THE OCCUR-CHECK PROBLEM REVISITED

JOACHIM BEER

A method is presented for executing PROLOG programs which avoids almost all unnecessary occur-checks. The method is based on a dynamic classification of the context in which logical variables occur. No static global analysis of the PROLOG program is required to detect the places where an occur-check has to be made. The presented method has also an important side benefit. It considerably cuts down on the number of memory references during the execution of PROLOG programs. Furthermore, in most cases it avoids "trailing" and "untrailing" of unbound variables altogether. Due to this fact the employed method actually speeds up PROLOG execution. The method is discussed in terms of an actual implementation based on the Warren abstract PROLOG instruction set. However, the method should be applicable to other implementation models as well. No assumptions are made with respect to particular hardware.

1. INTRODUCTION

In the last few years the programming language PROLOG has gained considerable acceptance as a programming language based on the logic-programming paradigm [1]. A better understanding of the language implementation issues resulted in important breakthroughs in implementation techniques and helped to make PROLOG a viable alternative to other AI languages (especially LISP). Whereas older, interpreter based PROLOG implementations were plagued by slow execution and an extraordinary appetite for memory space, current PROLOG implementations provide compilers (either as standalone compilers or incorporated in an interpreter system) to speed up execution.

One advantage of logic programming languages is their clean and simple semantic model. However, most PROLOG implementations, for efficiency reasons, deviate
from the semantic model of first-order formal logic. It is, for example, well known that PROLOG's search strategy is logically incomplete; however, this will not be the subject of the following discussion. Another point that has been sacrificed for efficiency's sake is the unification with occur-check. Most existing PROLOG implementations use a unification without occur-check. This means that a variable \( X \) can be unified with a term \( t \) even if \( X \) occurs within the term \( t \). For example it is possible to unify \( X \) with \( t(X) \), resulting in the cyclic structure \( t(t(t(t(\ldots)))) \).

Plaisted [3] showed how it is possible to write formally correct PROLOG programs using first-order predicate-calculus semantics and yet derive nonsense results such as \( 3 < 2 \).

One way to deal with this problem is to turn a bug into a feature. Colmerauer [2] developed a formal semantics of PROLOG using infinite trees; this allows terms with loops to be regarded as infinite trees. However, this is not the standard semantics any longer, and there are applications that require the semantic model of first-order predicate calculus, i.e. unification with occur-check.

It should be obvious why the occur-check is rarely implemented. Every time a variable is bound to a term, the term would have to be traversed to detect whether the variable is already part of the term. This, of course, constitutes a tremendous overhead. However, in most cases the occur-check can safely be omitted, since PROLOG programs are ordinarily written in such a way as not to give rise to cyclic structures.

The question arises how to detect those places where the occur-check can be safely omitted and where an occur-check has to be made. To do so it helps to know that the occur-check is only necessary in those situations where a clause head literal and a clause body literal have more than one occurrence of any variable [3]. This is complicated by the fact that variables might be aliased. Therefore it is not sufficient to just look for literals in which a particular variable name occurs more than once. In his paper Plaisted proposed a static global analysis of the PROLOG program to detect the places where loops may be created and added appropriate checks at those places.

The actual algorithm for the global analysis proposed by Plaisted is fairly complicated, and the reader is referred to [3]. In a nutshell the method is based on a worst-case analysis of the PROLOG program. A set of instances of each clause which can be generated by any execution of the PROLOG program is created. For the top-level goal statement the user has to provide a mode declaration to get the iteration process started. From this set of clause instances a bipartite graph is created which represents the set of "calling-literal-called-literal" pairs for which loops may be created and appropriate tests are added.

Another alternative would be to have the user indicate the places where an occur-check should be executed: an unsatisfactory solution, since it puts the burden on the user and is very error prone. The method presented in this paper is based on a fine-grained differentiation of the context in which the variables of a given clause occur. The method does not require a global analysis of the PROLOG program and should be applicable for most implementation models (structure sharing models, structure copying models, etc.). It requires only the analysis of variable occurrences within a given clause—something every compiler has to do anyway. PROLOG implementations—be they compiler or interpreter based systems—use data tags or descriptors to identify the objects to be unified. Those data types are 'atom',
integer', 'structure', etc.; however, unbound logical variables are generally only
tagged as 'unbound-variable'. No further information is provided as to the context
in which the variable occurs. This is where the scheme presented in this paper differs
from other implementations. Through the use of further tags to identify unbound
variables and the context in which they occur, it is possible to avoid unnecessary
occur-checks. Only one extra data tag will do the trick. However, this data tag will
be interpreted differently by different instructions, thereby establishing a compre-
hensive way to express the context in which variables might occur.

The presented method has also an important side benefit. It considerably cuts
down on the number of memory references during the execution of PROLOG
programs. Furthermore, in most cases it avoids “trailing” and “untrailing” of
unbound variables altogether. Due to this fact, the employed method actually
speeds up PROLOG execution. The method will be discussed in terms of an actual
implementation based on the Warren abstract PROLOG machine [4].

2. COMPILING PROLOG

The following discussion is based on the Warren abstract PROLOG machine
(WAM). For those readers not familiar with the WAM, I will now give a short
overview of the abstract PROLOG instruction set. The discussion will necessarily be
brief, and the interested reader is referred to [4]. A quick overview of the principles
of compiling PROLOG programs will be helpful in understanding the instruction
set.

