すごい Prolog つくって学ぼう?!

Daiki Matsunaga

2017 年 11 月

1 プロローグ

Prolog というプログラミング言語があります。C 言語や BASIC が手続き型言語、Java や Smalltalk がオブジェクト指向言語、OCaml や Haskell が関数型言語と呼ばれるのに対して、Prolog は論理型言語と呼ばれています。Prolog は、命題の証明を背景とする実行の仕組みやバックトラック、論理変数、ユニフィケーションなど面白い特徴を多く備えています。

この記事では、そんな Prolog の処理系をつくった話をします。巷に溢れるコンパイラやインタプリタの本は、実装される言語が手続き型や関数型言語であるものがほとんどで、論理型言語を扱っているものはあまり見ません。ですが、特に WAM(後述) は非常によく考えられていて、役に立つ立たないとか関係なく面白い話題であると思います。

2 Prolog の実行形態

Prolog 処理系の実現方法として、まずインタプリタが挙げられます。以前私も Prolog インタプリタを実装しました[*1]。ユニフィケーションや変数の管理で特に苦労したように思います (もっと良いやり方があったのかもしれませんが...)。その他には、中間言語に変換して仮想マシンで実行したり、C 言語のソースコードへ変換してから機械語にコンパイルしたりといった方法が挙げられます。

様々な方法がありますが、多くの Prolog 処理系は Warren's abstract machine(WAM) をベースにしています。WAM は David H. D. Warren によって考案された、Prolog の実行のために考えられた抽象機械で、Prolog の複雑な仕組みをうまく実現しています。

この記事ではそんな WAM の解説を主に行いたいと思います。

3 WAM

3.1 WAM って何?

WAM が Prolog の実行のための抽象機械であると 2 節で言いました。「こんな感じの命令セットとレジスタ、メモリがあれば Prolog の複雑な実行メカニズムを効率よく実現できるんじゃないの」と Warren さんが考えたものが Warren's Abstract Machine、WAM です。Prolog プログラムは述語・質問単位で WAM 命令

[*1] https://github.com/matsud224/petit-prolog

にコンパイルされ、仮想マシンで実行されます。また、Prolog の処理に特化したハードウェア (Prolog マシン) で WAM の影響を受けているものもあります。

WAM のメモリレイアウトとレジスタを図 1 に示します。プログラムカウンタやスタック、ヒープ位は耳にしたことがあるかも知れませんが、トレイルやバックトラックレジスタなど聞き慣れない名前がいくつか出てきます。徐々にこれらの説明をしていくので、時々この図を見返してみてください。

