# Variable Ages In A WAM Based System

Ilyas Cicekli
Dept. of Comp. Eng. and Info. Sc.,Bilkent University, 06533 Bilkent,Ankara,Turkey

## Abstract

We present a new method to represent variable bindings in the Warren Abstract Machine (WAM), so that ages of variable bindings can be easily found using this new representation. The age of a variable bound to a ground term will be the youngest choice point such that backtracking to that choice point can make that variable an unbound variable again. In other words, the age of a variable bound to a ground term will be the age of the youngest one of bindings causing that variable to be bound to that ground term. Variable ages are used in the process of figuring out backtracking points in an intelligent backtracking schema. We also introduce an algorithm to compute variable ages and justifications why it works.

# 1 Introduction

The Warren Abstract Machine (WAM) is an abstract machine for Prolog execution which consists an instruction set and several data areas on which instructions operate. The WAM is recognized as a breakthrough in the design of Prolog systems and other computational logic systems by the logic programming community. Many commercial [2, 12] and non-commercial Prolog systems based on the WAM are implemented during last decade. During discussions in this paper, we will assume that the reader has a familiarity with the WAM. Details of the WAM can be found in Warren's original paper [14] and Kaci's tutorial book on the WAM [1].

The backtracking method used in a standard Prolog implementation is known as *naive* backtracking. In *naive* backtracking, when a goal fails backtracking is done to the most recent choice point during that failure (last alternative) although this choice point may be nothing to do with that failure. In this approach, a lot of unnecessary backtrackings will be done while the same failure occurs many times. An intelligent backtracking method analyzes reasons of failures to choose proper choice points to avoid redundant backtrackings. The chosen choice point may not be the most recent choice point during that failure. In other words, alternatives of choice points between the most recent one and the chosen one are discarded without retrying them. If they were retried, the system would have reencountered with that same failure. Many intelligent backtracking schemes [3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15] are presented to avoid unnecessary backtracking steps. Early works in intelligent backtracking [3, 7, 11] are implemented as Prolog interpreters. Implementations of later works [5, 6, 8, 9] are WAM based systems.

Our intelligent backtracking schema whose some parts are presented in this paper is implemented as an extension of the WAM same as systems in [5, 8]. Our mechanism is similar to mechanisms used in those systems except that how we keep unification information for variable bindings and how we find reasons of failures. The mechanism we proposed is naturally integrated with the WAM architecture, and our performance

results are comparable with results with systems in [5, 8] even though we do not present our performance results here. We concentrate on a new representation for variable bindings in the WAM, so that ages of variables can be easily found using this new representation. The mechanism to find ages of variables causing failures is the central part of any intelligent backtracking schema.

In the following section, we will give a set of observations about WAM variables and their bindings. The correctness of the new representation for variable bindings in Section 5 depends on some of those observations. After procedure backtracking points which play important roles in the determination of variable ages are introduced in Section 3, we present unification graphs to make a concrete definition for variable ages for Prolog variables in Section 4. In Section 5, we give a new representation for bindings of WAM variables and present an algorithm to find ages of WAM variables bound to ground terms.

# 2 WAM Variables

A variable in the WAM is a chain of locations ending with an unbound location or a location holding a ground term. An unbound location holds a reference to itself, and a ground term can be a structure, a list or a constant.

A binding is an operation of putting a reference to a location into an unbound location. Locations can live on the heap or on the stack. If one of locations involved in a binding holds a ground term, a reference to that location is put into the other unbound location except when that ground term is a constant. In that case, that constant is directly copied into that unbound location. In a real implementation, references to structures and lists are tagged pointers. During binding of two unbound locations, following rules [1] are applied in the WAM.

**Binding Rule 1** A heap location is never bound to a stack location. If an unbound heap location and an unbound stack location are going to be bound, the stack location must point to the heap location.

**Binding Rule 2** If both of unbound locations live on the same data area (the stack or the heap), a younger (most recently created) location must point to an older (less recently created) location.

These rules not only avoid dangling references in memory areas but also improve efficiency by shrinking lengths of reference chains and avoiding unnecessary trailing operations. During binding of an unbound location, if that location is created before the most recent choice point, a trailing operation is performed for that location. *Trailing* in the WAM means that saving the address of a bound location which was older than the most recent choice point during the binding of that location. During a backtracking, a bound location can be unbound again if that location was bound after that backtracked choice point.