General unification algorithms deal with the problem of unifying two (or more)
unknown terms; however, when unifying a goal with a clause header the structure of
the header is, of course, known. Consider the case in which the following goal and
clause-head literals must be unified:

Goal literal h(X, Y, Z)
Clause-head literal h(U,[V|W],[a]).

The structure of the goal arguments is not known at compile time, since the
variables might become instantiated to just about anything at run time. The
structure of the header, on the other hand, has been completely specified when the
program was created. (The first argument is an unbound variable, the second
argument is a list, the list header as well as the list tail are unbound variables, and
the last argument is a list with one element, the constant a.)

Given this structural information we can generate code for a unification al-
gorithm that can unify only terms that have the very same structure as the
clause-head literal, but can do that very efficiently. In principle, what is being done
when PROLOG programs are compiled is that code for a highly specialized
unification algorithm is being generated for every clause head. In addition to the
unification code for the clause-head literal, the compiler will also generate code for
the memory management of the PROLOG system as well as in-line code for
procedure calls.

Since it would be too lengthy to cover every instruction in detail, we will restrict
ourselves to a global overview of the instruction set. The PROLOG instructions can
be grouped into five main groups according to their functionality.
(1) **Control instructions:** The instructions in this group are the PROLOG specific low-level memory-management and control-flow instructions. The instructions are Allocate, Deallocate, Proceed, Execute, and Call.

(2) **Goal instructions:** The instructions in this group serve to set the argument registers prior to a subgoal invocation, i.e., they are responsible for passing the actual arguments of a goal literal to the called procedure. In the Warren model they are denoted as ‘put ...’ instructions. The instructions are: put_const, put_structure, put_list, put_variable, and put_value; they load, respectively, a constant, a structure pointer, a list pointer, a pointer to an unbound variable and the value of a bound variable into a designated argument register. (There are some additional instructions covering special cases that are of no interest for our discussion.)

(3) **Clause-head instructions:** The instructions in this group unify the formal arguments of the clause-head literal with the actual parameters of the calling goal. In the Warren model they are denoted as ‘get ...’ instructions. These instructions mirror the structure and data types of the arguments of the clause-head literal. The instructions are get_const, get_structure, get_list, get_variable, get_value; they attempt to match (unify) the actual goal arguments with a constant, a structure, a list, an unbound variable, and a bound variable, respectively. The instructions that represent structured data objects (structure or list) occurring in the clause-head literal can be executed in one of two modes (READ or WRITE mode) depending on the type of the actual argument they are matched with. Consider the following goal and clause head literal that are to be unified:

<table>
<thead>
<tr>
<th>Goal literal</th>
<th>Clause-head literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>g(X)</td>
<td>g(s(a)) :- ...</td>
</tr>
</tbody>
</table>

If the actual parameter X is still an uninstantiated variable, the structure s(a) will have to be created, and X will be bound to this newly created structure. On the other hand, if X has already been bound to some structure, the structure s(a) will be matched only against this structure and no new structure s(a) will be created. The actual creation or unification of the structure's arguments will be done by the instructions of the next group.

(4) **Structure instructions:** The instructions in this group process the components of structured data objects. In the Warren model they are denoted as ‘unify ...’ instructions, a somewhat misleading term. The instructions are unify_const, unify_variable, unify_value; they represent a constant, an unbound variable, and a bound variable, respectively, that occur as arguments of a structure or list. A sequence of structure instructions must always be preceded by an instruction (put_structure, put_list, get_structure, get_list) which establishes the type of the structure to be worked on as well as the execution mode (READ/WRITE mode). Structures that occur as arguments of goal literals will always be created; hence, the instructions put_structure and put_list will always set the WRITE mode. The following ‘unify’ instructions will then create the structure or list. (Again, there are some other instructions in this group that are of no interest for our discussion).
(5) **Clause control instructions:** The instructions in this group control the access to the clauses of a procedure. They establish the selection order in which the alternative clauses should be selected for execution and control the PROLOG backtrack mechanism.

The Warren abstract machine (WAM), as well as most other PROLOG implementations, uses three stacks to accommodate the PROLOG runtime data structures. The *local stack* serves to accommodate the variables of a given clause. The *global stack* accommodates structured data objects such as PROLOG lists or structures. Variables that occur within a structured object will also be allocated on the global stack. The *trail stack* serves to record the binding of variables so that variables can be reset upon backtracking.

3. **A VARIABLE CLASSIFICATION SCHEME**

Through a set of examples we will now develop a modified instruction set. The reader anxious to see the relevance for dealing with the occur-check problem should patiently bear with me. The solution to the occur-check problem will present itself in an almost trivial way at the end of the following discussion.

Whenever an unbound variable occurs as a goal argument, an unbound variable is created on either the local or the global stack and the argument register is loaded with a reference to this location. The process of passing an unbound variable as a goal argument and the subsequent binding of the variable to a nonvariable term can best be understood by looking at an example.

Consider the following fragment of a PROLOG program:

\[ \ldots, g(X), \ldots \]

\[ g(a). \]

\[ g(b). \]

Assume the current goal is \( g(X) \), and \( g(X) \) is also the first goal in which the variable \( X \) occurs (i.e., the variable \( X \) is an unbound variable in the current goal). In order to resolve the current goal the following action takes place:

1. Create an unbound variable, \( X \), in the current environment, place a pointer referencing \( X \) in the argument register, and call the procedure \( g \).
2. Create a choice point for procedure \( g \), and unify the current goal with the unit clause \( g(a) \). The unification process involves dereferencing the argument register—yielding the unbound variable \( X \)—and binding variable \( X \) to \( a \). In this particular case the unbound variable \( X \) needs also to be trailed on the trail stack.

