ICOT Technical Report: TR-374

TR-374

変数管理をするGHCの自己記述

田中二郎、的野文夫

May, 1988

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

変数管理をするＧＨＣの自己記述

Variable Management in an Enhanced GHC Metainterpreter

田中　二郎*　　的野文夫

新世代コンピュータ技術開発機構　　富士通ＳＳＬ

ICOT　　Fujitsu SSL

*現在，富士通，国際情報社会科学研究所

あらまし　GHC の簡単な自己記述から始め，それを拡張して，fail-safe なメタインタプリタ，メタレベルから制御可能なメタインタプリタ，スケジューリング・キューやリダクション・カウンタを持つメタインタプリタなどを段階的に導いた．また自分で変数管理を行うメタインタプリタについて，その概略を記述した．なお，本論文は並列論理型言語 GHC〔Ueda 85〕についての基本的な知識を前提としている．

Abstract　Starting from the simple self-description of GHC, we derive a fail-safe metainterpreter, a metainterpreter which can be controlled by the outside world, and a metainterpreter which has a scheduling queue and a reduction counter by stepwise enhancement. We also derive a metainterpreter which has variable management facility. This paper assumes the basic knowlege of GHC.

Keywords:　GHC, metainterpreter, self-description, variable management, reflection

1．はじめに

プログラムの自己記述（自分で自分を記述すること）は，起源的には，Lispなどで始まった技法と思われる．すなわち言語の特徴をメタインタプリタの形で表記する．Lispではデータ構造もプログラムもすべてがＳ式で表されるので，メタインタプリタの記述は比較的容易であった．

一方，論理型言語Prologの世界では，以下に示すような4 行メタインタプリタが，いわゆる "Prolog in Prolog" として知られてきた〔Bowen 83〕．

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

このメタインタプリタの意味は簡単である．すなわち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でも複数個でも無い時には，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見つけ，その定義節のボディにゴールを展開し，そのボディを解く．また，それ以外のときには（ゴールはシステム述語であるので）それを解く．解けないときには与えられたゴールが解けないことを意味するのでexecが失敗する．これらより，この4行のプログラムは簡単ではあるが，ちゃんと"Prolog in Prolog"になっていることがわかる．すなわち，実行したいゴールG を述語execのなかで実行することができる．

本稿では並列論理型言語 GHC〔Ueda 85, Furukawa 87〕の自己記述について考察するが， GHCについてもメタインタプリタはPrologの4行プログラムと同様に，以下のように記述できる．

```
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.
```

ひとめ見てわかるように，こ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．ただしGHC ではすべての定義節はガードを持っているので，Prologの！はGHC では｜に置き代わっている．またPrologでは定義節を見つけるのに述語clauseを使っていたが，GHC では述語reduceを使っている．述語reduceはclauseと若干仕様が異なり，与えられたゴールにユニファイ（というよりはマッチングというべきか）可能な定義節を見つけ，それらの中でガードがとける定義節を捜し，その定義節のボディにゴールを展開する．またGHC ではゴールが一度展開されると再びバックトラックされることがない．

さてこれらの4 行メタインタプリタであるが，これらはこのままでは最上位レベルのプログラム実行をシミュレートしているだけであり，あまりにも情報が乏しい．そこで，このメタインタプリタを修正あるいは拡張してもっと有用な情報が取り出せるようにしたいという欲求が起きてくる．

2．メタインタプリタの拡張

次の問題は，メタインタプリタをどのように修正あるいは拡張するかという問題である．これについては我々はKurusaweの論文〔Kurusawe 86 〕にヒントを得ている．

Kurusaweは，まずPrologプログラムを実行できる抽象Prologマシンを考えた．そしてそれらの抽象PrologマシンとPrologプログラムの境界を適当に変更することにより，Prologプログラムを変換（部分評価）してWarren風のコードを導いた．抽象Prologマシンにもいくつかのレベルが考えられ，もっとも簡単なPrologのインタプリタから始まり，ユニフィケーションをexplicitにしたもの，メモリの構造をexplicitにしたものとだんだんにマシン・イメージを注入され詳細化されていった．

我々のアプローチもKurusaweのアプローチと似ている．ただしKurusaweの興味のあったのはPrologプログラムのWarren風のコードへの変換であったが，我々は実行されるプログラムにはあまり興味がない．興味の

