記 号 処 理 67－4
(1 9 9 3. 1. 2 2)

ヒープを使用する論理型言語でのトレール方式

山崎憲一　天海良治　竹内郁雄　吉田雅治
NTT 基礎研究所, NTT ヒューマンインタフェース研究所

現在, 我々が設計中の言語 TAO はバックトラックなどの論理型言語の機能と代入などの副作用を合わせもつ言語である. 論理型言語の実装としては, 構造データをグローバルスタックに割り当てる WAM (Warren Abstract Machine) 方式が一般的であるが, これを副作用のある言語に用いると dangling pointer が生ずる可能性がある. このため, TAO では構造データをヒープに割り当てる. しかし, この割り当て法によって WAM よりもトレールスタックを多く消費するようになる. 本論文ではトレール量を削減するアルゴリズムを提案する. このアルゴリズムは副作用を持つ論理型言語一般に適用可能である.

A Trailing Algorithm for Logic Programming Languages using Heap Memory

Kenichi Yamazaki　Yoshiji Amagai　Ikuo Takeuchi　Masaharu Yoshida
NTT Basic Research Laboratories,
NTT Human Interface Laboratories

We are designing a programming language TAO which has logic programming features such as backtracking and side effects such as assignments at the same time. Although WAM (Warren Abstract Machine) that allocates structured data on the global stack is commonly used to implement a logic programming language, it may cause dangling pointers if the language has side effects. TAO allocates structured data on the heap memory to solve this problem. However, this allocation method consumes the trail stack more than WAM. In this paper, we propose an algorithm which reduces the trail stack consumption. The algorithm can be applied to any type of logic programming languages that has side effects.

1 はじめに

Prologの特徴の1つはバックトラックである．広く用いられているWAM (Warren Abstract Machine)[1]による実現では，構造データ（リストおよび構造体）をスタック上に割り当て，バックトラック時にはこれをスタック機構によって解放する．Prologを代入などの副作用のある言語と融合し，構造データを共有する場合，この方式ではうまくいかない．解放されてしまった領域をポインタが指すようなデータ構造が，作成されうるからである．我々が現在設計中の言語TAO[2]では，これを避けるため，構造データをスタック上でなく，ヒープ上に割り当てる．しかし，このようにした場合，WAMに比べて，メモリ消費が多くなるという問題が生ずる．本稿では，この問題を解決するアルゴリズムについて述べる．このアルゴリズムは，上に述べたような融合型の言語一般に対して適用可能である．以下，第2節でWAMのメモリの使用方法について分析したのち，第3節で副作用のある言語とWAM方式で実装された論理型言語を融合する際の問題点を指摘する．第4節では，その問題点を解決するアルゴリズムを述べる．

なお，以下では，もとのデータを上書きするような代入であることを強調するために破壊的代入という言葉を使うことがある．単に代入と言った場合はユニフィケーションにより未定義の変数を具体化する操作も含むものとする．

2 WAMのメモリ管理方法

WAMについて，必要な部分だけを説明する．WAMはメモリを3つの部分に分けて管理する．それぞれは，

- ヒープ（グローバルスタック）
- スタック（ローカルスタック）
- トレール（トレールスタック）

のように呼ばれるが，ここでは，混乱を避けるため，上記の括弧内の名称を使う．また，ヒープはLispのセル領域のような方法で管理されるメモリを意味するものとする．すなわち，ヒープにおいては，スタックと異なり，アドレスの大小関係に意味はなく，メモリ解放がゴミ集め(garbage collector, GC)によって行われる．スタックは底の方向を下といい，その逆を上と呼ぶ．下の方がアドレスが大きく，上の方が小さい．

グローバルスタックには，構造データが割り当てられる．述語呼び出しが進むにつれて，構造データがグローバルスタックに積まれてゆく．バックトラックが起きると，選択点（後述）を用いて，グローバルスタックのスタックトップポインタ（グローバルSPと呼ぶ．ローカルなども同様）が，その選択点が作られたときのグローバルSPの値に戻される．これにより，その新しいSPより上のデータが自動的に捨てられる．