If we throughly analyze effects of binding rules above, we can deduct the following observations about variables in the WAM. Some of these observations will be referred

[1] These rules are satisfied by always binding a location of higher address to a location of lower address in the following WAM memory layout. The stack is allocated at higher addresses than the heap in the same global address space. The heap and the stack grows from lower addresses to higher addresses.

when the correctness of our new representation for variables is demonstrated in Section 5.

**Observation 2.1** *Stack locations can only occur as a subchain prefix in the reference chain of a variable.*

This observation says that stack locations in a variable are grouped as a subchain and occur before heap locations in that variable. In other words, stack locations will appear contiguously and early in the reference chain of a variable and will not mix with heap locations in that chain. This observation is an immediate consequence of the first binding rule.

**Observation 2.2** *Stack and heap locations in a variable are ordered from younger locations to older locations in terms of their creation time. This may only be violated by last two heap locations in the reference chain of a variable if the last heap location contains a ground term.*

This observation states that stack locations in a subchain prefix of a variable are ordered among themselves and heap locations are ordered among themselves. But it does not say that all stack locations are younger than heap locations in that variable. The reason that this condition may be violated by last two heap locations is that an unbound heap location is directly bound to another heap location containing a ground term during their binding operation. In fact, this is only true for locations containing lists and structures in the original WAM because constants themselves are copied into unbound locations.

From Observations 2.1 and 2.2, we can get the following observation about creation times of locations of variables in the WAM. If location $L_1$ is younger than location $L_2$, and $ct_{L_1}$ and $ct_{L_2}$ are creation times of $L_1$ and $L_2$, we will say that $ct_{L_1} < ct_{L_2}$.

**Observation 2.3** *Let $S_1, \ldots, S_n$ be stack locations and $H_1, \ldots, H_m$ be heap locations in the reference chain of a variable where $n \geq 0$, $m \geq 0$ and $n + m \geq 1$. Let $ct_{S_1}, \ldots, ct_{S_n}$ and $ct_{H_1}, \ldots, ct_{H_m}$ be their creation times, respectively. Following conditions hold for this variable.*

1. *The order of locations in the reference chain is $S_1, \ldots, S_n, H_1, \ldots, H_m$.*
2. *If this variable ends with an unbound location, $ct_{S_1} < \ldots < ct_{S_n}$ and $ct_{H_1} < \ldots < ct_{H_m}$ hold for this variable.*
3. *If this variable has two heap locations in its reference chain and its last location $H_m$ contains a ground term, $ct_{S_1} < \ldots < ct_{S_n}$ and $ct_{H_1} < \ldots < ct_{H_{m-1}}$ hold for this variable. The relation $ct_{H_{m-1}} < ct_{H_m}$ may or may not hold.*

When an unbound location $L_1$ is bound to another location $L_2$, if the creation time of $L_1$ is older than the most recent choice point, a trail entry for $L_1$ is created during binding of $L_1$ to $L_2$. So, $L_1$ can be unbound when a backtracking occurs to the choice point in question. If the creation time of $L_1$ is younger than the most recent choice point, no trail entry is created for that location. Thus, we will have two kinds of bindings, *trailed* and *untrailed* bindings. We will use notation $ct_{L_1 L_2}$ for the creation time of binding of $L_1$ to $L_2$.

**Observation 2.4** *Let $L_1, \ldots, L_n$ be locations of a variable, $ct_{L_1L_2} > \ldots > ct_{L_{n-1}L_n}$ holds for bindings in that variable. The only exception is that $ct_{L_1L_2} > ct_{L_2L_3}$ may not hold if binding of $L_1$ to $L_2$ is untrailed, and $L_1$ is a stack location.*

This observation guarantees that bindings in a variable will be sorted from older bindings to younger bindings. The reason that this observation holds is that variables are dereferenced by WAM unification instructions before binding operations. This means that an unbound location at the end of the reference chain of a variable is actually bound to another unbound location or a location containing a ground term. Of course, this binding will be younger than other bindings in that variable. Because variables are not dereferenced by instructions *get_variable* and *unify_variable*, we have the exception for the first binding. These instructions bind a new unbound location to the first location of a variable. Of course, this binding won't be younger than other bindings in that chain and won't be trailed.

