記号処理 31－1
（1985. 3. 12）

H-Prolog システムにおける コンパイラの方式について

中谷吉久，斉藤　剛，平松啓二（電機大・工）
中村克彦（電機大・理工）

1. はじめに

Prologの処理系は，DEC-10 Prolog[1]をはじめとして種々開発されており，現在われわれの研究室においても，データベースと作業用データのためにハッシュ記憶を用いて高効率化を図った H-Prolog システム[2]が稼働している．

Prologは，実行が非決定性であること，および基本演算が単一化であること等，従来の手続き型言語と非常に異なるため，高速化やメモリ効率向上のための技法に関する研究が多数報告されている．しかし，移植性に優れたコンパイラについての報告は少ない．そこで，現在われわれは，移植性の高いシステムによる高速処理をめざした Prologコンパイラの方式について検討し，その処理系の開発・改良を進めている[3]．

われわれが開発した Prologコンパイラの特長としては，次の三点を挙げることができる．

(1) コンパイラの大部分を Prologで記述し，その目的言語をC言語としているので，これにより設計・開発・改良が容易となるばかりでなく，機械に依存しない移植性の高いものとなっている．
(2) 処理速度を上げるために，Mellishの方式[4]に類似したNSS(Non Structure Sharing, コピー）方式を採用している．また，モノコピーリストを用いてコピーコストの軽減を図っている．
(3) trail スタックは用いずに，二つのスタック（local スタックと global スタック）のみで変数の環境や後戻り制御等を実現しているので，さらに実行時のメモリ効率が高まっている．

本報告では，その Prologコンパイラで採用したデータ構造と制御方式について述べる．

2. データ構造

本コンパイラでは，データの表現にリスト方式を，そして合成項（instance）の表現にMellishの方式に類似したNSS方式，すなわちコピー方式を採用している．

このリスト方式には，次のような利点がある．

(1) システムの記述が簡潔になる．
(2) モノコピーリストが利用でき，コピーコストが小さくなる．

また，コピー方式では，合成項の参照が効率よく行なえるが，しかしコピーコストをいかに低くおさえるかが重要な問題となる．本コンパイラでは，このコピ

ーコストの軽減するために，ハッシュ記憶とリスト方式を用いることによりモノコピーリストを実現している．

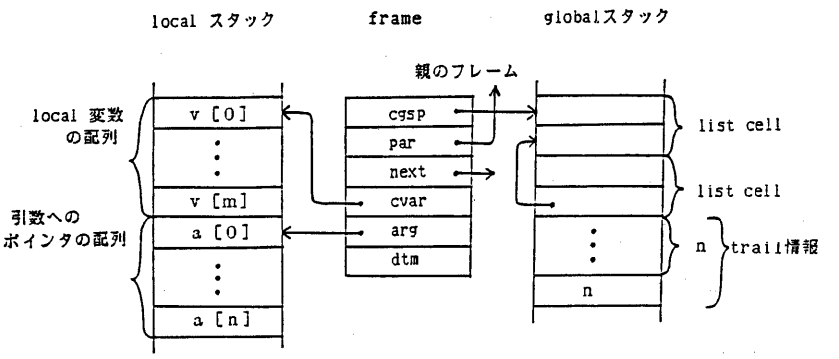

図 2.1　フレームとスタック

本コンパイラにおけるデータ構造は，特にフレームとスタックに特徴があり，次にこれらについてのべる．

まず最初に，スタックは次のように用いられる．すなわち，local 変数と global 変数の区別は静的に行なわれ，節の適用時に，local 変数の記憶域は local スタック上に，global 変数とリスト・セルの記憶域は global スタック上にそれぞれ割り付けられる．また trail 情報は，global スタック上に置かれる．これらの様子を図 2.1 に示す．ここで，global スタックを指すポインタが，リスト・セルを指すポインタかそれともダイレクト・リンク（同一の変数に同一の記憶域を割り付けるためのリンク）かの区別は，次のように行なっている．すなわち，スタックの要素はすべて２バイトであるため，スタックを指すポインタの最後のビットは意味を持たない．この最後のビットを用いて，セルへのポインタかダイレクト・リンクかを区別している．