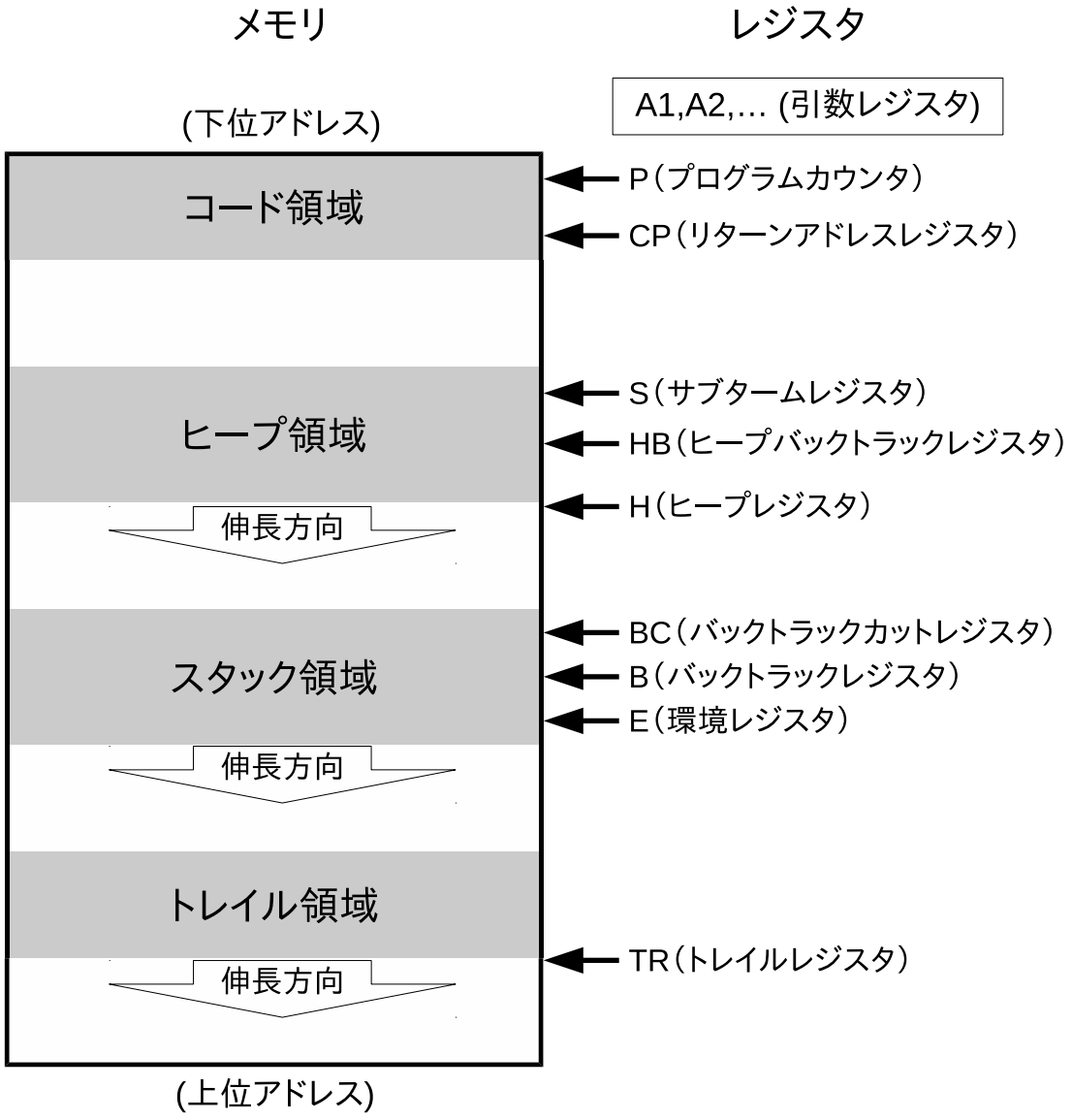

図 1　WAM のメモリレイアウトとレジスタ

3.2 レジスタとメモリ

まず、メモリとレジスタの使い方についていくつか決めておきます。メモリにはアドレスが振られていて、メモリ 1 単位にはタグと値の組が格納できるものとします。タグは、表 1 に挙げたものをを使用します。また、引数レジスタにもメモリと同様にタグと値の組が格納できるとします。その他のレジスタにはアドレスを格納できます。

タグ	値	意味
REF(reference)	アドレス	メモリへの参照
CON(constant)	アトム or 数値	定数
STR(structure)	構造体の先頭アドレス	構造体への参照
SST(structure start)	構造体名/引数の数	構造体の開始

表 1　タグとその意味

例えば、構造体 a(1,X) は表 2 のようにメモリ上に置かれます。ここで、X は未束縛変数であるとします。未束縛変数は自分自身を参照することで表現されます。

アドレス	タグ	値
100	SST	a/2
101	CON	1
102	REF	102

表 2　構造体 a(1,X) のメモリ上での表現

Prolog は未束縛変数同士のユニフィケーションが可能で、どちらか一方にもう一方の変数への参照が代入されます。すると、変数同士のユニフィケーションにより何重にも参照が連鎖することが考えられます。定数もしくは構造体・リストへの参照、もしくは未束縛変数に到達するまで参照を辿る操作をデリファレンスといいます。WAM では、変数に対してまずデリファレンス操作を行い、その中身を判断します。

3.3 ユニフィケーション

まずはユニフィケーションから始めましょう。

```
p(X, bar, a(bar,X)).
```

という事実 p があるとき、

```
?- p(foo, X, a(X,foo)).
```

と質問してみます。これは `X=bar` となります。

事実 p と質問はそれぞれ図 2 と図 3 のような WAM 命令にコンパイルされます。なお、左端の数字は行番号です。% から行末まではコメントで、Prolog プログラムとのおおまかな対応を示しています。

基本的には、呼び出し側が put-*命令で引数レジスタに値をセットして、呼ばれた側が get-*命令で引数レ

```
1: get-variable A4,A1   % p(X,
2: get-constant bar,A2  %   bar,
3: get-structure a/2,A3 %   a(
4: unify-constant bar   %     bar,
5: unify-value A4       %     X)
6: proceed              %   ).
```

図 2　事実 p の WAM 命令

```
1: put-constant foo,A1  % ?- p(foo,
2: put-variable A4,A2   %      X,
3: put-structure a/2,A3 %      a(
4: set-value A4         %        X,
5: set-constant foo     %        foo)
6: call p/3             %      ).
```

図 3　質問の WAM 命令

ジスタを検査するという流れです。引数レジスタは A1, A2, ..., An と表記し、順番に第 1 引数, 第 2 引数, ..., 第 n 引数の授受に使用します。引数レジスタの個数についてはここでは無視します。

get-*や unify-*命令はレジスタ・メモリ上の値を見てユニフィケーションの成功/失敗を判定します。ここでレジスタ・メモリ上の値同士のユニフィケーション操作が行われ、値の比較や未束縛変数への束縛が行われるのです。また、構造体は put-struct 命令に続く set-*命令、get-struct 命令に続く unify-*命令で扱います。

上記の例を見てみます。まず質問のゴール側 (呼び出し側) で、引数レジスタ A1 に 1 番目の引数を、A2 に 2 番目の引数を... と引数を順番にセットしていきます。一番な簡単なものとして、1 行目の put-constant foo,A1 はレジスタ A1 に定数 foo をセットします。

