

Prolog コール・インタプリタの検証について

古川 康一 新田 克己
(電子技術総合研究所)

1. はじめに

述語論理型言語 Prolog は 1971 年にマルセイユ大学で作られてから各地で注目を集め、現在までに多くの処理系が作られてきた。特に、エジンバラ大学の処理系 [War] はコンパイラと多くの組み込み関数を有する初めての実用的システムとして評価され、Prolog 支持者を増加させた。

さて、Prolog で応用プログラムを作るようになると、その欠点が色々と指摘され始めた。特に、Prolog プログラムの読み難さと、制御構造の貧弱さは多く指摘される所であり、cut の是非とも関連して、その改善は今後の大きなテーマである。

ここでは、Prolog の 2 つの大きな特徴であるバックトラックとユニフィケーションのプロセスを用いて説明する。この記述法は、Prolog インタプリタの動作を簡単に説明し、同時に、変数の束縛情報の有効範囲を明確にすることを目的としたものであるが、stream 処理や並列 Prolog 等、今後の Prolog システムの拡張にも容易に対応することができるものである。

次章では、インタプリタを AND-OR 探索木の 2 種類の節 (AND 節、OR 節) に対応した 2 種類のプロセスで記述し、その動作を概説する。

や 3 章では、このインタプリタが正しく動作することを検証する。この検証は、インタプリタの制御と、変数の束縛情報に分けて行う。

や 4 章では、インタプリタの性能改善についての考察、および、拡張について述べる。

2. Prolog インタプリタの記述

2. 1 コントロール

Prolog プログラムの実行は AND-OR 探索木の探索に例えることができる。例として、プログラム

```
(0) ?- a(X).  
(1) a(X) <- b(X), c(X).  
(2) b(X) <- e(X), f(X).  
(3) b(X) <- g(X).  
(4) c(X) <- h(X).  
(5) e(dick).  
(6) f(dick).  
(7) g(tom).  
(8) h(tom).
```

を実行すると、制御の流れは、Fig. 1 に示す AND-OR 探索木を左から右へ、depth first に探索する過程と一致する。この探索木は、OR 節が Prolog プログラムの 1 つの goal に対応し、AND 節が head に対応している。従って、上記のプログラムで (2), (3) のように同一の head を持つ複数の clause があるとき、AND-OR 探索木では OR 節からその数だけ枝が分かれる。また、(1) のようにある clause に goal が複数あれば、探索木では AND 節からその数だけ枝が分かれる。

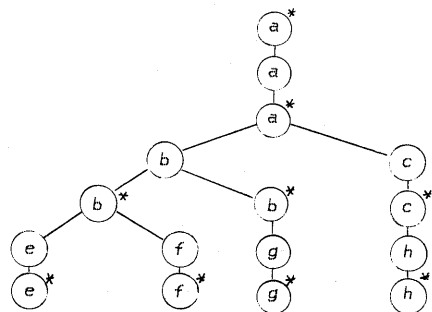


FIG.1 AND-OR SEARCH TREE

(○*: and-node ○: or-node)

AND-OR探索木を depth first に進めたとき、各節点は次のように結果を親の節へ送る。

AND節

- i) 子の節がないとき、又は、子の OR 節が全部成功したとき「成功」
- ii) 子のある OR 節が失敗なら、その OR 節の1つ前の OR 節へ後戻りをする。
- iii) 子の OR 節が全部成功にならないとき「失敗」

OR節

- i) 子の AND 節が成功したら「成功」
- ii) 子の AND 節が失敗したら、次の AND 節を探索
- iii) どの AND 節も成功しなければ失敗

Prolog インタプリタは上記の AND 節の動作を行う AND プロセスと、OR 節の動作を行う OR プロセスの2つを用いて表現される。AND プロセスは goal 列をパラメータとして受け取り、その goal の検証とバックトラックの制御を扱う。OR プロセスは1つの goal をパラメータとして受け取り、その goal とユニフィケーションできる clause を選択する。各プロセスの動作を Problem Analysis Diagram (PAD) [ニ村] を用いて表現したものを Fig. 2 (a) (b) に示す。

AND プロセスは、まず、パラメータの goal 列の初めの goal を curr-goal とする。curr-goal について OR プロセスを生成・起動させ、結果が戻るまでこの AND プロセスは wait 状態に入る。子プロセスの終了により、このプロセスは再起動され、子プロセスが成功ならば curr-goal を1つ進めて、再び OR プロセスを生成・起動させる。子プロセスが失敗ならば curr-goal を1つ戻して、前に成功した子プロセス (B-stack に管理されている) を再起動させ、その続きを実行させる。goal 列の最後の goal のプロセスが成功し