あるのは抽象マシンの「メタインタプリタ」による記述である．(Kurusawe は抽象マシンをメタインタプリタで記述することはしなかった．)

ところで，メタインタプリタの拡張については，既にPrologや並列論理型言語で様々な拡張が提案されている．以下それらについてまとめてみる．

① Bowen とKowalskiによる「デモ述語」の提案〔Bowen 82〕．このデモ述語は "demo(Prog,Goals)" の形で使われ,Prog からGoals が証明可能であることを示す．これは、プログラムの定義がexplicitにProgとして与えられていることを除けばexecと同じである．

② execを2 引数にして，exec(G,R) とする．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．これによってゴールG が失敗してもプログラム全体が失敗するのを防ぐことができる(fail-safe 機能)．これはイギリスのインペリアル・カレッジのKeith Clark たちのグループにより，並列論理型言語で提案されているものである．

③ execを3 引数にして，exec(G,I,O) とする．ここでI はこのexecへの外の世界からの入力，O は2 引数execのR を一般化したもので，execから外の世界への出力である．このexec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き有効である．すなわち，I から各種の命令を入力することにより，ゴールG の実行を，中断(suspend) したり，再開(resume)したり，また実行を放棄(abort) したりできる．これもイギリスのインペリアル・カレッジのKeith Clark たちのグループにより，並列論理型言語で提案されているものである．

④ 2 引数execをさらに修正して，実行結果だけでなく実行履歴をかえすようにする．これを利用してデバッガとして使えるようにする．これについては，既にICOTなどで，並列論理型言語やPrologで様々な提案がなされている．

以上，既に提案されているメタインタプリタの拡張について述べたが，メタインタプリタの拡張で何をexplicitにするかといえば，それはプログラム実行で我々が「知りたいもの」，「コントロールしたいもの」に他ならない．

3．ＧＨＣにおけるリフレクション機能

それでは我々がGHC プログラムの実行で「知りたいもの」，「コントロールしたいもの」とは何かというのが問題となる．そこで，まず我々の関心を明確にしておきたいが，我々の当面目的とするものは，プログラミング・システムのGHC による記述である．(プログラミング・システムとは，そこからユーザがプログラムや実行するゴールを入力することが出来る一種のオペレーティング・システムのミニ版のようなものである．) こうしたプログラミング・システムの記述にあたっては，システムの仕事とユーザの仕事を区別することが要求される．また同時に，システムがユーザの仕事を適当にコントロールしながら実行することが要求される．

GHC は並列論理型言語であり，言語のレベルでプロセスや同期が扱える．こうしたことからプログラミング・システムの記述も一見容易そうな気がする．しかしながら少しプログラミング・システムを真面目に書

こうとすると意外と大変である．原因としては，現在のGHC ではメタレベルとオブジェクトレベルの区別が曖昧で，かつメタレベルとオブジェクトレベルの間で情報をやりとりする機構が不十分である事が挙げられよう．

そこで我々の目的とするのは，我々の希望する機能を得るプログラミングの枠組みの開発，およびそのような機能を提供するようなGHC の言語仕様の手直しである．

様々な検討の末，我々が手本としたのは，結局3-Lisp〔Smith 84〕におけるリフレクションの考えかたである．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能力のことである．もしもそのような能力をプログラミング言語やシステムが持つならば，それらは強力なシステムを簡潔に記述することにつながる．たとえばオペレーティング・システムにおいては，メモリやCPU タイムなどのシステムのもつ資源を自己管理することが要求されるが，この時，システムが何らかの形で自己の状態や能力などを動的に感知し，それに応じて行動を取れるとしたら便利である．体力が余っていれば忙しく仕事をし，忙しくなったら余り新しい仕事は受け付けない．また自己の能力の限界を知っていて，できそうもないと思ったらあきらめてしまうなどと言うことも可能となる．

このような自己感知や自己変更を行うための枠組みとしてSmith はメタインタプリタを使っている．3-Lispのメタインタプリタは，プログラムの継続(Continuation)と変数の束縛環境(Environment) をexplicitに持ち運んでおり，ユーザはプログラムからこれらのメタな情報を自由に取り出し，変更を加えて戻せるようになっている．