2 行目の put-variable ではレジスタに (未束縛) 変数をセットしています。引数レジスタはゴールの呼び出しが起こる度に使用 (上書き) されます。変数は後に束縛が起きる可能性があるため、それでは困ります。そのため、変数はレジスタに直接置かれるのではなくヒープ領域またはスタック領域にその領域が確保されます。スタック領域に確保する場合は後で説明するとして、ここではヒープ領域に確保を行います。ヒープレジスタ H は使用済みヒープ領域の末尾を指すレジスタです。2 行目の put-variable A4,A2 は、まず H レジスタをインクリメントして、ヒープ領域に変数 1 つ分の領域をとります。そしてそれを未束縛状態にして、そのメモリへの参照を引数レジスタ A4 と A2 に代入します。引数は 3 つしか無いのに A4 を使ってもいいの?、と思われたかもしれません。引数の個数を超えた引数レジスタは自由な用途に使えます[*2]。

3-5 行目は構造体 a(X,foo) の処理です。構造体もヒープ領域に置かれます。まず 3 行目の put-structure a/2,A3 でヒープ上に構造体の開始の印 (SST) を置きます。A3 にはその構造体への参照 (STR) が入ります。set-*命令は構造体の引数をヒープ領域に置くために put-structure に続いて使用されます。まず変数 X を置くわけですが、X は既出で、A4 にはすでに変数 X への参照が入っています。そのため set-variable ではなく set-value を使います。そして定数 foo を置いて、call p/3 で述語 p/3 を呼び出します。

ここまでで、ひとまず呼び出し側の処理は終わりです。図 4 は 5 行目実行直後のレジスタとヒープの様子です。

さて、今度は呼ばれた側 (事実 p) の WAM 命令を見ていきます。引数レジスタを get-*命令で順番に見ていきます。1 行目の get-variable A4,A1 は単純に A1 の値を A4 に代入しています。2 行目の get-constant bar,A2 では A2 すなわち第 2 引数と bar でユニフィケーションを行います。A2 が未束縛変数であった場合は A2 に bar を束縛します。A2 が未束縛変数でなく、定数 bar でもない場合はユニフィケーション失敗なので、後述するバックトラックが発動されます。

構造体の場合は、get-structure の後に unify-*命令が続く形と なります。3 行目の get-structure a/2,A3 は

[*2] ここで A4 にも代入していますが、実はこれは無駄なので最適化で改善されるべき点なのです...

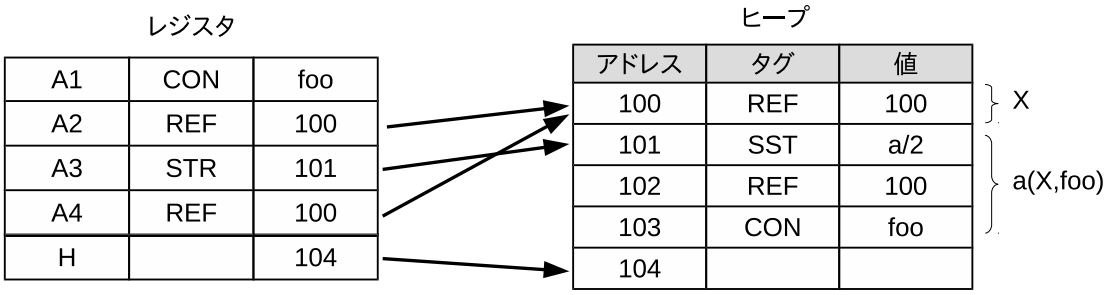

図 4　5 行目実行直後のレジスタとヒープの様子

A3 すなわち第 3 引数が構造体 a/2 もしくは未束縛変数であるか調べます。

もし構造体 a/2 がすでにヒープ領域にあるなら READ モードへ移行します。サブタームポインタ S が構造体のある場所を指すようにして、後続の unify-*命令で S をインクリメントしながらヒープ領域上の構造体の引数についてユニフィケーションを行います。

もし A3 が未束縛変数だったなら WRITE モードへ移行します。後続の unify-*命令ではヒープ領域に構造体を構築していきます。このように、get-structure によって選択されたモードによって続く unify-*命令の動作が変わるのです。ユニフィケーションは双方向であることを考えるとそうなるのも納得がいきます。

今回の質問`?- p(foo, X, a(X,foo)).`では構造体を置いているので、READ モードとなります。4 行目の unify-constant bar ではヒープ上の構造体 a/2 の第 1 引数 (アドレスは 102) を見ます。対象が違うだけで get-constant と同様の動作です。

5 行目の unify-value A4 では A4(変数 X) とヒープ上の構造体 a/2 の第 2 引数 (アドレスは 103) とをユニフィケーションします。このようなレジスタ・メモリ上に置かれた値同士のユニフィケーションについても、Prolog のユニフィケーション規則と同様です。例えば、CON foo と CON foo は成功、CON foo と CON bar は失敗、REF 10(未束縛変数) と CON foo の場合は 10 番地を foo にして成功、のような感じです。未束縛変数同士のユニフィケーションの場合は一方のアドレスを他方へ代入することになるのですが、参照の方向がアドレスが大きい方から小さい方となるようにします。これはダングリングポインタとなりうる、ヒープからスタックへの参照ができないようにするためです。

