ICOT Technical Memorandum: TM-0691

TM-0691

GHCによる並列同一化とメタインタプリタ

藤田　博

March, 1989

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による並列同一化とメタインタプリタ

藤田 博

Institute for New Generation Computer Technology,
1-4-28 Mita, Minato-ku, Tokyo, 108, Japan
phone: 03-456-4365　e-mail: fujita@icot.junet

概要

GHCによるメタインタプリタ作成のための基本的な技法を与えること，及び部分計算に基づく最適化技法の適用を目的として，同一化(unifier)，Prologインタプリタ，及びGHCインタプリタのGHCによる実現例を示す．

1. はじめに

論理プログラミングとメタプログラミングとは極めて相性がよいとされている．Prologでの経験はこのことを裏づけた．メタプログラミングの唯一の難点とされた実行効率の問題も，部分計算の技法を用いて解決できるという見通しが得られている．

並列論理型言語GHCの場合はどうであろうか．Prologのようにうまくいくものであろうか．この報告では，この問題について考える際の叩き台として幾つかのケーススタディを示す．

特に重要と考えられるのは，オブジェクトレベル変数の表現法と同一化の問題である．Prologでは，変数は単一代入を唯一の制約とする論理変数，同一化はいわゆるfull unificationといわれるものであった．他方，GHCでは，変数がゴール側，プログラム節側のいずれに属するものかによって，同一化の際の代入に制約が加えられている．

また，逐次実行のPrologにおいては，var述語が変数の束縛状況判定の目的に効果的に用いられた．他方，GHCでは束縛を待つwait述語は可能でも，変数の未束縛を判定して利用するということは許されていない．従って，Prologでは一定の注意のもとにオブジェクトレベル変数をメタ変数で表現することが可能だったが，GHCではこれは不可能である．

そういうわけで，GHCではオブジェクトレベル変数はメタプログラム上の基底項として表現せざるをえない．このことは，メタプログラム作成者にとっての現実的な問題としてかなり大きな負担である．Prologのように3行で済むようなメタインタプリタは，GHCでは(特別な組み込み述語の仮定なしには)もはや実現できないのである．

2節では，基本となるGHCプログラミング技法を簡単な例を用いて示す．3節では，同一化の並列アルゴリズムとその実現について述べる．この同一化アルゴリズムを用いて，4節ではPrologのインタプリタ，5節ではGHCのインタプリタを示す．

2. 要素となるプログラミング技法

2.1. ストリーム

並列プログラムの基本構成要素は，並列に動作する複数のプロセスとそれらの間の通信である．GHCではゴールをプロセス単位とし，ゴールの間で共有される変数を媒介して通信が行なわれると考える．様々な通信方式が考えられるが，なかでも不定長リストを利用したストリーム通信が重要である．

ストリームを使った例として，エラトステネスのふるいに基づく素数生成プログラム(1)を次に示す．

```
primes(J) :- true | gen(2,I), sift(I,J).
gen(N,I) :- true | I=[N|I1], N1:=N+1, gen(N1,I1).
sift([P|I],J) :- true | J=[P|L], filter(I,P,K), sift(K,L).
filter([N|I],P,K) :- 0=:=N mod P | filter(I,P,K).
filter([N|I],P,K) :- 0=\=N mod P | K=[N|K1], filter(I,P,K1).
```

プロセスgen(N,I)は，Nから始まる自然数列のストリームIを生成する．sift(I,J)は，ストリームIの先頭Pが素数であることを仮定してこれをストリームJの先頭に出力すると同時にPのふるいを生成する．ふるいfilter(I,P,K)は，ストリームIの先頭Nが素数Pの倍数であればこれを捨て，さもなければストリームKの先頭に出力する．各プロセスが再帰呼出しの度にストリームの次の要素を生成したり(gen)，入力ストリームの次の要素を見て出力ストリームの次の要素を決定したり(sift, filter)，新たなプロセスを生成したり(sift)しながら一本のラインを構成していく様子を思い浮かべるとよい．

この例題はコンパクトながら，ストリームの用法を通じて次の二つのプログラミング・パラダイムを取り込んでいる．

- 生成検定法
- パイプライン処理

genが生成部，siftとfilterが検定部となって生成検定法を実現していることは明らかであろう．また，siftと次々に生成されるfilterが一線上に並び，各々のポイントで流れ作業の一部を担当するという，まさにパイプライン処理を実現していることもわかる．

1.2. 信号

次の例では，共有変数が特殊な信号伝達に利用される場合を見ることにしよう．

```
find(X,T,A) :- true | search(X,T,no-No,F), result(No,F,A).
search(_,_,_,found) :- true | true.
search(X,[L|R],N1-N2,F) :- true |
  search(X,L,N1-N3,F), search(X,R,N3-N2,F).
search(X,X,_,F) :- true | F=found.
search(X,Y,N1-N2,_) :- Y\=X, Y\=[_|_] | N1=N2.
result(no,_,A) :- true | A=not_found.
result(_,found,A) :- true | A=found.
```

find(X,T,A) は，二分木 T の葉に X が現れているかどうかを判定する．search(X,T,N1-N2,F) は，T が木のとき，左右の枝を見にいく．T が X に等しいときは，F を found に具体化する．T が木でなく X に等しくもないときは，N1 と N2 を同一化する．木のどこかで X の葉が見つかればただちに (他の葉のチェックの終了を待たずに) found の結果がわかるし，木のすべての葉が X と異なることがわかったときのみ not_found の結果がわかるようになっていることに注意しよう．

