Black Boardを用いたリフレクティブGHCの実装

青木良憲, 田中二郎

筑波大学

茨城県つくば市天王台 1-1-1

0298(53)5165

aokiyo@softlab.is.tsukuba.ac.jp, jiro@softlab.is.tsukuba.ac.jp

あらまし

本論文では、並列論理型言語GHCに記述力強化のためにリフレクション機能を組み込んだリフレクティブGHCの実装について述べる。

既にリフレクティブGHCはいくつかの試験的な提案がなされているが、本論文では、BinPrologの特徴であるBlack Board機能とプログラムをバイナリ節に変換する機能を有効に用いることで、本格的なリフレクション機能を持つリフレクティブGHCを、より効率的に実現する手法について述べている。

キーワード　リフレクション, GHC, BinProlog, Blackboard, バイナリ節

Implementing Reflective GHC Based on Blackboard of BinProlog

Yoshinori Aoki, Jiro Tanaka

University of Tsukuba

1-1-1 Tennoudai, Tsukuba-shi, Ibaraki-ken, JAPAN

0298(53)5165

aokiyo@softlab.is.tsukuba.ac.jp, jiro@softlab.is.tsukuba.ac.jp

Abstract

The new implementation of Reflective GHC, which increases the description power of GHC, has been proposed. Though we already have several experimental proposals for Reflective GHC, none of those are satisfactory. In this paper yet another implementation of Reflective GHC in BinProlog has been proposed. The blackboard and the translation to the binary clause of BinProlog turned out to be very effective in implementing Reflective GHC. By using those functions of BinProlog, the new implementation of Reflective GHC has been realized more effectively.

key words　Reflection, GHC, BinProlog, Blackboard, Binary clauses

1 序論

現在、さまざまなプログラミング言語が存在しているが、理想的なプログラミング言語とは、ひとことでいえば、簡単に理解できて、しかも記述力が強力な言語であるといえるだろう。そこで、こうした簡単な枠組みで強力な記述力を得るための機能の一つとして、「リフレクション」という概念が考えられている。

リフレクションとは、一般に「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、現在の状態を自己感知し、それに応じて自己変更を行なえるようになる。よってそれらは、強力なシステムを簡単に記述することにつながるのである。

並列論理型言語 GHC[1] 上にリフレクション機能を実装したリフレクティブ GHC については、既にいくつかの提案 [2][3] がなされている。しかし、それらはリフレクションの枠組を試験的に実装したにとどまり、効率の面で問題を残すものであった。

本論文では、論理型言語 BinProlog[4][5][6] の Black Board 機能を用いて、本格的なリフレクション機能を持ち、かつ効率的なリフレクティブ GHC を実現する手法について述べている。

2 準備

本章ではリフレクション機能を実装する上でベース言語となる GHC について概要を述べる。次にリフレクティブ GHC の実装に用いた BinProlog について述べる。

2.1 並列論理型言語 GHC

GHC(Guarded Horn Clauses) は一階述語論理に基づく並列論理型言語である。

GHC プログラムはガード付き節 (guarded clause) の有限集合として表される。ガード付き節は次の形をもつ。

$$H :- G_1, \cdots, G_n \mid B_1, \cdots, B_m. \qquad n,m \geq 1$$

"|" はコミット (commit) 演算子と呼ばれる。"$G_1, \cdots, G_n$ および $B_1, \cdots, B_m$" はそれぞれガード部、ボディ部と呼ばれる。"H" は節の頭部 (head) と呼ばれる。同じ述語記号 (引数の数も同じ) を頭部にもつ節の集合をその述語の定義節と呼ぶ。