我々のGHC におけるリフレクションの考え方もこれと同様である．ただしGHC は並列論理型言語であり，プログラム実行中にゴール実行の成功，失敗などのメタな情報が発生し，プログラムが並列に実行されるので若干リフレクション実現のための枠組みは複雑になる．さてそれではGHC のメタインタプリタで何をexplicitに扱うかという問題であるが，まず4.ではスケジューリング・キューとリダクション・カウントをexplicitに扱った例を紹介する．次に5.で変数環境をexplicitに扱った例を紹介する．

4．メタインタプリタの段階的拡張

本節では1.で述べたGHC の4 行メタインタプリタを段階的に拡張し，最終的にはスケジューリング・キューとリダクション・カウントをexplicitに扱うメタインタプリタを導出する．

まず1．で挙げた4 行メタインタプリタであるが，前にも述べたようにこのexecはプログラムの実行を最も表層でシミュレートしているだけであり，これではあまりにも役に立たない．ゴールG をexec(G) とexecの中で実行しても，トップレベルで実行したのと何の相違も生じないからである．

そこで，簡単な拡張として，execを2 引数としてexec(G,R) とすることを考える．（これは2.の②の場合に相当する．）この2 引数execは4 行メタインタプリタを修正することにより以下のように記述できる．

```
exec(true,Result) :- true | Result=success.
exec(false,Result) :- true | Result=failure.
exec((P,Q),Result) :- true | exec(P,R1),exec(Q,R2),and(R1,R2,Result).
exec(P,Result) :- not-sys(P) | reduce(P,Body),exec(Body,Result).
```

```
exec(P,Result) :- sys(P) | sys-exe(P,Result).
```

このexecでは，与えられたゴール実行の結果によってResultにsuccess やfailure が入る．また述語and はゴールの成功，失敗の結果を集計する述語である．述語reduceは，与えられたゴールにユニファイ可能な定義節が無かったり，それらの定義節のガードがすべて失敗したときにはBodyにfalse を返すと考える．

次に，2.の⑤で挙げた3 引数execであるが，これもこの2 引数execを多少修正し，以下のように記述できる．

```
exec(true,I,O) :- true | O= [success ].
exec((P,Q),I,O) :- true | exec(P,I,O1),exec(Q,I,O2),and-merge(O1,O2,O).
exec(P,I,O) :- not-sys(P),var(I) |
            reduce(P,Body,I,O1),exec(Body,I,O2),and-merge(O1,O2,O).
exec(P,I,O) :- sys(P),var(I) | sys-exe(P,I,O).
exec(P,[susp|I],O) :- true | wait(P,I,O).
exec(P,[abort | I],O) :- true | O= [aborted ].

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

ここではメタレベルとの連絡がストリームI とO で常に保たれていることに注意したい．すなわちゴール実行の成功，失敗という概念は絶対的な概念でなく，メタレベルへのメッセージとして実現されている．

次の拡張は，スケジューリング・キューの導入である．3-Lispにおいてはプログラムの継続(Continuation)をメタインタプリタがexplicitに持ち運んでいたが，GHC ではスケジューリング・キューがプログラムの継続の役目をすると考えたわけである．スケジューリング・キューの導入と同時にexecにゴール処理の逐次性が導入されていることに留意されたい．

スケジューリング・キューをexplicitに導入したexecは4 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．（この差分リストによるスケジューリング・キューの実現は，〔Shapiro 83〕のPrologによるConcurrent Prolog の実現で使われた方法をまねたものである．）4 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([P|H],T,I,O) :- not-sys(P),var(I) |
            reduce(P,T,NT,O,NO),exec(H,NT,I,NO).
exec([P|H],T,I,O) :- sys(P),var(I) | sys-exe(P,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〕,0) :- true ｜exec(H,T,I,0).
wait(H,T,〔abort ｜ I〕,0) :- true ｜0=〔aborted〕.
```

ここではexecはスケジューリング・キューを内蔵している．いままでの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する．述語reduceは引数のゴールを評価し，それが展開可能であれば展開したゴールをスケジューリング・キューの尾部に詰め，ゴールが評価できないときはもとのゴールをそのままスケジューリング・キューの尾部に詰める．

そしてこの4引数execにさらにリダクション・カウントを扱うための引数二つを加えるとexecは6 引数となり，exec(H,T,I,O,MaxRC,RC)となる．ここでMaxRC はこのexecに許された最大リダクション数，RCはexecの実行時のリダクションの数である．リダクション・カウントはCPU 資源の代わりをすると考えたわけである．