It seems to be a wasted effort to first create an unbound variable before invoking the current goal and then bind the variable immediately after the goal has been invoked. I propose to optimize the passing of unbound variables in the following way: Instead of just having one “tag” indicating whether a variable is unbound, I introduce another “tag” indicating that the variable is unbound but does not exist as a properly tagged unbound variable on either the local or global stack. This “tag” is called `NEW_UNBOUND`. I also distinguish between unbound variables on the local
and global stack through appropriate "tags", but this is only a minor optimization which is independent of the main scheme presented here. Execution of the above example would then proceed as follows:

1. Load a pointer to the location where the variable X is to reside into the argument register; "tag" the argument register as 'NEW_UNBOUND'. Do not create an unbound variable at the location pointed to by the argument register. The location of X will stay undefined as of yet. Call procedure g.

2. Create a choice point for procedure g and unify the current goal with clause g(a). To bind the variable just write the constant a and the appropriate tag into the cell pointed to by the argument register.

In comparison with the first execution model the following steps have been saved:

- creation of an unbound variable,
- dereferencing of the argument register,
- the trail check,
- the trailing of the variable.

The question might arise why the trailing can be saved too. The reason is simple: the argument register contains the tag 'NEW_UNBOUND' and a pointer to an undefined cell. There is no need to trail an undefined memory cell; the fact that this call represents an unbound variable is contained in the argument register, which will be saved as part of the choice point anyway. In those cases where the variable would not have to be trailed, we still save at least checking whether it needs to be trailed.

This scheme can be generalized even further. We do not explicitly establish variable-variable bindings any longer. Consider the following program:

```
?- g(X), . . .
g(Y) :- t(U), s(Y).
s(a).
```

In the old model variable Y contains an explicit pointer (tagged as a reference) which points to the unbound variable X. In the new model the argument register of the current goal will just be saved in location Y and be restored when variable Y is used for the first time in the current clause body. When the variable Y is used for the first time in the clause body, it still contains the tag 'NEW_UNBOUND' and a pointer to the variable X. Subsequent binding of Y will thus immediately cause the variable X to be bound without any dereferencing and trailing.

Of course, there is a price to be paid too. For the proposed scheme to work in all situations, it is necessary to extend the instruction set. Consider the following program:

```
?- g(X), t(X).
g(.).
t(a).
```

When g(X) is invoked, the variable X does not exist yet (remember, only a pointer to the location of variable X is being passed as argument, without really creating the
variable \( X \) on the stack). However, when the goal \( t(X) \) is invoked, we need to know the value of \( X \). This is O.K. as long as the variable \( X \) will eventually become bound within the subtree spanned by the current goal \( g(X) \), because the binding will be written into the location of \( X \). In the present case, however, the variable \( X \) will not become bound within the subtree spanned by \( g(X) \). In order to give the variable \( X \) a value that can be used by the goal \( t(X) \), we need to put the cell representing the variable \( X \) into a well-defined state ("well-defined state" means that the referenced memory cell contains a properly tagged PROLOG data object). This is accomplished through the instruction 'get void \( A\)'. The instruction 'get void' examines the tag of the argument register \( A\). If the tag is 'NEW_UNBOUND', an unbound variable will be created in the cell pointed to by the value field of the argument register. Otherwise 'get void' behaves just like a NOOP instruction. In other words, the creation of unbound variables is delayed as long as possible in the hope that the variable will get bound before execution returns from the present computational subtree. The only possibility for an unbound variable to stay unbound within a computational subtree is when the variable is unified with a void variable. In this case we do have to create an unbound variable on the stack in order to guarantee a well-defined computational status upon return from the subtree. The other instructions are best explained through examples.

**EXAMPLE 1.**

```prolog
?- g(X), ... g(Y) :- t(U), s(Y, U), h(Y).
```

```
put_var X, A1
call g

allocate n
move A1, Y
put_var U, A1
call t
move Y, A1
put_val U, A2
call s
put_val Y, A1
deallocate
execute h
```

Note how the variable \( Y \) is treated in this example. In the clause head the argument register is saved in location \( Y \). The variable \( Y \) now contains a pointer to the variable \( X \) (which does not exist yet) and the tag 'NEW_UNBOUND'. When the goal \( s(Y, U) \) is invoked, the contents of location \( Y \) are just copied into the argument register \( A1 \). However, when \( h(Y) \) is invoked the contents of location \( Y \) are still unchanged ('NEW_UNBOUND' and pointer to \( X \)). If this value is used as the argument of \( h(Y) \), the variable \( X \) might become bound again, disregarding any bindings established by the goal \( s(Y, U) \). However, in order to solve this problem the tag 'NEW_UNBOUND' need only be changed to 'REF'. The 'put_val' instruction just changes all 'NEW_UNBOUND' or 'UNBOUND' tags to 'REF'. Note that it is also necessary to change
‘UNBOUND’ to ‘REF’ because an argument register tagged ‘UNBOUND’ requires that the variables’ location must be examined by the callee before any bindings can be established. If the variable is still ‘UNBOUND’, we get the same tag again and might be trapped in an infinite dereferencing loop. In the above case the argument of \( h(Y) \) will now be a reference to location \( X \). Since the variable \( Y \) was used before in the goal \( s(Y, U) \), it is guaranteed that the location of the variable \( X \) will contain a proper value (remember that upon exit from a subtree all variables used in this subtree are in a well-defined state). In this case we just proceed as we did in the old Warren model.