GHC プログラムの実行は Prolog プログラムの実行と似ている。主な特徴は次のような点である。

- 各ゴールは引数の結合により情報を共有する。
- 各ゴールはそれぞれ並列に実行される。
- 各ゴールは、各呼出しごとにただ一つの節にユニファイされるが、その節はコミット (操作) を用いて選択される。コミット操作とは、ガード部の実行が成功した節のうちから任意の一つを選ぶ操作である。
- ガード部の実行はゴールの引数を具体化しないように実行される。
- コミット操作ができなかったゴールは、他のゴールの実行によってコミットが可能になるまで実行を中断する。
- コミットにより選択された節は、ボディ部を実行して、その節を呼び出したゴールを具体化することができる。

つまり、GHC の実行規則は、各ゴールに対して、必要な結合が来るのをガードで待ち合わせ、その節が選択されたら、ボディを実行して結合を呼び出し側に返すというものである。

2.2 BinProlog

BinProlog は、カナダの Moncton 大学の Paul Tarau によって開発された、バイナリ節への変換に基づく、高速でコンパクトな Prolog コンパイラである。

BinProlog は C エミュレーションでの実行と、C へのコンパイルによるスタンドアロンアプリケーションの生成を特色とした Prolog である。

主な特色としては、以下のようなものがある。

- 継続情報 (Continuation) を扱うことができる。
- ガベージコレクションが可能なハッシュテーブルがあり、疎な配列を一定の時間で参照することができる。

2.2.1 バイナリ化

BinProlog では、通常の Prolog の定義節は継続情報を持ったバイナリ節に変換されて実行される。

以下に通常の定義節からバイナリ節への変換の例を示す。

```
Source clause:
     a(X):-b(X),c(X,Y),d(Y).
Binary clause:
     a(X,Cont):-b(X,c(X,Y,d(Y,Cont))).

Source clause:
```

```
        a(13).
    Binary clause:
        a(13,Cont):-true(Cont).

    Source clause:
        p(X,Y):-X,Y.
    Binary clause:
        p(X,Y,Cont):-call(X,call(Y,Cont)).
```

ユーザ自身がバイナリ節を記述するには、:- の代わりに::- を記述する。

```
    Head::-Body.
```

2.2.2 Black Board

常設の情報のために、BinProlog には新しいデータエリア (Black Board) がある。そこには任意の項が保存でき、2 キーのハッシュ関数で効率良く参照できる。

Black Board 上の情報を扱う時には、以下に述べる述語を用いる。

bb_def/3	(Key1,Key2,Value)	値の定義
bb_set/3	(Key1,Key2,Value)	値の書き換え
bb_rm/2	(Key1,Key2)	値の消去
bb_val/3	(Key1,Key2,Value?)	値の取り出し

3 Black Board を用いたリフレクティブ GHC

本章では、Black Board を用いたリフレクティブ GHC の実現法について述べる。

3.1 リフレクティブシステム

リフレクションにおける自己感知や自己変更については、高級言語のように「プログラム」と「データ」を分けて考えるのではなく、アセンブリ言語のように、時には「プログラム」を「データ」と解釈したり、自分で自分のプログラムを書き換えたりする能力が、アセンブリ言語よりも整理された形で必要となる。

このような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、そのように実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。このような仕組みはメタシステムの枠組を用いて実現する。メタシステムの中では、その問題領域であるオブジェクトシステムがデータとして扱われるので、後は、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。

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、リフレクションが起こったときだけ、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している。

メタシステムは、リフレクションに必要な計算だけを行ない、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、メタシステムの実行中に更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、原理的にはリフレクティブタワーの構築が可能である。

3.2 リフレクティブ GHC の構成

3.2.1 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、それらが全てデータとして表現されていることが必要である。既存の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する。

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。

変数、変数束縛　メタレベルで変数の操作を行なうために、「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「基底項」に対応させる。

3.2.2 リフレクティブシステムにおけるメタの階層

リフレクティブタワーにおいては、メタの階層ができているので、変数の「基底項」としての表現について、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となる。