以下に6 引数execのプログラムを示す．

```
exec(T,T,I,O,MaxRC,RC) :- true ｜0=〔success(count = RC)〕.
exec(〔true｜H〕,T,I,O,MaxRC,RC) :- true｜exec(H,T,I,O,MaxRC,RC).
exec(〔P ｜H〕,T,I,O,MaxRC,RC) :- not-sys(P),var(I),MaxRC>RC ｜
            reduce(P,T,NT,O,NO,RC,RC1),exec(H,NT,I,NO,MaxRC,RC1).
exec(〔P ｜H〕,T,I,O,MaxRC,RC) :- sys(P),var(I),MaxRC>RC ｜
            sys-exe(P,T,NT,O,NO,RC,RC1),exec(H,NT,I,NO,MaxRC,RC1).
exec(〔P ｜H〕,T,I,O,MaxRC,RC) :- MaxRC=<RC｜0=〔count-over〕.
exec(H,T,〔Mes ｜I〕,O,MaxRC,RC) :- true ｜
            control-exec(H,T,〔Mes ｜I〕,O,MaxRC,RC).
```

述語reduceは4 引数execのreduce述語と同じであるが，ゴールを展開したときにはリダクション・カウントを一だけ進める．ゴールが展開できずもとのゴールをそのまま詰めた時にはリダクション・カウントを進めない．sys-exe 述語でも同様の処理を行う．

さて，こうして拡張されたメタインタプリタさえ記述してしまえば，メタインタプリタのなかに，スケジューリング・キューやリダクション・カウントが表現されているので，例えば，プログラムの実行中に現在のリダクション・カウントやスケジューリング・キューの中身を求めたり，そのリダクション・カウントやスケジューリング・キューの中身を入れ換えたりすることは容易である．（本稿ではその詳しい記述は省略するが，例えば〔Tanaka 88〕などを参考にされたい．）

5．変数管理をするメタインタプリタ

いままで紹介したメタインタプリタにおいては，メタインタプリタの中で変数の束縛を管理することはなく，変数の束縛は外の世界に垂れ流しであった．従って3-Lispのように変数の束縛をいじることはできなかった．そこで変数環境をexplicitに扱うメタインタプリタm-ghc について考える．

本メタインタプリタm-ghc(In,Out) のトップレベルの記述は以下の通りである．

```
m-ghc( [goal(FGoal,I,O) | In] ,Out) :- true |
        transfer(FGoal,NGoal,1,Id,Env),
        schedule(NGoal,H,T),
        exec(H,T,I,O,Id,Mem),
        memory( [enter(Env) | Mem ] , {} ),
        m-ghc(In,Out).
m-ghc( [halt | In] ,Out) :- true |
        Out= [halted] .
```

ここでm-ghc の引数InとOut はユーザへの入出力である． m-ghcはゴールが goal(FGoal,I,O)の形で入力されると，まず述語transferを起動する．述語transferは5 個の引数を持ち，第1 引数の FGoalを変換して第2 引数 NGoalに格納する．NGoal ではFGoal の変数がすべて"@数字" を割り当てた特殊な形に変形される．本メタインタプリタでは変数をすべて "@ 数字" の形に変換し，操作するのである．述語transferの第3 引数は"@数字" の形で割り当てる数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1 からn までの変数番号を割り当てたとすると，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" とその値の対応がしまわれる．

こうして述語transferでNGoal やEnv が作られると，それに対応してプログラムを実行するexecや変数をしまうmemoryが起動する．

Env や memory のなかでは変数が（変数名，値）の形で，例えば，以下のように格納される．

```
(@1，undf)        ・・・  変数 @1 は undf （未定義）である．
(@2，100)         ・・・  変数 @2 の値は 100である．
(@3，ref(@2))     ・・・  変数 @3 の値は @2 への参照ポインタである．
```

例えば，述語transferはFGoal "exam([H | T] ,H)" から NGoal "exam([@1 | @2] ,@1)" を作るが，そのときEnv は [(@1,undf),(@2,undf)] になり，それらはやがてmemoryの中にenter される．