**Observation 2.5** *If there are both untrailed and trailed bindings in a variable, all untrailed stack locations precede trailed stack locations and all untrailed heap locations precede trailed heap locations in the reference chain of that variable.*

Since the correctness of this observation may not be easily seen, we will try to give justification behind this observation instead of giving a formal proof. Let assume that we have just a single binding $binding_{L_1L_2}$ in the reference chain of a variable and we are going to bind $L_2$ to $L_3$. We will also assume that these locations live on the same data area. From observations on creations times of locations, we will have $ct_{L_1} < ct_{L_2}$. We can have following two cases.

1. If $binding_{L_1L_2}$ is untrailed, $binding_{L_2L_3}$ can be trailed or untrailed. In fact, $binding_{L_2L_3}$ will be trailed if $ct_{L_2}$ is older than the most recent choice point and it will be untrailed otherwise.

2. If $binding_{L_1L_2}$ is trailed, $binding_{L_2L_3}$ will be trailed too. Since $binding_{L_1L_2}$ is trailed, $ct_{L_1}$ must be older than the most recent choice point and $ct_{L_2}$ must be older too by Observation 2.3. So, $binding_{L_2L_3}$ will be trailed too.

Note that $binding_{L_2L_3}$ may be untrailed although $binding_{L_1L_2}$ is trailed if $L_1$ is a stack location and $L_2$ and $L_3$ are heap locations. In general, a variable will be in the following form if it contains both trailed stack and heap locations.

$$S_1 \xrightarrow{u} \cdots \xrightarrow{u} S_l \xrightarrow{t} \cdots \xrightarrow{t} S_n \xrightarrow{t} H_1 \xrightarrow{u} \cdots \xrightarrow{u} H_k \xrightarrow{t} \cdots \xrightarrow{t} H_m$$

where $S_1, \ldots, S_n$ are stack locations, $H_1, \ldots, H_m$ are heap locations, and $\xrightarrow{u}$ and $\xrightarrow{t}$ represent untrailed and trailed bindings, respectively.

# 3 Procedure Backtracking Points

In a regular Prolog system, a backtracking occurs to the most recent choice point when a failure occurs due to a unification failure. This failure occurs because a variable bound to a ground term cannot be unified with a different ground term or the unification algorithm cannot unify two variables bound to two different ground terms.

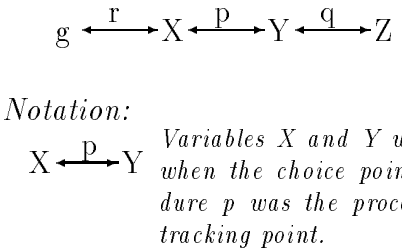
Since a variable bound to a ground term is responsible from that failure, the reoccurrence of that failure can only be avoided by backtracking to a choice point such that this variable won't be bound to that ground term when that same unification occurs. So, a backtracking to the most recent alternative in a regular Prolog system may not fix the problem and that same failure may occur again. Thus, we should backtrack to the youngest choice point such that the variable causing the failure can be unbound again. This choice point will be called as the reason of that failure. We can also avoid that failure by backtracking to first alternative of the clause in which that failure occurs. This means that we will completely skip the clause where the unification causing that failure occurs. Of course, the youngest one of these two points is going to be our backtracking point for that failure in an intelligent backtracking schema. Now, we will give a definition to formally describe one of these points.

**Definition 3.1 (Procedure Backtracking Point)** *The procedure backtracking point at a certain time of execution is the choice point of the current procedure in which we are or the choice point of its first ancestor which has a choice point.*

When a failure occurs, the system should backtrack to a choice point between the most recent one and the procedure backtracking point. The procedure backtracking point indicates the furthest choice point to which the system can backtrack during a failure. If the procedure backtracking point is equal to the most recent choice point, this is known as shallow backtracking. In that case, we have to try the most recent choice point. If the procedure backtracking point is not the most recent choice point, in that case the point that the system will backtrack depends on the reason of that failure. But, if the reason of that failure indicates a choice point which is older than the procedure backtracking point, we can only again backtrack up to the procedure backtracking point. By backtracking to the current procedure backtracking point, we completely skip the clause where the unification causing that failure occurs. In other words, we won't reencounter with that same failure because we are not going to execute that unification again.