レベルの示し方には絶対的レベルを使用する。すなわち、

オブジェクトレベル	… レベル 1
メタレベル	… レベル 2
メタメタレベル	… レベル 3

のように定義する。

3.2.3 変数、変数束縛

変数、変数束縛の管理は基底項で表現されている変数にレベルをつけて、キー1をLevel、キー2をId(番号)としてBlack Boardを用いて行なう。

```
@(Level,Id) → Value
```

以下に変数束縛の例を示す。

- `@(1,1) → undf`
  レベル1の変数1の値は未束縛である。
- `@(1,2) → a`
  レベル1の変数2の値はaである。
- `@(1,3) → ref(@(1,2))`
  レベル1の変数3の値はレベル1の変数2への参照ポインタである。
- `@(1,4) → f(@(1,1))`
  レベル1の変数4の値は構造体であり、その関数記号はf、第1引数はレベル1の変数1への参照ポインタである。
- `@(2,1) → b`
  レベル2の変数1の値はbである。
- `@(2,2) → ref(@(1,3))`
  レベル2の変数2の値はレベル1の変数3への参照ポインタである。

変数から変数への参照ポインタは、2つの変数がユニファイされたときに生じるものであり、これらに関しては、変数の値が要求された時には、参照ポインタをたどって値を求めるdereferenceという操作が必要となる。

このように管理することによって、変数がハッシュ関数で参照できるので、参照の効率が向上する。また、Black Boardを用いて管理されるので、変数環境を陽に表す必要がなくなる。

3.2.4 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、現在実行している計算システムの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以下の形式で、定義される。

```
reflect(<ゴール>,Cont)
          :- <ガード・ゴール列> |
             <ボディ・ゴール列>.
```

表 1: 定義節の有効範囲

レベル定義	述語の種類	有効なレベル
1:	通常の述語	オブジェクトレベル
2:	通常の述語	メタレベル
3:	通常の述語	メタメタレベル
1:	リフレクティブ述語	オブジェクトレベル以上
2:	リフレクティブ述語	メタレベル以上
3:	リフレクティブ述語	メタメタレベル以上

リフレクティブ述語は、2引数の述語reflectを用いて定義され、第1引数にリフレクションを起こしたゴールの表現が入る。第2引数には、その時の継続情報が入る。

<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>には一段上のレベルで行なうべき処理を記述する。このガードやボディでは、凍結された計算システムの状態を参照したり、変更したりすることができる。

3.2.5 述語定義

ユーザ定義述語は、BinPrologのデータベースを用いて、定義する。

また、絶対的レベル概念を導入することに伴い、ユーザ定義述語のデータベースを以下の形式で記述することとする。

```
Level: Head :- Guard | Body.
```

各述語は定義節に記述されているレベルがその時の計算レベルと等しい時に有効となる。但し、リフレクティブ述語については、定義節のレベルがその時の計算レベル以下である時に有効となる。定義節の有効範囲のについての例を表1に示す。

ユーザ定義述語の記述例を以下に示す。

```
1:append([H|T],Y,Z) :- true|
    Z=[H|Z2],
    append(T,Y,Z2).
1:append([],Y,Z) :- true|
    Z=Y.

1:reflect(get_q(Q),Cont) :- true|
    Q=Cont.
```

3.2.6 リフレクティブ述語の実行

ユーザがリフレクションを起動するためには、あらかじめそのゴールをリフレクティブ述語の形式で定義して

おく必要がある。

例えば、オブジェクトシステムの実行中にそのゴールが呼ばれると、まず、(一段上の)メタシステムが動的に作られメタレベルの計算が始まる。メタシステムの実行が終った時点で、再びオブジェクトレベルのゴールの実行が開始される。