ここで用いられた信号は，次の二つのタイプのものである．

- 打ち切り (abort) 通告
- 完了通告

打ち切り通告は変数を一群のプロセスたちで共有することにより実現される．これは加盟プロセスのいずれか一つの結果が残り全体のプロセスの続行を不要とするとき，それらを強制終了させるためのものである．従って，原則的にこの変数への書き込みはいずれかただ一つのプロセスによりただ一回だけ行われ，その後，他のプロセスはできる限り早い時期にこれを観測するものとする．(ただし，書き込みの値が同一である場合は多重書き込みがあっても不都合は生じない．) この変数のことを打ち切り端子と呼ぶことにする．また，これがプロセス間で共有されることを強調したいとき，この共有関係を打ち切り線と言うことにしよう．

一方，search の第3引数 (N1-N2) のような仕組みは，複数のプロセスによる共同作業の完了を合図する目的に用いられる．これを完了双極子と呼ぶことにする．ある共同作業に加わるプロセスは，仮想的に一線上に配置される．これを完了線と呼ぶことにする．各プロセスが与えられる完了双極子のそれぞれの極は，このプロセスの両隣のプロセスのそれぞれと共有される．そして，個々のプロセスは自分の受け持ち作業が終了すると，この2極の間を短絡する (2変数を同一化する)．最終的に全部のプロセスが短絡を行ったと

き，完了線の両端の極が導通する(両端の項が同一化される)．この両端極を保持するプロセスはこの導通を観測でき，全体作業の完了を知ることができるというわけである．

3. 同一化

1節で述べたように，GHCの処理系が提供する同一化は，GHC変数を単純に論理変数とみてfull unificationを与えるものではない．従って，基底項表現された変数を含む項の同一化手続きを与えるのが，GHCのメタプログラミングの手始めである．

同一化は既によく知られた手続きであり，逐次的アルゴリズムをGHCで記述するのに特に困難はない．しかし，ここではできる限りの並列性を引き出すべく，1節で示した要素的技法を駆使して再考してみよう．

同一化は二つの項を入力し，同一化可能な場合に変数置換(substitution)を出力するものである．同一化不可能な場合は単に失敗を知らせればよい．入力項が部分項をもつとき，全体の同一化は部分同一化に分解される．各部分同一化はそれぞれ独立に変数置換を出力する．これら独立に得られた変数置換をまとめて全体の結果を得るには，それら相互の整合性をチェックする必要がある．そこで，まず

```
unify(X,Y,Res) :- true | generate(X,Y,Subst), sift(Subst,Res).
```

と考える．

ところが，sift(Subst,Res)プロセスドでも整合性のために副次的に同一化問題が発生し，そこから新たな変数置換が生成される可能性がある．そこで，

```
unify(X,Y,Res) :- true |
  generate(X,Y,S1), merge(S1,S2,S3), sift(S3,S2,Res).
```

S2はsift下で新たに発生する変数置換である．これはフィードバックされ，もとのS1と併合され，S3となって再びsiftに流入する．

これに，成否信号線(完了双極子Succ-Sucz と打ち切り端子Abort)を設けて，次のようにトップレベルの形が決まる．

```
unify(X,Y,Res) :- true |
  unify(X,Y,S1,Succ-succ,Abort),
  merge(S1,OutS2,InS,Abort),
  sift(InS,OutS,OutS2,Abort),
  uResult(Succ,Abort,OutS,Res).

  uResult(    _,abort(0),_,Res) :- true | Res=fail.
  uResult(succ,Abort,OutS,Res) :- true | Res=OutS, Abort=abort(1).
```

uResult(Succ,Abort,OutS,Res) は，完了双極子 Succ-Sucz の一端 Succ (これを成功端子と呼ぶことにする)，打ち切り端子 Abort，及び変数置換ストリーム OutS を観測し，結果 Res を決定する．打ち切り端子に abort(0) が観測されたときは，結果は同一化の失敗 fail である．一方，成功端子に成功 succ が観測されたときは，結果は変数置換ストリームそのものと置かれる．このとき，同時に打ち切り端子に打ち切り信号 abort(1) を流す．この信号の必要性については後で述べる．

次の unify/5 は上の generate に対応するもので，変数置換の生成者である．

```
unify(_,_,_,_,abort(_)) :- true | true.
unify(X,X,OutS,Succ-Sucz,_) :- true | OutS=[], Succ=Sucz.
unify('$var'(I,V),T,OutS,Succ-Sucz,_) :- true |
  OutS=[('$var'(I,V)=T^NewT)*(Succ-Sucz)].
unify(T,'$var'(I,V),OutS,Succ-Sucz,_) :- true |
  OutS=[('$var'(I,V)=T^NewT)*(Succ-Sucz)].
unify(X,Y,OutS,Succ_Sucz,Abort) :-
    X\=Y, X\='$var'(_,_), Y\='$var'(_,_) |
  functor(X,F,N), functor(Y,G,M),
  unify2(F/N,G/M,X=Y,OutS,Succ_Sucz,Abort).

  unify2(_,_,_,_,_,abort(_)) :- true | true.
  unify2(F/N,F/N,X_Y,OutS,Succ_Sucz,Abort) :- true |
        unify3(N,X_Y,OutS,Succ_Sucz,Abort).
  unify2(F_N,G_M,_,_,_,Abort) :- F_N\=G_M | Abort=abort(0).

  unify3(_,_,_,_,abort(_)) :- true | true.
  unify3(N,X=Y,OutS,Succ-Sucz,Abort) :- N>0 |
      N1:=N-1, arg(N,X,X1), arg(N,Y,Y1),
    unify(X1,Y1,S1,Succ-Sucm,Abort),
    unify3(N1,X=Y,S2,Sucm-Sucz,Abort),
    merge(S1,S2,OutS,Abort).
  unify3(0,_,OutS,Succ-Sucz,_) :- true | OutS=[], Succ=Sucz.
```