We introduce a new register PB (Procedure Backtracking Point) to hold the procedure backtracking point in addition to registers in the original WAM architecture. The register PB will be updated with a new choice point when that choice point created by a *try* instruction. The old value of the register PB will be saved in the new choice point by that *try* instruction. During a backtracking, this register will be set to point at to the backtracked choice point. In fact, that choice point will be the most recent choice point at that time. The register PB is also restored from the saved value in a choice point when that choice point is discarded.

This PB register is saved in environments by *allocate* instructions same as saving the environment register E except that the register PB is not updated by *allocate* instruction. The reason saving it in environments is that it can be restored from the stored value in the current environment by *proceed* instructions. When a *proceed* instruction is executed, we get out from a context of a procedure and return back to the context of one of its ancestors. Since that ancestor has an environment and its procedure backtracking point is saved in its environment, the register PB can be restored from that value.




$ageset_{XY}$ is the set of labels of edges on the path from $X$ to $Y$.

$ageset_{Xg} = \{r\}$
$ageset_{Yg} = \{r, p\}$
$ageset_{Zg} = \{r, p, q\}$

Figure 1: A Unification Graph

# 4 Unification Graphs

We will introduce unification graphs to represent unifications of variables in a Prolog program. These graphs will be used to describe age sets of variable bindings via a single unification or a set of unifications. In this section, the discussion about variable ages will be given in terms of unification ages, and we will give new representation for bindings of WAM variables and discuss variable ages in binding ages of WAM variables in the following section.

A unification graph for a set of variables is a labeled undirected graph such that vertices of that graph are variables in that set and an edge represents the unification of variables indicated by two vertices. The label on an edge indicates the age of that unification.

**Definition 4.1 (Age of Unification)** *The age of a unification is the procedure backtracking point during that unification.*

The age of a unification indicates the choice point such that backtracking to that choice point can avoid the reoccurrence of that unification again. In other words, backtracking to the age of a unification will unbind a variable which was bound because of that unification. In fact, the alternative to which the system backtracks is the first alternative of the prolog clause containing that unification.

Figure 1 gives a unification graph for three variables $X$, $Y$, $Z$ and a ground term $g$. Labels on edges are names of procedures whose choice points are procedure backtracking points during those unifications. In that example, $X$ is unified with $Y$ and $g$ such that procedure backtracking points during these unifications are choice points of procedures $p$ and $r$, respectively. During the unification of $Y$ and $Z$, the choice point of procedure $q$ is the procedure backtracking point. Note that, all variables are bound to ground term $g$ as a result of three unifications in that graph. The reader should note that labels on edges don't reflect times of unifications. Three unifications given in the graph in Figure 1 can be performed in any order.

**Definition 4.2 (Age Set of Variable Binding)** *The age set of a binding of variable $X$ to another variable $Y$ (or a ground term) in a unification graph is the set of labels of edges on the path from $X$ to $Y$. Notation $ageset_{XY}$ will be used to describe the age set of binding of $X$ to $Y$.*

The age set of binding of X to Y is the set of all choice points such that backtracking to one of them will break the path from X to Y. In other words, X and Y won't be bound to each other after that backtracking. Figure 1 gives age sets of bindings of

variables in that graph to the ground term $g$. For example, backtracking to one of choice points of procedure $p, q$, and $r$ will break the link between $Z$ and $g$.

Although we are able to find age sets of bindings of any two variables, our main goal is to find age sets of bindings of variables to ground terms because those variables cause failures. The algorithm to find age set of two vertices in a unification graph is trivial. We find the path between these vertices and labels of edges in that path make the age set of those vertices.

**Definition 4.3 (Age of Variable Binding)** *The age of binding of variable $X$ to variable $Y$ is the youngest age in $ageset_{XY}$. Notation $age_{XY}$ will be used to describe the age of binding of $X$ to $Y$. We will also use the notation $age_X$ to describe the age of binding of $X$ to a ground term.*

The age of a variable bound to a ground term indicates the youngest choice point to which we should backtrack to unbind that variable from that ground term. So, if a failure occurs due to a variable bound to a ground term, the youngest age in the age set of their bindings gives the intelligent backtracking point for that failure.

