On Taylor's Scheme for Unbound Variables

Thomas Lindgren       Per Mildner
Johan Bevemyr
Computing Science Department, Uppsala University

Abstract

A novel scheme for representing unbound variables was proposed by Taylor: that aliased unbound variables are represented as circular chains of cells. Unfortunately, few details were given on its implementation and the involved trade-offs.

We compare the proposed scheme to that of Warren's abstract machine, and find that Taylor’s scheme (a) leads to increased trailing, (b) makes stack allocation more difficult, (c) requires more expensive copying of heap cells at times, and (d) has a more expensive variable binding mechanism. On the positive side, a very strong advantage is that Taylor’s scheme entirely dispenses with pointer chain dereferencing.

We conclude by enumerating the data flow properties a global analyzer should find in order to eliminate these difficulties.

1 Introduction

Taylor presented an intriguing representation of unbound logic variables in his thesis [5]. The standard WAM scheme [9] represents variables as pointers and unbound variables as self-pointers. Variable binding means dereferencing the pointer chain until an unbound variable appears and performing a trailed assignment (i.e., which is reversible on backtracking).

The drawback of this approach appears when we consider the case once such a pointer-chain is bound to a non-variable term, which is possibly stashed in a datastructure, passed around, returned, etc. The execution machinery must be ready to handle pointer-chain dereferencing at most points when the value of a variable is sought.
**Previous work.** Some native code compilers attempt to remove pointer dereferencing operations (e.g., Van Roy's Aquarius Prolog compiler [7]). The Aquarius Prolog compiler takes a conservative stance: once a dereferencing chain may have appeared in a term, every subterm needs dereferencing. Van Roy and Despain [7] found that, on a range of medium-sized benchmarks, 17% of the predicate arguments were recursively dereferenced, and a further 23% were uninitialized variables.

Taylor considered a more powerful analysis to remove dereferencing operations [4], which found that only 8–33% of the predicate arguments of a small set of programs required general dereferencing. However, he subsequently abandoned that approach in favor of that examined in this paper. The reason may have been the difficulty to avoid conservative assumptions, since dereferencing chains appear on a much lower level than the analyzed Prolog code. For instance, the analysis had to make assumptions on low-level execution engine issues to decide some analysis properties. Such assumptions must then be maintained throughout compilation. A second reason may be that module boundaries again introduce conservative assumptions, since nothing is known about the dereferencing chains in the input arguments.

**The cost of dereferencing operations.** Measurements show that most dereferencing chains are short, zero or one step in length [6], which means most dereferencing operations are trivial. In a native code implementation, it is vexing to emit dereferencing loops at many points: it increases compile times, it pollutes the instruction cache with unnecessary instructions and it may reduce the efficiency of low-level routines.

Consider the typical dereferencing loop in the WAM. These are frequently expressed as while-loops; we have performed the usual optimizations on such loops to expose the anticipated code layout.

```plaintext
if (!IsRef(X)) goto done; /* unknown */

loop:
    X1 = Next(X);
    if (X1 == X) goto done; /* unknown */
    X = X1;
    if (IsRef(X)) goto loop; /* not taken */

done:
...
```

Note that this code may have poor branch behavior: the two forward branches are predicted as not taken, which is not true if the chain has length zero or points to an unbound variable; the backward loop branch is predicted as taken, again mistakenly if the chain has length one.
Unfortunately, as we saw before, dereferencing chains often have length zero or one. (Native code compilers may attempt to rectify this situation by improving code layout or setting branch hint bits.)

Furthermore, the useless branches will occupy space in the hardware branch prediction tables, dilutes the information available when using correlated branch prediction and fills the I-cache with some useless operations (the parts after the forward branch in the loop are usually never needed).

2 Taylor's scheme
Taylor instead proposed a simpler scheme than that in the WAM, the following.

- Unbound variables are initially represented as self-pointers.
- When variable is bound to variable, they are linked together to form a circular chain of two variables; continuing this process means the chains can have arbitrary length.
- When variable is bound to nonvariable, all the variables of the chain are bound to the nonvariable. This is shown in Figure 1.

Unfortunately, there was little discussion on the advantages and drawbacks of this representation. We provide such a discussion below.

