プログラミング言語 17 － 3
（１９８８. 7. １５）

ガード部を持つFGHC節から ガード部を持たない節へ のプログラム変換

Martin Nilsson and Hidehiko Tanaka
東京大学電機工学科田中英彦研究室

あらまし　ガード部を持つFGHC節からガード部を持たない節へのプログラム変換について検討する。ターゲットプログラムは、FGHCと似たような、単純なCommitted-choice並列論理型言語である。この変換方式では、ターゲット言語は特に実装しやすいので、ここに述べた変換方法を用いれば、FGHCの強力な実装方法も得ることができる。変換方法も、変換ターゲット言語で表現する。

Converting FGHC Clauses with Guards into Clauses without Guards

Martin Nilsson and Hidehiko Tanaka
Hidehiko Tanaka Lab., Department of Electrical Engineering,
University of Tokyo, Hongo 7-3-1, Bunkyo-ku, 113 Tokyo

Abstract We will describe how to transform FGHC clauses with guard goals into clauses without guard goals. The target program is in a simple, parallel committed-choice logic language, similar to FGHC. Since this target language is especially easy to implement, we obtain a powerful method for implementing FGHC. The translation method is described in the target language itself.

1 始めに

この論文では、ガード部を持つFGHC[14]節からガード部を持たない節へのプログラム変換について検討する。ターゲットプログラムは、FGHCではなくて、Fleng [7],[8]というCommitted-choice並列論理型言語である。 Flengは、GHCとよく似ているが、ガードゴールはなく、ボディーゴールは、共有変数通信以外、完全に独立に実行される。すなわち、ボディーゴールは別々にフォークされ、FGHCのようなAND-関係なしに実行される。この方法の利点は、Flengは特に実装しやすい言語であるので、FGHCからFlengへの変換ができると、FGHCの実装方法にもなることである。この方法を用いて、筆者は色々なSIMDアーキテクチャのFGHC処理を検討した [9],[11],[10]。ベクトル並列スーパーコンピュータの上にGHCプログラムを 1.1 MHzの推論周波数で実行する処理系を乗せた。これは、Flengほど単純な言語を用いなければ、とても難しい。

ここで説明するアルゴリズムは、最適ではないが、説明のために簡単にしたアルゴリズムである。アルゴリズムをTop-down的に説明しながら、アルゴリズムそのものをFlengプログラムとして表現してゆく。

パターンとして、次のプログラムを見よう。

```
p(X) :- q1(X) | r1(X).
p(X) :- q2(X) | r2(X).
```

このプログラムは、次のように変換される：

```
p(X) :- guard(1,p(X),N), guard(2,p(X),N), body(N,p(X)).
guard(1,p(X),N) :- q1(X,N).
guard(2,p(X),N) :- q2(X,N).
body(1,p(X)) :- q1(X,N), r1(X).
body(2,p(X)) :- q2(X,N), r2(X).
```

この新しいプログラムに、相互排除変数Nを入れた。q1は、Nを1に代入しようとして、q2は、Nを2に代入しようとするが、そのうち先に成功するガードがボディー部候補を決定する。このプログラムを見ると、元のガードゴールq1(X)をq1(X,N)に、そしてガードゴールq2(X)をq2(X,N)に変換する必要がある。q1とq2が変数代入を輸出しないことの保証が変換の鍵である。

q1は、guardにもbodyにも必要な時がある: 例えば、

```
p(X,Z) :- Y=17, X>Y | Y=Z.
```

では、Yはガード部でバインドされるが、ボディー部で参照される。

普通は、もっと効率的に変換できるはずであるが、このような方法は、比較的簡単であり、また一般的でもある。

例として、次のプログラムを見よう。

```
test(X,Y) :- X \= 0 | Y = not_zero
test(0,Y) :- true | Y = zero.
```

これは上で調べたように、次のプログラムに変換される：

```
test(X,Y) :- guard(1,test(X,Y),N),
             guard(2,test(X,Y),N), body(N,test(X,Y)).
guard(1,test(X,Y),N) :- '\='(X,Y,R), commit([R],1,N).
guard(2,test(0,Y),N) :- true,        commit([],2,N).
body(1,test(X,Y)) :- '\='(X,Y,R), Y = not_zero.
body(2,test(0,Y)) :- true,        Y = zero.
```