# 5 Ages of WAM Variables

In this section, we will introduce required changes to the original WAM so that we can find ages of variables causing failures during unifications. The age of a variable is the first choice point which can avoid a failure occuring because of that variable. A variable in the WAM is a chain of locations ending with an unbound location or a ground term. Since ground terms cause failures during unifications, we will concentrate on reference chains ending with ground terms.

The age of a ground term is the procedure backtracking point at the time of the creation of locations for that ground term. We will use the age of a ground term when a failure occurs directly due to the location of a ground term. To find out ages of ground terms we put constants onto the heap same as structures and lists. Thus all ground terms in our system live on the heap. Each constant is put onto the heap during its first occurrence, and other variables are bound to the location of that constant without copying it into those variables. Putting constants onto the heap and not copying them into other locations help to find ages of constants. Since a same constant may end up at different locations during bindings in the original WAM, there is no way to calculate the age of a constant in that formalism. In our representation, there will be one value cell on the heap for a constant, and reference chains of variables bound to that constant will end with this value cell.

If an unbound location is bound to another location, this operation is called as binding. If a variable having a binding in its reference chain causes a failure because of that binding, we have to backtrack to the procedure backtracking point at the time of that binding. Note that the age of a binding may not be the most recent choice point at the time of that binding. Ages of bindings similar to bindings of variables X and Y in the following trivial example must be equal.

```
p :- X=1, q, Y=2, s(X,Y).   Ages of bindings of X to 1 and Y to 2
p.                          are equal, namely the choice point of p.
q.                          Note that most recent choice points are
q.                          different during these bindings.
```

Ages of both of those bindings must be the choice point of procedure p which is the procedure backtracking point during both of those bindings. Thus, if procedure s fails due to binding of Y to 2, we should backtrack to the choice point of p instead of the choice point of q.

When we talk about a new representation for variable bindings in the WAM, and the algorithm to find their ages, we will give new versions of definitions given in Section 4. These new definitions will be given in terms of ages of bindings of WAM locations.

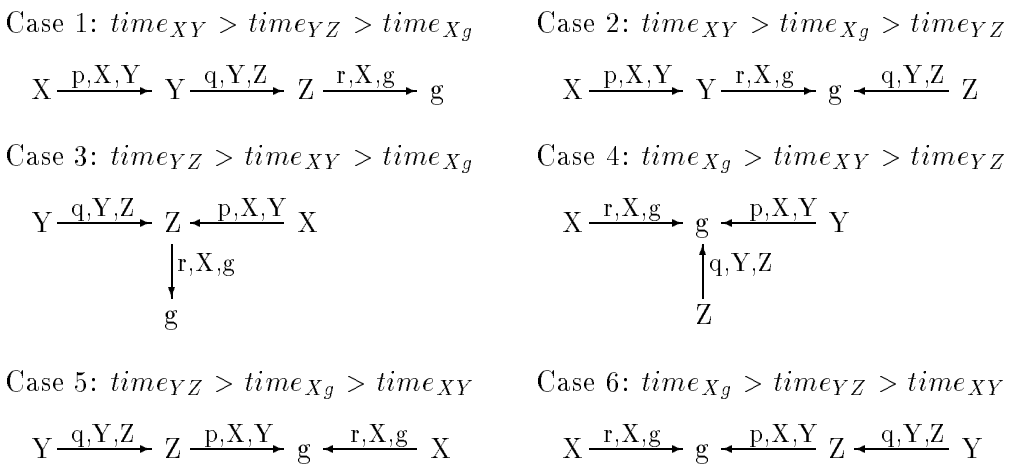
**Definition 5.1 (Age of Binding)** *The age of a binding is the procedure backtracking point in the register PB at the time of that binding.*

This definition corresponds to the definition of age of a unification in unification graphs. In fact, it defines the age of a binding which is a result of a unification. Of course, the age of that binding must be equal to the age of the unification causing that binding.