3.3 リフレクティブGHCの実現

前節で述べてきたような要素を元に、BinPrologを用いてリフレクティブGHCを実現する。

リフレクティブシステムr_ghc(Goal,Out)のトップレベルの記述は以下の通りである。

```
r_ghc(Goal,Out):-
    Level=1,
    transfer(Goal,GoalRep,1,
                          Id,Level),
    exec(GoalRep,Id,IdRep,
                         Level,Res),
    make_result(Res,Id,Level,Out),!.
```

r_ghcはゴールを入力すると、述語transfer、述語exec及び述語make_resultを起動する。

述語transferは実行するゴールの表現をメタレベル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。

述語make_resultは実行結果を表すResから出力情報Outを作り出す述語である。

また、述語execは5引数からなり、第1引数は実行ゴール、第2引数は次に割り当てられる変数番号の初期値、第3引数は実行後の変数番号の値であり、第4引数は実行ゴールが実行される計算レベル、第5引数は実行結果を表す。execは以下のように記述される。

```
exec(true,Id,Id,Level,Res):-!,
    Res=success.
exec(false,Id,Id,Level,Res):-!,
    Res=suspend.
exec((P,Q),Id,IdRep,Level,Res):-!,
    exec(P,Id,Id1,Level,Res1),
    exec1((P,Q),Id,Id1,IdRep,
                    Level,Res1,Res).
exec(P,Id,IdRep,Level,Res):-
    user_defined(P,Level),!,
    reduce(P,Id,Level,Id1,Body),
    exec(Body,Id1,IdRep,Level,Res).
exec(P,Id,Id,Level,Res):-
    sys(P),!,
    sys_exe(P,Res).
```

述語execは、以下のように動作する。

1. 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falseであればゴール実行が中断する。
3. 実行すべきゴールが複数個あれば、それをexecとexec1に分解して実行する。
4. 実行すべきゴールがユーザ定義述語であれば、リダクションを行ない、そのゴールを実行する。
5. 実行すべきゴールがシステム組み込み述語である時には、それを解く。

ここで、述語exec1の定義は次のようになる。

```
exec1((P,Q),Id,Id1,IdRep,Level,
                    success,Res):-!,
    exec(Q,Id1,IdRep,Level,Res).
exec1((P,Q),Id,_,IdRep,Level,
                    suspend,Res):-!,
    exec(Q,Id,Id1,Level,Res1),
    exec(P,Id1,IdRep,Level,Res2),
    and_result(Res1,Res2,Res).
```

述語exec1は、以下のように動作する。

- ゴールPの実行が成功しているならばゴールQを実行する。
- ゴールPの実行が中断したならばゴールP,Qの順序を入れ換えてQ,Pの順に再実行する。
  述語and_resultはRes1とRes2の双方がsuccessならばsuccessを、そうでなければsuspendをResに返す述語である。

また、述語reduceはゴールPにユニファイ可能な定義節からコミットできる節をひとつ選んで、その節のボディ部のゴールを返す。述語sys_exeはシステム組み込み述語を解き、結果を返す。

述語reduceは以下のように記述される。

```
reduce(G,Id,Level,IdRep,Body) :-
    replace(G,G1,Level),
    fclause(G1,Level,GrBs),
    transfer((G1:-GrBs),NClause,
                      Id,IdTemp,Level),
    try_commit(G,NClause,Level,
                  IdTemp,IdRep,Body),!.
reduce(G,Id,Level,Id,false).
```

```
try_commit(Goal,(Head:-GrBs),Level,
                    IdTemp,IdRep,Body):-
    h_unify(Goal,Head),
    guard(GrBs,Guard,Body),
    exec_guard(Guard),
    IdRep=IdTemp.
```

述語replace と述語fclause でゴールG にユニファイ可能な定義節を探す。そして、述語transfer でその定義節をオブジェクトレベルの記法に書き直した後、述語try_commit でコミットできるかを調べて、コミットできたらその定義節のボディゴール列を返す。

述語try_commit では、まず述語h_unify が呼びだし側のゴールを具体化しないようにヘッドユニファイを行なう。それが成功すると、述語guard が定義節からガードゴール列とボディゴール列を分離し、述語exec_guard でガードゴール列の実行を行なう。ガードゴール列の実行が成功したら、Id の値を変更して、終了する。