ここで、\= という述語は引き数を比較して、その結果を変数Rで報告する。 commit という述語がこのような結果を収集して、もし全部が true であれば、二番目と三番目の引き数（相互排除変数）を単一化する：

```
commit([],M,N) :- M = N.
commit([true|Rest],M,N) :- commit(Rest,M,N).
```

Flengの単一化では、失敗が広まらない: 一つのゴールが失敗しても、他ゴールに影響はない。そのために、安全に同時に複数の commit ゴールを実行することができる。

\=は、次のように定義されている[3]：

```
'\='(X,Y,R) :- equal(X,Y,S), not(S,R).

equal(X,Y,R) :- equal1(X,X,Y,Y,R).
equal1(X,X,Y,Y,R) :- unify(R,X,Y).

not(false,R) :- R = true.
not(true,R) :- R = false.
```

このように、引数を二回入れることにより、その引数がグラウンド（定数）であることを確認することができる。すなわち、ある節の頭部が「a(X,X) :- ...」であると、「?- a(X,X)」のような呼び出しはサスペンドする。それは、頭部単一化の間、同じ変数でも、外の変数を代入してはいけないからである。逆にいえば、これもグラウンドを確認する強力なプログラミング技術として使用できる。FGHCは、サスペンドするかしないかは処理系によって違う[3], [14]が、我々の経験として、しない方が強力であろう。

3引数unify(R,X,Y)は、XとYを単一化して見て、単一化が終了すると、その結果(true又はfalse)の否定をとってRに代入する。

変換したプログラムは、かなり最適化が可能である。例えば、上のtestは、次のプログラムとも等しい：

```
test(X,Y) :- equal(X,0,R), test1(R,Y).
test1(false,Y) :- Y = not_zero.
test1(true,Y) :- Y = zero.
```

2 変換主ルーチンと頭部変数の保護と解放

変換主ルーチンは、translateである。呼び出しは、

```
translate(Head,[Clause1,Clause2,...,ClauseN],Newclauses,Ready)
```

の形をとる。Headは、変換したい述語の、全ての引数が相異なる変数であるようなリテラルで、Clause1, Clause2, ..., ClauseNはその述語の定義節である。簡単にするために、ここでも、Head, Clause1, Clause2, ..., ClauseNは共

有変数を持たないと仮定する。変換結果は、Newclausesに出力される。Readyという信号変数がtrueになると、計算が終わったことがわかる。

例えば、前述のtestを変換しようとするとき、

```
?- translate(test(X,Y),
             [(test(X1,Y1) :- X1 \= 0 | Y1 = not_zero),
              (test(0,Y2) :- true | Y2 = zero)],
             Newclauses,Ready).
```

のように呼び出す。

この変換プログラムでは、"isvar"のような変数を扱うメタ述語を用いるので、その変数との単一化が終了したかどうかのハンドシェイク信号が必要であり、それがReadyである。

```
translate(Pred,Clauses,Newclauses,Ready) :-
      Newclauses = [(Pred :- Goals)|Otherclauses],
      trans1(Pred,0,N,Clauses,Otherclauses,Goals,Ready).

trans1(Pred,Count,N,[],Newclauses,Goals,Ready) :-
      Newclauses = [], unify(Ready,Goals,body(N,Pred)).
trans1(Pred,Count,N,[(Head:-Guard|Body)|Clauses],Newclauses,Goals,Ready) :-
      Goals = (guard(Count,Pred,N),Othergoals),
      protect(Head,ProtectReady),
      convert_whenready(ProtectReady,Guard,Results,Newguard,ConvertReady),
      unprotect(ConvertReady,[Head,Newguard,Body],
                             [Head1,Newguard1,Body1],UnprotectReady),
      Newclauses = [
        (guard(Count,Head1,N) :- Newguard1, commit(Results,Count,N)),
        (body(Count,Head1) :- Newguard1, Body1)|
        Othernewclauses],
      compute('+',1,Count,Newcount),
      and(UnprotectReady,NextReady,Ready),
      trans1(Pred,Newcount,N,Clauses,Othernewclauses,Othergoals,NextReady).
```