次に，フレームに関してであるが，これは図 2.1 で frame として示されている．このフレームは，各目標の実行ごとに作成され，実行を制御するために用いられる．このフレームの要素は，次の通りである．

(1)　cgsp :　globalスタックの先頭へのポインタ
(2)　par　:　親の frame へのポインタ
(3)　next :　後続目標の系列を実行する関数のアドレス
(4)　cvar :　local 変数の配列へのポインタ
(5)　arg　:　引数へのポインタの配列へのポインタ
(6)　drm　:　決定性か非決定性かのフラグ

このフレームは，次のように用いられる．節中の global 変数やリストは cgsp からのオフセット値で参照され，local 変数は cvar からのオフセット値で参照される．ある親目標に対して節の適用が成功し，次に親の後続目標を実行する際は next を参照し，また par を参照して親の変数に対する環境等を復元する．ある目標の実行に際しては，引数を配列に格納し，その配列へのポインタを arg に格納する．従って，引数は，arg からのオフセット値で参照されることになる．

drm は，節の適用が決定的かどうかのフラグである．

3. 制御方式

図 3.1 に append の翻訳例を示す．

目標の実行は，その目標を実行する関数へ，変数の環境へのポインタや後続目標を実行する関数のアドレス等を収めたフレームを引数として渡し，その関数を実行することにより行なわれる．その時，適用される節が単位節であるならば，単一化が成功した場合，そこで親から渡されたフレームを参照して，親の後続目標を実行する関数を呼び出すという方式を採用した．

この方式によると，後戻りやカット，決定性等に関する制御が，目標を実行する関数の返す値により容易に制御できるという利点がある．

すなわち，後戻りに関しては，単一化もしくは第一目標が失敗した時（変数 t には値 FALSE が代入される），変数を unbind し，local，global 双方のスタックを節の適用以前に戻す．次に，他の節の適用を試みるが，なければ値 FALSE を持って return する．

カットに関しては，後戻りがカットまで達した時，親の目標まで戻らなければならない（変数 cut_at が指標となる）が，この時別解を求めてはならないことが後戻りの場合とは異る．従って，この時は値 CUTEX を持って return する．

決定性に関しては，フラグ (frm.dtm) を用いて，節の適用が決定的かどうかを判定している．節の適用が決定的であると判定されたなら，local スタックのみを節の適用以前に戻し，DTRUE という値を持って return する．

append の例では目標が一つしかないため，フラグを用いる必要がなく，目標が返す値 (DTRUE) のみが決定性の判定の基準となる．

```
append(pfrm)
frame *pfrm;
{
    int t;
    list *lvar,*gvar,*parg=pfrm->arg;
    {
        pfrm->cvar=lvar=linit(1);
        gvar=ginit(0);

        if (! test(NIL,parg[0])) goto L1;
        lvar[0]=parg[1];
        if (! bind(&lvar[0],parg[2])) goto L1;

        t=(*pfrm->next)(pfrm->par);

        switch(t) {
        case TRUE : return(TRUE);
        case DTRUE: lsp=pfrm->cvar+1;
                    return(DTRUE);
        case FALSE: goto L1;
        case CUTEX: unbind();
                    lsp=pfrm->cvar+1;
                    gsp=pfrm->cgsp-1;
                    return(CUTEX);
        }
    }
    L1: unbind();
        lsp=pfrm->cvar+1;
        gsp=pfrm->cgsp-1;
    {
        pfrm->cvar=lvar=linit(1);
        gvar=ginit(4);

        gvar[0]=UNDEF;
        gvar[1]=UNDEF;
        gvar[2]=dlink(gvar[0]);
        gvar[3]=UNDEF;

        if (is_list(parg[0])) {
            gvar[0]=car(parg[0]);
            gvar[1]=cdr(parg[0]);
        } else
            if (! test(gcell(gvar[0]),parg[0])) goto L2;
        lvar[0]=parg[1];
        if (is_list(parg[2])) {
            if (! bind(dlink(gvar[2]),car(parg[2]))) goto L2;
            gvar[3]=cdr(parg[2]);
        } else
            if (! test(gcell(gvar[2]),parg[2])) goto L2;
        {
            frame frm;

            frm.cgsp=gsp+1;
            frm.par =pfrm->par;
            frm.next=pfrm->next;
            frm.arg =get1(3);
            frm.dtm =FALSE;

            frm.arg[0]=dlink(gvar[1]);
            frm.arg[1]=&lvar[0];
            frm.arg[2]=dlink(gvar[3]);

            t=append(&frm);

            switch(t) {
            case TRUE : return(TRUE);
            case DTRUE: lsp=pfrm->cvar+1;
                        return(DTRUE);
            case FALSE: goto L2;
            case CUTEX: if (cut_at == &frm) {
                            cut_at=frm.arg;
                            goto L2;
                        } else {
                            unbind();
                            lsp=pfrm->cvar+1;
                            gsp=pfrm->cgsp-1;
                            return(CUTEX);
                        }
            }
        }
    }
    L2: unbind();
        lsp=pfrm->cvar+1;
        gsp=pfrm->cgsp-1;
        return(FALSE);
}
```