ここで，変数の表現 '$var'(I,V) を説明しておこう．I は変数識別番号，V は変数の印字名である．このようにオブジェクトレベルの変数は基底項として表現されている．

unify(X,Y,OutS,Succ-Sucz,Abort) は，X と Y を同一化する過程で生じる変数置換をストリーム OutS に出力する．X と Y が既に同一であるとき，この出力ストリームを閉じると同時に，完了双極子 Succ-Sucz を短絡する．即ち，この部分同一化は成功して完了する．

X と Y のいずれか一方が変数のとき，変数置換を

```
OutS=[('$var'(I,V)=T^NewT)*(Succ-Sucz)]
```

のように出力する．ここで，T が変数 '$var'(I,V) に対する置換項，NewT は全体の同一化が成功した時点で定まる予定の (dereference 後の) 置換項 (の予約席) である．また，この変数置換には完了双極子 Succ-Sucz を付随させておく．変数置換が生成されたこの段階では，まだこの部分同一化は成功とは言えないことに注意．

X と Y が同一でなく，かついずれも変数でないときは，構造を分解して部分項ごとに同一化を試みる．unify2 はこの分解直後に両項の主要構造 (primary functor の名前と arity) を見て，その同一化可能性を判定する．もしこの時点で同一化不可能と判れば，打ち切り端子 Abort に打ち切り信号 abort(0) を流す．この信号 (レベル 0) は，全体の同一化の失敗を示す．主要構造の比較で同一化可能性が残っている場合は，unify3 において，各部分構造について unify/5 を起動する．この際，完了双極子は短絡されたときに各部分同一化が成功線を形成する (直列に結合される) ように設定され，打ち切り端子は各部分同一化に公開される．また，各部分同一化の出力ストリームは併合されて，親の出力ストリームに一本化される．

次に，ふるいの方の詳細について述べる．

```
sift(_,_,_,abort(0)) :- true | true.
sift(_,OutS,_,abort(1)) :- true | OutS=[].
sift([(V=T^NewT)*(Succ-Sucz)|InS],OutS,OutS2,Abort) :- true |
  Succ=Sucz,
  OutS=[V=NewT|OutS1],
  merge(OutS21,OutS22,OutS2,Abort),
  filter(InS,V,T,NewT,InS1,OutS21,Abort),
  sift(InS1,OutS1,OutS22,Abort).
sift([],OutS,OutS2,_) :- true | OutS=[], OutS2=[].
```

sift(InS,OutS,OutS2,Abort) の入力ストリーム InS に流れてくる変数置換は，主出力ストリーム OutS に送り出される．それと同時にその変数で後続の変数置換をふるいにかけるために filter プロセスを生成する．副出力ストリーム OutS2 は，filter がその処理過程で新たに生成する変数置換を回収するためのものである．

```
filter(_,_,_,_,_,_,abort(0)) :- true | true.
filter(_,_,T,NewT,_,_,abort(1)) :- true | T=NewT.
filter([(V=TT^_)*Succ_Sucz|InS],
       V,T,NewT,OutS,OutS2,Abort) :- true |
  merge(OutS21,OutS22,OutS2,Abort),
  unify(TT,T,OutS21,Succ_Sucz,Abort),
  filter(InS,V,T,NewT,OutS,OutS22,Abort).
filter([(VV=TT^NewTT)*Succ_Sucz|InS],
```

```
        V,T,NewT,OutS,OutS2,Abort) :- VV\=V |
    deref(TT,V=NewT,NewTT1,Abort),
    deref(T,VV=NewTT,NewT1,Abort),
    OutS=[(VV=NewTT1^NewTT)*Succ_Sucz|OutS1],
    filter(InS,V,NewT1,NewT,OutS1,OutS2,Abort).
filter([],_,T,NewT,OutS,OutS2,_) :- true |
    T=NewT, OutS=[], OutS2=[].
```

filter(InS,V,T,NewT,OutS,OutS2,Abort) は，自分の担当している変数 V で InS に流れてくる変数置換をふるいにかける.

流れてきたのが同じ V に対するもの V=TT^_ であるとき，この filter で保持している置換値 T と今入力した置換値 TT を同一化しなければならない．そして，流入した変数置換に付随していた完了双極子 Succ_Sucz は，この新たな同一化の完了双極子としてそのまま引き継がれる.

流れてきたのが V と異なる変数に対するもの VV=TT^NewTT であるとき，この二つの変数置換を互いに相手の置換値に適用させる (deref)．そして，その結果 VV=NewTT1^NewTT が OutS に出力され，V の filter は新しい置換値 NewT1 を保持しつつ次の入力変数置換に備える.

ここで，何故打ち切り信号 abort(1) が必要となるのか述べておこう.