次にexecの記述は以下のようになる．

```
exec(T,T,I,O,Id,Mem) :- true | O= [success ] ,Mem= [] .
exec( [true | H ] ,T,I,O,Id,Mem) :- true | exec(H,T,I,O,Id,Mem ).
```

```
exec( [P | H ] ,T,I,O,Id,Mem) :- not-sys(P),var(I) |
      reduce(P,T,NT,O,NO,Id,Id1,Mem,Mem1),exec(H,NT,I,NO,Id1,Mem1).
exec( [P | H ] ,T,I,O,Id,Mem) :- sys(P),var(I) |
      sys-exe(P,T,NT,O,NO,Mem,Mem1),exec(H,NT,I,NO,Id,Mem1).
exec(H,T, [Mes | I ] ,O,Id,Mem) :- true | control-exec(H,T, [Mes | I ] ,O,Id,Mem).
```

このexecは4.で示した4 引数execとほとんど同じであるが，reduceやsys-exe で変数の値を読んだり更新したりするときにはmemoryにメッセージを送っているのが相違点である．また，述語reduceでは，ゴールが実行されて展開されるに従って，動的に変数が生成され，それにも変数番号が新たに割り当てられていく．

また述語reduceは以下のように記述できる．

```
reduce(P,T,NT,O,NO,Id,Id1,Mem,NewMem):- true |
      clauses( P,FClauses),
      resolve( P,FClauses,Body,Id,Id1,Mem,NewMem),
      schedule(Body,T,NT).

resolve( P, [FClause | Cs] ,Body,Id,Id1,Mem,Mem2):- true |
      transfer(FClause,Clause,Id,IdTemp,LoEnv),
      try-commit(P,Clause,Body,LoEnv,Res,Mem,Mem1),
      resolve1( Res,P,Cs,Body,Id,IdTemp,Id1,Mem1,Mem2).
resolve( P, [] ,Body,Id,Id1,Mem,NewMem):- true |
      Body=P,
      NewMem=Mem,
      Id1=Id.
resolve1(success,_, _, _, _,IdTemp,Id1,Mem1,Mem2):- true |
      Id1=IdTemp,
      Mem2=Mem1.
resolve1(susp,P,Cs,Body,Id,_,Id1,Mem1,Mem2) :- true |
      resolve( P,Cs,Body,Id,Id1,Mem1,Mem2).
```

すなわち述語reduceは，まずclauses でゴールP にユニファイ可能な候補節のリストFClausesを渡し，resolve で候補節の中からコミット出来る節(すなわちゴールP がその節の頭部とユニファイ可能でかつガードがすべて成功する節) をひとつ選んで，ボディに展開する．scheduleはそのボディを再び述語reduceのスケジューリング・キューにいれる．

次に述語resolve であるが，resolve はユニファイ可能な候補節のリストから先頭の要素を取り，それについて述語transferでローカルな環境をまず作る，そして述語try-commitでその節がコミット出来るかどう

かを調べ，コミット可能であればその節のボディに展開する．

次に，述語try-commitを記述する．述語try-commitは，与えられたゴールと候補節の頭部とのヘッド・ユニフィケーションを行い，候補節のガードのゴールを解き，ゴールと節のコミットを試みる．ここでつぎのことに注意しなければならない．ヘッド・ユニフィケーションにおいて，節の変数（ローカル環境）に値が書き込まれるが，memoryの中にあるゴールの変数（グローバル環境）には書き込まれてはいけない．ただし節がコミットしたときには，その節のローカル環境は，enter コマンドを使ってグローバル環境に登録されなければならない．

```
try-commit(Goal,(H :- G | B),Body,LoEnv,Res,Mem,NewMem):- true |
     head-unification(Goal,H,LoEnv,LoEnv1,Res1,Mem,Mem1),
     solve-guard(G,LoEnv1,LoEnv2,Res2),
     and-result(Res1,Res2,Res3),
     commit(Res3,B,Body,LoEnv2,Res,Mem1,NewMem).

commit(success,B,Body,LoEnv,Res,Mem,NewMem):- true |
     Mem= [enter(LoEnv) | NewMem] ,
     Body=B,
     Res=success.
commit(susp,_, _, _,Res,Mem,NewMem):- true |
     NewMem=Mem,
     Res=susp.
```

また，ヘッド・ユニフィケーションの記述は，次のようになる．

```
head-unification(Goal,H,LoEnv,NewLoEnv,Res,Mem,NewMem):- true
       Mem= [deref(Goal,GV) | Mem1] ,
       deref(H,LV,LoEnv),
       h-unify(GV,LV,LoEnv,NewLoEnv,Res,Mem1,NewMem).
```