図 3.1 append の翻訳例

4. インタプリタ

最も基本的な Prolog インタプリタを Prolog で記述すると次のようになる．

```
execute([]).
execute([Headgoal|Tailgoallist]):-call(Headgoal),execute(Tailgoallist).

call(Goal):-clause(Goal,Subgoallist),execute(Subgoallist).
```

このインタプリタは本コンパイラで翻訳することにより，本コンパイラによる目的プログラムからの呼出しが可能になる．従って，目的プログラムからこのインタプリタを呼び出し，まだ翻訳されていない Prolog プログラムを実行することが可能になる．このことはまた，翻訳されたプログラムが Prolog プログラムを生成し，そのプログラムを実行することができることをも意味する．

方式としては，実行すべき目標のリストを global スタック上に構成し，それを上記のインタプリタで解釈し，実行する．従って，メモリ効率や処理速度は低下するが，デバッグが容易になる．また，インタプリタの修正・改良が容易であるため，より強力なデバッグ機能の追加も容易に行なえると思われる．

さらに，単位節はデータと解釈し，翻訳せずにデータベースに蓄えて，インタプリタによりアクセスすることも可能である．

5. おわりに

本コンパイラは， H-Prolog システムから起動し，またその目的プログラムも H-Prolog システム上で作動するものである．本コンパイラにより翻訳されたプログラムの実行速度は，同一の Prolog プログラムを H-Prolog インタプリタで実行した場合に比較し，約５倍の効率改善がみられた．また，本コンパイラは大部分を Prolog で記述し，目的言語をＣ言語としているので移植性も高い．さらに，目的プログラムからインタプリタが起動できることで，処理系としての柔軟性も備えていると言える．

現在，本コンパイラによる目的プログラムをさらにＣコンパイラで翻訳し，H-Prolog システムとリンクする作業は人手に頼っている．これらの作業を自動的に行ない， Prolog の言語処理系としての拡充を図ることが今後の課題である．

参考文献

[1] Warren, D.H.D.: "IMPLEMENTING PROLOG" ,DAI Research Report No. 39,40 , Univ. of Edinburgh (1977).

[2] Nakamura, K.: "H-Prolog (version 1.0) reference manual" , TDU-ISG Memo 83-85, Tokyo Denki University (1983) .

[3] 中谷，斉藤，中村，平松 : " Prolog コンパイラの一方式 ", 情報処理学会第29回全国大会，(1984).

[4] Mellish, C.S.:"AN ALTERNATIVE TO STRUCTURE-SHARING IN THE IMPLEMENTATION OF A PROLOG INTERPRETER" ,DAI Reseach Paper No. 150, Univ. of Edinburgh (1980) .