今回の質問では、ユニフィケーションに成功します。ユニフィケーション終了後のレジスタとヒープの様子を図 5 に示します。

もし、質問が`?- p(foo, X, Y).`だったとすると、WRITE モードとなります。この時の unify-*命令は set-*命令と同じ動作となり、呼び出し側で構造体 a(bar,X) をヒープに構築します。

3.4　環境

ここでは、規則 (節でボディをもつもの) を扱えるようにします。

```
a(...) :- b(...), c(...).
```

この規則 a を WAM 命令にコンパイルすると、

(a の get/unify 命令列)

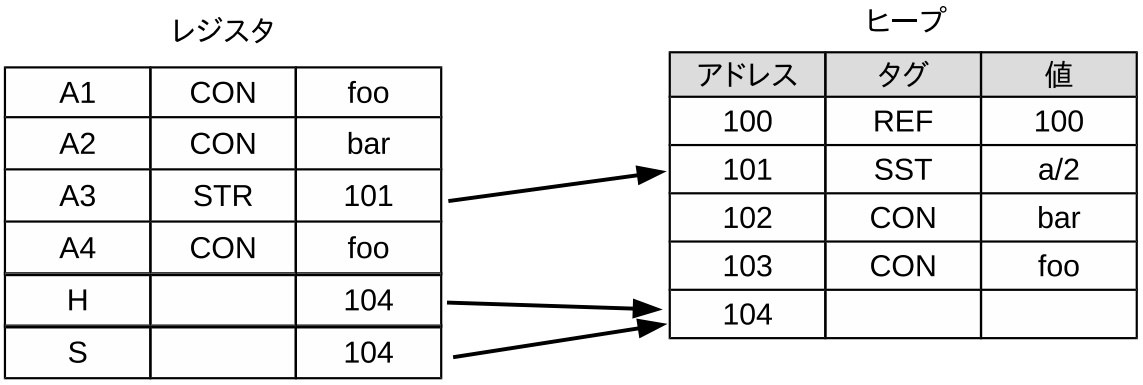

図 5　ユニフィケーション終了後のレジスタとヒープの様子

(b の put/set 命令列)

```
call b
```

(c の put/set 命令列)

```
call c
```

こんな感じになります。ボディの数だけ put/set と call を並べるだけです。

ところがここで、ある問題が発生します。次の規則 p をご覧ください。

```
p(X,Y) :- q(Y), r(X),s(Z).
```

変数 X は引数レジスタ A1 から参照されているとします。そのとき、r(X) の呼び出しの時に A1 に X への参照がある保証はありません。q の呼び出しで引数レジスタが上書きされてしまうからです。

そこで環境というものを導入します。これは C 言語のスタックフレームのようなものだと思ってください。これまでは変数をヒープに確保して引数レジスタから参照していたのですが、変数の領域をスタック上の環境内にとることで上記の問題を解決します。環境内に置かれる変数は、要するに C 言語のローカル変数みたいなものです。ここではこのような変数を恒久変数と呼びます。それと対比して、ヒープ上に取られる変数を一時変数と呼びます。ある変数が恒久変数として環境内に確保されるか、一時変数としてヒープに確保されるかはコンパイル時に決定されます。その基準は、「ボディで、複数のゴールにまたがって出現する変数は恒久変数、それ以外は一時変数とする。」です。上記の規則 p では、X が恒久変数、Y と Z が一時変数です。Y はヘッドとボディの両方に出現していますが、ヘッドでの出現はカウントしません。

環境について詳しく見ていきます。まず、現在の環境は環境レジスタ E が指しています。環境の生成は allocate 命令、破棄は deallocate 命令を使用します。allocate 命令ではスタック領域に新たな環境を確保・初期化し、E がそれを指すようにします。deallocate 命令では E が 1 つ前の環境を指すようにします。Prolog の変数のスコープは節内であるため、節が呼ばれる度に環境がスタック上にとられます。WAM コードとしては、節のはじめに allocate 命令、終わりに deallocate 命令を置いてやります。そして、環境は表 3 のような構造となっています。E+2+n で n 番目の恒久変数にアクセスできます。それぞれの環境は 1 つ前の環境のアドレスを持っているため連結リストになっています。

もうひとつ、リターンアドレスについて補足しておきます。call 命令では、別の節へ移ったあとに戻ってくるためにリターンアドレスをリターンアドレスレジスタ CP にセットしています。そして、事実の場合は

```
1: allocate 1            %
2: get-variable Y1,A1    % p(X,Y) :-
3: put-value A2,A1       % q(Y)
4: call q/1              % ,
5: put-value Y1,A1       % r(X)
6: call r/1              % ,
7: put-variable A1,A1    % s(Z)
8: call s/1              % .
9: deallocate            %
```

図 6　規則 p の WAM 命令

proceed 命令が CP を P に代入して復帰します。規則の場合は、CP が書き換えられる可能性があるためリターンアドレスは環境内に退避しておきます (表 3 参照)。そして、deallocate 命令で環境からリターンアドレスを P へ代入して復帰します。

E	1 つ前の環境のアドレス
E+1	リターンアドレス
E+2	恒久変数の数
E+3	1 つ目の恒久変数
E+4	2 つ目の恒久変数
E+5	3 つ目の恒久変数
...	...