In the original WAM, when an unbound location which is protected by a choice point is bound to another location, the address of that unbound location is pushed onto the trail during that binding operation. This operation is known as trailing. Since we need ages of all bindings, we will use trail entries to store information about bindings in our representation. In our method, we will push five value cells onto the trail during binding of an unbound location to another location for each binding (trailed or untrailed). The unbound location is bound to the first value cell on the trail, and that value cell is bound to the location to which the unbound location would have been bound in the original WAM. The second position holds the address of the unbound location for untrailing purposes. The procedure backtracking point during that binding is saved in the third position. That saved value is the age of that binding and it may be different than the most recent choice point during that binding. The fourth and fifth positions hold first locations of variables whose unifications causing this binding.

In our new representation for variable bindings, we keep pointers to first locations of variables whose unifications causing those bindings. The question is how it can be guaranteed that pointers in a binding won't be dangling references after a certain time. This question is answered by Observation 2.4 in Section 2. That observation guarantees that a binding will be discarded before locations involved in the unification causing that binding are discarded.

A unification graph can map to different structures of WAM variables depending on times of unifications and creation times of locations for variables in that graph. Let assume that locations for $X, Y, Z$ and $g$ n Figure 1 are created in the reverse order (ie. creation order is $g, Z, Y$ and $X$) in the same data area. We will have six different structures for reference chains of variables depending on times of three unifications in that graph. Figure 2 gives reference chains for these six cases. In that figure, an arrow represents a binding, and a label on each arrow represents the age of that binding and first locations of variables causing that binding. For example, the arrow from $Z$ to $g$ with labels $q, Y, Z$ represents the binding of $Z$ to $g$. The age of that binding is $q$ and first locations of variables causing that binding are $Y$ and $Z$. Note that $q$ is also the age of unification of $Y$ with $Z$ in the corresponding unification graph in Figure 1. Although a unification graph can map different structures of reference chains of WAM variables, we have to calculate same ages for locations in those different representations.




*Notation:*

| | |
|---|---|
| L1 $\xrightarrow{\text{p,X,Y}}$ L2 | *Location L1 is bound to location L2 because of a unification such that the age of that unification is p and variables involved in that unification are X and Y.* |
| $time_{XY}$ : | *Time of the unification of X to Y. Relations $<$ and $>$ will be used to indicate younger and older relations among times.* |
| *Assumption* : | *Creation order for variables is g, Z, Y, and X and all of them live on the same data area (the heap).* |

Figure 2: Corresponding Reference Chains For A Unification Graph

**Definition 5.2 (Age Set of Binding of Two Locations)** *Let X and Y be two WAM locations and their reference chains join at some location. The age set of binding of X to Y (or Y to X) is a set of ages of bindings causing that junction. We will use the notation $ageset_{XY}$ to represent the age set of the binding of X and Y.*

**Definition 5.3 (Age of WAM Variable)** *Assume that X is a WAM variable whose reference chain ends with location G holding a ground term. The age of X is the youngest age in $ageset_{XG}$. We will use the notation $age_X$ to represent the age of X.*

These definitions are counter parts of Definitions 4.2 and 4.3 in Section 4 in terms of ages of bindings of WAM locations. If two locations whose reference chains end with a same location, they will have a junction location. At the worst case, this junction location will be the last location in their reference chains. For example, $X$ and $Y$ in Case 5 of Figure 2 has the location of $g$ as a junction location, but the junction location for $Y$ and $Z$ is also $Z$.

**Definition 5.4 (Junction Set)** *If X and Y have a junction location Z, their junction set is the set of all bindings from X to Z and from Y to Z. We will use the notation $junctionset_{XY}$ for the junction set of X and Y.*