trans1は変換の主ループで、並列に各節を変換する。最初に頭部にある変数を保護して、次にガード部を変換し、最後に変数を全て解放する。このシーケンスは大切であるので、ProtectReadyとConvertReadyという同期信号を用いている。compute('+',X,Y,Z)はFlengの加算をする組込み述語であり、X+YをZと単一化する。これを使って、節を数えている。andは、終了信号をまとめる"rendez-vous"述語である：

```
and(true,true,R) :- R = true.
```

protectは変数を'$prot'(_)とバインドすることによって変数を保護する。変換のあとでは、unprotectでまた'$prot'を剥く。

```
protect(X,Ready) :- typeof(X,Type), protect(Type,X,Ready).
protect(prot,_,Ready) :- Ready = true.
protect(const,X,Ready) :- Ready = true.
protect(var,X,Ready) :- unify(Ready,X,'$prot'(_)).
protect(list,[X|Y],Ready) :-
      protect(X,Ready1), protect(Y,Ready2), and(Ready1,Ready2,Ready).
```

Flengでは、リストとレコードのデータ型を区別しないので、最後の節はその二つの場合を両方カバーする。データ型を得るのに、compute(sametype,X,Y,Z) という組込み述語を用いる。これは、XとYのデータ型が等しければ、Zをtrueにバインドし、さもなければ、falseにバインドする。

```
typeof(X,Xtype) :-
      compute(sametype,X,_,IsVar),
      compute(sametype,X,a,IsConst),
      compute(sametype,X,0,IsConst),
      compute(sametype,X,[_|_],IsList),
      typeis(IsVar,IsConst,IsList,X,Xtype).
typeis(true,_,_,_,Type) :- Type = var.
typeis(_,true,_,_,Type) :- Type = const.
typeis(_,_,true,[Functor|_],Type) :-
      compute('=',Functor,'$prot',IsProt), typeis1(IsProt,Type).
typeis1(true,Type) :- Type = prot.
typeis1(false,Type) :- Type = list.

unprotect(true,X,Y,Ready) :- typeof(X,Type), unprotect(Type,X,Y,Ready).
unprotect(prot,'$prot'(X),Y,Ready) :- unify(Ready,X,Y).
unprotect(const,X,Y,Ready) :- unify(Ready,X,Y).
unprotect(var,X,Y,Ready) :- unify(Ready,X,Y).
unprotect(list,[X|Y],Z,Ready) :-
      unprotect(true,X,A,Ready1), unprotect(true,Y,B,Ready2),
      unify(Ready3,Z,[A|B]), and(Ready1,Ready2,Ready3,Ready).
```

3 ガード部の変換

convert述語によって、ガード部のゴールを次々に変換する。変換が終了すると、それを最後の引き数で報告する。各FGHCガード組込み述語のために、convert節が一つずつある。

convertのResという引き数は、変換したガードゴールの結果変数の列を収集している。この列は、後でcommitの引き数になる。Newは新しいゴールの列である。

```
convert_whenready(true,Guard,Result,New,ConvertReady) :-
```

```
      convert(Guard,Result,New,ConvertReady).

convert([],Res,New,ConvertReady) :- Res=[], New=true, ConvertReady=true.
convert([wait(X)|Guard],Res,New,Ready) :-
      Res = [R|Res1], New = (wait(X,X,R),New1),
      convert(Guard,Res1,New1,Ready).
```

FGHC の組込み述語 wait は、変数ではない引き数を待っている。Fleng で、次のように定義すると、

```
wait([_|_],_,R) :- R = true.
wait(X,X,R) :- R = true.
```

wait は、X が複合データ構造かグラウンドデータ構造になるのを待つ。\=と:=以外の FGHC 組込み述語は全て次のような節で処理できる：

```
convert([X<Y|Guard],Res,New,Ready) :-
      Res = [R|Res1], New = ('<'(X,Y,R),New1),
      convert(Guard,Res1,New1,Ready).

'<'(X,Y,R) :- evaluate(X,A), evaluate(Y,B), compute('<',A,B,R).
```