たら、この AND プロセスは成功である。また、goal 列の最初の goal のプロセスが失敗したら、この AND プロセスは失敗である。

OR プロセスは、パラメータの goal について、それとユニフィケーションできる head を持つ clause をさがし、その clause の body について AND プロセスを生成・起動させ、wait 状態に入る。子の AND プロセスが終了すると、この OR プロセスは再起動され、子プロセスが成功ならばこのプロセスも成功する。また、子プロセスが失敗ならばこのプロセスは別の clause を選択し、ユニフィケーションを試みる。

このように AND/OR のいずれのプロセスも、制御を子から親へ、又は、親から子へと伝えることによってコーーチンとして動作する。生成されたプロセスは、失敗したときは消滅するが、成功したときはその状態のまま保存され、バックトラックによる再実行に備える。(Fig. 2 で E_2, e_2 の直前の○印が、このような状態に相当する所である)

また、1つの AND プロセスから、複数の OR プロセスが生成されるが、1つの OR プロセスからは、同時に存在する AND プロセスは1つしかないことに注意する。

プロセス間の制御の流れを Fig. 3 に簡単に示す。この図は、AND プロセスが OR プロセスを生成し、この OR プロセスが AND プロセスを生成したときの制御の流れを示したものである。

2. 2 ユニフィケーション

ユニフィケーションには、いくつかのアルゴリズムがあるが、現行の処理系では Robinson のアルゴリズムが多く用いられているようである。ここでは、Robinson のアルゴリズムを copy 方式 [Bru] で実現する。