上に示したように，filter は置換値同士の同一化のために新たな変数置換を生成する場合がある．新たに生成された変数置換は，最初に unify/5 で生成された変数置換と同格であり，同列のふるいにかける必要がある．従って，頭の unify/3 の本体において，sift の副出力に流されてきたこれら中間生成された変数置換たち S2 は，同じ sift にフィードバックされているのである.

ところが，容易にわかるようにこのフィードバックループは，ふるい全体のデッドロックを引き起こす可能性を導く．どのようなときにデッドロックとなるのであろうか．それは，sift をすべての変数置換が流れ過ぎた後に起こる．そのとき，すべての変数置換生成者 unify/5 は生成を完了しており，filter と sift (及び幾つかの merge) プロセスだけがフィードバックループに繋がったまま入力待ちになっている.

さて，このデッドロックは解消できるはずである．それは，同一化 (ここでは occur check の必要性が無いものとする) がそもそも有限の計算であることを考えれば明らかである．では，どのように解消するか.

それには，sift をすべての変数置換が流れ過ぎたことを観測できるようにすればよい．実は，unify/5 の完了双極子 Succ-Sucz がそのための観測点を提供している．unify/5

で，入力項対が既に同一であるときには完了双極子が短絡される．また，すべての変数置換には完了双極子が付随しており，これは sift を通過後，短絡される．すべての完了双極子は直列に結線されているので，全体の同一化の成功は結局，sift をすべての変数置換が流れ過ぎたときにトップレベルの完了双極子が導通して Succ(完了線の一方の端) が succ(完了線のもう一方の端) の値を得たときに確認できるわけである．

uResult では，全体の同一化の成功を観測するや，打ち切り端子 Abort に abort(1) を流す．これは実は，デッドロック状態に陥っているはずのふるいプロセス群に，打ち切りを知らせるための信号なのである．

なお，filter プロセスは入力項に含まれる変数のそれぞれにつき高々一つしか生成されない．しかも変数置換の全ての対についての相互比較および dereference が，ただ一回ずつ試みられるようになっている．

次に，deref のコードを示しておく．

```
deref(_,_,_,abort(0)) :- true | true.
deref('$var'(I,V),'$var'(I,V)=S,NewT,_) :- true | NewT=S.
deref('$var'(I,V),'$var'(J,W)=S,NewT,_) :- (I,V)\=(J,W) |
  NewT='$var'(I,V).
deref(T,U_S,NewT,Abort) :- T\= '$var'(_,_) |
  functor(T,F,N), rotcnuf(NewT,F,N),
  deref1(N,T,U_S,NewT,Abort).

  deref1(_,_,_,_,_,abort(0)) :- true | true.
  deref1(N,T,U_S,NewT,Abort) :- N>0 |
    N1:=N-1, arg(N,T,A), arg(N,NewT,B),
    deref(A,U_S,B,Abort), deref1(N1,T,U_S,NewT,Abort).
  deref1(0,_,_,_,_) :- true | true.
```

最後に，ストリーム併合子のコードを示す．よく知られた普通の定義だが，打ち切り端子 (Abort) をもっていることに注意．

```
merge(_,_,_,abort(_)) :- true | true.
merge([A|X],Y,    Z,Abort) :- true | Z=[A|Z1], merge(X,Y,Z1,Abort).
merge(X,    [A|Y],Z,Abort) :- true | Z=[A|Z1], merge(X,Y,Z1,Abort).
merge([],   Y,    Z,_) :- true | Z=Y.
merge(X,    [],   Z,_) :- true | Z=X.
```

4. Prolog インタプリタ

前節で述べた同一化プログラムを用いて，Prolog のインタプリタを記述してみる．ここでは，純 Prolog を仮定し，与えられたゴールに対し全解を求めるものとする．

トップレベルは次のようである.

```
bagof(Q,A) :- true |
  copy(Q,G,?), solve(G,top,[],OutSS-[]), answer(OutSS,G,A).

  answer([S1|OutSS],G,A) :- true |
    A=[A1|As], applyS(G,S1,A1), answer(OutSS,G,As).
  answer([],_,A) :- true | A=[].
```

インタプリタの主要部分は solve/4 で, "Prolog in Prolog" の体裁に近い部分である.

```
solve(true,       _,InS,OutSS-Z) :- true | OutSS=[InS|Z].
solve((G1,G2),Path,InS,OutSS_Z) :- true |
  solve(G1,1(Path),InS,OutSS1-[]),
  solveNext(OutSS1,G2,r(Path),OutSS_Z).
solve(G,      Path,InS,OutSS_Z) :- G\=true, G\=(_,_) |
  clauses(G,Clauses),
  tryClauses(Clauses,0,G,Path,InS,OutSS_Z).

  tryClauses([C1|Clauses],N,G,Path,InS,OutSS-Z) :- true |
    N1:=N+1,
    copy(C1,(H:-B),c(N1,Path)),
    unify(G,H,InS,U),
    expand(U,B,c(N1,Path),OutSS-M),
    tryClauses(Clauses,N1,G,Path,InS,M-Z).
  tryClauses([],_,_,_,_,OutSS-Z) :- true | OutSS=Z.

    expand(fail,_,       _,OutSS-Z) :- true | OutSS=Z.
    expand(U,   B,NewPath,OutSS_Z) :- U\=fail |
      applyS(B,U,B1),
      solve(B1,NewPath,U,OutSS_Z).

solveNext([InS1|InSS],G,Path,OutSS-Z) :- true |
  solve(G,Path,InS1,OutSS-M),
  solveNext(InSS,G,Path,M-Z).
solveNext([],_,_,OutSS-Z) :- true | OutSS=Z.
```

solve(Goal,Path,InS,OutSS) は, 証明経路 Path の端にあるゴール Goal を, 変数置換の入力ストリーム InS のもとで解き, 変数置換ストリームのストリーム OutSS を出力する. OutSS の要素が各々入力ゴールに対する一つの解(変数置換列)である.

