクティブ定義はコピーされ，メタ定義は，定義からmetaがはずされてメタ・レベルのデータベースに登録される．(G,Rest,Env,Db) がシフト・ダウンされて(Down-G,Down-Rest,Down-Env,Down-Db) が作られる．次にメタ・レベルの計算を担当するexecが起動されるが，初期ゴール列としてはリフレクティブ・ゴール，初期環境としてはリフレクティブ・ゴール中に現れる変数を入れた環境，データベースとしてはcreate-meta-dbによって作られたメタ・レベルのデータベースが格納されている．

メタ・レベルの計算が終了したときには，@1，@2，@3には，オブジェクト・レベルの環境の再開始状態が与えられているので，それらをメタ・レベルの変数環境でdereferenceして，さらにシフト・アップして(NRest,NEnv,NDb)を作り，再びオブジェクト・レベルの計算を担当するexecを起動する．

このプログラムからもわかるように，ここではオブジェクト・システムもメタ・システムも同じexec述語で記述されている．したがってメタ・システムで更にリフレクティブ述語が呼ばれればメタ・メタ・システムが起動される．このように，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．この様子を図に示したのが図2 である．

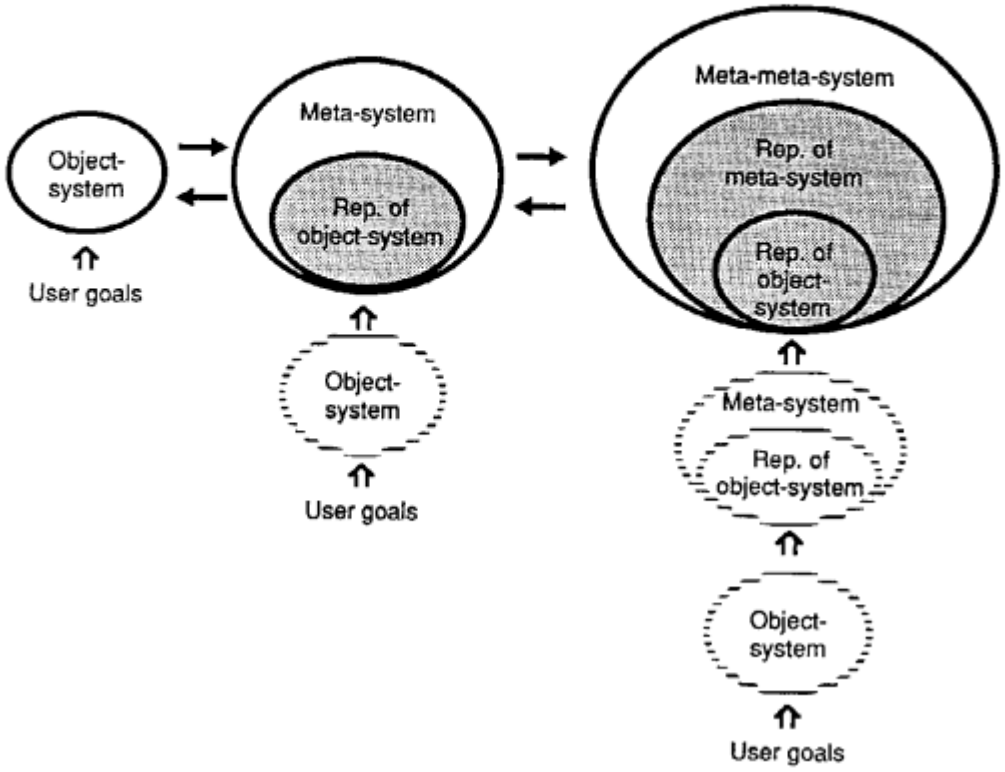

図2　リフレクティブ・システムの実現

3.6 実際のインプリメント

実際のリフレクティブGHC のインプリメントは、GHC とESP を用いてPSI-Ⅱ上で行った．処理系の主要部分はGHC で記述し，ソースコードで約960 行である．インプリメントは3.5 に示したものに多少のユーザ・インタフェイスの機能を付加している．また，他に一部の入出力部分や，低レベルの述語をスピードアップのためESP で記述した(約480 行)．GHCの処理系としては,GHCプログラムをESP にコンパイルするGHC コンパイラ〔Ueda 85-2 〕を用いている．なおESP はPrologのオブジェクト指向版であるが、オブジェクト指向言語の特色は特に積極的には使用していない．