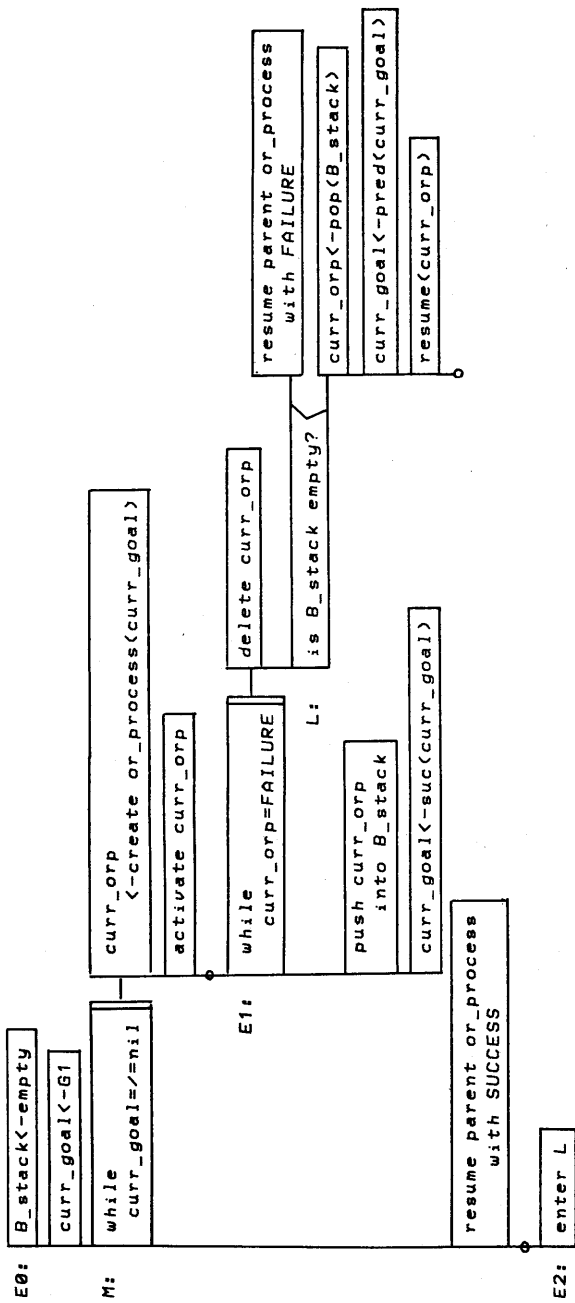


FIG.2 (a) AND-PROCESS(GOAL SEQUENCE) GOAL SEQUENCE=G1,G2,...,Gn

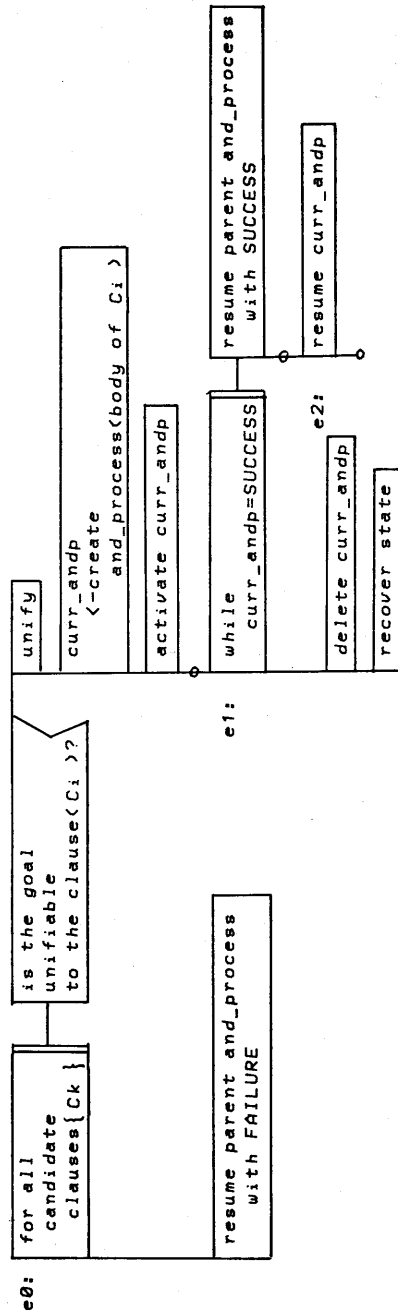


FIG.2 (b) OR-PROCESS(GOAL)

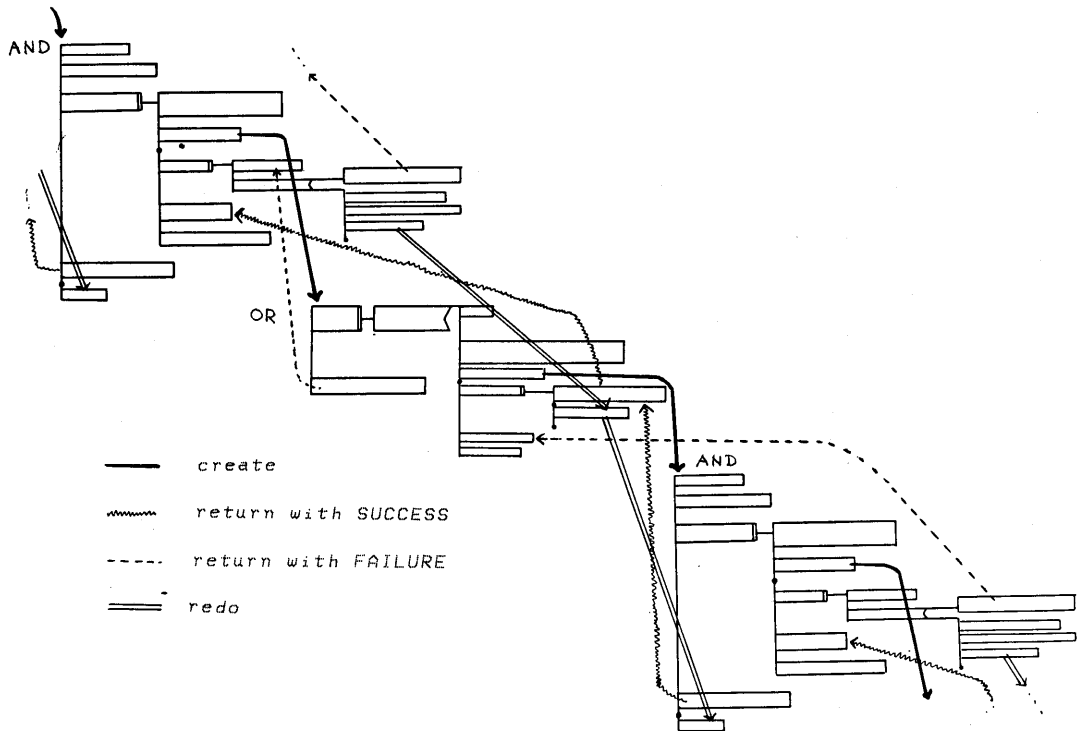


FIG.3 RELATION BETWEEN AND-PROCESS AND OR-PROCESS

ある OR プロセスで $\text{goal } \text{foo}(A_1, \dots, A_n)$ と $\text{head } \text{foo}(B_1, \dots, B_n)$ のユニフィケーションを、 procedure Unify を用いて、

```

unify ( $A_1$ ,  $B_1$ )
unify ( $A_2$ ,  $B_2$ )
...
unify ( $A_n$ ,  $B_n$ )