ここで呼ばれる unify/4 は既に定まった変数置換列 InS のもとで同一化を行うもので, 次のように定義される.

```
unify(X,Y,InS,Res) :- true |
```

```
    linkS(InS,S1,Succ-Sucm,Abort),
    unify(X,Y,S2,Sucm-succ,Abort),
    merge(S1,S2,S3,Abort),
    merge(S3,OutS2,S4,Abort),
    sift(S4,OutS,OutS2,Abort),
    uResult(Succ,Abort,OutS,Res).

linkS(_,_,_,abort(_)) :- true | true.
linkS([V=T|InS],OutS,Succ-Sucz,Abort) :- true |
  OutS=[(V=T^NewT)*(Succ-Sucm)|OutS1],
  linkS(InS,OutS1,Sucm-Sucz,Abort).
linkS([],OutS,Succ-Sucz,_) :- true | OutS=[], Succ=Sucz.
```

linkS は，定まった変数置換列を再びふるいに投入するために，新たに完了双極子を付加するなどの前処理を行う．

次に，変数置換の適用(applyS)，プログラム節の変数の名前替え(copy)のためのコードを示しておく．

```
applyS('$var'(I,V),S,Y) :- true | applyS1(S,'$var'(I,V),Y).
applyS(X,S,Y) :- X\='$var'(_,_) | functor(X,F,N), rotcnuf(Y,F,N),
  applyS2(N,X,S,Y).

  applyS2(N,X,S,Y) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
    applyS(A,S,B), applyS2(N1,X,S,Y).
  applyS2(0,_,_,_) :- true | true.

  applyS1([V=T|_],V,Y) :- true | Y=T.
  applyS1([U=_|S],V,Y) :- U\=V | applyS1(S,V,Y).
  applyS1([],     V,Y) :- true | Y=V.

copy(var(V),Y,I) :- true | Y='$var'(I,V).
copy(X,Y,I) :- X\=var(_) |
  functor(X,F,N), rotcnuf(Y,F,N),
  copy1(N,X,Y,I).

  copy1(N,X,Y,I) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
    copy(A,B,I), copy1(N1,X,Y,I).
  copy1(0,_,_,_) :- true | true.
```

Prologプログラムは次のように与えられる．

```
clauses(append(_,_,_),Clauses) :- true | Clauses=[
  (append([],var(y),var(y)) :- true),
  (append([var(h)|var(x)],var(y),[var(h)|var(z)]) :-
```

```
    append(var(x),var(y),var(z)))].
clauses(X,Clauses) :- X\=append(_,_,_) | Clauses=[].
```

最後に，実行例を示しておこう．

```
?- ghc bagof(append(var(x),var(y),[1,2,3]),A).

A = [append([],[1,2,3],[1,2,3]),append([1],[2,3],[1,2,3]),
     append([1,2],[3],[1,2,3]),append([1,2,3],[],[1,2,3])]
```

5. GHC インタプリタ

同様にして，GHC のインタプリタも記述できる．ただし，ガード実行部分に上の同一化をそのまま使うわけにはいかないので，幾らか複雑になる．

トップレベルは次のようである．

```
exec(Q,Res) :- true |
  copy(Q,P,?,_),
  solveB(P,top,InS,S1,Succ-succ,Abort),
  merge(S1,S2,InS,Abort),
  sift(InS,OutS,S2,Abort),
  eResult(Succ,Abort,OutS,P,Res).

  eResult(   _,abort(0),_,_,Res) :- true | Res=fail.
  eResult(succ,Abort,OutS,P,Res) :- true |
    Abort=abort(1), applyS(P,OutS,Res).
```

本体側ゴールの実行部分は次のようである．

```
solveB(_,_,_,_,_,abort(_)) :- true | true.
solveB(true,_,_,OutS,Succ-Sucz,_) :- true | OutS=[], Succ=Sucz.
solveB(X=Y,_,_,OutS,Succ_Sucz,Abort) :- true |
  unify(X,Y,OutS,Succ_Sucz,Abort).
solveB((Q1,Q2),Path,InS,OutS,Succ-Sucz,Abort) :- true |
  merge(OutS1,OutS2,OutS,Abort),
  solveB(Q1,1(Path),InS,OutS1,Succ-Sucm,Abort),
  solveB(Q2,r(Path),InS,OutS2,Sucm-Sucz,Abort).
solveB(Q,Path,InS,OutS,Succ_Sucz,Abort) :-
    Q\=true, Q\=(_=_), Q\=(_,_) |
  clauses(Q,Clauses),
  tryClauses(Clauses,0,Q,Path,InS,SuccG,AbortG,Abort),
  commit(SuccG,InS,OutS,AbortG,Succ_Sucz,Abort).
```