**EXAMPLE 2.**

\[
g(X) :- t(X, X), \ldots \\
t(a, b).
\]

\[
\begin{align*}
\text{put\_local\_ref} & \quad X, A1 \\
\text{put\_val} & \quad A1, A2 \\
\text{call} & \quad t \\
t : & \quad \text{get\_const} \quad a, A1 \\
& \quad \text{get\_const} \quad b, A2
\end{align*}
\]

If both argument registers of the goal \( t(X, X) \) were tagged ‘NEW UNBOUND’, binding conflicts might arise later on. This is because ‘NEW UNBOUND’ variables get bound directly without looking at the variable’s location. When a variable occurs more than once within the calling goal, it is necessary always to examine the variables location before a particular binding is established. The reason is that in this case the same variable might get accessed (and hence bound) from different locations. Hence, if a variable (on its first occurrence in the clause body) occurs more than once in a goal, we proceed as in the old model. An unbound variable is created and a reference to this variable is loaded into the argument register, thereby requiring the callee to always “dereference” to the variable’s location before attempting to bind the variable. This is accomplished through the instruction ‘put\_local\_ref’, which puts a reference to a local unbound variable into the argument register (this instruction behaves exactly like the ‘put\_var’ instruction of the old model).

**EXAMPLE 3.**

\[
g(X) :- s(X, X), \ldots.
\]

\[
\begin{align*}
\text{put\_nonlocal\_ref} & \quad A1, A1 \\
\text{put\_val} & \quad A1, A2 \\
\text{call} & \quad s
\end{align*}
\]

As in the previous example, we cannot pass a dangling reference to the called procedure. However, we don’t know the status of argument \( A1 \). It might be that the incoming argument \( A1 \) is a ‘NEW UNBOUND’ variable which cannot be passed on, as it is due to the doubling of the variable \( X \). The instruction ‘put\_nonlocal\_ref’ examines the tag of argument \( A1 \). If the argument type is ‘NEW UNBOUND’, it will
create the respective variable (similar to the 'get_void' instruction) and change the
tag to 'REF' to reference the newly created variable. The difference between
'put_local_ref' and 'put_nonlocal_ref' is that the first will always create an unbound
variable in the local environment and the latter examines the type of an argument
that has come in through the clause head and only creates an unbound variable if
the type of the head argument was 'NEW_UNBOUND'. But this 'NEW_UNBOUND'
variable must lie in some earlier environment; hence the name 'put_nonlocal_ref'.

It should be mentioned that the "doubling" of variables within a single goal is a
very rare occasion. According to our analysis of large PROLOG programs this
happens in less than 0.5% of all goals. However, in some instances the occur-check
is necessary for the correct behavior of the program.

The scheme of delaying the actual creation of unbound variables as long as
possible can also be carried over to variables that occur within structures and/or
lists (i.e., variables that would be created on the global rather than the local stack).
Consider the following program fragment:

?- ..., g(X), ...
  g(f(Y)) :- t(Y).

Assume the variable X is a new unbound variable. In this case the structure f(Y)
needs to be created on the global stack. However, it is not necessary to also create
an unbound variable Y on the global stack. It is sufficient to reserve a cell within the
structure without actually initializing this cell to 'UNBOUND'. The reason is that if
the structure f(Y) is unified with an argument register with tag type 'NEW
_UNBOUND', we are guaranteed that this variable occurs only once within the calling
goal (otherwise the tag would have been 'REF' with subsequent dereferencing
yielding 'UNBOUND'). However, this implies that there exists only one pointer to the
newly created structure f(Y) which lies outside of the present computational
subtree. Hence, I can leave the structure in a partially undefined state. I only need
to take care that the structure is in a well-defined state when I return to the parent
clause. For the above example this means that within the structure f(Y) the variable
Y remains in an undefined state when the structure is created. When t(Y) is
invoked, the argument register is loaded with a pointer to the undefined cell within
the structure f( ) and the tag 'NEW_UNBOUND'. Now we can proceed as described
in the other examples above. Whenever we return from the subtree spanned by
t(Y), the undefined slot within the structure f( ) will have been put in a well-
defined state.

This whole scheme is implemented by providing another 'mode' flag. The old
model had only a READ and a WRITE mode in which to execute 'unify' instructions.
Now the 'unify' instructions can also be executed in a WRITE_SAFE mode. The
WRITE_SAFE mode indicates that the primary functor has been matched with an
argument carrying a 'NEW_UNBOUND' tag, in which case the creation of variables
within the structure can safely be delayed (i.e. the structure can remain partially
undefined until we return to the parent clause).

Of course, when a structure is created as a goal argument, it must be completely
defined. It is not permitted to have partially undefined structures as input arguments
to procedures. Hence, the 'put_structure' and 'put_list' instructions set the mode to
To avoid dangling references within structures a new instruction was introduced.