述語try_commit が失敗したならば、バックトラックが生じ、もう一度述語fclause が呼ばれ、ユニファイ可能な別の定義節を探す。他にそのような定義節が見つからなかったならば、述語reduce の2つめの定義節により、false をリダクションの結果として返す。

リフレクションの実現は、上記のメタシステムexec の定義節に、以下の定義節を追加する。

```
exec(P,Id,Id,Level,Res,Cont)::-
    reflective(P,Level,
    '$bin_cut'('$cut',
    is(Meta_Level,Level+1,
    exec(reflect(P,Cont),1,IdRep,
                    Meta_Level,Res,
    delete_bb(Meta_Level,IdRep,
    (Cont))))))).
```

この定義節はリフレクティブタワーの構築を行なう。

この定義節は継続情報を取り出してそれを利用しているので、バイナリ節で記述されている。
'$bin_cut'('$cut', …) は、通常の表記では "!"(カット) に相当する表記である。

実行ゴールがリフレクティブゴールであれば、まずMeta_Level をLevel+1 とする。次にメタシステムの計算を行なう述語exec が起動され、リフレクションを起こしたゴールの実行を開始する。メタシステムでの実行が終ると、述語delete_bb がメタレベルの変数環境を破棄し、残りのオブジェクトシステムのゴールの実行を再開する。

実際のインプリメントでは、以上のプログラムにBinProlog-Tcl/Tk インタフェース [7] を用いたウインドウ関係の機能を付加している。

3.4 実行例

リフレクティブ GHC の実行例を図1及び図2に示す。

図1,2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。

図1の左側のウインドウで奥にあるものがオブジェクトレベル (レベル1) に対応するウインドウである。

入力ゴールとしてtest(Q,A) を与えると、それが「基底項」で表現されたtest(@(1,1),@(1,2)) が表示されている。述語get_q が呼び出されると、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。図1の左側で手前にあるウインドウがメタレベル (レベル2) に対応するウインドウである。メタレベルのウインドウで、reflect(…はリフレクティブゴールを表している。その後に表示されている項目は、リフレクションによって一時凍結されたオブジェクトレベルの変数束縛の内容である。

図1でメタシステムの実行が終ると、オブジェクトシステムの実行に戻る。図2にはオブジェクトシステムに戻った後、全ての計算が終了した状態が示されている。計算後の変数束縛の状態がそれぞれ示されている。

4 既存のリフレクティブ GHC との比較

過去に田中らは [3] において試験的なリフレクティブ GHC の実装 (以下、旧リフレクティブ GHC と呼ぶ) を行なっている。本章では旧リフレクティブ GHC と第3章で述べた BinProlog を用いたリフレクティブ GHC(新リフレクティブ GHC) との比較を行なう。

4.1 旧リフレクティブ GHC との相違点

新リフレクティブ GHC において、旧リフレクティブ GHC と異なる点について表2にまとめた。

4.2 変更による利点

- 変数管理を Black Board を用いて行なうことにより、変数の個数に関わらず一定の時間で変数を参照できるようになる。また、述語の呼びだしごとにリストで表現された変数環境を引数として渡す必要がなくなっている。

- 絶対的レベルの導入により、リフレクションが起こった時に変数の表現を書き直す必要がなくなっている。絶対的レベルと相対的レベルでは本質的な違いはないが、リフレクションが起こった時にメタシステムに実行が移るので、リフレクションを起こしたゴールやその時の計算システムの環境の表現を一段下の表現にしなければならない。このとき相対的レベルの記法を用いていると、そのゴールや環境