Each item in a junction set is a pair representing a binding. The first element in that pair is also a pair of first locations of variables whose unification causing that binding. The second element in the binding pair is a singleton set of the age of that binding. For example, the junction set of $X$ and $Y$ in Case 5 of Figure 2 is equal to $\{((Y,Z),\{q\}),((X,Y),\{p\}),((X,g),\{r\})\}$, but the junction set of $Y$ and $Z$ is $\{((Y,Z),\{q\})\}$. In fact, a junction set is a special form of sets of bindings with age sets. A set of bindings with age sets is same as a junction set except that the

```
g ← location of the ground term at the end of the reference chain of V;
if g = V then age_V ← age_g
else begin
   notvisited ← junctionset_Vg;
   visited ← empty;
   done ← false ;
   while (not done ) do begin
      delete first node ((x, y), ageset_xy) from notvisited;
      if (x,y) = (V,g) then
         begin
            ageset_Vg ← ageset_xy;
            done ← true;
         end
      else begin
         for each ((w, z), ageset_wz) in junctionset_xy do
            if (w, z) ≠ (x, y) and (w, z) ∉ visited and (w, z) ∉ notvisited then
               add ((w, z), ageset_wz) to notvisited as last node;
         for each ((w, z), ageset_wz) in visited do
            if (x, y) and (w, z) share a common location then
               begin
                  let (b, c) be new pair where b, c are other locations in (x, y) and (w, z);
                  if (b, c) ≠ (x, y) and (b, c) ∉ visited and (b, c) ∉ notvisited then
                     add ((b, c), ageset_xy ∪ ageset_wz) to notvisited as last node;
               end;
         add ((x, y), ageset_xy) to visited as last node;
      end;
   end;
   age_V ← youngest in ageset_Vg;
end;
```

Figure 3: Algorithm to Find Age of Variable Bound To Ground Term

second element in a binding pair can be any non-empty set of ages. A binding age pair $((Y, Z), \{q\})$ means that $Y$ and $Z$ are bound to each other because of a unification whose age is $q$. In general, age set in a binding age pair will be ages of unifications causing binding of two locations in that pair.

Figure 3 gives an algorithm written in a pseudo code to find ages of variables bound to ground terms. In that algorithm, first we find the location $g$ of ground term at the end of reference chain of the given variable $V$. If $V$ is equal to $g$, then the age of the location of the ground term is the age of our location. Otherwise, we have to construct $ageset_{Vg}$ which is the set of ages of unifications causing the binding of $V$ to $g$ to find the age of $V$. We start from the junction set $junctionset_{Vg}$ of $V$ and $g$ to accomplish this task because at least, age of one of the bindings in $junctionset_{Vg}$ will be in $ageset_{Vg}$. We continue to enlarge our set of bindings with age sets by adding junction sets of variables of bindings currently in our set and new bindings constructed with joining two bindings in our set. Two bindings can be joined to get a new binding if they share

a common variable. In other words, if there are bindings $(x, y)$ and $(y, z)$, we create a new binding $(x, z)$. The age set of this new binding will be the union of the age sets of those bindings involved in that join operation. Bindings which has been considered are put into the set *visited* and the rest of bindings stay in the set *notvisited*. New bindings are added to *notvisited* if they have not been produced earlier. During the enlargement of the set, we check whether we find the binding of $V$ and $g$, or not. The enlargement stops when we find it. This algorithm is a breath first search in the search space of all possible bindings. Since the age of a variable is the youngest age in its age set, we compute the youngest age in its age set instead of computing the whole set in the actual implementation of this algorithm.

*Example :*

In this example, we will give enlargement steps while the algorithm tries to find the age of $Z$ in Case 5 of Figure 2. In each step, we will give sets *visited* and *notvisited* to indicate the current search space of bindings.

- Calculate $junctionset_{Zg}$ and assign it to *notvisited*.
  $notvisited = \{((X, Y), \{p\})\}$ $visited = \{\}$

- Visit $(X, Y)$ by adding $junctionset_{XY}$ and results of join operations of $((X, Y), \{p\})$ with all bindings in *visited* to *notvisited*.
  $notvisited = \{((Y, Z), \{q\}), ((X, g), \{r\})\}$ $visited = \{((X, Y), \{p\})\}$

- Visit $(Y, Z)$ by adding $junctionset_{YZ}$ and results of join operations of $((Y, Z), \{q\})$ with all bindings in *visited* to *notvisited*.
  $notvisited = \{((X, g), \{r\}), ((X, Z), \{p, q\})\}$
  $visited = \{((X, Y), \{p\}), ((Y, Z), \{q\})\}$

- Visit $(X, g)$ by adding $junctionset_{Xg}$ and results of join operations of $((X, g), \{r\})$ with all bindings in *visited* to *notvisited*.
  $notvisited = \{((X, Z), \{p, q\}), ((Y, g), \{p, r\})\}$
  $visited = \{((X, Y), \{p\}), ((Y, Z), \{q\}), ((X, g), \{r\})\}$