ここで，簡単のため本体側で呼び出される組み込み述語は同一化だけとする．

本体側ゴールの実行は，単一解を求めていること，full unification でよいことから，あたかも全体で一つの同一化を試みているものと見なせる．このとき solveB は，変数置換の生成部と考えられる．(unify/5 に相当する役割である．) solveB の完了双極子と打ち切り端子は (同一化を含む) 本体側ゴール全体の成否に関わっているものである．

次に，プログラム節の選択部分を詳細化する．

```
tryClauses(_,_,_,_,_,_,AbortG,abort(_)) :- true | AbortG=abort(0).
tryClauses(_,_,_,_,_,_,abort(_),_) :- true | true.
tryClauses([Cl|Clauses],N,P,Path,InS,SuccG,AbortG,AbortB) :-
    true | N1:=N+1,
  copy(Cl,(H:-G|B),c(N1,Path),AbortB),
  selectSignal(SuccG1,SuccG2,SuccG,AbortG),
  tryClause(P,(H:-G|B),c(N1,Path),InS,SuccG1,AbortG,AbortB),
  tryClauses(Clauses,N1,P,Path,InS,SuccG2,AbortG,AbortB).
tryClauses([],_,_,_,_,_,_,_) :- true | true.

  selectSignal(_,_,_,abort(_)) :- true | true.
  selectSignal(S,_,OutS,_) :- wait(S) | OutS=S.
  selectSignal(_,S,OutS,_) :- wait(S) | OutS=S.
```

tryclauses は，プログラム節のそれぞれについて tryclause を起動する．

tryclause の成功は SuccG 端子を通じて commit 演算子に通告される．selectSignal は，複数のプログラム節でこの SuccG 端子を占有するための勝ち抜きを行うものである．

```
tryClause(_,_,_,_,_,_,AbortG,abort(_)) :- true | AbortG=abort(0).
tryClause(_,_,_,_,_,_,abort(_),_) :- true | true.
tryClause(P,(H:-G|B),NewPath,InS,SuccG,AbortG,AbortB) :- true |
  unify(P,H,S1,Succ-Sucn,Abort1),
  solveG(G,S0,S2,Sucn-succ,Abort1),
  merge(InS,S4,S0,Abort1),
  merge(S1,S2,S3,Abort1),
  merge(S3,S4,S5,Abort1),
  siftG(S5,OutS,S4,Abort1),
  gResult(Succ,Abort1,SuccG,succG(OutS,B,NewPath),AbortG,AbortB).

  gResult(_,Abort1,_,_,_,abort(_)) :- true | Abort1=abort(0).
  gResult(_,Abort1,_,_,abort(_),_) :- true | Abort1=abort(1).
  gResult(_,abort(_),_,_,_,_) :- true | true.
  gResult(succ,Abort1,SuccG,S,_,_) :- true | Abort1=abort(1),
    SuccG=S.
```

次に，コミット演算子の詳細を示す．

```
commit(_,_,_,AbortG,_,abort(_)) :- true | AbortG=abort(0).
commit(_,_,_,abort(_),_,_) :- true | true.
commit(succG(S,B,NewPath),InS,OutS,AbortG,Succ_Sucz,AbortB) :-
    true |
  AbortG=abort(0),
  applyS(B,S,BS),
  bodyvar(BS,NewB),
  solveB(NewB,NewPath,InS,OutS,Succ_Sucz,AbortB).
```

ガード側ゴールの実行部分は次のようである.

```
solveG(_,_,_,_,abort(_)) :- true | true.
solveG(true,_,OutS,Succ-Sucz,_) :- true |
  OutS=[], Succ=Sucz.
solveG((G1,G2),InS,OutS,Succ-Sucz,Abort) :- true |
  merge(OutS1,OutS2,OutS,Abort),
  solveG(G1,InS,OutS1,Succ-Sucm,Abort),
  solveG(G2,InS,OutS2,Sucm-Sucz,Abort).
solveG(X=Y,_,OutS,Succ-Sucz,Abort) :- true |
  unify(X,Y,OutS,Succ-Sucz,Abort).
```

ここで，簡単のためガード側で呼び出される組み込み述語は同一化だけとする.

solveG の完了双極子と打ち切り端子は本体側 solveB のそれとは無関係であることに注意.

ガード側同一化における変更は，ゴール側変数とプログラム節側変数の同一化の際，変数置換の方向性に注意することである．即ち，この場合常にプログラム節側の変数をゴール側変数で置き換えるような変数置換を生成しなければならない.

```
unify(_,_,_,_,abort(_)) :- true | true.
unify(X,X,OutS,Succ-Sucz,_) :- true | OutS=[], Succ=Sucz.
unify('$VAR'(I,V),T,OutS,Succ-Sucz,_) :- T\='$var'(_,_) |
  OutS=[('$VAR'(I,V)=T^NewT)*(Succ-Sucz)].
unify(T,'$VAR'(I,V),OutS,Succ-Sucz,_) :- T\='$var'(_,_) |
  OutS=[('$VAR'(I,V)=T^NewT)*(Succ-Sucz)].
unify('$var'(I,V),T,OutS,Succ-Sucz,_) :- true |
  OutS=[('$var'(I,V)=T^NewT)*(Succ-Sucz)].
unify(T,'$var'(I,V),OutS,Succ-Sucz,_) :- true |
  OutS=[('$var'(I,V)=T^NewT)*(Succ-Sucz)].
unify(X,Y,OutS,Succ_Sucz,Abort) :-
    X\=Y, X\='$var'(_,_), Y\='$var'(_,_),
    X\='$VAR'(_,_), Y\='$VAR'(_,_) |
  functor(X,F,N), functor(Y,G,M),
```

```
unify2(F/N,G/M,X=Y,OutS,Succ_Sucz,Abort).
```

'$VAR'(I,V) はゴール側(本体側)変数, '$var'(I,V) はガード側変数を表す.