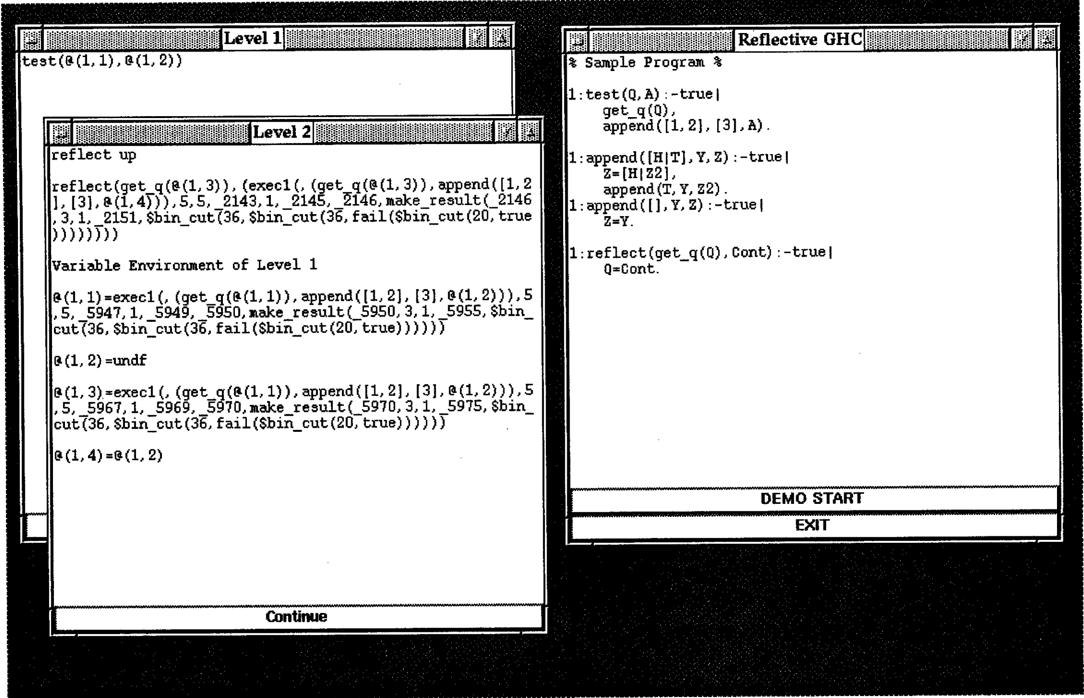

図 1: 実行例 (1)