Consider the following PROLOG clause:

\[
g(f(X)) :- h(t(X)).
\]

\[
\text{get\_structure} \quad f, A1
\]

\[
\text{unify\_var} \quad X
\]

\[
\text{put\_structure} \quad t, A1
\]

\[
\text{unify\_unsafe\_val} \quad X
\]

\[
\text{execute} \quad h
\]

Assume the structure \( f(X) \) is unified with an argument tagged \('NEW\_UNBOUND'\). Then the \('get\_structure'\) instruction will set the mode to \('WRITE\_SAFE'\), and the following \('unify\_var'\) instruction will not create an unbound global variable, but only set the temporary variable \( X_1 \) to \('NEW\_UNBOUND'\) and a pointer to the undefined slot of the structure \( f(X) \). However, when the structure \( t(X) \) is created, the value of the global variable \( X \) needs to be accessed. But this variable does not exist yet. Therefore a special instruction, \('unify\_unsafe\_val'\), is needed to bring the variable \( X \) into existence. It should be noted that in the following clause no special action is required:

\[
g(f(X)) :- h(X).
\]

\[
\text{get\_structure} \quad f, A1
\]

\[
\text{unify\_var} \quad A1
\]

\[
\text{execute} \quad h
\]

The argument of the goal \( h(X) \) is just a pointer to the undefined slot of the structure \( f(X) \) with tag \('NEW\_UNBOUND'\). As mentioned earlier, it is guaranteed that upon return from the goal \( h(X) \) everything involved in the computation of \( h(X) \) is in a well-defined state. In particular, the empty slot of the structure \( f(X) \) must have gotten instantiated one way or the other.

One problem that has not been addressed yet is the \"aliasing\" of variables (which is different from just passing a variable down a calling chain). However, this problem is trivial. Whenever two arguments with tag \('NEW\_UNBOUND'\) are unified one variable will be created (i.e. tag \('UNBOUND'\) and value \"self reference\") and the other will be initialized to reference this variable and carry the tag \'REF'. Hence, we are back to the old Warren model.

The reason why a difference is being made between local and global unbound variables is to avoid the restriction that the respective stacks have to lie in a certain order to each other. Also, testing whether a variable needs to be trailed becomes much easier.

Table 1 gives an indication of the savings in memory references by avoiding the explicit creation of unbound variables in memory, dereference, trail, and untrail operations. Actually the runtime savings are even greater, since in most cases checking whether a variable needs to be trailed can be dispensed with in our model. However, the runtime savings are highly dependent on the machine architecture (e.g. how fast certain logical comparisons can be made and whether the involved
## TABLE 1. Comparison between the old and the new model

<table>
<thead>
<tr>
<th>Program</th>
<th>Operation</th>
<th>New model</th>
<th>Old model</th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>create unb var</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>(31 unifications)</td>
<td>deref</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>naive rev.</td>
<td>create unb var</td>
<td>0</td>
<td>465</td>
</tr>
<tr>
<td>(496 unifications)</td>
<td>deref</td>
<td>0</td>
<td>466</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>quicksort</td>
<td>create unb var</td>
<td>50</td>
<td>478</td>
</tr>
<tr>
<td>(376 unifications)</td>
<td>deref</td>
<td>50</td>
<td>479</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>225</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>8-queens</td>
<td>create unb var</td>
<td>0</td>
<td>554</td>
</tr>
<tr>
<td>(2674 unifications)</td>
<td>deref</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>673</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>656</td>
</tr>
<tr>
<td>bucket</td>
<td>create unb var</td>
<td>0</td>
<td>52</td>
</tr>
<tr>
<td>(504 unifications)</td>
<td>deref</td>
<td>0</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>palindrome</td>
<td>create unb var</td>
<td>25</td>
<td>225</td>
</tr>
<tr>
<td>(227 unifications)</td>
<td>deref</td>
<td>98</td>
<td>324</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>16</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>query</td>
<td>create unb var</td>
<td>0</td>
<td>127</td>
</tr>
<tr>
<td>(127 unifications)</td>
<td>deref</td>
<td>644</td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>449</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>440</td>
</tr>
<tr>
<td>times10</td>
<td>create unb var</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>(19 unifications)</td>
<td>deref</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>divide10</td>
<td>create unb var</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>(19 unifications)</td>
<td>deref</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>log10</td>
<td>create unb var</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>(11 unifications)</td>
<td>deref</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ops8</td>
<td>create unb var</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>(13 unifications)</td>
<td>deref</td>
<td>0</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>trail</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>untrail</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
object can be kept in registers). Therefore I have restricted myself to listing the actual saving in memory references.

The programs analyzed are from a set of classical PROLOG benchmark programs [5] plus three other PROLOG programs: 'append' (a list with 30 elements concatenated with a list with 1 element), '8-queens', and 'bucket'. The number of logical inferences (successful unifications) is also given. This number includes only unifications of user defined predicates. System predicates are compiled in-line. The programs that are not part of the benchmark set are listed in Appendix A.

The reason for analysing this particular benchmark set and not large, real-world PROLOG programs is that the savings of the new model depend on the programming style. The reader can judge for himself to what extent the benchmark programs are representative of his/her application. The fact that substantial savings were obtained for all programs within the benchmark set should indicate the viability of the presented scheme.

It should be pointed out that the new tag introduced to implement the presented scheme does not impose any great runtime overhead as compared with the old Warren model (with the exception of the 'get_void' instruction, which is not present in the old model; however, when mode declarations are provided, the 'get_void' instruction can often be dispensed with, since already instantiated input arguments do not need to be handled through a 'get_void' instruction). I have merely extended the set of possible tag values. However, the actions to be taken upon encountering a particular tag are determined by a switch table; branching to a particular action is independent of the table size (at least for small tables).