4 リフレクティブGHC の実行例

リフレクティブGHC の実行例を図3及び図4に示す．図3の右側のウインドウにオブジェクト・レベルのデータベースに登録されるプログラムが示されている．ここでは述語test，述語append，およびリフレクティブ述語get-q が定義されている．図3の左側に重なった二つのウインドウが示されているが，奥の方にあるのがオブジェクト・レベルの入出力ウインドウである．ここでは，この入出力ウインドウから入力ゴールとしてtest(Q,A,B) を入力している．この入力ゴールtest(Q,A,B) が実行されると,test 述語の定義に基づき，二つのappendゴールと get-qゴールが起動される．そして get-qゴールが実行され

るとリフレクションが起きてメタ・システムが動的に作られる．ここでは，図3の左側の重なったウインドウのうち，前方に表示されているのがメタ・システムの出力ウインドウである．

出力ウインドウのreflect(get-q(...)...)は，リフレクト・ゴールを表示し，図3 はこのメタ・システムの実行が3 ステップで成功して終了したことを示している．なお表示で@(1),@(2),@(3)などはメタ・システムのレベルの変数，'@!'(5),'@!'(7) などはオブジェクト・レベルの変数を示している．3.1 に示した変数の表示とは()や' などを含んでいる点で若干異なっているが，これはインプリメントしたシステムの制約によるもので本質的ではない．

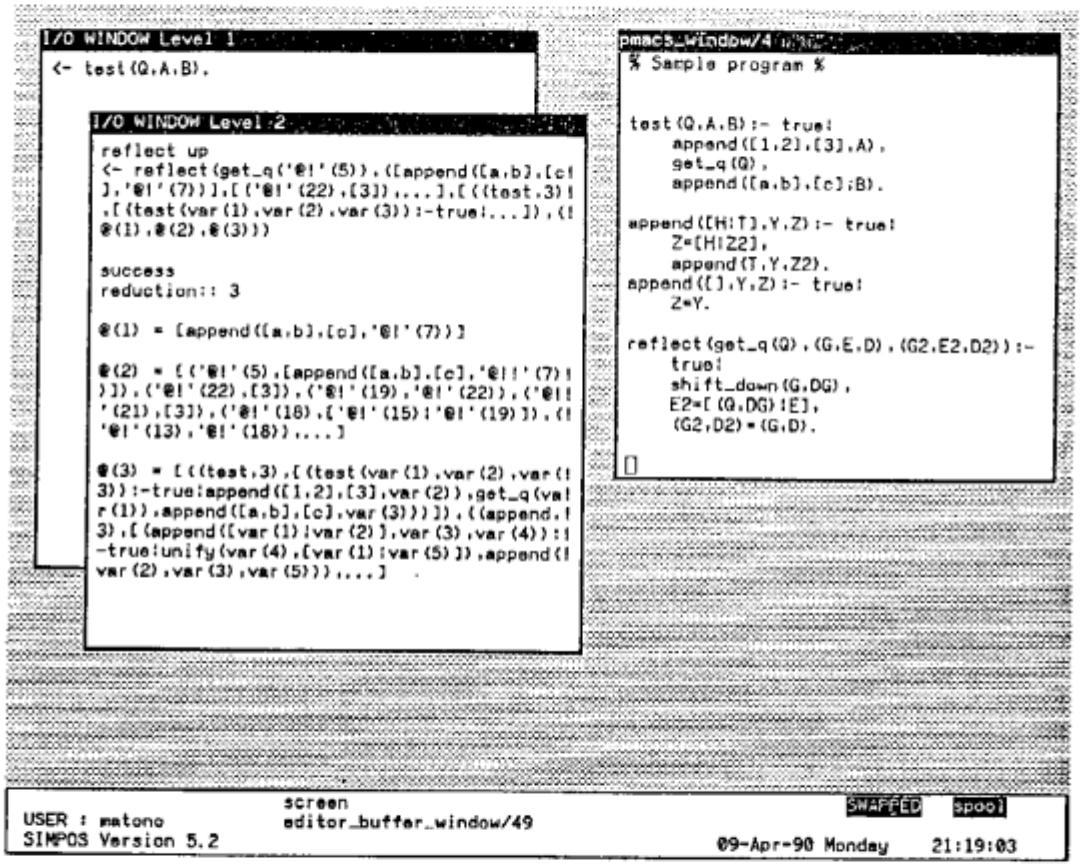

図3　リフレクティブGHC の実行例(1)

図3 でメタ・システムの実行が終わるとメタ・システムを壊してオブジェト・レベルの実行に戻る．図4 には，オブジェト・レベルの実行に戻ったあと，すべての計算が終了した状態が示されている．Q,A,B に対して変数@1,@2,@3が割り当てられており，計算後の変数束縛の状態がそれぞれ表示されている．