表 3　環境の構造

先ほどの規則 p は図 6 のような WAM コードにコンパイルされます。まずは 1 行目の allocate 1 で恒久変数 1 個が入る環境を確保します。2 行目には Y1 という表記がされていますが、これは 1 番目の恒久変数を示しています。get-variable Y1,A1 は A1 から Y1 に代入を行います。Yn へは E からのオフセットでアクセスがなされます。9 行目の deallocate 命令で、E でアクセスできる 1 つ前の環境のアドレスを E にセットします。要するに、E が 1 つ前の環境を指すようにします。

3.5　バックトラック

あとバックトラックができれば、もう Prolog です。この節では、以下のように OR 関係にある節を並べることができるようにします。

```
a(foo).
a(bar).
a(baz).
```

バックトラックは、ユニフィケーションが失敗するなどの要因で起こります。あるゴールに対して候補節が複数あってそのうちの一つを選ばなければならないとき、その時点をチョイスポイントと呼びます。バックト

```
    try_me_else L1
    < a(foo). のコード >
L1: retry_me_else L2
    < a(bar). のコード >
L2: trust_me
    < a(baz). のコード >
```

図 7　述語 a/0 の WAM 命令

ラックが起きると、もっとも最近に生成されたチョイスポイントに戻り、次候補を選択します。Prolog では定義順で選択を行います。バックトラック時にチョイスポイントに戻ってきて次の候補節を選択できるようにするためには、その時点でのレジスタの値や次の候補節などの情報を保持しておかなければなりません。

まず、チョイスポイントに戻ってこれるようにするためにチョイスポイントフレームというものを導入します。チョイスポイントフレームには、チョイスポイントでのレジスタの値や次候補節などの情報を格納します。バックトラック時には直近のチョイスポイントへ戻るので、チョイスポイントフレームは生成するごとに積み重ねていきます。

チョイスポイントフレームの構造を表 4 に示します。

B	引数の個数 n
B+1	チョイスポイントでの A1 レジスタ
B+2	チョイスポイントでの A2 レジスタ
...	...
B+n	チョイスポイントでの An レジスタ
B+n+1	チョイスポイントでの E レジスタ
B+n+2	チョイスポイントでの CP レジスタ
B+n+3	1 つ前のチョイスポイントフレームのアドレス
B+n+4	次候補節のコードの開始アドレス
B+n+5	チョイスポイントでの TR レジスタ
B+n+6	チョイスポイントでの H レジスタ

表 4　チョイスポイントフレームの構造

節のはじめの述語 a/0 の WAM コードは図 7 のような感じになります。まず try-me-else 命令でチョイスポイントフレームを生成します。別候補の選択が起こった場合は、L1 から実行が行われます。L1 や L2 はラベルです。retry-me-else 命令は途中の節で、trust-me 命令は最後の節で使用します。retry-me-else 命令はチョイスポイントフレームを更新します。trust-me 命令はチョイスポイントフレームを破棄します。

チョイスポイントフレームは一体どこに置かれるのでしょうか。図 1 を見てみると、トレイル領域というのが余っているのでそこかと思ってしまいますが、実はスタック領域に置かれるのです。つまり、環境とチョイスポイントフレームがスタック領域に混在することになります。一番新しいチョイスポイントフレームを指しているのがバックトラックレジスタ B です。

環境とチョイスポイントフレームはそれぞれが連結リストを形成しているので、ひとつのスタック領域で混

在していても独立に辿ることは可能です。しかし、スタックからポップする際に問題が起こります。例えば、環境-チョイスポイントフレーム-環境-チョイスポイントフレーム-環境の順に並んでいるときに環境をポップすると隙間ができてしまいます。

それではなぜ環境をチョイスポイントフレームをそれぞれ独立した領域に置かないのでしょうか。それにはちゃんと理由があります。私はこれを知った時に、すごいなぁと思いました (小並感)。

以下のプログラムと質問を使用して、環境とチョイスポイントフレームを独立した領域に配置した際に起こる問題を見てみましょう。

```
a :- b(X),c(X).
b(X) :- b1(X).
c(X) :- c1(X).
b1(1).
b1(2).
c1(2).
c1(3).

?- a.
```

まず a が呼ばれて a の環境が環境スタックに乗ります。b(X) が呼ばれ、b の環境を生成します。b1 のチョイスポイントを生成した後、b1(X) と b1(1) のユニフィケーションで X=1 となります。b の環境が破棄され、次は c(X) が呼ばれます。c の環境を生成し、c1 のチョイスポイントを生成した後、c1(X) と c1(2) とのユニフィケーションに失敗します (X は 1 に束縛されているため)。この時点では図 8 のようになっています。ここでバックトラックが起こり、B レジスタが指している c1 のチョイスポイントフレームを見て、レジスタの復元と次候補節へのジャンプを行います。c1(3) が選択されますが、再び失敗し、バックトラックします。c1 の候補節はもう無く、c1 のチョイスポイントフレームは破棄されており B は b1 のチョイスポイントフレームを指しています。そして b1 のチョイスポイントの次候補節 b1(2) へジャンプします。と、ここで大変なことに気が付きます。b の環境がなくなっているのです！ b1 は b のボディで呼ばれているため、b の環境がないとダメです。