>,=<,>=,=:=,=\=,\=についてもほとんど同様であるので、ここではその定義を省略する。evaluate は、一つ目の引き数がグラウンドかどうかを確認して、その表現を評価する。

```
evaluate(X,Y) :- checkexpr(X,X,IsExpr), eval(IsExpr,X,Y).
checkexpr(X,X,IsExpr) :- unify(IsExpr,X,[_|_]).
eval(true,X+Y,Z) :- evaluate(X,A), evaluate(Y,B), compute('+',A,B,Z).
eval(true,-X,Z)  :- evaluate(X,A), compute('-',0,A,Z).
eval(true,X-Y,Z) :- evaluate(X,A), evaluate(Y,B), compute('-',A,B,Z).
eval(true,X*Y,Z) :- evaluate(X,A), evaluate(Y,B), compute('*',A,B,Z).
eval(true,X/Y,Z) :- evaluate(X,A), evaluate(Y,B), compute('/',A,B,Z).
eval(false,X,Z) :- Z = X.
```

今までのガード部組込み述語は、変数の代入を輸出しないので、変換が単純であった。しかし、ガード部にある単一化ゴールは、変換がもう少し複雑である。

4 単一化ガードゴールの変換

ガード部にある単一化ゴールが、:=という述語の場合、その右側の表現は評価済みであるので、普通の単一化として扱える。

```
convert([X:=Y|Guard],Res,New,Ready) :-
      New = (evaluate(Y,A),New1),
      convert([X='$prot'(A)|Guard],Res,New1,Ready).
```

実は、この:=もFGHCのボディー部の組込み述語である。そのために、次のように定義する：

```
X:=Y :- evaluate(Y,Z), X=Z.
```

また、単一化述語("=")の変換は、まずその単一化を部分計算して、その後で対応するFlengゴールを生成することにする。

```
convert([X=Y|Guard],Res,New,Ready) :-
      typeof(X,Xtype), typeof(Y,Ytype),
      partevunify(Xtype,Ytype,X,Y,Guard,Res,New,Ready).
```

簡単にするために、ガード部がfalseにならないと仮定すれば、二つの定数の単一化を省略しても良い：

```
partevunify(const,const,_,_,Guard,Res,New,Ready) :-
      convert(Guard,Res,New,Ready).
```

複合データ構造であれば、二つの単一化に分解する：

```
partevunify(list,list,[X|Y],[U|V],Guard,Res,New,Ready) :-
      convert([X=U,Y=V|Guard],Res,New,Ready).
```

単一化の引き数の一つが保護されていない変数であれば、すぐそれを単一化する。そのとき、単一化の終了を待たなければならない。そうしないと、他の単一化も同じ変数をバインドしようとする危険がある。

```
partevunify(var,_,X,Y,Guard,Res,New,Ready) :-
      unify(UnifyReady,X,Y),
      convert_whenready(UnifyReady,Guard,Res,New,Ready).
partevunify(_,var,X,Y,Guard,Res,New,Ready) :-
      unify(UnifyReady,X,Y),
      convert_whenready(UnifyReady,Guard,Res,New,Ready).
```

もし引数が両方保護されているか、一つが保護で、一つが定数であれば、その変数に代入してはならない。従って、unify は使用せず、前述の equal を使用する：

```
partevunify(prot,prot,X,Y,Guard,Res,New,Ready) :-
      Res = [R|Res1], New = (equal(X,Y,R),New1),
      convert(Guard,Res1,New1,Ready).
partevunify(prot,const,X,Y,Guard,Res,New,Ready) :-
      partevunify(prot,prot,X,Y,Guard,Res,New,Ready).
partevunify(const,prot,X,Y,Guard,Res,New,Ready) :-
      partevunify(prot,prot,X,Y,Guard,Res,New,Ready).
```

保護された変数と複合データ構造の単一化のために使う、 iscons 述語を紹介する：

```
iscons([U|V],A,B,R) :- A = U, B = V, R = true.
```

この述語を先程の equal のように用いれば、残りの単一化部分計算を次のように処理できる：

```
partevunify(list,prot,[A|B],Y,Guard,Res,New,Ready) :-
      Res = [R|Res1], New = (iscons(Y,U,V,R),New1),
      convert([A='$prot'(U),B='$prot'(V)|Guard],Res1,New1,Ready).
partevunify(prot,list,X,Y,Guard,Res,New,Ready) :-
      partevunify(list,prot,Y,X,Guard,Res,New,Ready).
```