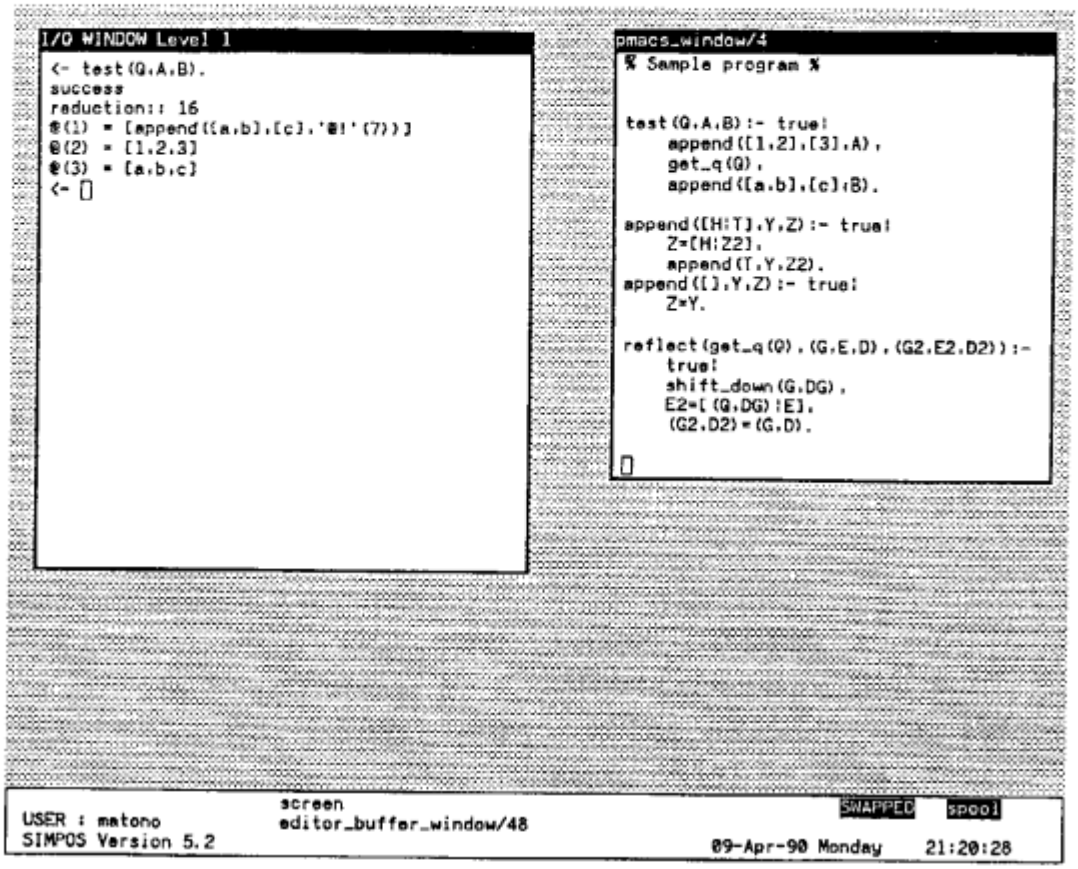

図4　リフレクティブGHC の実行例(2)

5 関連研究の動向

リフレクションは，とくに最近脚光を浴びるようになってきた研究領域〔Maes 88, Tanaka 88-3〕である．メタ推論とリフレクションの研究動向については解説〔Sugano 89 〕があるので，ここでは詳しい記述は省略する．本章では，リフレクティブGHC と他のアプローチとの関連について述べる．

まず，メタシステムとの比較であるが，メタシステムの主なものとしてはFOL 〔Weyhrauch 80〕,BowenとKowalskiのシステム〔Bowen 82〕，Goedel〔Lloyd 88-1〕などがある．FOL は，一階述語論理を対象とした推論システムであるが，Lispをベースに記述されており，インタラクティブな操作により自然演繹法のスタイルで推論を実行する．FOL と我々のシステムには直接の関連は無いが，メタ的な取り扱い，リフレクション原理などに関する研究の出発点を提供した．

また，Bowen とKowalskiのシステムは，本稿の2 章で述べたメタ・システムと本質的に同じである．また彼らの提唱するメタとオブジェクトの融合（アマルガメーション）は，

本稿の3.5 節の記述と多少の関連がある．ただしBowen とKowalskiのシステムでは，メタ・システムの具体的なインプリメントについては何ら述べられていない．このBowen とKowalskiのシステムを具体的に発展させたものにEshghiのシステム〔Eshghi 86 〕があるが本研究は，Eshghiのシステムとは独立に行われたものである．