2.1 Implementation
We provide some missing links on to how to implement the variable binding operations. We shall not be rigorous in the following treatment, but rather sketch operations and use suggestive names. The reader is invited to fill in the details.

2.1.1 Variable-nonvariable binding
Binding a variable X to a non-variable T requires the following loop.

```c
curr = X;
do {
    Trail(curr,T);
    curr = Next(curr);
} while (curr != X);
```

We assume that \texttt{Trail}(X,Y) trails X as needed, then writes Y to that location, and that \texttt{Next()} follows the pointer chain one step.

```c
#define Trail(addr) 0
#define Trail(addr) \
    if (MustTrail(addr)) \
        { tp[0] = addr; tp[1] = *addr; tp += 2 }
#define Next(x) *(UntagThe(REF,(x)))
```
Figure 1: Binding a chain of variables X to a nonvariable Nonvar
2.1.2 Variable-variable binding When variables $X$ and $Y$ are bound to each other, the following constant-time operation seems natural. It is shown in Figure 2.

```plaintext
t1 = Next(X);
t2 = Next(Y);
TrailedStore(X,t2);
TrailedStore(Y,t1);
```

However, there is a snag. Consider when $X$ and $Y$ are aliased to each other; more precisely, the case shown in Figure 3. Instead of the binding joining two chains into one, the one chain is split into two.

Thus, the general case of binding variables $X$ and $Y$ to each other is the following.

```plaintext
curr = X;
do {
   if (curr == Y) goto Aliased;
   curr = Next(curr)
} while (curr != X);
Bind(X,Y); /* as above */
```

Aliased:
```
...
```

If $X$ and $Y$ already are bound to each other, nothing should be done. Otherwise, a normal binding is performed.

A second drawback is when a variable $X$ is repeatedly bound to (unaliased) unbound variables: $X$ may be trailed repeatedly. This problem is similar to that of unnecessary trailing of updates to constraints or terms in some Prolog implementations, and can be solved by similar means – time stamping – at an extra cost. We are leery of this solution, since it has serious repercussions when applied to logic variables (e.g., it may require all objects to occupy two words rather than one, or that variables never appear inside structures; both of these solutions have high memory costs). Note that if a choice point is created in between of two trailed stores to $X$, the second trailing is of course still needed.

We may simply ignore this problem and reasonably hope that it does not occur much in practice; the garbage collector can short out unnecessary trailing if needed.

2.1.3 Copying references The next difficulty occurs in the following situation. Consider the second clause of the `append/3` predicate.

```prolog
append(A,B,C) :- A = [X|Xs], C = [X|Zs], append(Xs,B,Zs).
```
Figure 2: Binding variables A and B in constant time.
Figure 3: Variable binding when A and B are aliased may go wrong.
Assume that A is a list, and C is unbound. Now, there are two cases. If X is bound, then C can simply copy X:

\[
\begin{align*}
C &= \text{REF}(hp+0); \\
\text{hp}[0] &= \text{STR}("\cdot\cdot\cdot",2); \\
\text{hp}[1] &= X; \\
\text{hp}[2] &= \text{REF}(hp+2); \\
\text{hp} &= 3;
\end{align*}
\]

But this code is invalid when X is unbound – in that case, the new occurrence of X would not be part of the variable chain. Instead, we must generate this code when X is unbound:

\[
\begin{align*}
\ldots \\
\text{TrailedStore}(X,\text{REF}(hp+1)); \\
\text{hp}[1] &= \text{REF}(X); \\
\ldots
\end{align*}
\]

When the mode of X is unknown, we get the following code.

\[
\begin{align*}
\text{if } &\left(\text{Unbound}(X)\right) \{ \\
&\quad \text{TrailedStore}(X,\text{REF}(hp+1)); \\
&\quad \text{hp}[1] = \text{REF}(X); \\
&\} \text{ else } /* X bound */ \{ \\
&\quad \text{hp}[1] = X; \\
&\}
\end{align*}
\]

This case is similar to when unbound variables are represented with a special ‘unbound’ tag. In the WAM, a simple copy (as for the bound case) is sufficient whatever the mode of X.

2.1.4 Stack allocation Stack allocation is traditionally a tricky subject in the WAM. Using Taylor’s scheme, the main difficulty is that pointers naturally appear into the stack. These pointers can originate from three sources: from the heap, the stack or from registers.