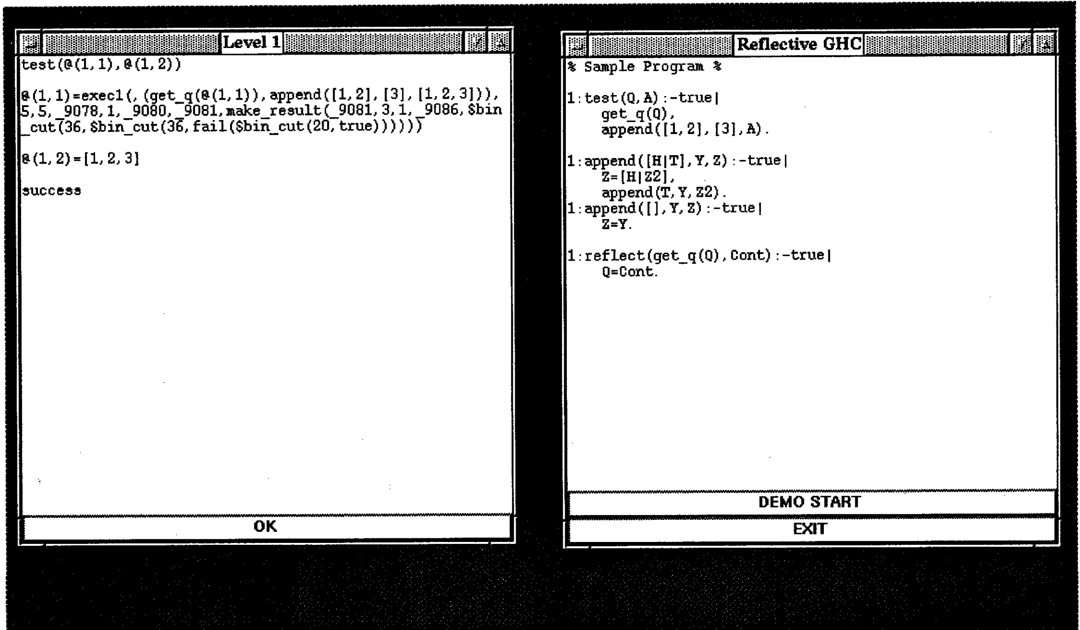

図 2: 実行例 (2)

表 2: 新旧リフレクティブ GHC の相違点

	新リフレクティブ GHC	旧リフレクティブ GHC
レベルの表現方法	絶対的レベル	相対的レベル
変数管理	Black Board に書き込む	リストとして保持する
データベースの構成	絶対的レベルにより有効範囲を限定された述語定義による各レベル共通のデータベース	メタ述語定義及びリフレクティブ述語定義によるレベルごとに異なったデータベース
ユーザ定義述語	Prolog の述語定義データベースを用いる	リストとして保持する
実行ゴール列	Prolog の AND 並列の実行機構を使用する	差分リストによって明示されたゴールキュー

の表現を逐一、一段下の表現に書き直さねばならない。これが絶対的レベルの記法を用いていると、その時の計算システムのレベルを示す要素を変更するだけで、その他の操作は一切必要がなくなる。

- 絶対的レベルを用いた各レベル共通のデータベースを用いることで、前項と同様に、リフレクションが起こった時に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を作成する必要がなくなる。

- ユーザ定義述語を Prolog の述語定義データベースを用いて定義することで、変数管理の項と同様に、述語のリダクションにかかる時間が短縮される。

- 実行ゴール列の管理をを BinProlog の実行機構に任せることで、リフレクションの処理を BinProlog のバイナリ節で記述しておくことで、リフレクションが起こった時に、継続情報をたやすく取り出し、取り出した継続情報の変更も原理的には可能である。

5 結論

本研究では、BinProlog を用いて、本格的なリフレクション機能を持ち、かつ効率的なリフレクティブ GHC を実現する手法について述べた。Blackboard 機能を有効に用いることで、いくつかの点について処理の効率化をすることができた。

今後はこのリフレクティブ GHC をさらに細部まで実装し、これを用いて応用的なものを設計することが考えられる。

参考文献

[1] 淵一博監修 古川康一・溝口文雄共編: 並列論理型言語 GHC とその応用, 知識情報処理シリーズ 6, 共立出版, 1987

[2] Jiro Tanaka: Meta-interpreters and Reflective Operetions in GHC, In *Proceedings of International Conference on Fifth Generation Computer Systems 1988*, pp.774-783, ICOT, November, 1988

[3] Jiro Tanaka and Fumio Matono: Constructing and Collapsing a Reflective Tower in Reslective Guarded Horn Clauses, In *Proceedinds of International Conference on Fifth Generation Computer Systems 1992*, pp 877-886, ICOT, 1992

[4] P.Tarau: Wam-optimizations in BinProlog: towards a realistic continuation passing prolog engine, Technical Report 92-3,Dept.d'Informatique, Université de Moncton, July, 1992

[5] P.Tarau: Low level issues in implementing a high-performance continuation passing binary prolog engine,In M.-M.Corsini,editor, In *Proceedings of JFPL'94*, June, 1994

[6] P.Tarau: BinProlog3.00 User Guide, 1994

[7] P.Tarau and B.Demoen: Language embedding by dual compliation and state mirroring, In *Proceedings of 6-th Workshop on Logiv Programing Environments, Santa Margherita Ligure,1994*, June, 1994