ローカルスタックには，環境と選択点が積まれる．環境は呼び出された述語の変数などである．本稿では選択点が重要である．選択点は，非決定的な述語が呼び出されたとき，すなわち，述語呼び出しの引数パターンとのユニフィケーションが成功する節が複数あるとき，ローカルスタックに積まれる情報である．選択点には，それが生成されたとき（すなわち積まれたとき）の，グローバルSP，トレールSP，最新の選択点へのポインタなどが含まれる．最新の選択点とは，直前に積まれた選択点であり，スタックの最も上にある選択点のことである．この選択点の生成が終了した時点で，今生成した選択点が最新の選択点となる．ユニフィケーションが失敗するとバックトラックが起き，最新の選択点を使ってSPなどのレジスタを戻し，次に述べるUNDO操作を行う．

トレールスタックには，ユニフィケーションによる代入が起きたときに，その代入がなされたアドレスが積まれる．この操作をトレールと呼ぶ．バックトラックのときには，最新の選択点に記録された昔のトレールSPに達するまでトレールスタックをポップしながら，代入を戻していく．つまり，そのアドレスのメモリの値を未定義にしていく．この操作をUNDOと呼ぶ．

トレールは必ずしもすべての代入において必要なわけではない．WAMは，次のような特性を利用してトレールの一部を省略している．バックトラックが起きると，最新の選択点が作成された時点の値に，ローカルSPとグローバルSPが戻される．これにより，各SPより上の値はすべて捨てられる．そののち，UNDO操作をするが，上で捨てられた値までUNDOする必要はない．なぜなら，そのような値は二度と使用されないからである．したがって，そのような値への代入は最初からトレールする必要もない．これは，選択点が作られた時点よりも

新しく生成された未定義変数への代入であれば，トレールする必要がないことを意味する．このために，最新の選択点が作られたときのグローバルSPの値を保持するレジスタがある (HBレジスタ)．HBと代入するアドレスを比較して，HBの方が下であればトレールをする必要はない．

3 副作用のある言語との融合

3.1 融合とその実現法

TAOは複数のプログラミングパラダイムを融合した言語である．TAOにおいては論理型プログラムが作った構造データを変数や別の構造データに破壊的に代入することができる．しかも，この代入はトレールされない[1]．

WAM方式の実装をした論理型言語に破壊的代入を導入すると次のような問題が生ずる．Prologに似た仮想的な言語を使って，次のような例を考える．

?- X = [a,b,c], $\alpha \leftarrow$ X, fail.

ここで，$\alpha \leftarrow$ X は，変数Xの値をグローバルスタックやローカルスタックでない別の空間のメモリセルαに代入するものとする．上記のゴールを実行すると，αにXの値が代入されるが (図1(a))，述語failによってバックトラックが起動されるため，グローバルスタック上のデータが捨てられ，αはデータが存在しない空間を指すようになる (図1(b))．いわゆるdangling pointerである．

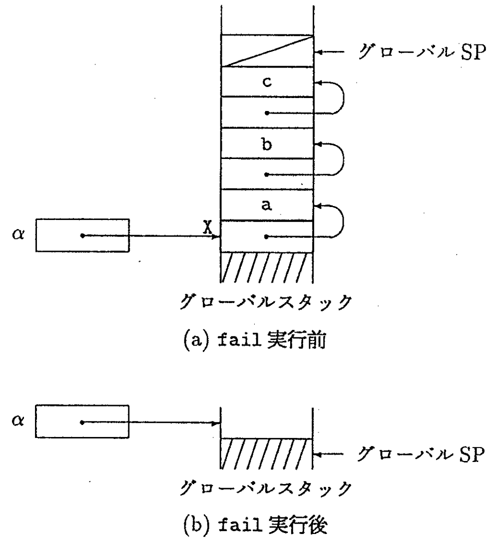
失敗せず，必ず成功して終了する述語であっても，同じ問題が発生する．あるゴールをトップレベルから呼び出すと，たとえそれが成功しても，終了して答えを表示した時点で，すべてのスタックを初期化してしまうから，グローバルスタック中の構造データは消去されてしまうのである．