```

で行うものとする。

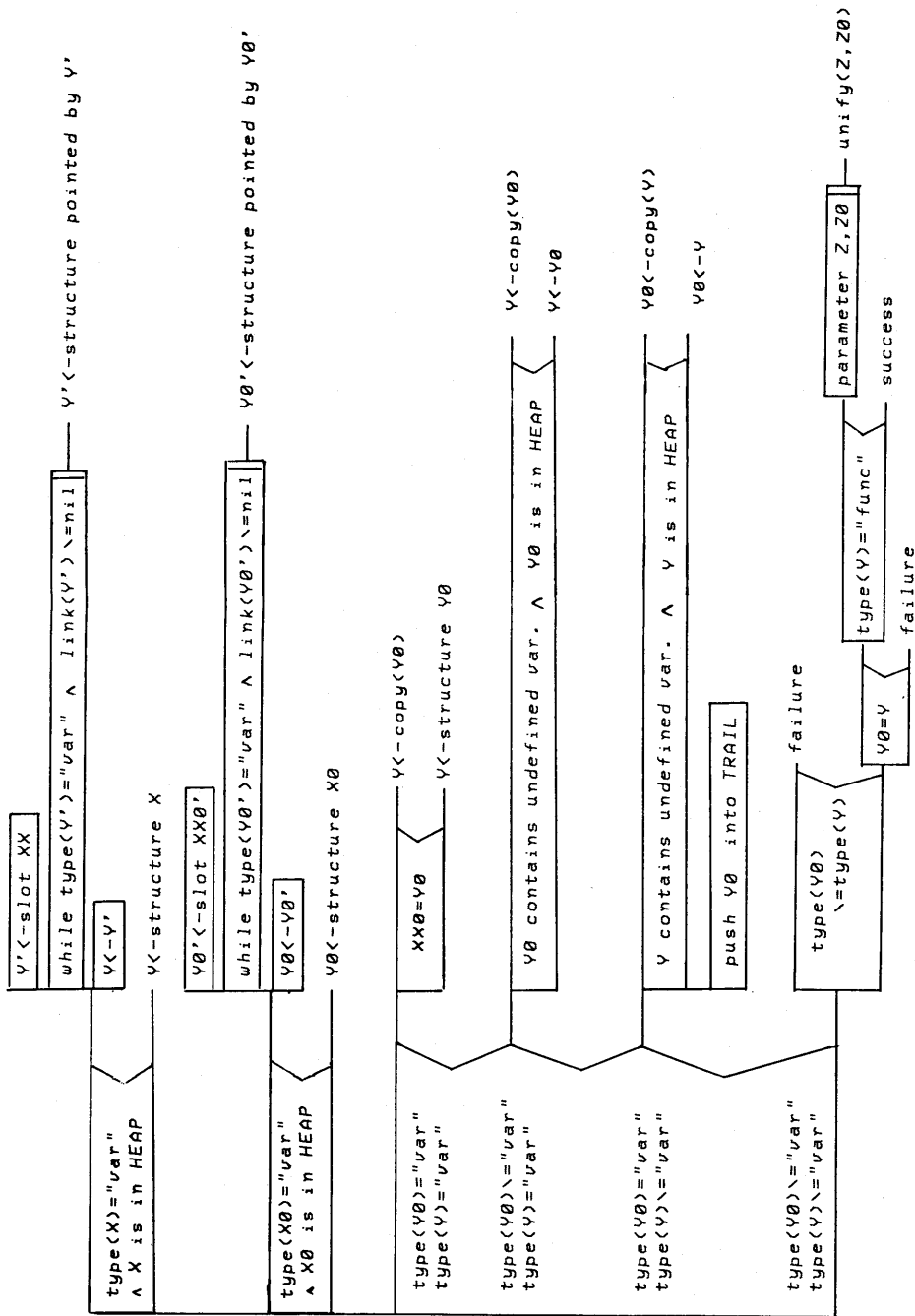
このような結果を得るための $\text{unify}(XO, X)$ のアルゴリズムを Fig. 4 に示す。ここで、パラメータ XO, X はソースプログラムで記述されたリテラルであり (heap に存在する)、“var”、“func”、“int”等、いくつかのタイプを有する。

ユニフィケーションを行おうとする OR プロセスは選択された clause の head 中の変数の値を、その親の AND プロセスは goal 列にある変数の値を記憶するスロットを、それぞれ持つ。例えば、goal 列の中に変数 $C_1, C_2, \dots, C_n$

があれば、この AND プロセス中にはこれらに対応する m 個のスロットがある。また、head の中に変数 $D_1, D_2, \dots, D_e$ があれば、OR プロセス中に e 個のスロットがある。(実際には、スロットはすべて OR プロセスの中にあり、その OR プロセスの子の AND プロセスは、親のプロセスのスロットに自由にアクセスできるものとする)

Fig. 4 において、 X のタイプが“var”のとき、その値がすでに定義されていれば X に対応するスロット XX によってその値 Y を知ることができる。 X の値が未定義 ($XX = \text{nil}$) または X のタイプが“var”でなければ Y は X そのものの (Fig. 4 では structure X と表現した) である。(YO についても同様である)