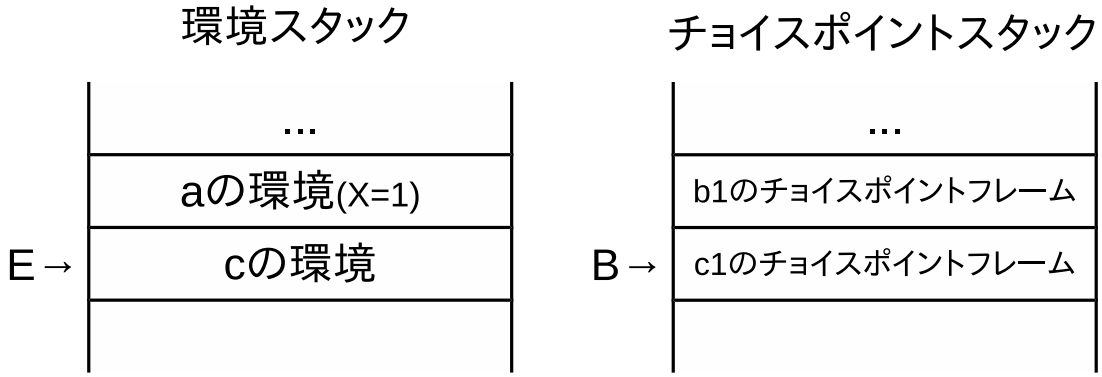

図 8　スタックを分ける (スタックは下へ伸びる)

チョイスポイントフレームを作っておきながらその時の環境は破棄されてしまうことが問題です。これを解決するためには、チョイスポイントフレームが存在している限りはその時点までの環境を残しておくことが必要です。そうであれば、環境とチョイスポイントフレームを混在させるとこれがシンプルに解決できます。

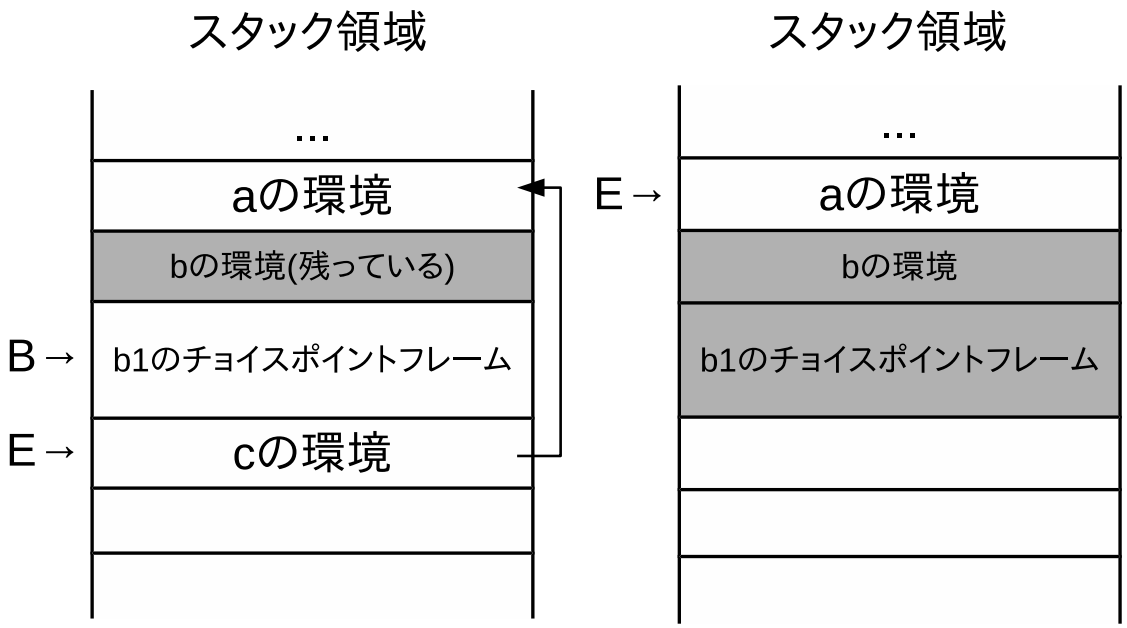

図 9　スタックを一緒に使う (左:c の環境確保後、右:b1 のチョイスポイントフレーム破棄後)

b1 のチョイスポイントフレーム生成後に c の環境を確保しようとしたとします。E が B よりも下 (アドレス値が大きい) の時には、これまでどおり E が指す環境の後ろに新たな環境を確保します。今回は、b の環境を破棄した後なので、E は a を指しています。B は b1 のチョイスポイントフレームを指しています。このように B が E よりも下 (アドレス値が大きい) ときは、新たな環境は B が指すチョイスポイントフレームの後ろに確保します。チョイスポイントフレームはそれまでに生成された (下位アドレスの) 環境を保護しているのです。c の環境を確保した後のスタック領域の様子が図 9 の左側です。

c1 が失敗して c の環境の deallocate が起こると、E は a の環境を指します。c の環境の確保時には E は a の環境を指していたからです。再び b に戻ってきて、このときチョイスポイントフレーム内に保存していた E(b の環境を指す) が復元されます。b1(2) が選択されたら、b1 にはもう候補節はないため、b1 のチョイスポイントフレームは破棄されます。この状態 (図 9 の右側) で c の環境の allocate が行われると、b の環境があった領域が再利用されます。すなわち、環境とチョイスポイントフレームを混在させてはいますが、チョイスポイントフレームが有効な間は必要な環境が保護され、不要になったらちゃんと再利用されるのです。