この問題を解決するには，以下の2つの方法が考えられる．

● 破壊的代入に制限を加える

基本的にスタックベースのWAM方式の実装をし，破壊的代入の際にはデータをグローバルスタック外へ持ち出す．これは例えばESP[3]で取られている方法である．

[1]実は破壊的であることよりトレールされないということが本質的であるが，ここでは破壊的代入という言葉で表す．



(a) fail 実行前

(b) fail 実行後

図1: dangling pointerの例

ESPにはヒープに置かれるヒープベクタと，グローバルスタックに置かれるスタックベクタがある．ヒープベクタの要素には未定義変数とスタックベクタは代入できない．また，ヒープベクタへの代入はバックトラックで取り消されない．さらに，スタック上のデータをヒープ上の特殊なデータに変換する述語 (freeze, melt) がある．これらを組み合わせて，スタック上のデータをヒープに明示的に移し，バックトラックと関係なくデータを保持することなどが可能である．この方法は，WAMの実装がそのまま使えるという点で優れているが，ユーザはスタック上のデータとヒープ上のデータを意識して使い分けなければならない．

● スタックを使わない実装をする

Dangling pointerが発生する原因は，構造データをスタックに割り当てるからである．ヒープを使って，メモリが解放されないようにすれば，この問題は解決される．この方法は，代入に一切制限がないという点で優れている．ユーザはデータが割り当てられている場所を意識する必要はない．しかし，後述するようにメモリ消費が多くなるという欠点がある．

TAOは関数型プログラムと論理型プログラムを密に融合することを目標としている．しかし，前者の方式で

実装するとユーザは常にデータの種類を意識しなければならず，関数型と論理型のプログラム間で自由にデータを渡せなくなる．このような理由によりTAOでは後者を採用する．

なお，TAOで唯一ユーザが意識しなければならないのは，論理型プログラムで作成した構造データをある場所に破壊的代入した後で，そのプログラムがバックトラックした場合である．この場合，代入されたデータの一部が未定義値に戻されることがある．しかし，多くの場合，プログラムは成功して終了することが多く，我々のプログラミング経験上，これが問題となったことはほとんどない．また，未定義値の参照関係を保持したままリストをコピーする関数なども用意されている．

3.2 実現上の問題点

前節で言及したように，ヒープ上に構造データを置く実装をした場合，メモリ消費が多くなることが最大の問題である．単純に実装した場合，例えば，

```
append([],X,X).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).


foo(X) :- append(X,[],Z), fail.
foo(X) :- append(X,[],Z).


?- foo([a,b,c]).
```

を実行した場合，結果的にはfooの引数であるリストの3セル以外に，ヒープ6セル (12ワード) とトレールスタック3ワードを消費してしまう (図2)．一方，WAMではグローバルスタック6ワードのみである．このメモリ消費の増加は，それぞれ次のような理由による．

● ヒープ

WAMではfooの第1節の失敗によって，グローバルスタック上の3セルが解放され，結果的に第2節で再利用される．一方，本方式ではメモリは獲得する一方で再利用されず，解放はGCによってのみなされる．このため，ヒープのメモリ使用量が増加する．しかし，バックトラックによって構造データが解放されないようにするためにヒープ上に構造データを置くのであるから，これはやむをえないことである．また，ヒープの無駄が生じるのはバックトラックが起きたときであり，成功するのが主であるような計算ではWAMと同程度のメモリしか消費しない．