Goedelは，メタ・システムをメタインタプリタの拡張により定義し，Prolog の非論理的な述語を論理的に再構築しようとするもので，変数の基底表現（グランド・リプリゼンテーション）を提唱するなど，我々のアプローチともっとも関連が深い．しかしながら両者の研究は全く独立に行われたものであり，〔Lloyd 88-1, Tanaka 88-2 〕ともに発表時期も同時期である．Goedelにおいては，形式的な取り扱いやその意味論に重点が置かれいるのに比べ，我々のアプローチでは実際のインプリメントや応用に重点があるという点に違いがある．またLloyd たちはメタ・システムを信奉し，リフレクティブ・システムについては，その応用や意味が明確でないという立場を取っている．1988年に「論理プログラミングにおけるメタプログラミング」に関して，国際ワークショップ『Meta 88 』〔Lloyd 88-2〕が開催されているが，メタ・システムに関しては，論文集から判断する限り，本稿の2 章で述べた観点が重要であるという点では，一応のコンセンサスが得られていると思われる．

次に，リフレクティブ・システムとの比較であるが，FOL 〔Weyhrauch 80〕に関してはリフレクション原理やリフレクティブ・タワーについての概念的記述があるものの，そのインプリメントの詳細についてはほとんど記述がない．また3-Lisp〔Smith 84〕は，第1章で述べたように本研究の直接の発端となったシステムである．当然ながらこのシステムはLispのシステムであり，論理型言語とは全く関連がない．また，Prolog にリフレクションを導入しようとする試みとしてReflective Prolog 〔Costantini 89 〕と R-Prolog*〔Sugano 90 〕がある．Reflective Prolog はその名前にもかかわらず，主とするところはメタ・システムのデータベースを再定義してオブジェクト・システムの計算戦略を操作するもので，リフレクティブ・システムと呼ぶにはふさわしくない．(Costantini は Procedural reflection をインプリメントしたが余りにも低レベルなので言語レベルでは，それがそのまま表れないようにしたと書いている)． またR-Prolog* は、われわれが提案するリフレクティブ述語の形式をPrologに導入したものであるが，主たる関心はその意味論的な取り扱いにある．

6 本研究の特徴及び今後の課題

以上リフレクティブGHC に関してその概略を述べ，関連研究の動向についても簡単にまとめた．これらより，我々のアプローチの特色は以下のようにまとめられよう．

(1) シンプルなメタ・レベルの表現

本システムの定式化は，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，本システムでは，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を，変数の基底表現と変数束縛によって実現している．また変数の基底表現は，変数の属するレベルの情報を含んでおり，様々なレベルの変数が混ぜて使われても，混乱が起こることはない．すなわち本方式は，より論理型言語の特徴を生かした，また実際のインプリメント

も容易な簡潔な定式化となっている．

(2) リフレクティブ述語によるリフレクションの実現

リフレクティブGHC では，リフレクティブ述語を用いることでリフレクションが起動される．この方式は3-Lispの方式を論理型言語用に焼き直したものである．リフレクティブGHC では，リフレクションが起動された際，終了した際にシフト・アップ，シフト・ダウンが自動的におこり，ユーザは，通常，レベルの上げ下げを陽に指定する必要がない．

(3) リフレクティブ・タワーの実現

本システムでは，リフレクティブ述語による，任意のレベルのリフレクティブ・タワーの動的な構築が，非常に簡潔なコード(exec 述語）により記述されている．本方式ではオブジェクト・システムもメタ・システムも同じexecとして記述されるため，メタのレベルが上がっても実行速度が遅れることなく，同一の速度で処理が行われる．

(4) メタ述語定義とリフレクティブ述語定義によるデータベースの構築

本システムでは，メタ述語定義とリフレクティブ述語定義を併用することにより，メタ・レベルやメタ・メタ・レベルなどのデータベースをオブジェクト・レベルと完全に区別した形で構築することが可能である．

(5) 実際のインプリメンテーション

5 章でのべた，さまざまなメタ・システムやリフレクティブ・システムの大部分は，単に提案段階に留まっているが，本システムは，3.6 節，4 章で示したように，実際にインプリメントされ，稼働している．

(1)について，多少説明を追加する．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持たない．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おり，『' 』(quote) は、評価を行うかどうかの指示に必要であった．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．これらの理由により，本システムの定式化は，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を，変数の基底表現と変数束縛によって実現しており，構文要素としてquote を含まない定式化となっている．