ヘッド・ユニフィケーションは，第1 引数にゴール，第2 引数にヘッド，第3 引数にヘッド側のローカル環境が入力されて呼ばれる．そして，まずゴールとヘッドがそれぞれdereference される．(dereferenceとは，変数の指している内容を求めることである．もし，内容がポインタであれば，dereference は，そのポインタを手繰っていく．) dereference するにあたって，ゴール側の変数はグローバル環境におさめられており，一方，ヘッド側の変数はローカル環境におさめられていることに注意されたい．グローバル変数のdereference は，memoryにderef 命令を送ることによって，一方，ローカル変数は，ローカルな変数環境でderefeを行うことによってなされている．

ゴールとヘッドのdereference がそれぞれ終わると，ヘッド・ユニフィケーションh-unify が行われる．GHC のh-unify は，表1 に従う．

表1　ヘッド・ユニフィケーション表

		ゴール側		
		変数 @1	アトム	複合項
ヘッド側	変数 @2	@2 <- ref(@1)	@2 <- a	@2 <- 複合項
	アトム	suspend	success / fail	fail
	複合項	suspend	fail	分解しユニファイ

ここで，@2 <- ref(@1)は，変数名 @2 の値 undf を ref(@1) に書き変えることを意味する．なお，実際のコーディングではfailは，suspend の特殊な場合として扱っている．

これらヘッド・ユニフィケーションは，次のように記述できよう．(ここではコードの一部のみを示す．

```
h-unify(GV,LV,LoEnv,NewLoEnv,Res,Mem,NewMem):-
      variable(GV),
      nonvariable(LV) |
      Res=susp,
      NewLoEnv=LoEnv,
      NewMem=Mem.
h-unify(GV,LV,LoEnv,NewLoEnv,Res,Mem,NewMem):-
      variable(GV),
      variable(LV) |
      assign(LV,ref(GV),LoEnv,NewLoEnv),
      Res=success,
      NewMem=Mem.
```

つぎにsys-exe の記述をする．この述語は，ボディ部におけるシステム述語を実行する．ここでは，ユニ

フィケーションと，加算の記述を示す．

```
sys-exe((X=Y),T,NT,O,NO,Mem,NewMem):- true |
        Mem= [unify(X,Y,Res) | NewMem] ,
        sys-exel(Res,(X=Y),T,NT,O,NO).
sys-exe(+(Z,X,Y),T,NT,O,NO,Mem,NewMem):- true |
        Mem= [deref(X,XV),deref(Y,YV) | Mem1] ,
        add(Z,XV,YV,Mem1,NewMem,Res),
        sys-exel(Res,+(Z,X,Y),T,NT,O,NO).

add(Z,XV,YV,Mem,NewMem,Res):-
        readyarg(XV,YV) |
        Ad:=XV+YV,
        Mem= [unify(Z,Ad,Res) | NewMem] .
add(Z,XV,YV,Mem,NewMem,Res):-
        notreadyarg(XV,YV) |
        Res=susp,
        NewMem=Mem,

sys-exel(success, __,T,NT,O,NO):- true |
        NT=T, NO=O.
sys-exel(susp,G,T,NT,O,NO):- true |
        schedule(G,T,NT),
        NO=O.
```

つぎにグローバル変数を管理している述語memoryを記述する．これまでに記したように，exec部分は，memoryへ命令を送ってグローバル変数をアクセスした．すなわち，memoryはexec部分からの命令により，Dbの中に格納されたグローバルな変数の値を読んだり書いたりする．こうしたmemoryへのアクセスは，

①ヘッド・ユニフィケーション時など，deref 命令により，ゴールの変数をdereference するとき，

②節のcommitが成功し，enter 命令により，ローカルな環境をグローバル環境に登録するとき，

③ボディ部のシステム述語が，unify 命令により，ユニフィケーションを行うとき，

などにおきる．

```
memory( [deref(Term,Value) | NMem] ,Db) :- true |
        deref(Term,Value,Db),
        memory(NMem,Db).
```

```
memory( [enter(Env) | NMem] ,Db) :- true |
        enter(Db,Env,NDb),
        memory(NMem,NDb).
memory( [unify(X,Y,Res) | NMem] ,Db) :- true |
        unification(X,Y,Res,Db,NDb),
        memory(NMem,NDb).
memory( [] ,Db) :- true | true.
```