For conventional microprocessors the introduction of another unification mode, WRITE SAFE, could slow things up a bit. Of course, this is dependent upon the implementation, but in general, a three-way decision is more costly than a two-way decision. Specialized hardware can avoid this overhead.

I deliberately do not provide performance data in terms of LIPS for the usual benchmarks; hence the data given in Table 1 should be regarded as a qualitative result on the new scheme. This has several reasons: the main savings are in the number of memory references; hence the actual runtime savings depend very strongly on such parameters as the processor-speed/memory access-time ratio, bus conflicts, caches and buffers employed, etc. However, for PROLOG processor under construction at the GMD-FIRST we obtained a speedup of roughly a factor 2 for the deterministic append program (specialized hardware with a processor/memory speed ratio of 1:2 and with instructions completely contained in an onboard instruction cache).

4. THE OCCUR-CHECK PROBLEM

By now it should be obvious how the presented variable classification scheme relates to the occur-check problem. Whenever a structure or list is matched against a variable with tag 'NEW UNBOUND', no occur-check needs to be executed, since the tag 'NEW UNBOUND' just indicates that there can be no other pointer to the variable's storage location; hence, no loops can be created. However, whenever a structure or list is unified with a variable carrying the tag 'UNBOUND', the occur-check will have to be done. Aliasing of variables presents no problems. As we have seen in
the discussion above, whenever two ‘NEW UNBOUND’ variables are unified with each other, both their tags will change. One variable will be set to ‘UNBOUND’ and the other to ‘REF’, referencing the other unbound variable. It should be stressed again that this action does not represent an overhead compared with the old model. Hence, aliased variables will always dereference to ‘UNBOUND’. Therefore, aliased variables require an occur-check when they are unified with structured data objects.

The actual places where the occur-check has to be included are easy to find. Consider the following cases:

EXAMPLE (i).

\[ :-p(Z, s(Z)), \ldots \]
\[ p(X, X). \]

EXAMPLE (ii).

\[ :-p(Z, Z), \ldots \]
\[ p(X, s(X)). \]

Since the variable Z occurs more than once in the given goals, it will carry the tag ‘UNBOUND’.

In Example (i) the variable X will inherit the tag ‘UNBOUND’ of Z. When the ‘UNBOUND’ variable X is subsequently unified with the structure s(Z), an occur-check will be executed. In Example (ii) the loop is created in the head of the called clause. The variable X will again inherit the tag ‘UNBOUND’ of Z. However, in this scenario—and this is particular to the Warren instruction set—when the ‘UNBOUND’ variable Z is unified with the structure s(X), the occur-check is not executed right away. The variable Z will be set to point to the top of the global stack, and execution will proceed in ‘WRITE’ mode, i.e., the structure s(X) will be created on the global stack by the code of the clause head (structure copying approach). Only when a variable which has previously been bound to a structure or list is encountered as part of the current structure will the occur-check be executed. This is the case in Example (ii). The variable X had been bound to Z; subsequently Z was bound to the structure s(X). The structure was then copied onto the global stack. During the copying X was encountered, which (via Z) was bound to the structure s(Z); hence an occur-check had to be executed.

This is a point where the presented scheme is too conservative and always assumes the worst case. Consider the following example:

\[ :-p(s(a), Y, Y). \]
\[ p(Z, U, s(Z)) :- \]

Since Y occurs more than once in the current goal, it would carry the tag ‘UNBOUND’. The variable Z gets bound to s(a); however, when copying s(Z) onto the global stack, it is not known that the variable Z, whose value is a structure point to s(a), does not actually point back into the current structure s(Z), thereby creating a loop. In this situation an unnecessary occur-check would be executed. This situation can only be ruled out through a global data analysis of the program. It should be noted that if Y had occurred only once within the given goal, no occur-check would have resulted. In this case Y would have been a ‘NEW UNBOUND’
variable, and \( Y \) could have been safely bound to \( s(Z) \) without occur-check. The reader can easily convince himself that the presented scheme is also independent of the order in which the arguments are unified.

Table 2 shows a comparison between the old and the new model with occur-check included. None of the benchmark programs require an occur-check for their correct execution. As can be seen from Table 2, in all but one program the new model detects automatically that no occur-check is necessary.

The numbers given in Table 2 are the number of times the occur-check would have to be invoked and not the number of memory references made within one occur-check invocation. This was done in order to avoid any assumptions about the actual implementation of the occur-check routine itself, since the occur-check algorithm very strongly depends on the internal data representation. For example, in some cases the occur-check in the WAM becomes quite simple. When a variable dereferences to a location on the local stack, this variable can be bound to a structure or list without further occur-checks. This is because of the stack organization of the WAM. There can be no pointers from the global stack into the local stack. Hence, the structure/list being processed cannot contain the local variable. Even so, it is safe to say—by a comparison between Table 1 and Table 2—that in terms of necessary memory references the new model with occur-check should execute faster than the old model without occur-check. The few unnecessary occur-checks that are still being executed even in the new model are more than compensated by the savings in memory references of the new model.

### 5. CONCLUSION

In summary, the proposed scheme is based on differentiating between variables that occur once or more than once in a given goal. Creation of unbound variables (i.e., tag = 'UNBOUND', value = 'SELF REFERENCE') is delayed as long as possible in the hope that the variable will eventually get bound. Since the argument register explicitly says whether a variable is still unbound, those variables need not be trailed upon binding. If a structure or list within a clause head is unified with an unbound variable which occurs only once within the calling goal, the structure or list can safely be left partially undefined until control is passed back to the parent clause.
The new variable status can be used to detect those situations where an occur-check has to be executed.