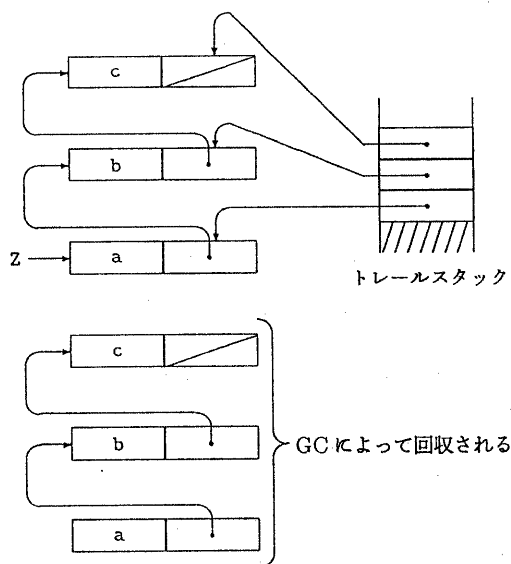


図2: ヒープとトレールの状態

この問題は，例えばある構造データが他の場所に代入されないことが，プログラムの解析などによってわかるのであれば解決可能であるが，これは今後の研究課題である．また，一般にWAMのGCは複雑であるが，ヒープを使った場合は簡単で高速なGCが期待できる．これについての定量的解析も今後の課題である．

● トレールスタック

前述のようにWAMでは，最新の選択点よりも後に生成された構造データへの代入であることがわかる場合はトレールしない．先の例で考えてみる．まず，fooは非決定的な述語であるから，選択点を生成する．この場合のappendは決定的であるから，選択点は生成しない．したがって，append実行中に生成されたセルへの代入は，すべて最新の選択点より後に生成されたデータへの代入である．したがって，トレールは不要である．

一方，ヒープに構造データを単純に割り当てる方式のままでは，構造データがいつ生成されたのかがわからないため，ユニフィケーションによるヒープへの代入をすべてトレールせざるをえない．同じ例で，appendのボ

ディからappendが呼ばれた場合を考えると理解しやすい．ヘッドの第3引数のユニフィケーションでセルを生成し，1つ前のappend呼び出しで生成したセルのCDRへ代入する．このセルはいつ生成したものであるかわからないから，トレールせざるをえない．その後，今生成したセルのCARへ変数Aの値を代入するが，これは自分の節内で生成したセルへの代入であることがわかるから，トレールの必要はない．この繰り返しにより，結局生成したリストの50% (この場合は3ワード) のトレールスタックが必要となる．

一般に，構造データに含まれる未定義値のほとんどは結局代入されるから，トレール量が極めて多くなることが予想される．実際，すべてをトレールする方式の処理系で測定すると，トレールはヒープと同程度のメモリを消費していることがわかる [4]．WAMのトレール消費量はグローバルスタックの5〜30%程度 [5] であることが多いから，これは重大な問題である．

4 トレールの削減方法

前節で述べたトレール量増加の問題を解決するための新しいアルゴリズム (ヒープトレールアルゴリズム) を提案する．ヒープトレールアルゴリズムは基本的にはWAMのトレール簡略化の方法をそのままヒープに適用したものである．以下では，これをWAMとの差分の形で説明する．まず，WAMの命令を挙げ，トレール簡略化の方法を説明する．次に，カットがない場合のヒープトレールアルゴリズムを説明する．これはWAMのトレール簡略化法と極めてよく対応がとれるものである．次にカットが入った場合のアルゴリズムを説明する．なお，TAOには構造体がないため構造データとしてはセルだけを考えているが，構造体にもまったく同じアルゴリズムが適用できる．

以下では，セルの各要素 (CAR, CDR) に割り当てられたメモリをスロットと呼ぶ．また，++ は1加算を，−− は1減算を表す．

4.1 ヒープトレールアルゴリズム

● WAM

まず，ヒープトレールアルゴリズムを使ったときに変更を受けるWAM命令を示す[2]．各命令は変更をうける部分についてだけ記述してある．HBは前述したように，最新の選択点が作られたときのグローバルSPの値である．また，WAMではメモリ内容が未定義である状態を，自分自身への間接ポインタによって表している．

レジスタ: HB.