Y, YO の一方が“var”、他方が“var”でないときには、一方が他方にリンクされる。例えば、 $\text{type}(Y) \neq \text{“var”}$ 、 $\text{type}(Y) = \text{“var”}$ ならば、 Y は YO を指す。



($link(Y) = Y0$ となる) このとき、 $Y0$ に未定義変数を含まなければ、 Y は $Y0$ を直接指すが、 $Y0$ に未定義変数を含むならば、 $Y0$ を現在の環境で評価してその値 $Y0'$ (Fig. 4ではこれを $Copy(Y0)$ と表現している)を $copy\ stack$ 上に作り、 Y は $Y0'$ を指す。 $Y0'$ を作る際、その中の未定義変数に対応するスロットがANDプロセスの中にあるはずである。この未定義変数とスロットをポイントでリンクしておき、後にこの変数に値が定ったときに、両者が同じ値になることを保証する。

バックトラックの際には、 $copy\ stack$ 中の不要なコピーは棄却される。また、 $TRAIL$ により、環境を元に戻さなくてはならない。(Fig. 2(b)の $recover\ state$ がこれにあたる)バックトラックが生じないときには、 $copy\ stack$ 中の構造は、プロセスの生成・消滅にかかわりなく保存される。

3. コルーチン・インタプリタの検証

3. 1 コントロール

ここでは、コルーチン・インタプリタがAND-OR探索木を完全に探索することを証明する。証明すべきものは次に述べる定理である。

＜定理1＞

PrologプログラムのAND-OR探索木が有限であれば、その木に対応する各AND/ORプロセスは、起動されるごとに新しい解を1つずつ生成し、すべての解を生成した後に起動されると「失敗」となって消滅する。□

この証明はAND-OR探索木の深さについての帰納法で行う。ユニフィケーションが失敗したときを除いて、探索木の端の節は必ずAND節であることに注意を要する。

定理の証明

(a) base

(i) 深さ0のANDプロセス

深さ0のANDプロセスは、 $goal$ 列が空である。Fig. 2(a)のMにおいて $curr-goal = nil$ となるのでこのプロセスは直ちに成功する。

このプロセスは再起動されると E_2 から実行が再開され、Lへ行くが、 $B-stack$ が空なので、バックトラックすることなく失敗となる。

深さ0のANDプロセスはあるunit clauseのbodyに相当するものであるから、解は1つしか存在しない。従って定理は成立する。

(ii) 深さ1のORプロセス

深さ1のORプロセスについて、選択できる候補のclauseを $C_1, C_2, \dots, C_n$ とする。($C_1 \sim C_n$ はすべてunit clauseである) いま、 $C_1, C_2, \dots, C_i$ ($i < n$) までのclauseを調べ、 $C_1 \vee C_2 \vee \dots \vee C_i$ のすべての可能な解をすでに生成したとする。

Fig. 2(b)のeのループにおいて、 $goal$ が C_{i+1} のheadとユニフィケーションできたとすると、それに対応するANDプロセスが生成される。このANDプロセスは高さが0だから、(i)の考察により、直ちに成功し、 e_i のループに入る。このORプロセスは成功し、解が親のANDプロセスに渡されて、 e_2 の直前でwait状態に入る。

再び起動されると、このORプロセスは e_2 で子のANDプロセスを再起動するが、今度は失敗し、制御は再びこのORプロセスへ戻る。ORプロセスはこのANDプロセスを消去し、このANDプロセスに係る変数束縛を戻して(環境を回復して) e へ進み、次のclause C_{i+2} を調べる。このようにして、 C_n に達するまでこの過程を繰り返し、さらに再起動されるとeのループを抜けて失敗となる。

このORプロセスは、 $C_1 \vee C_2 \vee \dots \vee C_n$ の中でユニフィケーションが成功したものを解として親のANDプロセスに返

することになり、定理が成立する。

(b) induction

(iii) 深さが $2N$ の AND プロセス

深さが $2N-1$ 以下の OR プロセスについて定理が成立しているとき、深さが $2N$ の AND プロセスでも定理が成立することを示す。

そのためには、次の lemma が成り立つことを示せば良く、これは、appendix で述べるように証明できるから、定理は成立する。

lemma 深さが $2N$ 以下の AND プロセス (その goal 列を $G_1, G_2, \dots, G_n$ とする) について、制御が Fig. 2(a) の M にあり、 $\text{curr-goal} = G_i$ 、 $B\text{-stack}$ の状態が α 、変数環境が β であるとす。このプロセスを実行させると環境 β の下で、 $G_1 \wedge G_2 \wedge \dots \wedge G_n$ となるすべての解を、このプロセスが起動されるごとに 1 つずつ生成し、さらに、 G_i が失敗して L へ来たときの $B\text{-stack}$ の状態は α である。□

(iv) 深さが $2N+1$ の AND プロセス

深さが $2N$ 以下の AND プロセスで定理が成立しているとき、深さが $2N+1$ の OR プロセスでも定理が成立することを示す。

この OR プロセスが選択できる候補の clause を $C_1, C_2, \dots, C_n$ とすると (ii) での考察と同様に、各 clause ごとに AND プロセスが生成され、この AND プロセスは同一の環境の下で、それぞれすべての可能な解を生成する。従って、この OR プロセスはすべての解を 1 つずつ生成し、 C_n が失敗すると e のループを抜けて失敗となる。

以上で定理が証明された。 a.e.d.