APPENDIX A

This appendix lists the benchmark programs that are not part of the benchmark set in [5].

append
:- append(List30,[1], R).
/* List30 is a list with 30 elements. */
app([], X, X).
app([H|T], X,[H|T2]) :- app(T, X, T2).

queens
:- queens([1, 2, 3, 4, 5, 6, 7, 8],[], R). /* 8-queens problem */
queens([], X, X).
queens(List_to_select_from, Occupied_Positions, X) :-
  select(List_to_select_from, Selected_Position, V),
  safe(Occupied_Positions, 1, Selected_Position),
  queens(V, [Selected_Position|Occupied_Positions], X).

select([X|Y], X, X).
select([X|Y], Selected_Position, [X|V]) :-
  select(Y, Selected_Position, V).

safe([],_).
safe([U|T], N, Selected_Position) :-
  nodiag(U, N, Selected_Position),
  M is N + 1,
  safe(T, M, Selected_Position).
nodiag(P, N, Selected_Position) :-
  Occupied_down_diagonal is P + N,
  Selected_Position = Occupied_down_diagonal,
  Occupied_up_diagonal is P - N,
  Selected_Position = Occupied_up_diagonal.

bucket
:- bucket(4, 0, R).
/* This program solves a puzzle. There is a seven and a five liter bucket. By repeatedly filling, emptying and pouring one bucket into the other, one is to reach a state where the seven liter bucket contains 4 liters and the other bucket is
empty. Initially both buckets are empty. The program returns a list of cycle free state transitions from the start to the final state. */

bucket( X, Y, Z) :- solve(s(0, 0), s( X, Y), Z, [s(0, 0), s( X, Y)]).
solve(Start, End, [Start, End], States_visited) :-
   reach(Start, End).
solve(Start, End, [Start|Tail], States_visited) :-
   not_element_of(Next_state, States_visited),
   solve(Next_state, End, Tail, [Next_state|States_visited]).

APPENDIX B

This appendix specifies the instructions necessary to implement the proposed scheme. Only those instructions that are new or have been changed from the original Warren instruction set are described. It should be evident that none of the control and memory management instructions ('call', 'try', 'allocate', 'proceed', etc.) are affected by optimizations concerning the treatment of unbound variables. Instructions such as 'put_const', 'put-nil', 'put-structure', 'put-list' are also not affected by the new scheme. The instruction 'get_value' is also unchanged, since this instruction just calls the unification routine proper.

There are some instructions handling special cases that are ignored here. The reader familiar with the WAM should have no difficulties incorporating the new scheme into these instructions. The unification algorithm in the new model is a straightforward extension of the algorithm in the old model: one only has to consider all the new possible "tag" combinations. If unification with occur-check is desired, the occur-check needs to be executed whenever a structure or list is unified with a variable with tag 'UNBOUND' within the unification routine.

Put Instructions

put_var X, Ai This instruction represents the first occurrence of a variable, X, as a goal argument. The variable X does not occur in the clause head, and the variable occurs only once within the current goal. If the variable occurs more than once, the 'put_local_ref' instruction must be used. The argument register
Ai is initialized with a tag 'NEW_UNBOUND' and a pointer to the variable's location in memory. The variable's location in memory remains undefined.

**put_global_var Ai**  This instruction is a special case of the instruction above. The instruction represents an unbound variable that occurs as a goal argument of the last goal of the clause body. It occurs in the last goal only once; therefore it could also be interpreted as a goal "void variable". If the variable occurs more than once as an argument of the last goal, the 'put_global_ref' instruction must be used. Register Ai is loaded with a tag 'NEW UNBOUND' and a pointer to the top of the global stack. The global stack pointer is incremented.

**put_local_ref X,Ai**  For this instruction to be used the following conditions must hold: the instruction represents the first occurrence of a variable, X, as a goal argument, the variable occurs more than once within the respective goal, and the variable did not occur in the clause head. In this case an unbound variable is created on the local stack and the argument register in initialized to reference the unbound variable.

**put_nonlocal_ref X,Ai**  For this instruction to be used the following condition must hold: the instruction represents the first occurrence of a variable, X, as a goal argument. The respective goal contains the variable more than once, and the variable did occur once within the clause head. If register Ai is a pointer with tag 'NEW_UNBOUND', the variable pointed to will be brought into existence, and X will be set to reference this newly created variable.

**put_global_ref Ai**  This instruction represents an unbound variable that occurs more than once as a goal argument of the last goal of a clause. An unbound variable is created on the global stack, and a pointer to this variable is loaded into register Ai. For the other argument positions of this variable the argument register Ai will then be copied into the respective argument registers through a 'put_value' or (move reg,reg) instruction.

**put_value X,Ai**  This instruction represents a goal argument that is a bound variable. The variable, X, has occurred either previously within the clause body, or more than once within the clause head. If the value of the variable is still 'UNBOUND' or 'NEW_UNBOUND', the instruction will change the tag to 'REF' and load register Ai with the value of variable X (see Example 1 in Section 3). If the variable has occurred only once in the clause head, its first occurrence within the clause body will be handled through either a 'move' instruction (if the variable occurs only once within the goal) or a 'put_nonlocal_ref' instruction. Only occurrences of the variable in later goals will be processed by the 'put_value' instruction (Examples 1–3).