セルを取るとき:

各スロットに自分のアドレスを指す間接ポインタを入れる[3]．

TRY:

HB ← グローバルSP.

選択点を作り，HBをセーブ．

TRUST:

最新の選択点を捨て，更新する (残った選択点で一番上のものを最新とする)．

HB ← 最新の選択点にセーブされたHB.

CUT:

指定された選択点を最新の選択点とする．

HB ← 最新の選択点にセーブされたHB.

TRAIL:

if (代入するアドレス > HB)

then 代入するアドレスをトレールスタックに積む．

UNDO:

トレールスタックに積まれたアドレスのメモリ内容を未定義の状態にする．

● アルゴリズム1 (カット無し)

未定義を表すタグ「未定義」を導入する．新レジスタGNR (generation number register) を用意し，タグは「未定義」とする．生成されるセルのスロットはGNRの値で初期化する．この値をセルの世代番号という．また，選択点にセーブされたGNRの値をその選択点の世代番号という．

レジスタ: GNR (タグは「未定義」).

セルを取るとき:

各スロットにGNRの値を入れる．すなわち

[2] この他にTRY_ME_ELSEなどがあるが，ここでの議論がそのまま適用できる．

[3] 実際のWAMでは本当に必要なときのみ行う．

(cons GNR GNR)

としてセルを作る.

TRY: GNR++.

選択点を作り, GNR をセーブ

TRUST:

最新の選択点を捨て, 更新する.

GNR ← 最新の選択点にセーブされた GNR.

TRAIL:

if (GNR > セルの世代番号)

then セルのアドレスと世代番号をトレールスタックに積む.

UNDO:

トレールスタックに積まれたアドレスと世代番号を使って元に戻す.

このアルゴリズムの正しさは WAM との対比により明らかであろう. すなわち, WAM では HB とのアドレス比較で最新の選択点以降に生成されたことを判定しているのに対し, ヒープトレールアルゴリズムでは構造データに明示的に世代番号を記録して判定している. GNR は HB と完全に同じように操作される.

なお, アルゴリズム 1 では, GNR は TRUST で必ず 1 減少するので, 選択点にセーブせずに, GNR−− とするだけでもよい. また, アルゴリズム 1 では TRAIL の > は ≠ でもよい.

● アルゴリズム 2 (カットあり)

カットは上からいくつかの選択点を削除し, 残った選択点でスタックの最も上にあるものを, 最新の選択点とする. これは TRUST に似ているが, TRUST 終了後には最新の選択点を含めてそれより小さい選択点の世代番号を持つセルだけがメモリ中に存在するのに対し, カットの実行後には, 最新の選択点より大きい世代番号を持つセルも存在する可能性がある. カット後に TRY により選択点を生成すると, カットされた選択点と同じ世代番号を持つ選択点が生成される. ところが, この世代番号の構造データは既に存在しているから, この構造データの生成時期を誤認してしまい, 結局必要なトレールを省いてしまうことになる. これを解決するには, 現在存在する可能性のある最大の世代番号を記憶しておき, 選択点を生成する際には, これより大きい世代番号をつければよい. このため, 過去の最大の GNR の値を保持するレジスタ MaxGNR を設ける. また, この場合には, TRAIL の判定は > でなければならない.

レジスタ: GNR, MaxGNR

セルを取るとき: アルゴリズム 1 に同じ.

TRY:

MaxGNR++.

GNR ← MaxGNR.

選択点を作り, GNR をセーブ.

TRUST:

最新の選択点を捨て, 更新する.

GNR ← 最新の選択点にセーブされた GNR.

MaxGNR ← GNR.

CUT:

指定された選択点を最新の選択点とする.

GNR ← 最新の選択点にセーブされた GNR.

TRAIL: アルゴリズム 1 に同じ.

UNDO: アルゴリズム 1 に同じ.

カットがない場合は, MaxGNR は GNR と同じ値を持つ. カットがあると, GNR は最新の選択点の値に戻されるが, MaxGNR は保持される. 次に TRY で選択点を作成するときには, カットが実行される前に存在した選択点の世代番号の最大のものの次の番号がつけられる.