変数置換に対するふるいも, ガード側においては特別の注意が必要になる. 即ち, ガードにおける同一化の結果の変数置換に, ゴール側の変数を置き換えるようなものがあってはならないので, これに対処すべく変更を要する.

```
siftG(_,_,_,abort(0)) :- true | true.
siftG(_,OutS,_,abort(1)) :- true | OutS=[].
siftG([('$VAR'(I,V)=T^NewT)*(Succ-Sucz)|InS],
      OutS,OutS2,Abort) :- true |
  merge(OutS21,OutS22,OutS2,Abort),
  filterGV(InS,'$VAR'(I,V),T,NewT,InS1,OutS21,Succ-Sucz,Abort),
  siftG(InS1,OutS,OutS22,Abort).
siftG([(V=T^NewT)*(Succ-Sucz)|InS],
      OutS,OutS2,Abort) :- V='$var'(_,_) |
  Succ=Sucz,
  OutS=[V=NewT|OutS1],
  merge(OutS21,OutS22,OutS2,Abort),
  filterG(InS,V,T,NewT,InS1,OutS21,Abort),
  siftG(InS1,OutS1,OutS22,Abort).
siftG([],OutS,OutS2,_) :- true | OutS=[], OutS2=[].
```

ゴール側変数 '$VAR'(I,V) の変数置換は siftG の出口 OutS に流してはいけない. 生成されるフィルタも ゴール側変数専用のタイプ filterGV とする.

```
filterG(_,_,_,_,_,_,abort(0)) :- true | true.
filterG(_,_,T,NewT,_,_,abort(1)) :- true | T=NewT.
filterG([('$VAR'(I,V)=TT^NewTT)*(Succ-Sucz)|InS],
        V,T,NewT,OutS,OutS2,Abort) :- true |
  merge(OutS21,OutS22,OutS2,Abort),
  filterGV(InS,'$VAR'(I,V),TT,NewTT,InS1,OutS21,Succ-Sucz,Abort),
  filterG(InS1,V,T,NewT,OutS,OutS22,Abort).
filterG([(V=TT^_)*Succ_Sucz|InS],V,T,NewT,OutS,OutS2,Abort) :-
    true |
  merge(OutS21,OutS22,OutS2,Abort),
  unify(TT,T,OutS21,Succ_Sucz,Abort),
  filterG(InS,V,T,NewT,OutS,OutS22,Abort).
filterG([(VV=TT^NewTT)*Succ_Sucz|InS],
        V,T,NewT,OutS,OutS2,Abort) :- VV\=V, VV='$var'(_,_) |
  deref(TT,V=NewT,NewTT1,Abort),
  deref(T,VV=NewTT,NewT1,Abort),
  OutS=[(VV=NewTT1^NewTT)*Succ_Sucz|OutS1],
  filterG(InS,V,NewT1,NewT,OutS1,OutS2,Abort).
```

```
filterG([],_,T,NewT,OutS,OutS2,_) :- true |
  T=NewT, OutS=[], OutS2=[].
```

filterG も siftG と同じくゴール側変数 '$VAR'(I,V) の変数置換に対し，ゴール側変数専用タイプの filterGV を生成する.

その filterGV は次のようである.

```
filterGV(_,_,_,_,_,_,_,abort(0)) :- true | true.
filterGV(_,_,T,NewT,_,_,_,abort(1)) :- true | T=NewT.
filterGV([(V=TT^_)*(Succ-Sucm)|InS],
         V,T,NewT,OutS,OutS2,Sucn-Sucz,Abort) :- true |
  Sucm=Sucn,
  unify(TT,T,OutS2,Succ-Sucz,Abort),
  OutS=InS.
filterGV([(VV=TT^NewTT)*Su_Sz|InS],
         V,T,NewT,OutS,OutS2,Succ_Sucz,Abort) :- VV\=V |
  deref(TT,V=NewT,NewTT1,Abort),
  deref(T,VV=NewTT,NewT1,Abort),
  OutS=[(VV=NewTT1^NewTT)*Su_Sz|OutS1],
  filterGV(InS,V,NewT1,NewT,OutS1,OutS2,Succ_Sucz,Abort).
filterGV([],_,T,NewT,OutS,OutS2,_,_) :- true |
  T=NewT, OutS=[], OutS2=[].
```

この他のコードについては，Prologインタプリタと同名のものはそれを用い，残りは付録に示す.

ここで，プログラム例をあげておく.

```
clauses(append(_,_,_),Clauses) :- true | Clauses=[
  (append([],var(y),var(z)) :- true | var(z)=var(y)),
  (append([var(h)|var(x)],var(y),var(z)) :- true |
     var(z)=[var(h)|var(z1)],
     append(var(x),var(y),var(z1)))].
clauses(X,Clauses) :- X\=append(_,_,_) | Clauses=[].
```

最後に実行例を示す.

```
?- ghc exec(append([1,2],[3],var('Y')),A).

A = append([1,2],[3],[1,2,3])
```

6. おわりに

GHCでのメタプログラミングでは，オブジェクト変数の表現とこれに伴う同一化の実装が最初の問題となった.