deref 命令を受けたmemoryは，dereference する項とグローバル環境を入力してderef を呼ぶ．deref は以下のように記述できる．

```
deref(Term,Value,Db):-
      variable(Term) |
      search-cell(Term,Cont,Db),
      deref1(Term,Cont,Value,Db).
deref(Term,Value,Db):-
      nonvariable(Term) |
      Value=Term.

deref1( _,ref(C),Value,Db):- true |
      deref(C,Value,Db).
deref1(Cell,undf,Value,Db):- true |
      Value=Cell.
deref1( _,Cont,Value,Db):-
      Cont/=ref( _),
      Cont/=undf |
      Value=Cont.
```

述語deref は，第1 引数にdereference する項を入力して呼び出し，第2 引数にdereference された値を出力する．第3 引数は，変数環境（グローバル環境，あるいはローカル環境）を入力する．これは，dereference する項が変数（‘@数字’）であれば，変数環境を調べて，その変数の値を求める．変数の値がrefポインタであれば，さらにdereference し，値がundfのときは変数名を，そうでなければ，その値を出力する．dereference する項が変数でなければ，その項を出力する．

つぎに，enter 命令を受けたmemoryは，下に記述したenter を呼び，グローバル環境にコミットされた節のローカル環境をappendする．

```
enter(Db,Env,NDb):- true |
        append(Db,Env,NDb).
```

unify 命令を受けたmemoryは，述語unification を呼び，２つの項をユニファイする．unification は，第1，第２引数にユニファイする２つの項を，第4 引数にグローバル環境が入力されて呼ばれる．そして，それぞれの項の変数がdereference され，ユニファイが行われる．

```
unification(X,Y,Res,Db,NewDb):- true |
        deref(X,XV,Db),
        deref(Y,YV,Db),
        unify(XV,XV,Res,Db,NewDb).
```

ユニフィケーションは，表2 に従う．

表2　ユニフィケーション表

	変数 @1	アトム	複合項
変数 @2	@2 <- ref(@1)	@2 <- a	@2 <- 複合項
アトム	@1 <- a	success / fail	fail
複合項	@1 <- 複合項	fail	分解しユニファイ

例えばunify は，次のように記述できる．（ここではコードの一部のみを示す．）

```
unify(X,Y,Res,Db,NewDb):-
      variable(X),
      nonvariable(Y)  |
      assign(X,Y,Db,NewDb),
      Res=success.
unify(X,Y,Res,Db,NewDb):-
      variable(X),
      varialbe(Y) |
      assign(X,ref(Y),Db,NewDb),
```

```
        Res=success.
    unify(X,Y,Res,Db,NewDb):-
        nonvariable(X),
        nonvarialbe(Y),
        atomic(X),
        atomic(Y),
        X=Y |
        Res=success,
        NewMem=Mem.
```

6．まとめ

GHC の簡単な（4 行の）自己記述から始め，それを拡張して，fail-safe なメタインタプリタ，メタレベルから制御可能なメタインタプリタ，スケジューリング・キューやリダクション・カウンタを持つメタインタプリタなどを段階的に導いた．また自分で変数管理を行うメタインタプリタについて，その概略を記述した．（このメタインタプリタについては紙数の関係で概略のみを記したが，この稼動バージョンに興味のある方は筆者らに問い合わせられたい．）

我々の当面の目的は，GHC を使用しての簡単かつ強力なプログラミング・システムの記述にある．拡張されたメタインタプリタは，こうした目的にも非常に有用である．今後これらのメタインタプリタを用いて，特にリフレクション機能実現のための研究を続けていく予定である．

〔参考文献〕

〔Bowen 82〕 K. Bowen, R. Kowaiski: Amalgamating Language and Metalanguage in Logic programming, Logic Programming, pp.153-172, Academic Press, London, 1982.

〔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.

〔Kurusawe 86〕 P. Kurusawe: How to Invent a Prolog Machine, Third International Conference on Logic Programming, LNCS-225, p.134-148, Springer-Verlag, 1986.

〔Shapiro 83〕 E. Shapiro: A Subset of Concurrent Prolog.and Its Interpreter, ICOT Technical Report TR-003, 1983.

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

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

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