例を示す. 次の 2 つのプログラムの述語 q の第 1 節でのユニフィケーションで, a を代入する直前の状態を図 3 に示す (図で UNDF(n) は世代番号 n の未定義値を表す).

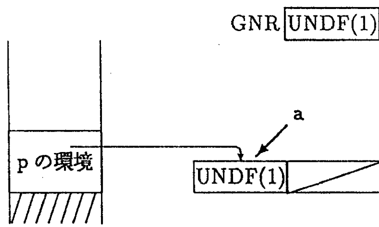
```
?-p(X).
p([A]):-q(A).
q(a).
```

プログラム (a)

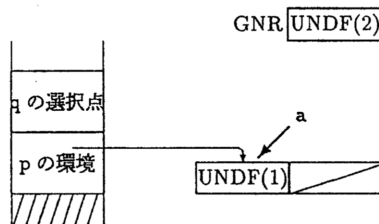
```
?-p(X).
p([A]):-q(A).
q(a).
q(b).
```

プログラム (b)

プログラム (a) の場合, q は節が 1 つしかないため決定的である. このため, q は選択点を作らないから, GNR は 1 のままである. 一方, セルの世代番号も 1 であるから, トレールは不要である. この後でバックトラックが



(a) トレールが不要な場合



(b) トレールが必要な場合

図 3: 世代番号によるトレール (UNDF(n) は世代番号 n の未定義値であることを表す．太矢印は代入を表す．)

起きたとしても，q と p は同時に取り消されるから，トレールが不要なことがわかる．

プログラム (b) では，q が非決定的であるから選択点を作る．このとき GNR は 2 となり，セルの世代番号よりも大きくなるから，トレールが必要となる．もし，この後でバックトラックが起きると，q の第 1 節の実行を取り消して，第 2 節を実行し，b を代入しなければならない．したがって，トレールが必要である．

4.2 いくつかの改良点

このアルゴリズムには次のような点で改良や修正が必要である．

● 世代番号の桁溢れ

TAO では関数からある述語が呼ばれ，それが成功して終了すると，その時点でのトレールと選択点がすべて消去される．しかし，ヒープはそのままである．ある述語が生成して関数に返した構造データを別の述語呼び出しに渡したとする．このときも，正しくトレールする必要があるから，世代番号は再利用できず，すべての異なる世代のデータには重複することなく世代番号を割り振らなければならない．結局，関数から述語を繰り返し呼ぶと，最大世代番号 MaxGNR は増加し続けることになり，最終的には桁溢れする．これについては，桁溢れした時点で，すべてのメモリをスキャンし，未定義値があったら，その世代番号を 0 にし，MaxGNR を 1 にリセットすればよい．実用的には MaxGNR があるスレッショルドを越えた最初の GC でこれを兼ねて行えば十分であろう．この操作の直後にはすべての構造データへの代入が必ずトレールされてしまい，一時的にトレールが増加するが，やがて正常な状態に近づいていく．

一般にはこれで十分であるが，TAO の GC は実時間 GC であるため，GC の最中にも世代番号が使われていく可能性がある．世代番号をリセットするようなモードで GC が並行動作している間は，MaxGNR のかわりに，世代番号 0 を使うなどの対策が必要である[4]．

● トレールの削減

ヒープトレールアルゴリズムでは多くの場合，WAM と同程度にまでトレール回数を削減できると予想されるが，1 回につき代入のアドレスと世代番号をトレールスタックに積まなければならない．したがって，トレールスタックは WAM の 2 倍消費することになる．これについては，世代番号を積まず，UNDO 時には世代番号を 0 にするという方法が考えられる．このようにすると，TRUST と CUT がある場合に無駄なトレールをする可能性がでてきてしまう．この影響とトレールスタックの各エントリが半分になることの効果はトレードオフの関係にあり，プログラムに依存する．これについては，大規模なプログラムをいくつか実行して統計をとる必要があり，今後の課題である．