**Get Instructions**

The old 'get var' instruction has been replaced by a simple 'move' instruction (what the 'get var' instruction really only was). The 'move' instruction also reflects better what is going on in the new model. When a clause-head argument is needed at some later time, the argument just gets saved (moved) into the environment and later restored. Only when nasty things such as variable doubling and/or aliasing of variables occur are special actions called for to maintain program correctness.
get_const $C, A_i$ This instruction is basically unchanged from the old 'get_const' instruction; however, if the constant is matched with a pointer of type 'NEW_UNBOUND', the constant $C$ will immediately be written into the location pointed to by $A_i$. No dereferencing takes place, and the trail routine does not get invoked. If $A_i$'s tag is 'REF', the value of register $A_i$ will be dereferenced. If the result is a reference to a variable, that variable is bound to the constant $C$, and the variable is trailed if necessary. Otherwise, the result is compared with the constant $C$, and—if the two values are not identical—backtracking occurs.

get_list $A_i$ This instruction represents the beginning of a list that occurs within the clause head. If the tag of register $A_i$ indicates a 'NEW_UNBOUND' variable, that variable is bound to a new list pointer pointing at the top of the global stack, and execution proceeds in 'WRITE_SAFE' mode. No dereferencing and trailing takes place. Otherwise register $A_i$ is dereferenced. If the result is a variable with tag 'UNBOUND', that variable is bound to the list pointer, the binding is trailed if necessary, and execution proceeds in 'WRITE' mode. If the result of the dereferencing is a list, execution proceeds in 'READ' mode and the list of the clause head will be unified element by element (through the following 'unify' instruction) with the list referenced by register $A_i$. Otherwise, backtracking occurs.

get_structure $f, A_i$ This is basically the same as the 'get_list' instruction. However, instead of processing a list, a structure with functor $f$ is processed.

get_void $A_i$ This instruction represents a "void" variable that occurs in the clause head. In order to avoid dangling references, any unbound variable that does not yet exist on the local or global stack must now be created. This guarantees that all variables involved in solving the current goal will be in a well-defined state upon return from the computational subtree (i.e., if register $A_i$ is a pointer with tag 'NEW_UNBOUND', the location pointed to will be initialized as an 'UNBOUND' variable).

Unify Instructions

unify_var $X$ This instruction represents an unbound variable that occurs as an argument of a structure or list. This instruction is almost identical to the old 'unify_var' instruction; however, when it is executed in 'WRITE_SAFE' mode, the variable $X$ will receive a 'NEW_UNBOUND' tag and a pointer to the top of the global stack. No explicit 'UNBOUND' variable is created on the global stack. If executed in 'WRITE' mode, a new variable will be created on the global stack, and a reference to the new variable will be stored in $X$. If the instruction is executed in 'READ' mode, the next argument of the structure being processed is obtained and stored in $X$.

unify_unsafe_value $X$ This instruction represents a variable that is a structure or list argument but whose value might not necessarily exist yet either on the local or the global stack. If the instruction is executed in 'READ' mode, it gets the next argument of the structure currently being processed and unifies this value with the value of variable $X$. If the instruction is executed in 'WRITE' or
'WRITE_SAFE' mode and $X$ is a pointer with tag 'NEW UNBOUND', the pointer is examined. If the pointer references a location on the global stack, a reference with tag 'REF' to this location is pushed onto the stack, and a variable with tag 'UNBOUND' is created at this address. Remember, a pointer with tag 'NEW UNBOUND' points to an undefined location. Now that a reference to this location has been created, the referenced unbound variable must be brought into existence. If the pointer (with tag 'NEW UNBOUND') points to a cell within the local stack, a new variable with tag 'UNBOUND' will be pushed onto the global stack and the location pointed to by $X$ will be set to reference this newly created variable. If register $Ai$ dereferences to an unbound variable, the location of the variable is examined and the same action takes place as above (of course, the variable itself does not need to be brought into existence). Otherwise, the value of $X$ is pushed onto the global stack. If unification with occur-check is desired, the actions described in the next instruction need to be included.

**unify_value** $X$  This instruction represents a bound variable that occurs as an argument of a structure or list. If the instruction is executed in 'READ' mode, it gets the next argument of the structure currently being processed and unifies this value with the value in $X$. If the instruction is executed in 'WRITE_SAFE' mode, the value of $X$ is pushed onto the global stack. If the value of $X$ has a tag 'NEW UNBOUND', the tag will be changed to 'REF', because the location referenced by the 'NEW UNBOUND' pointer must have been instantiated by the time the 'unify_value' instruction is used. If the instruction is executed in 'WRITE' mode, it needs to be distinguished whether the instruction is executed as part of a previous 'get ...' or 'put ...' instruction, since both those instructions can set the execution mode to 'WRITE'. If the instruction is executed in 'WRITE' mode as part of a 'get ...' instruction, an occur-check needs to be executed if the variable $X$ dereferences to a structure or list. This is necessary because $X$ might dereference to the structure or list that is currently being created, thereby creating a loop. However, if the instruction is executed as part of a 'put ...' instruction, no occur-check is necessary, and the value of variable $X$ will just be pushed onto the stack. If no occur-check is desired, no distinction needs to be made between 'WRITE' and 'WRITE_SAFE' mode, or whether the instruction is executed following a 'put ...' or 'get ...' instruction.

**REFERENCES**