5 関係する研究、討論と結論

枚田[5],[6] は、プログラム変換などのために、ガードゴールを持たない言語 Oc 開発した。しかし、 Oc をメタコール述語で Full GHC と同じ位強力になっている。その上、単一化はアトミクになっているので、実装は Full GHC より難しい。 Clark と Gregory[1],[4] は、 Parlog から Kernel Parlog へのコンパイレーションについて調べた。しかし、 Kernel Parlog はまだガードゴールもゴールの AND-関係を持っているので、 Kernel Parlog は FGHC の方に近いであろう。 Sterling と Codish と Shapiro[13],[2] は Concurrent Prolog から Flat Concurrent Prolog への変換を調べた。 Flat Concurrent Prolog も、 FGHC の方に近い。 Concurrent Prolog 系の言語もアトミク単一化の言語である。

FGHC からガードゴールを持たない言語 Fleng への変換について説明した。この方法で FGHC の簡単な実装方法を得た。最適なターゲットプログラムよりも、理解しやすいように調べようとした。論理変数をデータ構造として処理できることは、保護機構 protect/unprotect などのために大変役に立った。

最適化する変換プログラムは、上で調べたプログラムとそれほど違わない：ケース数を大きくすれば（特に空のガードと相互排除的なガードのため）、元の FGHC プログラムと近いゴール数になる。

6 謝辞

東大の並列推論エンジングループSIGIEとICOTの並列プログラミングWGの教員、特に、ICOTの上田さんと枚田さんと東大の小池さんに感謝致します。

参考文献

[1] Clark, K.L. and Gregory, S.: *Parlog: Parallel Programming in Logic.* ACM Trans. on Programming Languages,Vol. 8, No. 1, 1986.

[2] Codish,M. and Shapiro,E.: *Compiling OR-parallelism into AND-parallelism.* In Shapiro, E. (ed): Proc. 3rd Int. Conf. on Logic Programming, London. July 1986. p 593-599.

[3] Furukawa,K. and Mizoguchi,F. (Eds.): The Parallel Programming Language GHC and its Applications. Kyoritsu publishing Co. Tokyo, 1987. (In Japanese).

[4] Gregory, S.: *Parallel Logic Programming in Parlog* Addison-Wesley, 1987.

[5] Hirata, M.: *Self-description of Oc and its Applications.* In Proc. Second National Conf. of Japan Society of Software Science and Technology, 1985. p 153-156. (In Japanese)

[6] Hirata, M.: *Self-description of the Parallel Programming Language Oc.* Computer Software, No. 3, Vol. 4. September 1987. p 41-64. (In Japanese)

[7] Nilsson, M. and Tanaka, H.: *Fleng Prolog - The Language which turns Supercomputers into Prolog Machines.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, 1986. p 209-216. Proceedings also published as Springer LNCS 264.

[8] Nilsson, M. and Tanaka, H.: *The Art of Building a Parallel Logic Programming System.* In Wada,E. (Ed.): Proc. Japanese Logic Programming Conference. ICOT, Tokyo, June, 1987. p 155-163. Proceedings also to appear as Springer LNCS.

[9] Nilsson, M. and Tanaka, H.: *A Proposal for implementing GHC on the Connection Machine.* In Proc. IEEE Region 10 Conf. p 821-825. Seoul, August, 1987.

[10] Nilsson, M. and Tanaka, H.: *SIMD Architecture and Superparallel Logic programming.* In Information Processing Soc. of Japan Workshop on Computer Systems, July 1988.

[11] Nilsson, M. and Tanaka, H.: *A Flat GHC Implementation for Supercomputers.* To appear in Proc. Int. Conf. Symp. Logic Programming, Seattle, August 1988.

[12] Shapiro, E.: *A Subset of Concurrent Prolog and its Interpreter.* ICOT Technical Report TR-003, February 1983.

[13] Sterling, L. and Codish, M.: *Pressing for Parallelism: A Prolog Program Made Concurrent.* J. Logic Programming, No. 1, 1986. p 75-92.

[14] Ueda, K.: *Guarded Horn Clauses.* D.Eng. Thesis, Information Engineering course, University of Tokyo, Japan. March 1986.