● 並行プロセスへの対応

TAO のメモリモデルは共有メモリであり，あるプロセス上の述語が構造データを生成し，それを別のプロセスの述語にそのまま渡すことが可能である．システム全体を通して，すべての選択点に順に世代番号を割り当てていくため，アルゴリズム 2 では考慮していないような

[4]TAO が実装される CPU (SILENT) では，このチェックは他の命令の陰で実行可能である

世代番号を持つ構造データが渡されうる．このため，次の修正をする必要がある．

TRUST:

最新の選択点を捨て，更新する．

GNR ← 最新の選択点にセーブされたGNR.

TRAIL:

if (GNR ≠ セルの世代番号)

then アルゴリズム2と同じ．

MaxGNRレジスタはシステム全体で共通に使用しているため，自由に戻せなくなる．また，自分のプロセスで作成したデータであることがわかるのは世代番号が同じときだけなので，それ以外のときはすべてトレールするようにする．

TRAILの判定を ≠ としたことにより，並行プロセスとは関係のない，単一のプロセス上で実行する述語でもトレール量が増加してしまう． ＞という判定が必要であったのは，アルゴリズム2の項で述べたように，カットがあるからである．逆に，＞を ≠ にしたことにより，カットのあるプログラムにおけるトレール量が増えることになる．どの程度増加するかはプログラムに依存するため，一概に言うことはできないが，少なくとも，同じプログラムをTAOとPrologで記述した場合には，TAOの方がカットが少ない．これは，TAOでは，深いバックトラックを明示的に記述しなければならないからである．Prologでは，深いバックトラックが標準であるため，本質的に決定的な述語には，わざわざカットを入れなければならないが，TAOでは，その必要はない．このため，≠ による判定法でも，TAOではある程度のトレール削減の効果があると予想される．

4.3 WAMとの実行速度の比較

HBとGNRに関する扱いはWAMと完全に同じであり，異なるのはトレールの判定法とトレールのエントリが2倍になることである．トレールの判定については，WAMはレジスタ比較だけなので高速である．しかし，本方式でも，実はトレールするためには，代入先が未定義であることがわかっているはずであるから，そのときに世代番号を得ることが可能である．これを保持しさえすれば，WAMとまったく同じ速度で実行可能である．エントリの大きさについては，WAMの方が優れているが，プログラムによっては，トレールはヒープの5%程度にすぎないこともあるから，これが全体の実行速度にどの程度の影響を及ぼすかは実際のプログラムで測定しないとわからない．これは，上に述べた世代番号をトレールに記録しない方式と合わせて，今後の検討課題である．

5 おわりに

論理型言語において構造データをヒープに割り当てた場合に生ずる問題点と，トレール量を削減するヒープトレールアルゴリズムについて述べた． 本アルゴリズムにより，トレールの回数はWAMと同じにまで削減できる．トレールの各エントリはWAMの2倍となるが，全体ではかなりの削減が期待できる．

今後は，これまで述べてきた課題について検討を進める．ヒープトレールアルゴリズムはトレードオフのある選択肢(トレールに世代番号を積むか否かなど)があるため，WAMエミュレータ等を使って，大規模なプログラムについての統計情報を取る必要がある．

参考文献

[1] D.H.D.Warren: An Abstract Prolog Instruction Set, SRI Technical Note, October 1983.

[2] 山崎, 天海, 竹内, 吉田: TAO/SILENTの論理型プログラミング, 情報処理学会記号処理研究会, 64-1, March, 1992.

[3] 小型化PSI ESP説明書, 新世代コンピュータ技術開発機構 (1988).

[4] 山崎, 奥乃, 竹内: TAOにおける論理型プログラミングとその処理方式, 情報処理学会論文誌 Vol.32, No.9, 1990.

[5] E.Tick: Memory- and Buffer-Referencing Characteristics of a WAM-Based Prolog, The Journal of Logic Programming, Vol.11, pp.133-162, 1991.