3.6　束縛の取り消し

前の節ではひとつ大事なことを忘れていました。バックトラックでチョイスポイントに戻る際、レジスタを戻すだけでは不十分です。チョイスポイントからバックトラック発生時点までの間に変数を束縛した場合にそれを未束縛状態に戻さなければなりません。

束縛した変数に関する情報の記憶にトレイル領域が使用されます。変数を束縛する度にトレイル領域にその

変数のアドレスが格納されていきます。Prolog では変数は一度束縛されると再代入できないため、変数のアドレスを覚えておくだけで十分なのです。トレイルレジスタ TR は使用済みトレイル領域の末尾の次のアドレスを保持しています。ある時点からある時点までの束縛を取り消したい時には、それぞれの時点での TR レジスタの値を覚えていおけば大丈夫です。その 2 つの TR レジスタが保持するアドレスの間にあるアドレスの変数を未束縛にすればいいだけです。

全ての変数束縛を記録しておいてもいいのですが、それは明らかに無駄です。バックトラック時には、B が指す直近のチョイスポイントフレームに戻ります。そして、E レジスタは B より下位アドレスの環境を指すことになります。ヒープについても、直近のチョイスポイントの H レジスタの値が復元されます。そのために、ヒープバックトラックレジスタ HB には直近のチョイスポイントでの H レジスタがセットされています。このように、バックトラック時にはチョイスポイントから現在まで使用されてきたスタック領域とヒープ領域は破棄されます。よって、チョイスポイント以前より存在していた変数への束縛だけをトレイル領域に記録しておけば十分です。具体的には、スタック領域の変数であればそのアドレスが B 未満、ヒープ領域の変数であればそのアドレスが HB 未満であるときに記録対象とします。

3.7 その他の話題

ここまででユニフィケーションやバックトラックなど Prolog の大事な部分が WAM においてどのように実現されているのか説明しました。

この他にもカットの導入や、末尾呼び出し最適化・環境刈り込み、インデキシングといった最適化手法の話がありますが、詳しくは参考文献 [1][2][4] をご覧ください。特に参考文献 [1] は WAM について非常に丁寧に解説しており、本記事の作成や処理系の実装の際、大いに参考にさせて頂きました。

4 処理系を作った

WAM バイトコードへコンパイルし WAM 仮想マシンで実行する Prolog 処理系を CommonLisp で実装しました。なお、ソースコードは公開[*3]しております。この処理系は、上記で説明した基本的な Prolog の機能に加え、以下のような機能を実装しています。

- カット
- 環境刈り込み (⊃ 末尾呼び出し最適化)
- インデキシング (第一引数を見て候補節を絞り込む)

4.1 構文解析器

S 式を受け取るようにして構文解析をサボるという手もありますが、そこはやっぱり Prolog の文法で書けるようにしたいものです。Prolog の文法はシンプルなので構文解析器は比較的簡単に作れます。しかし、is/2 のような述語のサポートや op/3 組み込み述語による演算子定義を可能とするため、構文解析器をを工夫しました。具体的には、演算子名・優先順位・結合法則の 3 つ組が登録してある演算子テーブルを見て構文解析をできるようにしました。Prolog は結合法則の指定が充実していて、{ 右結合, 左結合, 結合なし } と { 単項前

[*3] https://github.com/matsud224/wamcompiler

置, 単項後置, 二項 } の組み合わせで 7 種類から選べます。非常に柔軟に演算子定義ができるのですが、実装する側としては辛いです。演算子順位構文解析法を試したりもしましたが、最終的に再帰下降構文解析法を工夫してこれを実現しました。

parse 関数が構文解析を行う関数です。

4.2 コード生成

compile-*関数でコンパイルを行います。その後、optimize-*関数などで WAM 命令レベルのちょっとした最適化を行います。例えば、リスト 1 を最適化するとリスト 2 となります。

Listing 1　最適化前の WAM コード

```
get-constant [],A1
get-variable X4,A2
get-variable X4,A3
proceed
```

Listing 2　最適化後の WAM コード

```
get-constant [],A1
get-value X2,A3
proceed
```

4.3 WAM 仮想マシン

参考文献 [1] の WAM 命令の疑似コードを参考に命令を実装しました。

4.4 REPL

Prolog は対話的に使用することができます。REPL を起動し質問を入力すると、解がある場合は解を表示し入力待ちとなります。本処理系では、

- y で終了
- n で次の解を表示
- a で全ての解を表示

という動作を行います。解がこれ以上ない場合は”no.”と表示されます。

REPL は repl 関数で実装しています。repl 関数は入力を読み取り、構文解析を行います。入力されたものが事実や規則であればコンパイルし、ハッシュテーブルに登録します。質問であればコンパイルし仮想マシンで実行します。