3. 2 ユニフィケーション

ここでは、Fig. 4 に示したユニフィケーションについて検証する。

Robinson のアルゴリズムは、パラメータを 1 つずつ最新の環境で評価して

その値を比較し、変数の bind を行うことにより、環境を更新する。ここで示すことは、Fig. 4 のアルゴリズムがプロセスの生成・消滅に影響されることなく、Robinson のアルゴリズムを実行することである。以後、 $\text{unify}(X, X_0)$ の実行前・実行後の環境をそれぞれ E_s , E_e とし、 X を環境 E で評価したときの値を $X\{E\}$ 、 $X\{E\}$ が構造 Y で表わされることを $X\{E\} \Rightarrow Y$ とかく。

< 定理 2 >

Fig. 4 において、 X が heap 上の構造で変数を含むならば、 $X\{E_s\}$ は Y の中の変数を (unify を call した) OR プロセスのスロットの指す値で置換して得られる。 X が変数を含まない構造か、 copy stack 上の構造ならば、 $X\{E_s\} \Rightarrow Y$ である。 X_0 についても同様のことが成り立つ。(ただし、 X_0 のときは OR プロセスではなく、親の AND プロセスのスロットを用いる) □

定理を証明するために、まず、次の補題が成立することを簡単に述べる。

補題 スロットから出たリンクの鎖は枝分かれすることなく 1 つの構造に達する。□

補題の略証

初めはスロットの値 = nil であるがユニフィケーションによって外部の構造を指すようになる。これには 2 つの場合があり、1 つは変数の bind が行われるとき、もう 1 つは copy を作る際に、 copy される構造内の未定義変数とリンクされるときである。いずれの場合も、ポインタはスロットから出て外の構造を指す。しかも、以後のポインタ操作は Y , Y_0 について行われ、上述の 2 種類のリンクにより、 Y , Y_0 から他の構造へのリンクが行われるから、補題が成り立つ。 a.e.d.

この補題を用いて、次のように定理

は証明される。

定理の証明

(a) base

初期状態として、 $E_s = \varepsilon$ 、 X と XO は heap 中の構造とする。 X が変数ならば、 Y' はスロット XX であり、 $\text{link}(Y') = \text{nil}$ だから、 Y はスロット XX である。(ここでは、スロットは copy stack 上の構造と同じ扱いをする) X が変数でなければ Y は X そのものである。 Y の中に変数があっても、 $E_s = \varepsilon$ ならばそれらは未定義である。従って $X\{E_s\} \Rightarrow Y$ となり、定理は成立する。(Xも同様)

(b) induction

E_s について定理が成立するとき、unify 実行後に、 E_e についても成立することを示す。

(i) $\text{type}(Y_0) = \text{type}(Y) = \text{"var"}$ のとき

仮定により、 $X\{E_s\} \Rightarrow Y$ 、 $XO\{E_s\} \Rightarrow Y_0$ とする。 $Y_0 = XXO$ ならば、 Y_0 のコピー Y_0' を copy stack 上に作り、 Y は Y_0' を指す ($Y_0\{E_e\} \Rightarrow Y_0'$ となる)。コピーを作る際、 Y_0 は Y_0' とリンクされ、 $Y_0\{E_e\} \Rightarrow Y_0'$ となるから、 $X\{E_e\} \Rightarrow Y_0'$ 、 $XO\{E_e\} \Rightarrow Y_0'$ となって定理が成立する。

$Y_0 \neq XXO$ ならば、 Y_0 は copy stack 上にある。 Y は直接 Y_0 を指し、 $Y\{E_e\} \Rightarrow Y_0$ となるから、 $XO\{E_e\} \Rightarrow Y_0$ 、 $X\{E_e\} \Rightarrow Y_0$ となって、定理は成立する。

(ii) $\text{type}(Y_0) \neq \text{"var"}$, $\text{type}(Y) = \text{"var"}$ のとき

仮定により、 $X\{E_s\} \Rightarrow Y$ 、 $XO\{E_s\} \Rightarrow Y_0$ にスロットの値を代入したもの (Y_0 が heap 上にあり、未定義変数を含むとき)、 $XO\{E_s\} \Rightarrow Y_0$ (それ以外するとき) とする。