Prologでは，メタプログラムの実行効率は部分計算によって十分な程度に改善された．

GHCでのメタプログラミングは，変数の扱いの分だけ記述量が増加することを除けば，Prologと同様にして実現できるであろう．しかし，その実行効率についてもPrologの場合と同様に，部分計算によって十分な程度に改善され得るであろうか．オブジェクト変数を基底項で表現し，同一化を陽に実装したことが，効率改善の障害とならないであろうか．

この問題は，GHCの部分計算法に関する課題として検討中である．

並列同一化手続きの実現については，ここで示した方法の他に，変数表を集中管理するプロセスを用いるものが実現されている．両者のパフォーマンス比較は興味ある問題である．

Prologインタプリタについては，バックトラックを行うものを要求駆動型のコーディングを用いて実現できるであろう．また，全解探索の方法については，レイヤードストリーム方式を応用した版も検討中である．

ここで示したGHCインタプリタは，オブジェクトプログラムがデッドロックするようなとき，インタプリタ自体がデッドロックする．このデッドロックの検出と回避の問題を解決するには，もうひと工夫要する．しかし，基本的にはここで扱った技法で対処可能であろう．

さて，ここで扱ったメタインタプリタはいわゆるかおり(flavor)をもたない．即ち，証明木の生成，reductionステップの計数といった実質的なメタ作業をユーザに提供するような形で行ってはいない．スタンダードな処理系で得られる解を模倣的に求めるだけである．

かおり付きメタインタプリタの実現，reflection機能の装備，等々，個々の応用において，ここであげたものが参考になることを期待している．

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第6巻，共立出版，1987

付録1．GHCインタプリタコード捕足

```
applyS('$VAR'(I,V),S,Y) :- true | applyS1(S,'$VAR'(I,V),Y).
applyS('$var'(I,V),S,Y) :- true | applyS1(S,'$var'(I,V),Y).
applyS(X,S,Y) :- X\='$var'(_,_), X\='$VAR'(_,_) |
```

```
    functor(X,F,N), rotcnuf(Y,F,N),
  applyS2(N,X,S,Y).

  applyS2(N,X,S,Y) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
    applyS(A,S,B), applyS2(N1,X,S,Y).
  applyS2(0,_,_,_) :- true | true.

  applyS1([V=T|_],V,Y) :- true | Y=T.
  applyS1([U=_|S],V,Y) :- U\=V | applyS1(S,V,Y).
  applyS1([],     V,Y) :- true | Y=V.

copy(_,_,_,abort(_)) :- true | true.
copy(var(V),Y,?,_) :- true | Y='$VAR'(?,V).
copy(var(V),Y,I,_) :- I\= ? | Y='$var'(I,V).
copy(X,Y,I,Abort) :- X\=var(_) | functor(X,F,N), rotcnuf(Y,F,N),
  copy1(N,X,Y,I,Abort).

  copy1(_,_,_,_,abort(_)) :- true | true.
  copy1(N,X,Y,I,Abort) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
    copy(A,B,I,Abort), copy1(N1,X,Y,I,Abort).
  copy1(0,_,_,_,_) :- true | true.

bodyvar('$VAR'(I,V),Y) :- true | Y='$VAR'(I,V).
bodyvar('$var'(I,V),Y) :- true | Y='$VAR'(I,V).
bodyvar(X,Y) :- X\='$var'(_,_), X\='$VAR'(_,_) |
  functor(X,F,N), rotcnuf(Y,F,N), bodyvar1(N,X,Y).

  bodyvar1(N,X,Y) :- N>0 | N1:=N-1, arg(N,X,A), arg(N,Y,B),
    bodyvar(A,B), bodyvar1(N1,X,Y).
  bodyvar1(0,_,_) :- true | true.

deref(_,_,_,abort(0)) :- true | true.
deref('$VAR'(I,V),'$VAR'(I,V)=S,NewT,_) :- true | NewT=S.
deref('$var'(I,V),'$var'(I,V)=S,NewT,_) :- true | NewT=S.
deref('$VAR'(I,V),'$VAR'(J,W)=S,NewT,_) :- (I,V)\=(J,W) |
  NewT='$VAR'(I,V).
deref('$VAR'(I,V),'$var'(_,_)=S,NewT,_) :- true | NewT='$VAR'(I,V).
deref('$var'(I,V),'$VAR'(_,_)=S,NewT,_) :- true | NewT='$var'(I,V).
deref('$var'(I,V),'$var'(J,W)=S,NewT,_) :- (I,V)\=(J,W) |
  NewT='$var'(I,V).
deref(T,U_S,NewT,Abort) :- T\='$var'(_,_), T\='$VAR'(_,_) |
  functor(T,F,N), rotcnuf(NewT,F,N),
  deref1(N,T,U_S,NewT,Abort).
```

```
deref1(_,_,_,_,abort(0)) :- true | true.
deref1(N,T,U_S,NewT,Abort) :- N>0 |
  N1:=N-1, arg(N,T,A), arg(N,NewT,B),
  deref(A,U_S,B,Abort), deref1(N1,T,U_S,NewT,Abort).
deref1(0,_,_,_,_) :- true | true.
```