仮想マシンでの実行中に解が見つかったり失敗したりすると、REPL 側に制御を移します。単純に見つかった解をリターンするだけで良さそうですが、ここはちょっと面倒です。解が見つかったときには一旦 REPL に制御を移しますが、次の解を要求された場合は再度仮想マシンに制御を移し、**続きから**実行を行う必要があるのです。

続きから実行を行うのに、仮想マシンの状態をまるごと保存したり継続渡しスタイルで書いたりといった方法が考えられますが、今回は CommonLisp の Condition System を用いました。

Condition System は CommonLisp における例外処理機構で、一般的な言語のそれと比べて非常に強力です。Condition System には、例外ハンドラ側で原因を解決した後に例外発生箇所から実行を再開させる場面で用いられる restart という機能があります。これを今回は利用しました。なお、本処理系では通常の例外処理にも Condition System を用いています。Condition System については参考文献 [5] などをご覧ください。

4.5 組み込み述語

実用性を高めるためには組み込み述語が必要です。例えば、型判定の述語、数値計算用の述語、入出力を行う述語などです。

組み込み述語はある程度の数を定義することとなります。そのため、できるだけストレスなく定義できるとうれしいです。そこで、組み込み述語を定義しやすくするためのマクロや関数をいくつか書きました。例えば、整数ならば成功する integer/1 述語はリスト 3 のように定義できます。これは最も単純な述語のひとつであって、少し複雑な組み込み述語を定義しようとすると途端にしんどくなるので、まだまだ改良の余地があります。例えば、findall/3 はすべての解をリストにまとめる述語で、これはうまく Lisp で書けないので動的に WAM 命令を生成するという汚い実装となっています。

Listing 3　integer/1 組み込み述語の定義

```
(define-prolog-builtin "integer" (x)
  (if (and (wamvalue-constant-p x) (integerp (wamvalue-content x)))
      (prolog-success)
      (prolog-fail)))
```

現在、実装した組み込み述語を以下に示します。

- 型判定系... atom/1, atomic/1, number/1, integer/1, float/1, compound/1, var/1, nonvar/1
- 数値計算系... is/2[*4], </2, >/2, >=/2, =</2, =:=/2, ==/2
- 出力系... write/1, nl/0
- その他... op/2, consult/1, findall/3, call/1, =../2, fail/0, halt/0

4.6 実行例

以下の実行例では、作成した Prolog 処理系で append/3 述語を定義し質問しています。その後、REPL を抜け show-wamcode 関数で append/3 述語がどのような WAM コードへコンパイルされたのかを表示させています。なお、"CL-USER>"は Lisp のプロンプト、"prolog>"は Prolog のプロンプトです。

Listing 4　実行例

```
CL-USER> (repl)
prolog> append([], Xs, Xs).
```

[*4] 四則演算、剰余、べき乗、sqrt, sin, cos, tan... などを使える

```
prolog> append([X | Ls], Ys, [X | Zs]) :- append(Ls, Ys, Zs).
prolog> ?-append(X, Y, [k,i,t,c,c]).
X = NIL
Y = [k,i,t,c,c]
?a
X = [k]
Y = [i,t,c,c]

X = [k,i]
Y = [t,c,c]

X = [k,i,t]
Y = [c,c]

X = [k,i,t,c]
Y = [c]

X = [k,i,t,c,c]
Y = NIL

no.
prolog> ?-halt.

yes.
NIL
CL-USER> (show-wamcode "append" 3)
          switch-on-term G950,G954,G953,FAIL
G954:     switch-on-constant {[] => G952}
G950:     try-me-else G951
G952:     get-constant [],A1
          get-value X2,A3
          proceed
G951:     trust-me
G953:     get-list A1
          unify-variable X4
          unify-variable X1
          get-list A3
```

```
unify_value X4
unify_variable X3
execute append/3
```

5 エピローグ

「Prolog が面白い言語なら、Prolog 処理系作りも面白いのでは?」ということで、実際に作ってみたのですが、とても楽しかったです。Prolog インタプリタを作るのも楽しかったのですが、コンパイラだと生成された WAM コードを眺めて楽しめるので良いです。

参考文献

[1] Hassan Aït-Kaci, Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, http://wambook.sourceforge.net/

[2] David H.D.Warren, AN ABSTRACT PROLOG INSTRUCTION SET(日本語訳), http://www.takeoka.org/~take/ailabo/prolog/wam/wam.html

[3] 柴山 潔, 並列記号処理, コロナ社

[4] Kostis Sagonas, Logic Programming Implementation Part I: The WAM, http://www.it.uu.se/edu/course/homepage/logpro/ht04/LP_Impl.pdf

[5] 工藤千加子, 黒田寿男, Common Lisp における例外処理 -Condition System の活用-, 株式会社数理システム 知識工学部, http://cl-www.msi.co.jp/solutions/knowledge/lisp-world/tutorial/condition-system.pdf