Y_0 が heap 上にあり、未定義変数を含むとき、 Y_0 のコピー Y_0' が copy stack 上に作られる。 Y_0' は、 Y_0 に含まれる変数が定義済なら、その値を AND プロセス中のスロットから求めて代入し、変数が未定義なら、AND プロセス中の対応するスロットとそれをリンクさせたものである。従って、 Y_0' は $XO\{E_s\}$ を表わし、補題により、スロットの値

とそれに対応する Y_0' 中の変数は常に同じ値をとる。よって、 $X\{E_e\} \Rightarrow Y_0'$ 、 $XO\{E_e\} \Rightarrow Y_0$ にスロットの値を代入したものとなり、右辺は同じ値となるから定理は成立する。

Y_0 が変数を含まないとき、又は、copy stack 上にあるときには、 Y は Y_0 を指す ($Y\{E_e\} \Rightarrow Y_0$ となる)。よって、 $X\{E_e\} \Rightarrow Y_0$ 、 $XO\{E_e\} \Rightarrow Y_0$ となって定理が成立する。

(iii) $\text{type}(Y_0) = \text{"var"}$, $\text{type}(Y) \neq \text{"var"}$ のとき

(ii) と同様にして証明できる。

(iv) $\text{type}(Y_0) \neq \text{"var"}$, $\text{type}(Y) \neq \text{"var"}$ のとき

Y 、 Y_0 が function ならば、個々のパラメータについて unify がリカーシブに call され、これは Robinson のアルゴリズムの順序と一致する。 Y 、 Y_0 が function でなければ、その値が一致するか否かによってユニフィケーションの正否が決まり、環境の変化は起こらない。リカーシブに call された個々の unify は、(i) ~ (iii) のいずれかによって証明できる。

(iii) の場合、 Y_0 の bind は TRAIL に記憶する。これは、(i)(ii) の場合には bind 情報は OR プロセスに記録されるので、バックトラック時にこの OR プロセスを消去すれば環境は元に戻る。(iii) の場合は、bind 情報は親の AND プロセスに記録されるので、OR プロセスを消滅させるだけでなく、TRAIL を用いて、環境を戻さなくてはならないからである。

このアルゴリズムでは、リンクの鎖で他のプロセスのスロットを参照することがない。また、copy stack の情報が消去されるのは、バックトラック時のみである。従って、次章でふれるような、決定的に成功したプロセスが消滅する場合でも、必要な構造は必ずアクセスできる。

4. コルーチン・インタプリタの実現

Fig. 2 に示したインタプリタは、その動作を明確に記述することを目的とし、cut や効率的実行のための手法は省略されている。

例えば、可能な解を全て生成した OR プロセス（最後の解を出力した直後の状態を、「決定的に成功した状態」という）は、次に再起動されると必ず失敗するので、最後の解の生成後に消滅させてメモリを解放させることが考えられる。

また、ここに示したモデルでは親プロセスと子プロセスの間の交信しかしないが、プロセスの発生順序を考慮して、他のプロセスとの交信を許せば、高速インタプリタが実現できる。この場合、[田村]のように AND/OR プロセスを 1 つにまとめ、冗長性を廃することも考慮すべき点である。

ここで述べたインタプリタは SIMULA でインプリメントされている。現在、改良されたインタプリタを BLISS でインプリメント中である。

5. おわりに

AND/OR プロセスによる Prolog のコルーチン・インタプリタを紹介し、それが正しく動作することを検証した。このインタプリタは、制御構造と変数情報のスコープをわかりやすく記述し、概念の整理に役立つものである。

なお、ここでは copy によるユニフィケーションを述べたが、これはプロセス表現とマッチするからである。（ユニフィケーション・アルゴリズムとしての structure sharing と copy の優劣はいくつか比較されているが、決定的な結論はでない）

謝辞 本研究の機会を与えられました石井治・パターン情報部長、棟上昭男・ソフトウェア部長 に感謝致し

ます。また、討論していただいた松本裕二氏をはじめとする研究グループの方々に感謝致します。

参考文献

- ・ニ村良彦: "Problem Analysis Diagram"
- ・Warren D.H.D: "Implementing PROLOG"
D.A.I. Reports 39,40 Univ. Edinburg 1977.
- ・Emden M.H.: "An Algorithm for Interpreting PROLOG program" Report CS-81-28 Univ. Waterloo 1981.
- ・Bruynooghe M.: "The Memory Management of PROLOG Implementation" Logic Programming Workshop
- ・Conery J.S. et al.: "Parallel Interpretation of Logic Programs" ACM Conference on Functional Language (1981).
- ・田村直之: "述語論理型プロログラミング言語の研究"
神戸大学修士論文 1982.
- ・Furukawa K. et al.: "Prolog Interpreter based on Concurrent Programming" Prolog コンファレンス 1982.