Heap-stack bindings. When a stack variable is bound to a heap variable, the heap variable points into the stack. Let us assume we wish to deallocate the stack frame; in this case, the reference into the stack must disappear, which we do by (a) traversing the chain of the stack variable, and (b) unlinking the stack variable. (We shall return to this topic.)

The next potential problem is whether there may be trailed bindings to deallocated environments. Consider the situation shown in Figure 4.

Assume there is one heap variable X and one stack variable Y. X is older than the topmost choice point, so bindings to it must be trailed. (If X and
Figure 4: Possibility of trailed binding into stack
Y have the same age, there is no problem.) The environment where Y is allocated is called E.

1. First, both X and Y are unbound.
2. Then X and Y are unified, and bound together. This means X is trailed, indicated by the arrow from the lower right corner.
3. Then, X is bound to a nonvariable term, Term. X is trailed again, while Y is not.
4. The environment is deallocated, e.g., due to being a last call.
5. The computation fails, and the binding of X is undone.

The question is now: does X point into the stack? Two trailings have been done. The older one, T1, resets X to the state of (1), the younger one, T2, to the state of (2).

We say that each choice point has an associated trail segment, the trail entries that will be undone when we fail to the choice point. When a new choice point is created, it gets a new trail segment. When a cut occurs, the trail segments of the cut-away choice points are merged into that of the youngest surviving choice point.

The following reasoning is fairly tricky, but ensures us of correctness.

If E still exists when (5) is reached, there is no problem: X will point either to itself or to Y after untrailing, and Y still exists.

If E does not exist at (5), then a dangling pointer will occur if only T2 is performed. Otherwise, both untrailings are done, and X is reset to point at itself as in (1).

Thus, we want to ensure that T1 and T2 are in the same trail segment whenever E does not exist. In that case, T1 will be untrailed whenever T2 is untrailed. The cases are as follows in state (5).

1. A choice point younger than E exists. Then, E cannot be deallocated.
2. A choice point younger than E existed, but was cut away. Thus, T1 and T2 must belong to the same trail segment due to trail segment merging after the cut.
3. No choice point younger than E has existed. In this case, T1 and T2 belong to the same trail segment by default.

Hence, heap-stack pointers are not a problem.
Register-stack pointers. The next case is register-stack pointers, which occur through `put.y_variable` or `put.y_value` WAM instructions. It is unreasonable to scan the registers whenever an environment is trimmed or deallocated, and so we choose to disallow registers pointing into the stack.

When a pointer to a stack variable is put in a register, there are two cases: either the stack variable is chained to a heap variable, in which case we would prefer to point to the heap variable; or it is not chained to a heap variable. In this case, we globalize the variable, i.e., link it to a freshly created heap variable and point to the heap variable instead.

One may consider mandatory globalization when a pointer to a stack variable is found, since we then avoid chasing pointers through the stack to see if there is a heap variable to point at. On the other hand, we run the risk of unnecessary allocation.

Stack-stack pointers. The final case is when there are pointers inside the stack between variables. In the WAM, pointers from older to younger environments can be avoided by ordering bindings; Taylor’s scheme is forced to make such bindings when stack variables are unified.

This turns out not to be a problem in practice: the implementation can use the same machinery as for heap-stack bindings, i.e., globalization when a variable is trimmed or deallocated.

We note that a potential pitfall exists: if there are \( n \) stack variables unified with each other, and they are all about to die, then we must take care to unlink them all at once.

If we unlink one stack variable at a time, then the chain of \( n \) gradually shrinks to length \( n - 1, n - 2, \ldots \) and so on. We perform \( n \) such unlinking operations, which in total takes time quadratic in \( n \). Instead, the implementation should short out all stack variables in a single pass over the chain.

(Nota that if each of the \( n \) variables is bound to a separate chain of \( n \) variables that survive the call, then deallocation time is again quadratic – but so is the size of the involved data. In the previous case, the data size is linear in \( n \).)

Stack variables are used to hold local results over a procedure call, in particular in order to store the results of a predicate. We note that this role can be subsumed by returning such (uninitialized) arguments in registers instead [8].

Thus, the number of unbound stack variables that appear throughout execution can be substantially reduced.