なお，我々はリフレクションを導入する土台として並列論理型言語GHC を想定しているが，本稿で述べた範囲にかぎり，並列，分散環境上での応用を考えなければ，論理型言語Prologでも同様なことが可能であろう．(実際，R-Prolog* はそのようなものになっている．)

また今後の課題については以下のようなことが考えられる．

(1) リフレクションの応用

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．これらに関しては，我々はすでに〔Tanaka 88-1, 88-2, 90 〕などの研究を行ってきたが，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．こういった研究の方向は，OSを主として『組み込み述語』や『荘園』などによって実現するPIMOS のアプローチ〔Chikayama 88〕とくらべ，はるかに柔軟な枠組みを提供する．

(2) 意味論的考察

本論文では述べなかったがリフレクティブGHC の意味論的考察は重要である．菅野はR-Prolog* の意味を，エルブラン基底を拡張し，入出力付きのエルブラン基底を用いて定義しているが，これはGHC の意味論としても親和性が良いと思われる．とくに並列，分散環境を想定してのリフレクティブGHC の意味論的考察は非常に興味深い研究テーマである．

(3) インプリメントの高速化

リフレクティブGHC の最も効果的な実現方法は，Warrenコードに対応する抽象機械語から設計をしなおし，それを直接にインプリメントすることである．この場合，リフレクションの実現のために，自己の環境をデータとして扱ったり，また逆にデータを環境に変換するような機構を抽象機械語レベルで持つことが必要になる．実際の論理プログラミングでは，メタインタプリタや拡張ユニフィケーションなどの技法が多く使われるが，リフレクションを用いることにより，こういった問題に対してより効率的なインプリメントを提供できる可能性がある．

［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，本稿に関して，コメントを頂いた國藤進氏に感謝する．

［参考文献］

〔Bowen 82〕

K.A.Bowen and R.A.Kowalski: Amalgamating Language and Metalanguage in Logic Programs, Logic Programming, eds. Clark and Tarnlund, pp.153-172, Academic Press, 1982.

〔Bowen 83〕

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

〔Costantini 89 〕

S. Costantini and G.A. Lanzarone: A Metalogic Programming Language, Logic Programming, Proc. of the Sixth International Conference, pp.218-233, The MIT Press.

〔Chikayama 88〕

T. Chikayama, H. Sato and T. Miyazaki: Overview of the Parallel Inference Machine Operating System (PIMOS), Proc. of International Conference on FGCS 1988, ICOT, Vol.1, pp.230-251, 1988.

〔Eshghi 86 〕

K.Eshghi: Meta-Language in Logic Programming, Ph.D. Thesis, Department of Computing, Imperial College, July 1986.

〔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 88-1〕

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

〔Lloyd 88-2〕

J. W. Lloyd ed.: "Meta 88: Workshop on Meta-Programming in Logic Programming," 396p., University of Bristol, June 1988.

〔Maes 86 〕

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

〔Maes 88 〕

P. Maes and D. Nardi ed.: Meta-Level Architectures and Reflection, North-Holland 1988.

〔Smith 84〕

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

〔Sugano 89〕
菅野博靖，田中二郎：メタ推論とリフレクション，情報処理学会誌，Vol.30，6，pp.694-705，1989.
〔Sugano 90〕
H. Sugano: Meta and Reflective Computation in Logic Programs and Its Semantics, Proc. of Meta90: Workshop on Meta-Programming in Logic, Leuven, Belgium, April 1990.
〔Tanaka 88-1〕
J. Tanaka: A Simple Programming System Written in GHC and Its Reflective Operations, Proc. of 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, ICOT, Vol.2, pp.774-783, 1988.
〔Tanaka 88-3〕
田中二郎：メタプログラミングとリフレクション，bit，Vol.20，5，pp.41-50，1988.
〔Tanaka 90〕
J. Tanaka, Y. Ohta, F. Matono: An Overview of ExReps System, Fujitsu Scientific and Technical Journal, Vol.26, No.1, 1990 (to appear).
〔Ueda 85-1〕
K. Ueda: Guarded Horn Clauses, ICOT Technical Report TR-103, 1985.
〔Ueda 85-2〕
K. Ueda and T. Chikayama: Concurrent Prolog Compiler on Top of Prolog, Proc. of 1985 Symposium on Logic Programming, Boston, pp.119-126, 1985.
〔Weyhrauch 80〕
R. Weyhrauch: Prolegomena to a Theory of Mechanized Formal Reasoning, Artificial Intelligence, Vol.13, pp.133-170, 1980.