APPENDIX

ここで lemma の証明を帰納法で証明する。

まず、準備として、次の補題を証明する。

補題 AND プロセスにおいて、いま、Fig. 2(a) の M に制御があるとする。
 $\text{curr-goal} = G_{i+1}$ ならば、B-stack には $G_1, G_2, \dots, G_i$ に係るプロセスがこの順に管理されている。□

補題の証明

(a) base

この AND プロセスの初期状態は B-stack = empty, $\text{curr-goal} = G_1$ である。 G_1 について OR プロセスが生成・起動され、この OR プロセスが失敗すれば、この AND プロセスも失敗となる。逆にこの OR プロセスが成功すれば、B-stack に push され、 $\text{curr-goal} = G_2$ となって M へ行くから、補題は成立する。

(b) induction

いま、 $i \leq k-1$ なる i について補題が

成立していると仮定し、 $i=k$ のときも補題が成立することを示す。

$\text{curr-goal} = G_k$ について OR プロセスを生成・起動させたとき、このプロセスが成功ならば、この OR プロセスは $B\text{-stack} \wedge \text{push}$ され、 $\text{curr-goal} = G_{k+1}$ となるから補題は成立する。この OR プロセスが失敗ならば、 L のループで $B\text{-stack}$ は pop up され、 G_{k-1} に係る OR プロセスが再起動される。また、 $\text{curr-goal} = \text{pred}(G_k) = G_{k-1}$ となるから、 E_1 のループにおいては、実行中の OR プロセスの goal と curr-goal は常に一致する。従って、この OR プロセスが成功して M へ行くときは補題が成立する。
Q.E.D.

lemma の証明

(a) base

$\text{curr-goal} = G_n$ (goal 列の最後の goal)、 $B\text{-stack}$ の状態 $\alpha = (G_1 \sim G_{n-1}$ の OR プロセスが push)、 $\beta =$ 現在の変数環境 とする。 G_n に係る OR プロセスが生成・起動され、これが成功すると $B\text{-stack} \wedge \text{push}$ されてこの AND プロセスは E_2 の直前で wait 状態となる。これが再起動されると、 L へ行き、 $B\text{-stack}$ からこの OR プロセスは pop up され、再起動される。この OR プロセスが解を出す間は、(i) $B\text{-stack}$ への push と、(ii) $B\text{-stack}$ から pop して再起動 が繰り返される。ここで、定理証明における (b)(iii) の仮定により、OR プロセスはすべての解を同じ環境の下で生成するから、この AND プロセスは同一環境 β の下で G_n のすべての解を生成することになる。

また、この OR プロセスが失敗すると E_1 のループへ入ってこの OR プロセスは消滅し、 L へ行くが、このときの $B\text{-stack}$ は状態 α と同じである。

よって lemma が成立する。

(b) induction

$i > k$ なる i について lemma が成立するとき、 $i=k$ のときも成立することを示す。

いま、 $\text{curr-goal} = G_k$ で、 M に制御があり、 $B\text{-stack}$ の状態を α 、環境を β とする。 G_k について OR プロセスが生成・起動される。これが成功すると、 $B\text{-stack}$ に push され、 $\text{curr-goal} = G_{k+1}$ となって M へ行く。帰納法の仮定から、この状態 ($G_1 \wedge \dots \wedge G_k$ が成立し、 G_{k+1} を証明しようとしている段階) からは、この AND プロセスを起動することにより、同一環境の下で $G_{k+1} \wedge \dots \wedge G_n$ のすべての可能な解を1つずつ生成することになる。

解をすべて生成した後の起動により、 G_{k+1} の OR プロセスが失敗すると、制御は E_1 のループから L へ行く。このときの $B\text{-stack}$ の状態は G_{k+1} の実行直前のものである。補題より、 $B\text{-stack}$ の一番上は G_k に係る OR プロセスである。この OR プロセスが pop されて再起動され、 G_k について別の解を出すと、その解について上述の過程が繰り返される。定理証明の (b)(iii) の仮定から、この OR プロセスは G_k についてすべての解を生成し、その各々の解について $G_{k+1} \wedge \dots \wedge G_n$ の可能な解が生成されることになる。従って、この AND プロセスは $G_k \wedge \dots \wedge G_n$ のすべての解を生成する。

その後、 G_k に係る OR プロセスが失敗すると、 E_1 のループでこのプロセスは消滅し、 L へ達したときの $B\text{-stack}$ の状態は α である。よって lemma が成立する。
Q.E.D.