2.1.5 Avoiding deallocation The problems detailed above disappear if we move to a heap-based implementation, where ‘stack frames’ are allocated on the heap and unbound variables are moved out of dead stack frames by the garbage collector [3].
A similar approach is to allocate all stack variables on the heap and ensure that stack references to these variables are handled properly [8].

2.2 Ordering variables.
Applications occasionally demand a global and total ordering of unbound variables using $\leq$, $\Rightarrow$ and other similar primitives. The conventional implementation of the WAM uses a compacting collector, which ensures that the relative positions of variables remains the same throughout execution. The total ordering is then given by comparing their addresses.

This feature is made less useful when we consider copying collectors, such as the Bevemyr-Lindgren generational top-down collector [1], where variable order is not maintained over collections. In this case, there are some options in the WAM:

- Put ordered variables on a stack that is collected using compaction and so retains the desired ordering [1].
- Attach an ordering number to each ordered variable.
- When variable X is ordered, create an ordering structure $\text{ord}(T,Y)$ on the heap, where $T$ is an ordering number and $Y$ is a fresh unbound variable, and bind X to Y. The runtime system must take care to retain the enclosing $\text{ord}$ structure on garbage collection or when subsequent ordering operations are performed. (This method is apparently used in the Demoen-Engels bottom-up collector, as appears in BinProlog v. 4.00.)

When we use circular chains of variables, we can adapt the above solutions to our needs.

- If we use a compacting collector, locate the oldest variable in the chain.
- If we use a copying collector, we can put an ‘ordering item’ in the variable chain. Comparison and binding operations must take care to update the ordering item properly. When ordered variables are bound to each other, the ordering items must be merged.
- Ordering items can be represented as special unbound variables. This means variable binding is not affected by ordered variables.

2.3 Using global analysis to strength-reduce operations
While the general variable binding scheme is quite elaborate, there are ample opportunities to reduce the complexity of binding operations when global information is given. Likewise, we can strength-reduce variable trimming and other operations using global analysis.

The following properties are of interest.
What is the mode of the copied subterm X (nonvar/unbound/unknown) ?

When X is bound to T, is T bound (yes/no/maybe)? Does X have a variable chain (yes/no/maybe)? Does X require trailing (yes/no/maybe)?

When X is bound to Y, and Y is unbound, are they aliased (yes/no/maybe)? Does Y require trailing? Does Y have a variable chain?

When X is bound, has X been bound previously (without an intervening choicepoint creation) (yes/no/maybe)?

Is X bound when it may be trimmed (yes/no/maybe)? Is X part of a variable chain (yes/no/maybe)?

Is either or both of X and Y ordered variables (yes/no/maybe)?

Answers to each of these questions allow us to remove the difficult aspects of variable binding operations, copying heap cells and trimming the stack (if the implementation has one).

2.4 Discussion

Drawbacks.

- In the WAM, younger variables are bound to older ones, which reduces trailing. In Taylor’s scheme, both variables are bound to each other, resulting in an extra trailing in some cases.

- When a variable is repeatedly bound to other variables, it may be trailed unnecessarily after the first binding.

- Copying references entails extra work; in particular if an unbound variable is the object referenced.

- Ordering variables entails (some) extra work, but this may be the case in a WAM implementation as well.

Advantages. There is only one advantage, but it’s a good one.

- Dereferencing operations become trivial.

Since pointer dereferencing is pervasive in Prolog, much more so than variable binding, the common case of pointer dereferencing is made cheap at the cost of the less common variable binding operation.
3 Conclusion
Taylor’s variable representation provides an intriguing alternative to Warren’s original representation. We have enumerated its drawbacks.

- Trailing increases when variable-variable bindings occur, as compared to WAM.
- The general case of variable-variable binding is more expensive than in WAM.
- Copying references is more involved than in WAM.
- Variable ordering operations require some extra effort. However, most of this also occurs in a WAM.

Since variable-variable binding occurs rarely in Prolog as compared to pointer-chain dereferencing, and most of the drawbacks are amenable to global analysis, we find that Taylor’s variable representation is suitable when the compiler can perform reasonably precise global analysis. The question remains open whether the representation is efficient enough when analysis results are lacking. Taylor indicates some promising results in his thesis, but that evaluation is insufficient to draw strong conclusions.

References