- Visit $(X, Z)$ by adding $junctionset_{XZ}$ and results of join operations of $((X, Z), \{p, q\})$ with all bindings in *visited* to *notvisited*.
  $notvisited = \{((Y, g), \{p, r\}), ((Z, g), \{p, q, r\})\}$
  $visited = \{((X, Y), \{p\}), ((Y, Z), \{q\}), ((X, g), \{r\}), ((X, Z), \{p, q\})\}$

- Visit $(Y, g)$ by adding $junctionset_{Yg}$ and results of join operations of $((Y, g), \{p, r\})$ with all bindings in *visited* to *notvisited*.
  $notvisited = \{((Z, g), \{p, q, r\})\}$
  $visited = \{((X, Y), \{p\}), ((Y, Z), \{q\}), ((X, g), \{r\}), ((X, Z), \{p, q\}),$
  $((Y, g), \{p, r\})\}$

- Visit $(Z, g)$. We found $ageset_{Zg}$.
  $ageset_{Zg} = \{p, q, r\}$

- Choose the youngest age in $ageset_{Zg}$ as $age_Z$.

To optimize our algorithm, we can stop the search immediately after the binding $((Z, g), \{p, q, r\})$ is found during the visit to the binding $((X, Z), \{p, q\})$.

# 6 Conclusion

The new method for representations of bindings of WAM variables plays an important role in the process of finding ages of variables bound to ground terms. Age of a variable and the procedure backtracking point introduced here determine backtracking point during a failure due that variable. This new representation presented here is smoothly integrated with the original WAM architecture with small overheads. Since most of overheads of our mechanism occur during a failure not during unification process, it is suitable for an intelligent backtracking schema.

In our mechanism, we keep unification information for all bindings. The space overhead resulted from this is not so ignorable. If could have accomplished this task by just keeping information for certain bindings, it would have been much desirable result. We are currently investigating this possibility.

# References


[1] Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, 1991.

[2] *ALS Prolog Manual*, Applied Logic Systems. Inc., 1992.

[3] Bruynooghe M. and Pereiara L. M., *Deduction Revision by Intelligent Backtracking*, in Implementations of Prolog, ed. Cambell J. A., Ellis Horwood, 1984.

[4] Chang J.-H. and Despain A. M., *Semi-Intelligent Backtracking of Prolog Based on a Static Data Dependency Analysis*, Proc. of $2^{nd}$ Int. Symp. on Logic Programming, Boston, 1985.

[5] Codognet P., Codognet C. and Filè G., *Yet Another Intelligent Backtracking Method*, Proc. of $5^{th}$ Int. Conf. and Symp. on Logic Programming, Seattle, 1988.

[6] Codognet P. and Sola T., *Extending the WAM for Intelligent Backtracking*, Proc. of $8^{th}$ Int. Conf. on Logic Programming, Paris, 1991.

[7] Cox P. and Pietrzylowski T., *Deduction Plans: A Basis for Intelligent Backtracking*, IEEE PAMI, Vol 3, 1981.

[8] Lin Y-J. and Kumar V., *An Intelligent Backtracking Schema for Prolog*, Proc. of $4^{th}$ Int. Symp. on Logic Programming, San Francisco, 1987.

[9] Lin Y-J. and Kumar V., *A Data-Dependency Based Intelligent Backtracking Schema for Prolog*, J. Logic Programming, Vol 4, 1988.

[10] Matwin S. and Pietrzylowski T., *Intelligent Backtracking in Plan Based Deduction*, IEEE PAMI, Vol 7, 1985.

[11] Pereiara L. M. and Porto A., *Selective Backtracking*, in Logic Programming, ed. Clark K. L. and Tarnlund S.-A., Academic Press, 1982.

[12] *Quintus Prolog Reference Manual*, Quintus Computer Systems Ltd., 1991.

[13] Toh J. and Ramamohanrao K., *Failure Directed Backtracking*, Tech. Report 86/9, CS Dept., Univ. of Melbourne, Australia, 1986.

[14] Warren, D.H.D., *An Abstract Prolog Instruction Set*, SRI Technical Report 309, 1983.

[15] Woo N. and Choe K., *Selecting The Backtrack Literal in The AND/OR Process Model*, Proc. of $3^{rd}$ Int. Symp. on Logic Programming, Salt Lake City, 1986.